

An Implicit Method for Hazard-Free Two-Level Logic Minimization*

Michael Theobald Steven M. Nowick

Department of Computer Science
Columbia University
New York, NY 10027

Abstract

None of the available minimizers for exact 2-level hazard-free logic minimization can synthesize very large circuits. This limitation has forced researchers to resort to heuristic minimization, or to manual and automated circuit partitioning techniques. This paper introduces a new implicit 2-level logic minimizer, IMPYMIN, which is able to solve very large multi-output hazard-free minimization problems exactly. The minimizer is based on a novel theoretical approach: it incorporates hazard-freedom constraints within a synchronous function by adding new variables. In particular, the generation of dynamic-hazard-free prime implicants is cast as a synchronous prime implicant generation problem.

The minimizer can exactly solve all currently available examples, which range up to 32 inputs and 33 outputs, in less than 813 seconds. These include examples that have never been exactly solved before.

1 Introduction

Asynchronous design has been the focus of much recent research activity. A number of methods have been developed for the design of hazard-free controllers [13, 7, 11, 25, 8, 18]. These methods have been applied to several large and realistic design examples, including a low-power infrared communications chip [9], a second-level cache-controller [12], a SCSI controller [23], a differential equation solver [24], and an instruction length decoder [2].

An important aspect of these methods is the development of optimized CAD tools. In synchronous design, CAD packages have been critical to the advancement of modern digital design. In asynchronous design, much progress has been made, including tools for exact hazard-free two-level logic minimization [16], optimal state assignment [7, 18] and synthesis-for-testability [15]. However, these tools have been limited in handling large-scale designs.

In particular, hazard-free 2-level logic minimization is an important step in all the above-mentioned

CAD tools. However, while the currently used Quine-McCluskey-like exact hazard-free minimization algorithm, HFMIN [7], has been effective on small- and medium-sized examples, it has been unable to produce solutions for several large design problems [8, 18]. This limitation has been a major reason for researchers to invent and apply manual as well as automated techniques for partitioning circuits before hazard-free logic minimization can be performed [8].

A first step towards handling large hazard-free 2-level minimization problems was the introduction of a heuristic minimizer, called ESPRESSO-HF[21]. Although this minimizer was able to handle some large designs, it does not always find a minimum solution. In addition, it is also somewhat limited to designs where the number of required cubes¹ is manageable, since the complexity of algorithms depends on the number of required cubes. In particular, ESPRESSO-HF has steps which may exhibit quadratic run-time in the number of required cubes (e.g. Expand).

Contributions of This Paper

This paper makes both practical and theoretical contributions.

The first contribution of the paper is to present the new practical minimizer, IMPYMIN, to solve the exact hazard-free two-level logic minimization problem for very large examples. Our algorithm is an implicit approach which makes use of data structures such as BDDs [1] and zero-suppressed BDDs [10]. In contrast, an existing minimization algorithm for hazard-free logic [16] is based on explicit data structures, and there does not seem to be any simple way to incorporate implicit representations into it. IMPYMIN can exactly solve all currently available examples, which range up to 32 inputs and 33 outputs, in less than 813 seconds. These include examples that have never been solved exactly before.

The second contribution of this paper is a novel theoretical approach to hazard-free minimization. We

* This work was supported by NSF under Grant no. MIP-9501880 and by an Alfred P. Sloan Research Fellowship.

¹A required cube is a set of minterms and represents a covering constraint. Required cubes are defined in the background section.

reformulate the generation of dynamic-hazard-free prime implicants as a *synchronous* prime implicant generation problem. This is achieved by incorporating hazard-freedom constraints within a synchronous function by adding new variables. This technique allows us to leverage off an existing method for fast implicit generation of prime implicants. As a result, our approach can be directly used to create a very efficient implicit minimizer for hazard-free logic. In particular, our approach makes it possible to use the implicit set covering solver of SCHERZO [6, 4, 3, 5], the state-of-the-art minimization method for synchronous two-level logic, as a black box.

Paper Organization

The paper is organized as follows. Section 2 gives background on asynchronous logic synthesis. Section 3 introduces a new approach of incorporating hazard-freedom constraints within a synchronous function, leading to a new method for computing dynamic-hazard-free prime implicants. Section 4 gives background on implicit logic minimization. Section 5 combines the techniques in Section 3 and 4 into a new implicit method for hazard-free 2-level logic minimization. Section 6 presents experimental results and compares our approach with related work, and Section 7 gives conclusions.

2 Background on Asynchronous Logic Synthesis

The material of this section focuses on hazards and hazard-free logic minimization, and is taken from [7, 16, 14]. For simplicity, the focus is on single-output functions. A generalization of these definitions to multi-output functions is straightforward, and is described in [7].

2.1 Circuit Model

This paper considers combinational circuits having arbitrary finite gate and wire delays (an *unbounded wire delay model* [16]). A pure delay model is assumed as well (see [22]).

2.2 Multiple-Input Changes

Definition 2.1 *Let A and B be two minterms. 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 $t = [A, B]$ is the Boolean function $A_i + B_i$ (alternatively, $[A, B]$ is the uniquely defined smallest cube that contains A and B : $\text{supercube}(A, B)$). An input transition or multiple-input change from input state (minterm) A to B is described by transition cube $[A, B]$.*

A multiple-input change specifies what variables change value and what the corresponding *starting* and *ending* values are. Input variables are assumed to change simultaneously. (Equivalently, since inputs may be skewed arbitrarily by wire delays, inputs can be assumed to change monotonically in any order and at any time.) Once a multiple-input change occurs, no further input changes may occur until the circuