On the Existence of Hazard-Free Multi-Level Logic

Steven M. Nowick* Charles W. O'Donnell

Department of Computer Science
Columbia University, New York, NY, USA
e-mail: nowick@cs.columbia.edu, cwo4@columbia.edu

Abstract

This paper introduces a new method which, given an arbitrary Boolean function and specified set of (function hazard-free) input transitions, determines if any hazard-free multi-level logic implementation exists. The algorithm is based on iterative decomposition, using disjunction and inversion.

Earlier approaches by Nowick/Dill [7] and Theobald/Nowick [8] have been proposed to determine if a hazard-free two-level *logic implementation exists. However, it is well-known that the effects of multi-level transformations are quite complex: since they can both decrease and increase logic hazards in a given circuit. In this paper, a method is proposed to solve the hazard-free multi-level existence problem. The method is proven to be both sound and complete for a large class of multi-level implementations. A novel contribution is to show that, if any hazard-free multi-level solution exists, then a hazard-free solution always exists using only 3 logic levels, in a 3-level NAND or OR-AND-OR structure. Moreover, in this case, it is shown there always exists a unique* canonical *hazard-free 3-level implementation.*

1 Introduction

A key challenge in designing asynchronous circuits is to eliminate hazards, that is, the potential for glitches [9]. For combinational circuits, not only is hazard elimination difficult; it is well-known that it is sometimes impossible [7].

As in the synchronous world, two-level logic is often better understood than multi-level logic. Given a Boolean function, and a specified set of input transitions to be made hazard-free, general algorithms to check for existence of a hazard-free two-level solution have been proposed [7, 8]. However, no complete solution to the "existence problem" for multi-level hazard-free logic has been previously proposed.

The impact of multi-level circuit transformations on hazards is complex, and has been studied in several previous papers [4, 9, 5, 2, 1]. A number of transformations have been identified which are *hazard-preserving*, i.e. do not modify hazard behavior, such as associative law [9]. Other transformations are *hazard-non-increasing*, i.e. which may either preserve or *reduce* hazards, such as a "collapse" of several gates into a single atomic gate, as well as factoring out of common products (i.e. via the 1st distributive law) [4]. Interestingly, these latter transformations — in reverse — become *hazard-increasing*, i.e. they may introduce new hazards, such as "multiplying out". Thus, multi-level logic can both increase and decrease hazard behavior over simpler two-level logic.

An initial solution to the multi-level hazard-free existence problem was proposed by Bredeson [2], using a recursive algorithm. However, a highly-restrictive problem formulation was used: the algorithm does not consider or introduce don't-cares, and there are also apparent bugs and missing theorems. The approach only considers a specialized case: identifying the set of *all* function-hazard-free input transitions of a Boolean function (to be discussed below), and attempting to make them all free of logic hazards. Thus, the starting point of this method is the set of all prime implicants. For almost all asynchronous applications, this is a quite unrealistic formulation, which has limited usefulness. In addition, no completeness proof was provided to justify that the algorithm always finds a solution if one exists.

In this paper, a new method is introduced which, given an arbitrary Boolean function and a user-specified set of (function hazard-free) input transitions, determines if a hazard-free multi-level logic implementation exists. The approach is targeted to a large class of multi-level circuits which are implemented using simple gates (AND, OR, INVERTER; or equivalently, NAND or NOR) The paper significantly extends the recursive approach by Bredeson, subsuming it, and also proves that the new method is both sound and complete. It also includes the first proposed formulation of necessary and sufficient conditions for hazard-free decomposition of incompletely-specified functions, under these two decompositions.

A key contribution of this paper is to show that, if a given function has *any* hazard-free multi-level solution, then it *always* must have *some* hazard-free implementation using only 3 logic levels, i.e. in 3-level NAND or OR-AND-OR form. Thus, in spite of the seemingly wider expressive range of the larger class of multi-level implementations, these two 3-level circuit structures are sufficient to synthesize all of the same hazard-free implementable behaviors. A simple iterative multi-level decomposition algorithm is proposed, to detect if such a hazard-free solution exists, and if so, to constructively generate an implementation. As a final contribution, it is shown is that, whenever some hazard-free solution exists, there is also a unique *canonical* hazard-free 3-level implementation, which can also be generated by an algorithm.

The structure of the paper is as follows. Section 2 gives background on combinational hazards and hazard-free logic, as well as on an existence check, for two-level implementations. Section 3 gives the formal problem statement to be solved, and an intuitive overview of the new method. Section 4 presents formal rules for hazard-free decomposition under both disjunction and inversion, proves their soundness, and introduces the notion of dominance. Section 5 presents two algorithms: an initial recursive multi-level hazard-free

*This work was supported by NSF ITR Award No. NSF-CCR-0086036, NSF Award No. CCR-9988241, and a grant from New York State's NYSTAR Microelectronics Design Center.

decomposition algorithm, which also indicates if a solution exists; and then a final simpler iterative algorithm which is restricted to only 3 levels. In addition, a canonical 3-level implementation is proposed, with its own corresponding decomposition algorithm. Section 6 illustrates the method on two examples. Section 7 briefly shows how implementations produced by the algorithm can be transformed to more common 3-level NAND or OR-AND-OR structures, and Section 8 presents a proof of the completeness of the approach. Finally, Section 9 gives some initial experimental results with a prototype CAD program, and Section 10 presents conclusions and future work.

2 Background

The potential for a glitch in a combinational circuit is called a *hazard* [9]. Hazards fall into two classes: *function hazards* and *logic hazards*. This section focuses on the problem of combinational hazards, and then outlines a method for hazard-free two-level logic minimization. Finally, the issue of the existence of a hazard-free two-level implementation is addressed.

2.1 Combinational Hazards

Since we are concerned with the dynamic behavior of a combinational circuit, we need to formalize the notion of a "multiple-input change" (MIC) or "input transition". A *transition cube* [1, 7] is a cube with *start point* A, *end point* B, and which contains all minterms (i.e., input combinations) that can be reached during a transition from A to B. Inputs are assumed to change monotonically (*i.e.,* at most once) in any order and at any time.

Several assumptions are made on the circuit and environmental models. First, an *unbounded wire delay model* is assumed: gates and wires may have arbitrary finite delays (there are no isochronic fork assumptions). Under this model, a combinational circuit is called hazard-free if it will never glitch (for a given input transition) regardless of the delays on the gates and wires. Second, *generalized fundamental mode* is assumed, where an input transition (with multiple-input changes) must be fully processed and the circuit implementation stabilized, before the environment applies a new input transition. That is, once a specified multiple-input change is complete, no further inputs may change until the circuit has stabilized.

There are a number of practical advantages in using the unbounded delay model to synthesize combinational logic — even when combined with fundamental mode environmental assumptions. Highly robust combinational circuits can be synthesized, without concern for local "isochronic fork" assumptions (equal delays on wire fanouts) or switching thresholds; in addition, a large range of powerful hazard-non-increasing circuit transforms have been identified under this delay model. Once the combinational circuits are synthesized, they do not need to be used in an unknown delay environment: rough timing bounds can be extracted, and they can then be placed in a fundamental mode (i.e. timed) environment.

A function *f* which does not change monotonically during an input transition is said to have a **function hazard** [1, 9]. 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, $f(1111) = 1$. Likewise, f has a *dynamic function hazard* in 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, passing through 0010 and 0011, where f changes more than once.

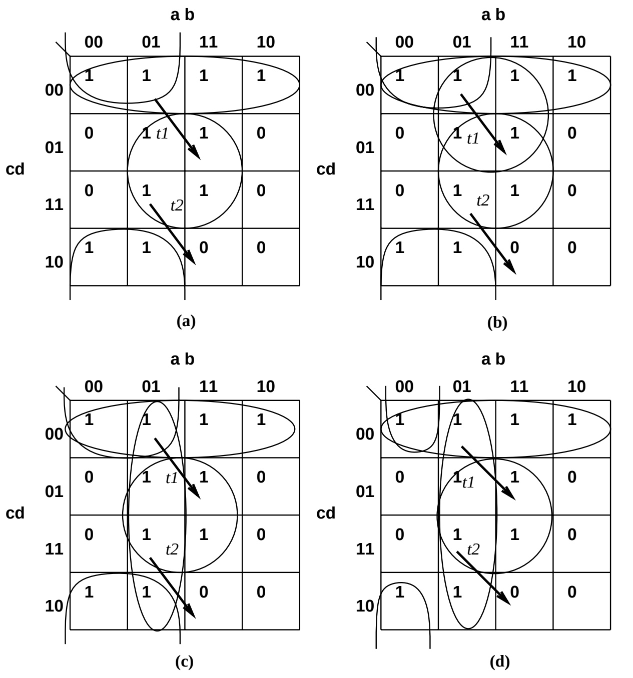

Figure 1. Combinational Hazard Example

Intuitively, a function hazard is a glitch that is inherent in the function itself. Assuming gates and wires may have arbitrary finite delays, there is no guaranteed method to synthesize a circuit which is glitch-free for a transition with a function hazard [9]. Luckily, most sequential synthesis methods, such as burst-mode, only see transitions which are function-hazard-free [6], since they naturally deal with monotonic behavior. Hereafter, *only function-hazard-free input transitions are considered.*

Even if an input transition is function-hazard-free, a circuit implementation 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 given input transition. Given a Boolean function and set of function-hazard-free input transitions, the key synthesis goal is to find a circuit implementation — either two-level or multi-level — which is free of logic hazards.

2.2 Conditions for a Hazard-Free Transition

This subsection now illustrates the complete set of conditions to avoid logic hazards in a two-level sum-of-products implementation (see [1, 7, 6] for details).

Example 1. Again, consider the example in Figure 1(a). The sum-of-products implementation, shown by the given cover, 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, $\bar{c}\bar{d}$ 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: in this example, a fourth cube (i.e., product term [3] $b\bar{c}$ is added, which *remains at 1* throughout the entire input transition. Therefore, the transition is logic-hazard-free. Note that the static $1 \rightarrow 1$ logic hazard is avoided by ensuring that the transition cube $[0100, 1101]$ is *completely contained* in some product of the cover. Such a static transition cube is called a **required cube** [7, 6]; it must be contained in some product of the cover.

Example 2. Next, consider the *dynamic transition* $t2$ in Figure 1(a) from $abcd = 0111$ to $abcd = 1110$. A necessary condition to ensure that $t2$ is hazard-free is to ensure that each $1 \rightarrow 1$ *static sub-transition* of $t2$ is also hazard-free. For example, if *only* input d goes low in $t2$, the circuit is not hazard-free: the sub-transition cube $[0111, 0110]$ is not contained in any product of the cover. The cover of Figure 1(c) solves this problem: for each static $1 \rightarrow 1$ 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 sub-cubes $[0111, 0110]$ and $[0111, 1111]$ are the **required cubes** of this dynamic transition.

Although all static sub-transitions of $t2$ are now hazard-free, the entire transition $t2$ *still* has a dynamic logic hazard: in Figure 1(c), product $\bar{a}\bar{d}$ is initially low; when d goes low, $\bar{a}\bar{d}$ can go high; finally, when a goes high, $\bar{a}\bar{d}$ goes low. As a result, $\bar{a}\bar{d}$ may glitch during the transition, and the glitch may propagate to the OR gate output. This problem is visible in the Karnaugh map: product $\bar{a}\bar{d}$ intersects dynamic transition $t2$ in the middle, but *not* at its *start point* 0111. This is called an **illegal intersection** [7, 6], and the entire dynamic transition cube is called a **privileged cube**. The cover of Figure 1(d) solves the problem. First, as before, each static $1 \rightarrow 1$ sub-transition is hazard-free (since $[0111, 0110] \subseteq \bar{a}b$, $[0111, 1111] \subseteq bd$). Second, no product in the cover illegally intersects the privileged cube. Note that, to avoid the illegal intersection, product $\bar{a}\bar{d}$ is reduced to $\bar{a}\bar{b}\bar{d}$, which is *non-prime*.

Summary. Examples 1 and 2 illustrate the complete set of conditions to avoid MIC logic hazards for $1 \rightarrow 1$ and $1 \rightarrow 0$ transitions. A $0 \rightarrow 1$ transition is regarded as a $1 \rightarrow 0$ transition in reverse, and so similar conditions apply. For $1 \rightarrow 1$ transitions, the entire transition cube is a *required cube* which must be contained in some implicant of the cover. For $1 \rightarrow 0$ and $0 \rightarrow 1$ transitions, two conditions must hold: (i) each maximal $1 \rightarrow 1$ subtransition is a *required cube*, which must be contained in some implicant of the cover; and (ii) the entire dynamic transition forms a *privileged cube*, which cannot be illegally intersected by any implicant of the cover. Finally, for $0 \rightarrow 0$ transitions, there are no constraints: a function hazard-free $0 \rightarrow 0$ transition is free of logic hazards in any sum-of-products implementation [9, 1, 7, 6].

2.3 Hazard-Free Two-Level Logic Minimization

A *hazard-free cover* of a Boolean function is a cover which is hazard-free for a given *specified set* of input transitions:

Hazard-Free Covering Theorem [7, 6]. Given a Boolean function *f* and a specified set of (function hazard-free) input transitions, a cover *C* is free of logic hazards for the set of 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* any specified dynamic transition.

An implicant which has no illegal intersections with any specified dynamic transition is called a *dynamic-hazard-free implicant (dhf-implicant)* (cf. [7, 6]). Only dhf-implicants may appear in a hazard-free cover. A *dhf-prime implicant* is a dhf-implicant contained in no other dhf-implicant.

Given the above discussion, the **two-level hazard-free logic minimization problem** is to find a minimum-cost cover of a Boolean function, using only dhf-prime implicants, where each required cube is covered. The problem is a variant of the classical two-level minimization problem, as solved by Quine-McCluskey and *mincov* methods, where, using only prime implicants, each ON-set minterm must be covered [3].

An exact *hazard-free two-level logic minimization algorithm* has been proposed [6, 7], with 3 steps: (i) generate dhf-prime implicants; (ii) construct a dhf-prime implicant table; and (iii) find a minimum-cost cover. (For a complete step-by-step example, including details of dhf-prime generation, see [7].) A more efficient exact algorithm using implicit data structures has also been proposed [8].

A simple example illustrating hazard-free two-level logic minimization is shown in Figure 2. There are 5 specified input transitions, including 3 dynamic and 2 static ones, indicated in Figure 2(a). Each transition cube is indicated by a dotted oval in Figure 2(b). The required cubes for the $1 \rightarrow 1$ static transition (i.e., the entire transition cube), and for the $1 \rightarrow 0$ and $0 \rightarrow 1$ dynamic transitions (i.e., for each maximal static sub-transition), are indicated by shaded ovals in Figure 2(b); a minimum-cost cover of dhf-prime implicants is also shown by 3 thick unshaded ovals.[1] The corresponding minimum-cost gate-level implementation is illustrated in Figure 2(c).

2.4 Existence of a Hazard-Free Two-Level Implementation

In general, given an arbitrary Boolean function and specified set of (function-hazard-free) input transitions, the conditions of the Hazard-Free Covering Theorem may be *unsatisfiable*, and therefore *no hazard-free solution may exist* [7, 6]. As an example, consider Figures 6(a) and (b), taken from [7, 6]. Here, there is no implicant which simultaneously (i) contains required cube ABD, while at the same time (ii) avoids illegal intersections with both privileged cubes. In particular, the implicant ABD illegally intersects the bottom privileged cube, and implicant BD illegally intersects the top left dynamic transition, so the required cube cannot be covered.

Two methods have been proposed to check if a two-level hazard-free solution exists. A simple approach is to use the hazard-free minimization algorithm of Nowick and Dill [7, 6]. Once all dhf-prime implicants are generated, a dhf-prime implicant table is formed. If no two-level hazard-free solution exists, there will be some required cube (e.g., in the above example, ABD) *not covered* by any dhf-prime in the table.

A more efficient existence check has also been proposed, which avoids the explicit generation of all dhf-prime implicants (see [8] for details). Instead, each required cube is examined in turn; if it has any illegal intersection, it is iteratively expanded until it has no more illegal intersections. If any such expanded required cube hits a 0-point (OFF-set), then no hazard-free two-level implementation exists; otherwise a hazard-free solution has been constructively generated.

3 Problem Statement and Overview of Approach

The previous section reviewed issues for two-level hazard-free logic. In the remaining sections, the issue of multi-level hazard-free logic is addressed.

Given the complex interaction of multi-level transformations on logic hazards, the goal of this paper is to determine precisely *when a hazard-free multi-level logic implementation exists*, for a given Boolean function. As in Section 2, several assumptions are made on circuit and environmental models. First, an *unbounded wire delay model* is assumed: gates and wires may have arbitrary finite delays (there are no isochronic fork assumptions). Second, *generalized fundamental mode* is assumed, where an input transition (with multiple-input changes) must be fully processed and the circuit implementation stabilized, before the environment applies a new input transition.

[1] For simplicity, the example illustrates the minimization of number of products, but other cost functions can be incorporated as well [8].

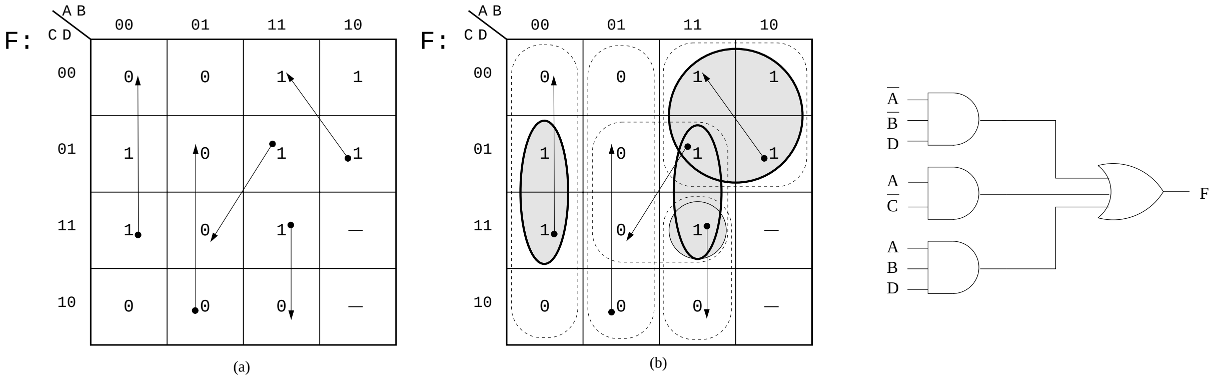

Figure 2. Hazard-Free Logic Minimization Example

Finally, in this paper, a limited class of multi-level circuit implementations is considered: those constructed only using AND, OR, and INVERTER gates. This class is quite large: circuits with NAND and NOR gates can be shown to map with identical hazard behaviors by transforming them to AND-INV and OR-INV sequences, respectively, and hence the results apply to these gates as well.[2]

Problem Statement. Given a Boolean function F, and specified set of (function-hazard-free) input transitions T, determine if there exists a *multi-level* combinational logic implementation which is free of logic hazards for every transition in T.

Overview of Approach. Under the above assumptions, the next sections present a complete method to solve this problem. Before proceeding to technical details, it is useful to consider an informal overview of the proposed approach, as shown in Figure 3. Given function F and set of input transitions T, a gate-level implementation is derived from output to inputs, through decomposition. As shown in Figure 3(a), initially no gates have been allocated for the implementation. If a two-level hazard-free solution exists (see Section 2.4), no decomposition is required, and the algorithm terminates, since a solution has been found. If no two-level solution exists, then an OR gate is allocated, and a disjunctive functional decomposition is performed, as in Figure 3(b). The procedure recurs; in this example, two of the decomposed subfunctions (f_1, f_3) have two-level solutions, as shown in Figure 3(c), so their recursion terminates. However, subfunction f_2 does not; in this case, f_2 is *inverted*, in the hopes that $\overline{f_2}$ *may* have a hazard-free solution. The process then recurs for $\overline{f_2}$.

If at any point, no further decomposition is possible and a subfunction is inverted twice (i.e. looping behavior), the algorithm terminates and no multi-level hazard-free solution exists. However, for the given example, after further iteration, the algorithm terminates and a hazard-free multi-level decomposition is obtained, as shown in Figure 3(d). This figure shows the constructed circuit implementation's structure: chains of OR gates sandwiched between inverters, where at the terminals (i.e. circuit inputs) there are AND-OR subcircuits.

[2]An alternative formulation is to solve the multi-level hazard-free existence problem for *any* network of *2-input combinational gates*. It can be shown that all 2-input gates, except XORs, can be modelled using AND/OR/INV, and that each XOR can be replaced with an equivalent AND-OR subcircuit with no worse hazard behavior. Hence, the proposed method can find if any hazard-free multi-level circuit exists, considering arbitrary networks of 2-input gates.

4 Rules for Hazard-Free Decomposition

This section formalizes the two decomposition operators shown in Figure 3: disjunction and inversion. In the next section, these two operators are combined in a constructive algorithm to determine if a hazard-free multi-level solution exists.

In general, given a Boolean function F and specified set T of input transitions (hereinafter called (F, T)),[3] a hazard-free decomposition allocates a logic gate, and generates new subfunction(s) and their respective new specified sets of input transitions $(F1, T1), \ldots, (FN, TN)$. A decomposition is called *hazard-free* if it introduces no logic hazards at the allocated gate. Effectively, the problem of finding a hazard-free implementation is then pushed down to the next level, where each subfunction (Fi, Ti) has its own (hopefully) simplified hazard requirements, which must in turn be satisfied. If the application of the various decomposition rules terminates, and produces a gate-level circuit, then the final resulting multi-level circuit implementation is guaranteed logic-hazard-free.

The goal of this section is to define precise conditions for hazard-free decomposition using two operators: disjunction and inversion.

4.1 Rules for Disjunctive Decomposition

4.1.1 Overview

Disjunctive decomposition corresponds to the operator which transforms Figure 3(a) to Figure 3(b): an OR gate is allocated, new subfunctions are generated, and then decomposition can recursively be applied to the subfunctions, to produce a final gate-level multi-level circuit.

The disjunctive decomposition rules are illustrated in Figure 4. A function F with its specified set T of input transitions is shown in Figure 4(a). It has 3 specified input transitions: $1 \rightarrow 1$, $0 \rightarrow 0$, and $1 \rightarrow 0$. Each input transition has its own decomposition rule, mapping it to one or more decomposed subfunction(s), in this case $F1$ and $F2$ in Figures 4(b) and (c), respectively ($F2$ will then be later adjusted in Figure 4(d); see below).

An intuitive overview of the rules is as follows. An initial function F, with set T of input transitions, gives rise to several required cubes. Required cubes arise from two cases: (i) $1 \rightarrow 1$ transitions, where the entire transition cube is a required cube (e.g., $BC'D'$ in this example); and (ii) $1 \rightarrow 0$ and $0 \rightarrow 1$ transitions, where each maximal $1 \rightarrow 1$ subtransition forms a required cube (AB and AC' for the $1 \rightarrow 0$

[3]In the rest of this paper, for simplicity, both (F, T) and F will be called 'functions', though the former consists of both a function and a set of specified input transitions.

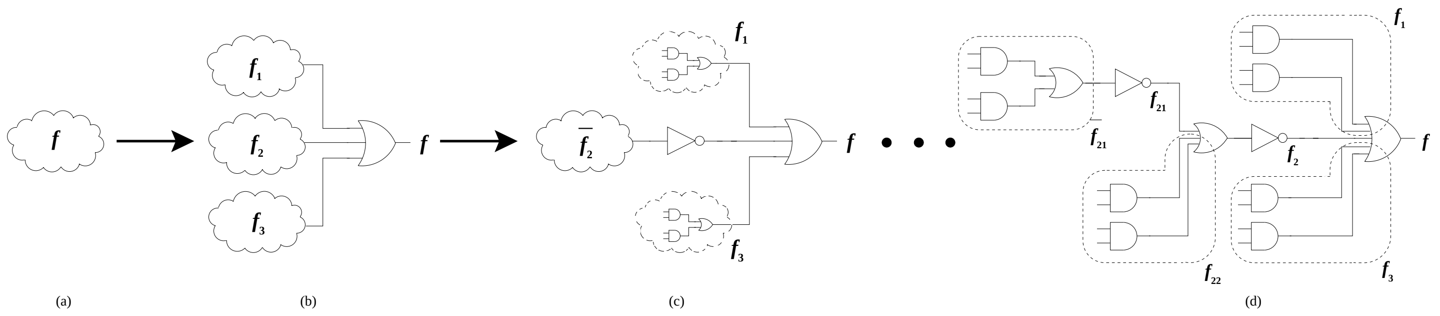

Figure 3. Overview of Proposed Decomposition

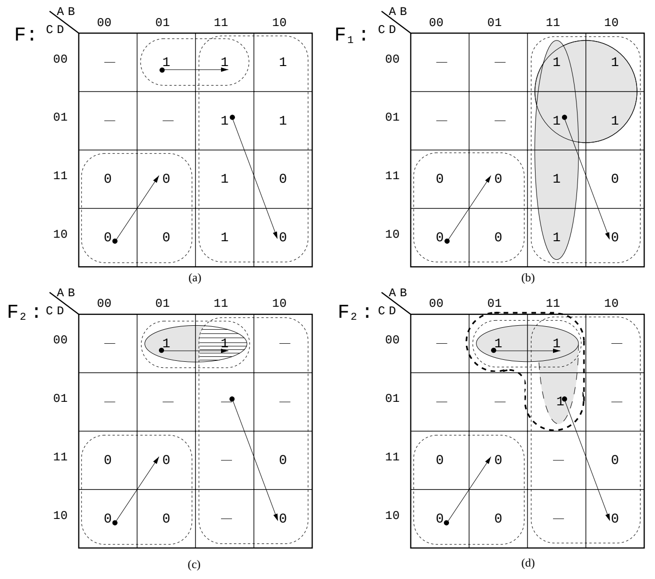

Figure 4. Disjunctive Decomposition Example

transition in this example). Disjunctive decomposition requires that each such required cube be mapped to *at least one subfunction;* in the remaining subfunctions, the required cube can be filled in with all "don't-care" values. In fact, different required cubes may be mapped to the same or to different subfunctions; any choice is permissible, as long as each required cube is mapped to at least some subfunction.

In summary, each $1 \rightarrow 1$ transition must be mapped to least *one* subfunction (in this example, the $1 \rightarrow 1$ transition is only mapped to $F2$, not to $F1$). In contrast, each $0 \rightarrow 0$ transition must be mapped to *all* subfunctions (in this example, the $0 \rightarrow 0$ transition is mapped to both $F1$ and $F2$). Finally, each $1 \rightarrow 0$ or $0 \rightarrow 1$ transition must be mapped to *all* subfunctions, and its 0 values must similarly be specified in every subfunction. However, each one of its required cubes must only appear in at least in *one* of the subfunctions; in other subfunctions, the required cube may be filled in with don't-cares (in this example, the $1 \rightarrow 0$ transition is mapped to both $F1$ and $F2$, and its OFF-set minterms appear in both subfunctions, but the required cubes appear only in $F1$).

Finally, as an optional post-processing step, certain subfunctions may be *further expanded* to explicitly indicate how certain don't-cares will be forced to 1 values (see subsequent subsection).

4.1.2 Basic Disjunctive Decomposition Rules

The following is a formal summary of the rules to ensure a hazard-free disjunctive decomposition. Consider again the initial decomposition of (F, T), shown in Figure 4(a), into two subfunctions: $(F1, T1)$ (in Figure 4(b)) and $(F2, T2)$ (in Figure 4(c)).

(i) $1 \rightarrow 1$ **Transition.** A $1 \rightarrow 1$ transition must be mapped to *at least* one subfunction. In any remaining subfunctions, the transition need not appear, and its required cube can be filled with don't-cares. (In the above example, the $1 \rightarrow 1$ transition is only mapped to $F2$; it does not appear in $F1$ and its minterms are all set to don't-care, except those covered by other required cubes.)

(ii) $0 \rightarrow 0$ **Transition.** A $0 \rightarrow 0$ transition must be mapped to every subfunction. (In the above example, the $0 \rightarrow 0$ transition appears in both $F1$ and $F2$.)

(iii) $1 \rightarrow 0$ **(**$0 \rightarrow 1$**) Transition.** A dynamic transition, along with all its OFF-set minterms (i.e. 0 minterms), must be mapped to *every* subfunction. However, each required cube of the dynamic transition must be mapped to *at least* one subfunction; in the remaining subfunctions, the required cube may be filled in with don't-cares. (In this example, both required cubes, AB and AC', are mapped to the same subfunction $F1$, but they could also have been mapped to two distinct subfunctions.)

Interestingly, Rule (iii) results in a new class of input transitions: *dynamic transitions with don't-cares,* as shown in $F2$ in Figure 4(c). Such an input transition still retains all the original OFF-set minterms (i.e., 0 points), and it still has a corresponding privileged cube which cannot be illegally intersected. However, *it contains no required cubes.*[4] These new input transitions define a region where the OFF-set values must be preserved, but there are no further covering requirements: don't-cares may or may not be covered by products, but if they are, no illegal intersection can occur. If no product intersects the transition, it is a $0 \rightarrow 0$ transition; if a product intersects, it becomes a $1 \rightarrow 0$ transition. In either case, the privileged cube ensures that the transition is monotonic, and thus logic hazard-free.

4.1.3 An Optional Expansion Step

The above disjunctive decomposition rules are complete, and no further requirements are needed for correct decomposition. However, the rules leave *implicit* some functional values.

As an example, consider again the Boolean function (F, T) in Figure 4(a), with its two decomposed subfunctions, $(F1, T1)$ in Figure 4(b) and $(F2, T2)$ in Figure 4(c).

[4]The intersecting required cube $BC'D'$ is a separate issue, which will be discussed shortly.

Note that while $(F2, T2)$ is a correct decomposed subfunction, one of its apparent don't-cares, $ABC'D$, will always be forced to a 1 value in any hazard-free implementation, as shown in Figure 4(d). For a hazard-free two-level implementation, the reason is that the required cube, $BC'D'$, illegally intersects privileged cube A (with start point $ABC'D$). Even for the case of hazard-free multi-level implementations (which do not have notions of cube covers), the arrow of the dynamic transition ($ABCD : 1101 \rightarrow 1010$) still indicates that the transition must be function and logic hazard-free; required cube $BC'D'$ produces a function hazard in the transition, unless the ON-set of the function is "expanded" to cover $ABC'D$. This expansion is not explicitly required: any hazard-free implementation will accomplish it. But if it is desired to have all don't-cares explicitly assigned to their final values, an expansion step can be performed.

4.1.4 Correctness

The above rules are now shown to be both necessary and sufficient for a hazard-free disjunction decomposition.

Disjunctive Decomposition Theorem. Given function F and specified set of input transitions T, and given any disjunctive decomposition into subfunctions $(F1, T1), \ldots, (FN, TN)$. Then the decomposition is hazard-preserving if and only if the above rules are met.

Proof. Consider Figure 3(a) and (b).

Rule (i). This rule precisely guarantees that at least one subfunction, and corresponding input wire into the OR gate, is held stable and glitch-free at 1. If the rule is met, the OR gate and hence output F will be stable and hazard-free at 1. Conversely, if the OR gate holds its output at 1, then one of the subfunctions must hold its output stable at 1, and, to do so, the rule must be met by the set of subfunctions.

Rule (ii). This rule precisely guarantees that all subfunctions, and hence all input wires of the OR gate, are held stable and glitch-free at 0. If the rule is met, then no OR gate input will have a hazard, and each subfunction will be stable at 0, hence the OR gate output will stay hazard-free at 0. Conversely, if the OR gate holds its output at 0, all its inputs must remain hazard-free at 0, hence the rule must be met for each subfunction.

Rule (iii). Consider without loss of generality a $1 \rightarrow 0$ transition. This rule precisely guarantees that all subfunctions, and hence all input wires of the OR gate, make monotonic and glitch-free changes (at the correctly specified input combinations) from 1 to 0. If the rule is met, then since there are no illegal intersections in any subfunction, and no possibility of incorrect setting of don't cares, each OR gate input makes a monotonic transition ($1 \rightarrow 0$ or $0 \rightarrow 0$); and each $1 \rightarrow 1$ subtransition of F is made glitch-free, since some subfunction contains the corresponding required cube; hence the OR gate output will be hazard-free. Conversely, if the OR gate output is hazard-free, each $1 \rightarrow 1$ subtransition must be hazard-free, and only monotonic OR gate input transitions are allowed for the entire $1 \rightarrow 0$ transition, which is equivalent to requiring that the rule be met.

4.1.5 Dominating Functions

During decomposition, it will be useful to consider how some subfunctions can dominate others.

Definition: Functional Dominance. Given two functions (F_i, T_i) and (F_j, T_j), function (F_i, T_i) **dominates** function (F_j, T_j), indicated as $(F_j, T_j) \preceq (F_i, T_i)$, if (i) for each minterm m, $(F_j(m) = 0) \rightarrow (F_i(m) = 0)$, and $(F_j(m) = 1) \rightarrow (F_i(m) = 1)$; and (ii) each specified input transition of T_j is also a specified input transition of T_i (i.e., $T_j \subseteq T_i$).

Intuitively, (F_i, T_i) dominates (F_j, T_j) if the Boolean function F_i covers F_j (the two functions agree wherever F_j is 0 or 1), and if the specified input transitions in T_i also cover (i.e. contain) the specified input transitions in T_j. In this case, (F_i, T_i) captures all the functional and hazard requirements of (F_j, T_j), and possibly more (i.e. fewer don't-cares, additional specified input transitions).

Example. As a simple example, the function (F, T) of Fig. 4(a) dominates the decomposed subfunction $(F1, T1)$ Fig. 4(b), since: (i) every 0 (1) minterm of $F1$ is a corresponding 0 (1) minterm of F (in fact, F has fewer don't-cares than $F1$); and (ii) every specified input transition of $T1$ is also a specified input transition of T.

Two useful lemmas and a corollary are immediate from the above definition. They indicate that any hazard-free implementation (either two-level, or arbitrary multi-level) of a dominating function is *always* a hazard-free implementation of a dominated function:

Lemma 1. If $(F_j, T_j) \preceq (F_i, T_i)$, then for every hazard-free *two-level* implementation C of (F_i, T_i), C is also a hazard-free two-level implementation of (F_j, T_j).

Proof. Since function F_i covers function F_j, then ON-set(F_j) $\subseteq$ ON-set(F_i), so C is also a valid implementation of function F_j. Since $T_j \subseteq T_i$, and C is free of logic hazards for all input transitions T_i, it is also free of logic hazards for all transitions of T_j.

Corollary 1. If $(F_j, T_j) \preceq (F_i, T_i)$, and if (F_j, T_j) has no hazard-free two-level implementation, then neither does (F_i, T_i).

Proof. Immediate from contrapositive of Lemma 1.

Lemma 2. If $(F_j, T_j) \preceq (F_i, T_i)$, then for *every* hazard-free (i.e., arbitrary *multi-level*) implementation C of (F_i, T_i), C is also a hazard-free implementation of (F_j, T_j).

Proof. Follows same reasoning as Lemma 1 proof.

4.1.6 Maximal Decomposition: a Special Case

While disjunctive decomposition allows any decomposition which satisfies the above rules, a special case will be important: "maximal (disjunctive) decomposition."

Definition. A **maximal (disjunctive) decomposition** is a disjunctive decomposition where (a) each required cube is assigned to a *distinct* subfunction, and (b) each subfunction has maximum permissible don't-cares.

The basic idea of maximal decomposition is that the resulting set of subfunctions are "primitive" or "atomic", and cannot be further simplified by disjunctive decomposition: each subfunction contains exactly one distinct required cube of the original function, which cannot be further reduced. Furthermore, each subfunction includes as many don't-cares as allowed under definition of disjunctive decomposition; i.e. only 0/1 minterms are set if required. As a result, maximal decomposition is a canonical and primitive transformation. It will shown later that it is sufficient to limit the hazard-free decomposition algorithm to only use maximal decomposition.

The following lemma characterizes how to construct a maximal decomposition of a function (f, T):

Lemma: Maximal Decomposition Characterization. Given a function (f, T), and a maximal decomposition into subfunctions $(f1, T2), \ldots, (fn, Tn)$. Each decomposed subfunction (fi, Ti) has the following properties: (a) it contains *exactly one* (unique) required cube of (f, T), which defines the *entire* ON-set of the subfunction fi; (b) it contains *every* OFF-set cube of (f, T), therefore OFF-set(f) = OFF-set(fi); (c) if the included required cube is derived from a

$1 \rightarrow 1$ specified input transition of T, then this single $1 \rightarrow 1$ input transition is also included in the new set of transitions, Ti;[5] (d) *every* specified dynamic input transition of T $(1 \rightarrow 0, 0 \rightarrow 1)$ is included in Ti (hence Ti has the same privileged cubes dynamic transitions and privileged cubes as T); and (e) *every* $0 \rightarrow 0$ specified input transition of T is included in Ti (hence Ti has the same static-0 transitions as T).

Proof. Immediate from the definitions of disjunctive decomposition and maximal decomposition.

Examples. An example of a maximal decomposition is shown in Figure 6 (to be discussed in detail later); each of the six original required cubes is assigned to a unique decomposed subfunction, F1-F6. In contrast, the decomposition in Figure 4 is not maximal: subfunction F2 contains 2 distinct required cubes.

The following useful theorem compares a maximal decomposition X of a function with any other arbitrary disjunctive decomposition Y, and indicates that each subfunction of X is always dominated by at least one of the decomposed subfunctions of Y. Intuitively, this theorem indicates that maximally decomposed subfunctions are "primitive": i.e., they are the bases on which any other decomposed subfunctions are constructed. This result will be important in Section 8 to justify why the hazard-free decomposition algorithm will be restricted to only using maximal decomposition.

First Maximal Decomposition Theorem. Consider any two hazard-free disjunctive decompositions of function (F, T), call them X and Y, where X is a maximal decomposition. Let (Fi, Ti) be any subfunction of (F, T) resulting from the maximal decomposition X. Then there exists a subfunction (Gj, Vj) of (F, T) resulting from decomposition Y such that $(Fi, Ti) \preceq (Gj, Vj)$.

Proof. Consider any subfunction (Fi, Ti) resulting from the maximal decomposition X. The above Maximal Decomposition Characterization lemma precisely defines the function Fi and set of specified input transitions Ti. Suppose (Fi, Ti) contains the original required cube ri. Then, from the above Disjunctive Decomposition Rules, under any arbitrary disjunctive decomposition Y, there must be some subfunction (Gj, Vj) which also contains ri (whether it was originally in a static-1 or a dynamic input transition of T). The Disjunctive Decomposition Rules require that function Gj covers Fi, i.e., wherever Fi is specified at 0 (or 1), Gi will have the same specified value. Furthermore, these rules indicate that Vj contains all the specified input transitions of Ti. Therefore, $(Fi, Ti) \preceq (Gj, Vj)$.

Example. As a simple example, consider the maximal decomposition (F1-F6) of Figure 6, compared with the original function (F, T). The latter can be regarded as a trivial decomposition: (F, T) is decomposed into itself (i.e. no decomposition). Clearly each maximally decomposed subfunction, $(F1, T1)$ through $(F6, T6)$, is dominated by (F, T).

The First Maximal Decomposition Theorem considered the restricted case of two decompositions of the *same* function, (F, T). The following more general theorem compares decompositions of two *distinct* initial functions, (F, T) and (G, V), where (G, V) dominates (F, T), and shows that this dominance is preserved after maximal decomposition (F, T), against any arbitrary disjunctive decomposition of (G, V):

[5]If the required cube is derived only from a *dynamic* $(1 \rightarrow 0, 0 \rightarrow 1)$ transition of T, then no $1 \rightarrow 1$ input transition is included in Ti.

Second Maximal Decomposition Theorem. Consider any two functions, (F, T) and (G, V), where $(F, T) \preceq (G, V)$, and respective hazard-free disjunctive decompositions, call them X and Y, where X is a maximal decomposition of (F, T) and V is any disjunctive decomposition of (G, V). Let (Fi, Ti) be any subfunction of (F, T) resulting from maximal decomposition X. Then there exists a subfunction (Gj, Vj) of (G, V) resulting from decomposition Y such that $(Fi, Ti) \preceq (Gj, Vj)$.

Proof. An immediate extension of the proof of the First Maximal Decomposition Theorem.

4.2 Rules for Inversion

Inversion is the second decomposition operator, as illustrated in transforming subfunction $F2$, from Figure 3(b) to Figure 3(c): an INVERTER gate is allocated, a new subfunction $(\overline{F2})$ is generated, and then decomposition can recursively be applied to this subfunction, to produce a final gate-level multi-level circuit.

Intuitively, inversion modifies a function in the obvious way: 1 and 0 entries are toggled, don't cares are preserved, and specified input transitions are also preserved. More subtly, required cubes are added around former OFF-set minterms, and eliminated from former ON-set minterms.

An example is shown in Figure 6(h) and Figure 7(a). Subfunction $F6$ is transformed through hazard-free decomposition into subfunction $\overline{F6}$. Note that the maximal OFF-set cubes of $F6$ – $AB'C$, ACD' and $A'B'C'D$ –, become required cubes of the inverted function $\overline{F6}$. Likewise, the required cube, ABD, of $F6$ becomes an OFF-set cube of $\overline{F6}$.

Also note that the top dynamic transition, from $ABCD =$ 0100 to 0001, contains only don't-cares and 0 values in Figure 6(g), so effectively the "start point" which must be covered by any intersecting implicant is $ABCD = 0100$. After inversion, in Figure 6(h), the new "start point" is *swapped:* to become $ABCD = 0001$, which must be covered by any intersecting implicant. In summary, even in the presence of don't-cares, the "start point" (i.e., ON-set end of the arrow) is swapped on each inversion.[6]

The following useful theorem indicates that dominance is preserved under the inversion operator:

Inversion Theorem. Let (F_i, T_i) and (F_j, T_j) be two Boolean functions with specified input transitions, and let (F_i', T_i') and (F_j', T_j') be the respective results of the inversion step. If $(F_j, T_j) \preceq (F_i, T_i)$ (i.e., before inversion), then $(F_j', T_j') \preceq (F_i', T_i')$ (i.e., after inversion).

Proof. Immediate from the definitions of inversion and dominance. Functionally, the 0 (1) minterms of F_j are covered by the 0 (1) minterms of F_i before inversion; since inversion preserves all don't-cares and toggles 0's and 1's, this coverage is also preserved after inversion. Likewise, no specified input transitions are added or removed by inversion, hence the input transitions of T_j' are covered by T_i' (i.e. after inversion).

5 A Hazard-Free Multi-Level Decomposition Algorithm

Given the above formal definitions of the two decomposition operators — disjunction and inversion – a hazard-free multi-level decomposition algorithm is now presented. The goal is to construct a hazard-free multi-level implementation, for

[6]Interestingly, it is even possible to have privileged cubes containing *only don't cares*; for example, if the required cube $A'B'C'D$ of $\overline{F6}$ is mapped to a further decomposed subfunction, a privileged cube with only don't-cares would be copied across $A'C'$ would be mapped to all remaining subfunctions. In this case, the designated "start point" (which would have had a 1 value) is still tracked and swapped on each inversion.

a given Boolean function F with specified set T of input transitions. First, a generic recursive algorithm, called Algorithm 0, is presented. Then, it is refined into the final iterative algorithm, Algorithm 1, which uses simpler hazard existence checks, and which will be used to generate 3-level implementations.

5.1 An Initial Recursive Algorithm

The algorithm was informally introduced in Figure 3. A simplified pseudo-code version is given in Figure 5(a).

```
multi-level-decomp$(F,T)$
if (exists-2-level-sol $(F,T)$)
    return (2-level-min $(F,T)$);
else {$((F1,T1),\ldots,(FN,TN))$ = max-decompose $(F,T)$;
    if $(N == 1)$ return ({"no solution"});
    for $i = 1$ to $N$ do {
        if (exists-2-level-sol $(Fi,Ti)$)
            $gi$ = 2-level-min $(Fi,Ti)$;
        else $gi$ = multi-level-decomp (invert$(Fi,Ti)$); }
    return ( $\{g1,\ldots,gn\}$ ) }

main-decomp$(F,T)$
if (exists-2-level-sol $(F,T)$)
    return (2-level-min $(F,T)$);
else {$((F1,T1),\ldots,(FN,TN))$ = max-decompose $(F,T)$;
    for $i = 1$ to $N$ do {
        if (exists-2-level-sol $(Fi,Ti)$)
            $gi$ = 2-level-min $(Fi,Ti)$;
        else $gi$ = multi-level-decomp (invert$(Fi,Ti)$); }
    return ( $\{g1,\ldots,gn\}$ )
```

(a) Algorithm 0. An Initial Approach (Recursive).

```
multi-level-decomp$(F,T)$
Sol = {};
$((F1,T1),\ldots,(FN,TN))$ = max-decompose $(F,T)$;
for $i = 1$ to $N$ do {
    $gi$ = get-single-cube-sol $(Fi,Ti)$;
    if ($gi$ exists) Sol = Sol $\cup$ $\{gi\}$;
    else { $(Fi',Ti')$ = invert$(Fi,Ti)$;
        $((Fi'_1,Ti'_1),\ldots,(Fi'_M,Ti'_M))$ =max-decompose $(Fi',Ti')$;
        Sol1 = {};
        for $j = 1$ to $M$ do {
            $gj$ = get-single-cube-sol $(Fi'_j,Ti'_j)$;
            if ($gj$ exists) Sol1 = Sol1 $\cup$ $\{gj\}$;
            else { $(Fi_j,Ti_j)$ = invert$(Fi'_j,Ti'_j)$;
                $gk$ = get-single-cube-sol $(Fi_j,Ti_j)$;
                if ($gk$ exists) Sol1 = Sol1 $\cup$ $\{\{gk\}\}$;
                else exit ({"no solution"}); }}
        Sol = Sol $\cup$ Sol1; }}
return (Sol);
```

(b) Algorithm 1. The Final Approach (Iterative).

Figure 5. Two New Algorithms for Hazard-Free Multi-Level Decomposition.

The algorithm follows the basic approach given in Figure 3. The "main-decomp" driver is called on a given function F and specified set T of input transitions. If a hazard-free two-level solution exists, it is immediately returned. One of the two-level existence checks of Section 2.4 is used. If no two-level solution exists, disjunctive decomposition is performed. The core of the algorithm is the for-loop, which is activated when disjunctive decomposition succeeds. In this case, if a subfunction has a two-level hazard-free solution, it is returned. Otherwise, an inversion is performed, followed by a call on the inverted function to the core recursive function "multi-level-decomp". In each case, the allocated new gates (OR,INV) can be reconstituted from the nesting of the returned subnetworks. Finally, the "N==1" test is to check for termination with no solution, which occurs when there is a loop with no progress. in this case, a subfunction (F,T) cannot be disjunctively decomposed and has no two-level solution; after inversion it still cannot be disjunctively decomposed and has no two-level solution.

The algorithm is sound: if it finds a multi-level solution, it is always hazard-free. This is immediate, since it uses only hazard-free decomposition steps (disjunction, inversion) which were proved sound in the previous section, as well as safe hazard-free two-level logic minimization. In Section 8, it will shown to be complete: it always finds a multi-level hazard-free solution, if one exists.

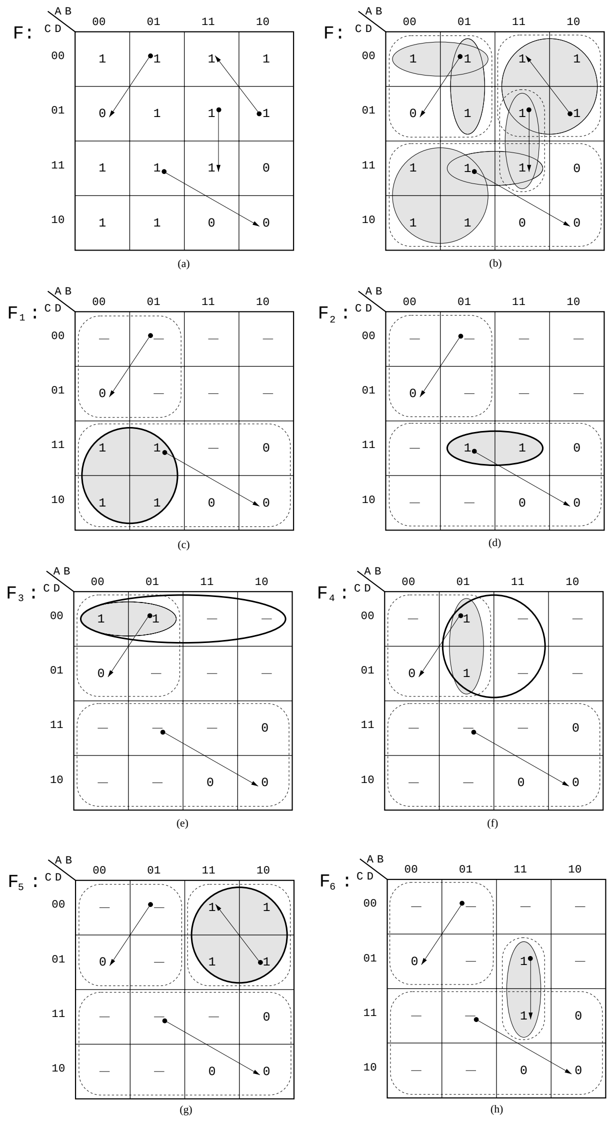

Figure 6. First Example: Finding a Hazard-Free Multi-Level Solution (*nowick_dill_nosol*)

5.2 A Final Iterative Algorithm

A final iterative algorithm, Algorithm 1, is presented in Figure 5(b).

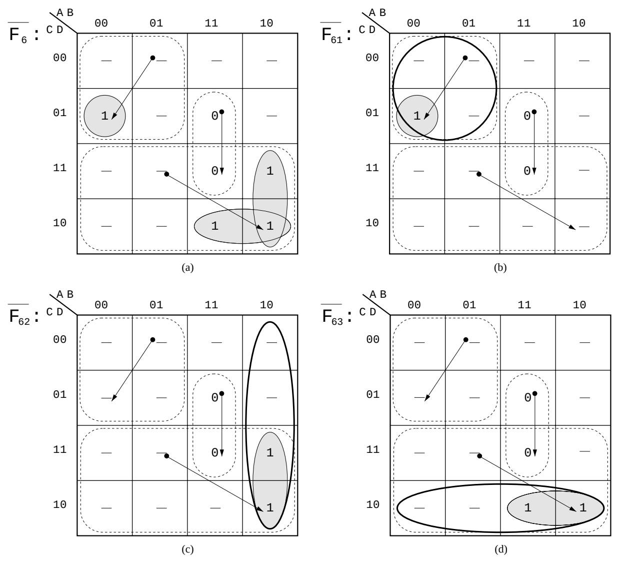
Figure 7. First Example (cont.): Finding a Hazard-Free Multi-Level Solution (*nowick_dill_nosol*)

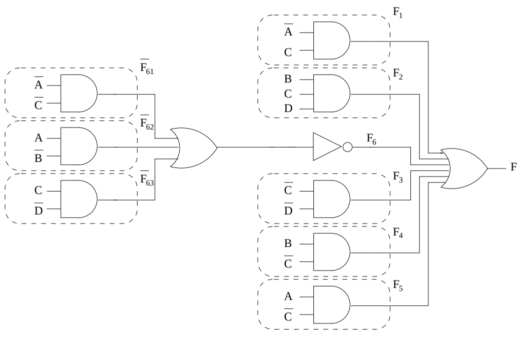
Figure 8. First Example (cont.): Final Hazard-Free Multi-Level Implementation (*nowick_dill_nosol*)

There are several changes over Algorithm 0. First, there are fewer hazard-free existence checks: only after maximal decomposition (no longer after inversion).

Second, as a result, a very simple hazard-free two-level existence check, "get-single-cube-sol", can now be used: determining if a *single-cube* (i.e., single-product) hazard-free solution exists. The justification is that, under maximal decomposition, each subfunction has only one required cube; therefore, if a hazard-free two-level solution exists, only one product will be needed to cover it.

Finally, the algorithm is now iterative, and *only* traverses three levels of search: *level-0* (before inversion), *level-1* (after one inversion), and *level-2* (after two inversions). Thus, at most, only two maximal disjunctive decompositions and two inversions are performed. An informal proof that no further recursion is needed is as follows: The first maximal decomposition, by definition, pulls apart the original required cubes, and creates subfunctions with *only one* required cube each; however, each subfunction has all the OFF-set cubes of the original function. After inversion, the OFF-set cube and required cubes are swapped; each inverted subfunction now has *only one* OFF-set cube, but possibly several required

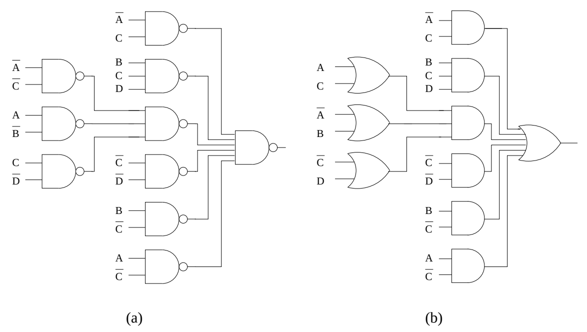
Figure 9. First Example (cont.): Final Hazard-Free Multi-Level Implementation (*nowick_dill_nosol*)

cubes. After another maximal decomposition, each resulting subfunction has *only one* required cube and *only one* OFF-set cube. After another inversion, the same invariant holds, but the resulting subfunctions have swapped ON-set and required cubes. At this point, no further maximal decomposition or inversion will alter these two subfunctions: they will simply alternate on each further inversion step. Hence, no further recursion is required.

Note that, as in Algorithm 0, a final solution is returned as a hierarchical set, here called *Sol*, where deeper subcircuits (from level-1 and level-2) are encapsulated with added parentheses.

As for algorithmic complexity, it is direct to see that, if the original function (F, T) has m required cubes and n maximal OFF-set cubes, then the algorithm can create $O(m \cdot n)$ gates, through the iterative decomposition process, with $O(m \cdot n)$ existence checks performed ("get-single-cube-sol").

5.3 Canonicity

Algorithm 1 does not necessarily produce a canonical (i.e. uniquely defined) decomposition, because "get-single-cube-sol" may produce one of several possible hazard-free single-product solutions for the leaf nodes. For example, in Figure 7(b)-(d) (to be discussed in the next section), several alternative single-product covers are possible for each subfunction. However, with a minor modification, the algorithm can be made canonical.

In [8], a simple hazard-free cover for a single required cube r was defined, called *supercube*$_{dhf}$*(r)*. Given a function, (F, T), this operator returns the single *smallest* implicant which (i) contains r, and (ii) has no illegal intersections (i.e., which is a dhf-implicant), if one exists. The result of *supercube*$_{dhf}$*(r)* is uniquely defined, hence canonical [8]. Hence, by replacing "get-single-cube-sol(r)" by *supercube*$_{dhf}$*(r)* in Algorithm 1, the algorithm now will produce a *unique* (i.e., canonical) hazard-free implementation for a given function (F, T), if any multi-level hazard-free solution exists.

In conclusion, if a given function (F, T) has *any* hazard-free multi-level implementation, then it always has a unique *canonical 3-level implementation*, produced by the above variant of Algorithm 1.

6 Examples

To illustrate the multi-level decomposition algorithm, it is now applied to two examples.

In Figure 6(a), function (F, T) has four specified input transitions. The example has no two-level hazard-free solution, and is taken from [7, 6]. There are six required cubes, each highlighted in Figure 6(b). Following Algorithm 1, under maximal decomposition, the initial function (F, T) is mapped to six corresponding decomposed subfunctions, $(F1, T1)$ through $(F6, T6)$, each containing one distinct required cube, as shown in Figures 6(c)-(h). For five of the resulting subfunctions, $(F1, T1)$ through $(F5, T5)$, a single-cube (i.e., single-product) hazard-free (two-level) solution exists, as shown. However, for subfunction $(F6, T6)$, no hazard-free two-level solution exists. As can be seen in Figure 6(h), only two implicants cover required cube ABD: ABD and BD. Neither is a dhf-implicant: implicant ABD illegally intersects the ABCD=0111 to 1010 dynamic transition, while BD illegally intersects the ABCD=0100 to 0001 dynamic transition. Therefore, subfunction $(F6, T6)$ is then inverted to produce a new subfunction, $(\overline{F6}, \overline{T6})$, as shown in Figure 7(a). The three former OFF-set cubes are now required cubes, and the one former required cube is now an OFF-set cube. As shown in Figures 7(b)-(d), after disjunctive decomposition, each of the subfunctions now has a single-cube hazard-free (two-level) solution. The final hazard-free multi-level solution is shown in Figure 8, where the various decomposed subfunctions are highlighted.

As a second example, Figure 10(a), modified from [2], has no two-level or multi-level solution. Due to space limitations, only part of the algorithm is illustrated. Figure 10(b) shows the result of the first disjunctive decomposition, showing one of the three resulting maximal decomposed subfunctions: this subfunction has no (single-cube) hazard-free two-level solution. Figure 10(c) shows the result of the subsequent inversion step, followed by another maximal disjunctive decomposition, again illustrating only one of the resulting subfunctions, which also has no (single-cube) hazard-free two-level solution. The algorithm then inverts this subfunction (not shown), again finds no hazard-free two-level solution, and then terminates.

7 Three-Level Circuit Implementations

While the circuit structure of Figure 8 corresponds directly to the algorithm flow, it is less standard or practical than may be desired for an actual circuit implementation. In this section, a key result is presented: an implementation produced by Algorithm 1 can *always* be transformed into two basic 3-level circuit structures: (i) 3-level NAND, and (ii) OR-AND-OR circuits.

The two transformations are illustrated on the circuit of Figure 8. The corresponding 3-level NAND circuit is shown in Figure 9(a). This transformation is performed using standard function- and hazard-preserving algebraic techniques, such as DeMorgan and other laws: the output OR of the former circuit is converted to a NAND with inverted inputs; in turn, double inverters are cancelled out; and a similar OR transformation is performed on the leftmost OR gate. Likewise, the transformation from Figure 8 to an OR-AND-OR circuit, shown in Figure 9(b), is accomplished by replacing the leftmost OR gate and its output inverter into an AND gate with inverted inputs; the leftmost AND gates and their new output inverters are then replaced by OR gates with inverted inputs.

Note that, while these 3-level circuit structures allow arbitrary fan-in gates, the designer can always map these structures to limited fan-in gates: by associate law, a large fan-in gate can be broken into a network of limited fan-in gates, with the same function and hazard properties. Thus, these structures can always be mapped to larger networks of 2-input gates, if desired.

The above example only illustrates part of the execution of Algorithm 1, which in this case terminates after only one inversion step. However, if a second inversion is performed by the algorithm, the resulting circuits can *still always be made 3-level*. Due to space limitations, no complete worked-out example of this scenario is presented, but below is a sketch of the transformation for this general case. Suppose the circuit Figure 8 now also had an additional AND2 gate followed by inverter feeding into the leftmost OR gate, where this AND2 gate has the two inputs x and y. Extending the above construction, a 3-level NAND circuit can still always be produced: the new AND2+inverter are deleted, and this AND2's inputs (x and y) would become *direct inputs* into the NAND3 of Figure 9(a) whose inputs are the 3 leftmost NANDs – thus transforming it into a NAND5 gate. Similarly, a 3-level OR-AND-OR circuit can still always be produced: the new AND2+inverter are deleted, and this AND2's inputs (x and y) would now become *direct inputs* into the AND3 of Figure 9(b) whose inputs are the 3 leftmost ORs – transforming it into a NAND5 gate.

8 Completeness of the Method

The multi-level decomposition algorithm of Section 5 has been shown to be *sound:* if it produces a multi-level implementation, the implementation is hazard-free. In the section, it is shown that the algorithm is *complete:* if any hazard-free multi-level implementation exists (with the given assumed types of gates), then the algorithm will always produce a solution.

Completeness Theorem. Given any function (F, T), for which there exists some hazard-free multi-level implemention C consisting of only ANDs, ORs, and INVERTERS. Then Algorithm 1 will always produce a hazard-free multi-level implementation for (F, T).

Proof.

Step 1. Circuit Transformation. The first step is constructive: to transform C into a form similar to the one produced by the algorithm, shown in Figure 3(d). In this circuit, ignoring inverters on primary inputs, AND gates only appear at inputs, and the only possible long chain of gates is a series of OR gates sandwiched between inverters.

The transformation rules in Figure 11 are used to transform C into the desired canonical form. The rules are applied in reverse topological order, from circuit output to inputs. As a pre-processing step, before the main traversal, the associative laws of Figure 11(a)-(c) are used to collapse adjacent AND gates (and OR gates), double inverters are replaced by wires, etc.[7] Next, if the gate output is not an OR gate, a 1-input OR is attached to the primary output.

Finally, the main reverse topological traversal is performed, following the rules of the figure. DeMorgan's laws are used to transform internal AND gates into OR gates surrounded by inverters. Note that Rules (e) and (f) should only be applied if the corresponding AND gate is non-terminal. Also, note that Rule (e) may produce a new OR gate which must then merged with those in its immediate fanout (by Rule(b)); similarly, Rules (e) and (f) produce new inverters at the inputs, which may result in further applications of Rules (a)-(c) at the input side. Finally, all inverters on primary inputs can be replaced by complemented literals.

It can easily be shown that the desired circuit properties hold on $C1$ after the above construction: ANDs can only appear at the circuit inputs, ORs and INVERTERS alternate in chains internally in the circuit, and all adjacent gates of the same type are eliminated or combined. At the leaves, is either an SOP (AND-OR) circuit or a single AND gate.

Furthermore, the resulting circuit, $C1$, has equivalent behavior to the original circuit C: (a) it has the *same function-*

[7]For simplicity, only two representative matches are shown in Figure 11(a)-(b), but obviously the same collapsing is performed for arbitrary fan-in gates.

ality as C, since only Boolean equivalences were applied; and (b) it has the *same hazard behavior* as C, since only hazard behavior-preserving transformations were applied [9]. Hence, the above construction take an arbitrary multi-level circuit C, and deterministically maps it to an equivalent one, $C1$, which is in a canonical form.

Step 2. Proof of Completeness. Given that $C1$ is a hazard-free multi-level implementation of (F, T), it is now shown how the new algorithm is always guaranteed to produce some hazard-free multi-level implementation.

Proof Sketch. The proof strategy is to follow two distinct traversals, in tandem: (a) a reverse topological traversal of circuit $C1$, from output to circuit inputs, and (b) the recursion steps of Algorithm 1. The idea is that, at each traversal step, the current subnetwork of circuit $C1$ has a subfunction which *dominates* a corresponding subfunction generated by the algorithm, and this dominance is passed from level to level during the traversal. Hence each subfunction produced by the algorithm should be "easier" to implement correctly than each circuit subfunction. Since the circuit traversal terminates with hazard-free leaf nodes on each path (each corresponding to a single-cube, or single-wire, implementation), it follows that each corresponding path through the algorithm must also successfully terminate with a hazard-free single-cube solution.

Detailed Proof. Before proceeding to the joint traversals of the algorithm and circuit, first consider each of them separately.

Algorithm 1 starts with function (F, T). It first applies a *maximal disjunctive decomposition*, generating a set of subfunctions $(F1, T1), \ldots, (FN, TN)$. These subfunctions, in turn, are then *inverted*, to generate corresponding subfunctions $(F1', T1'), \ldots, (FN', TN')$ (cf. subfunctions F1-F6 in Figure 6(c)-(g)). (For now, ignore where the algorithm terminates and returns a solution.) Each subfunction, (Fi', Ti'), in turn, then undergoes another *maximal disjunctive decomposition*, to generate a set of corresponding subfunctions, (Fi'_j, Ti'_j); that is, $(Fi'_1, Ti'_1), \ldots, (Fi'_M, Ti'_M)$ (cf. subfunctions F'61-F'63 in Figure 7(b)-(d)). Finally, each such subfunction, (Fi'_j, Ti'_j), is again *inverted*, to generate corresponding subfunctions, (Fi_j, Ti_j); that is, $(Fi_1, Ti_1), \ldots, (Fi_M, Ti_M)$. At this point, Algorithm 1 terminates, since it was shown that subfunction (Fi_j, Ti_j) and (Fi'_j, Ti'_j) are *non-reducible* by further maximal decomposition or inversion, and would simply alternate on every further level of recursion. However, for the following proof, this further recursive alternation (Fi_j, Ti_j) and (Fi'_j, Ti'_j) will be allowed; it is is safe, since only hazard-free decomposition steps are used. This modified algorithm allows a deeper simulation in order to track the circuit traversal: maximal decomposition will be applied (which will not further decompose or alter these subfunctions), alternating with inversion (which simply swaps them). [8]

Next, consider circuit, $C1$. By construction, it has an output OR gate. In reverse topological traversal towards the primary inputs, the structure is as shown at the right of Figure 3: there are no internal ANDs, only chains of inverters alternating with OR's, until the leaves (primary inputs) are reached. Only at the primary inputs, there are either terminal AND gates (or a degenerate case, simple wires) or terminal AND-OR circuits. Note that, like the algorithm, the circuit decomposition must also follow the well-defined decomposition rules. Since the circuit output is attached to an OR gate, therefore, at this first level, circuit $C1$ is decomposed using disjunctive decomposition. Therefore, each resulting subfunction, $(G1, V1)$ through (Gp, Vp), for the distinct subnetworks feeding into the OR gate must obey the disjunctive decomposition rules (Section 4.1), since these rules are necessary and sufficient, by the Disjunctive Decomposition Theorem. At the next level, assuming for now that the circuit is not at a leaf node, the circuit is always decomposed using inversion, and must likewise obey the rules of inversion (Section 4.2), which are also necessary and sufficient. Similarly, at each further step of the reverse topological traversal, on every path, the circuit undergoes alternating disjunctive decomposition and inversion, where each resulting subnetwork must have a corresponding subfunction which obeys these rules.

Finally, consider the traversal in tandem of the modified Algorithm 1 and circuit $C1$. From the above discussion, both the algorithm and circuit structure follow alternating disjunctive and inversion decompositions. It is now shown that, on each path through the algorithm, the resulting subfunction is *always* dominated by a corresponding subfunction in the circuit. At the top-level, for each maximally decomposed subfunction of the algorithm, (Fi, Ti), there must exist a corresponding subnetwork of $C1$ with subfunction (Gk, Tk) such that $(Fi, Ti) \preceq (Gk, Tk)$, by the First Maximal Decomposition Theorem. After a subsequent inversion, by the Inversion Theorem, this dominance relation persists: $(F'i, T'i) \preceq (G'k, T'k)$. Next, upon a subsequent maximal decomposition by the algorithm, now by the Second Maximal Decomposition Theorem, for each resulting subfunction (Fi'_j, Ti'_j), there again must be some decomposed subnetwork of $C1$ with a corresponding subfunction which dominates (Fi'_j, Ti'_j).

Thus, continuing this argument through the entire traversal, dominance persists: each subfunction produced by the algorithm is dominated by a corresponding subfunction of circuit $C1$. Furthermore, note that every circuit path is finite and hence terminates. In each case, by construction of $C1$, the terminating leaf circuit is always a single AND gate (or, if degenerate, a single wire); thus, the subfunction corresponding to each circuit leaf is a hazard-free two-level implementation. Hence, in this traversal, on every execution path of the algorithm, it will eventually generate a subfunction which is dominated by the subfunction of a corresponding *leaf node* in the circuit. By Lemma 1, since the latter subfunction has a hazard-free two-level solution (in fact, a single-cube solution), so must the former. Hence, on each such execution of the modified Algorithm 1, the algorithm must successfully terminate, since a hazard-free single-cube solution exists.

The only slightly subtle case is where the modified Algorithm 1 above has been allowed to recur through many levels (unlike the actual Algorithm 1, which allows at most two inversions). In this case, since either some Fi_j or Fi'_j must be the subfunction produced by the modified algorithm at that level. When the circuit terminates at an AND-leaf, then by the above, the subfunction produced by the algorithm (Fi_j or Fi'_j) must also have a single-cube hazard-free solution. Yet, in the *actual* iterative Algorithm 1, as discussed above, this subfunction (Fi_j or Fi'_j) would have been produced at an *earlier* step: at the first or second level of execution. These subfunctions Fi_j and Fi'_j are simply passed on, alternating through further levels of execution in the modified algorithm. Since subfunction Fi_j or Fi'_j has just been shown to have a single-cube hazard-free solution, then the

[8]This deeper recursion is not used in the actual Algorithm 1, but this proof will yield results that will then apply to the non-recursive Algorithm 1.

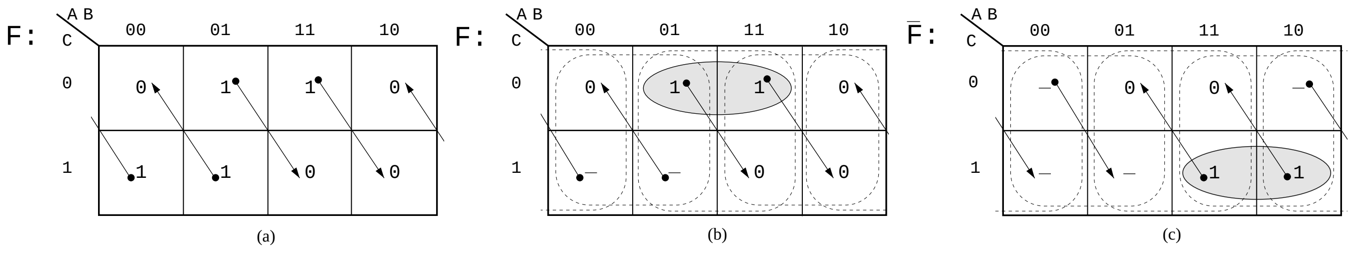

Figure 10. Second Example: No Hazard-Free Multi-Level Solution (*bredeson_nosol_redux*)

actual Algorithm 1 must therefore have successfully terminated earlier on this execution path (i.e., at first or second levels of iteration).

In summary, in the actual Algorithm 1 execution, on each execution path, a subfunction will be generated which is always dominated by the subfunction at *some* corresponding leaf node of circuit $C1$. By Lemma 1, since the latter node had a single-cube hazard-free solution, so must the former, hence the actual Algorithm 1 will always successfully return a solution on every execution path.

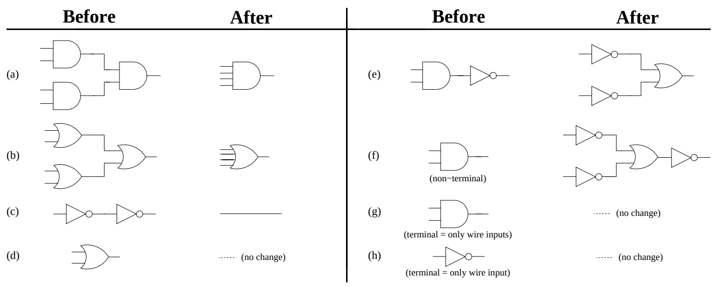

Figure 11. Canonical Circuit Transforms

9 Results

The table below presents some results of applying the multi-level hazard-free decomposition algorithm, Algorithm 1, to several examples. A prototype CAD tool was written in C++, and run on a 500MHz Pentium III with 256MB RAM running Redhat Linux 8.0. The input format is in a PLA-style format, indicating both function and input transitions. The multi-level output is in a subset of Berkeley BLIF format. Only small examples are considered, with 6 or fewer inputs.

Function Name	# Of Inputs	Solution Exists	Total # Subfunctions	Total # Func Inversions	Max # Inversions per Circuit Level	Recursion Depth	CPU Time (ms)
bredeson_nosol_redux	3	No	4	3	1	2	0.36
simple_ex	4	Yes	3	0	0	0	0.11
nowick_dill_nosol	4	Yes	8	1	1	1	0.59
z1	4	Yes	19	3	3	1	2.28
z2	4	No	12	10	4	2	1.42
z3	4	Yes	8	2	2	1	0.68
z4	4	Yes	4	0	0	0	0.18
z5	4	No	11	14	2	2	2.23

Columns indicate whether a solution exists, and various characterizations of recursion depth. Recursion depths range from 0 (i.e. a 2-level solution was found) to 2. The column "Total # of Func Inversions" indicates how many inversions were performed across the entire decomposition, while the column "Max # Inversions per Circuit Level" indicates the maximum number of subfunctions in any iteration which required inversion. Runtimes are indicated in milliseconds. The results are on some fabricated examples, but do indicate a range of parameters and characteristics for different functions.

10 Conclusions and Future Work

This paper has introduced the first general method for determining if an incompletely-specified Boolean function, with a specified set of input transitions, has a multi-level hazard-free realization. As part of this work, two hazard-free decompositions were defined: disjunctive and inversion; and the first set of necessary and sufficient conditions were formalized to characterize them. This paper is also the first to demonstrate that, if a function has any hazard-free multi-level solution,then it also always has a hazard-free 3-level circuit solution: in 3-level NAND and OR-AND-OR form. A simple iterative decomposition algorithm was proposed to find such a hazard-free implementation, if one exists. Furthermore, if one does, a unique canonical 3-level hazard-free form was identified, and an algorithm to obtain it was presented.

In the future, it would be useful to strengthen the claims on the classes of gates that can be considered, and to automate and apply the method to a larger set of real-world asynchronous benchmarks. In addition, as a practical application, it would be interesting to explore how Algorithm 1 can serve as the starting point for multi-level synthesis: it can immediately provide a "seed" implementation with the fewest possible hazards, which can then be further manipulated using well-known hazard-non-increasing transformations.

References

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

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

[3] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[4] D. Kung. Hazard-non-increasing gate-level optimization algorithms. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, November 1992.

[5] B. Lin and S. Devadas. Synthesis of hazard-free multi-level logic under multiple-input changes from binary decision diagrams. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, November 1994.

[6] S.M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, March 1993. (revised tech. report, Stanford Computer Systems Lab. CSL-TR-95-686, Dec. 1995).

[7] S.M. Nowick and D.L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. *IEEE Trans. on Computer-Aided Design*, 14(8):986–997, August 1995.

[8] M. Theobald and S.M. Nowick. Fast heuristic and exact algorithms for two-level hazard-free logic minimization. *IEEE Trans. on Computer-Aided Design*, 17(11):1130–1147, November 1998.

[9] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley Interscience, 1969.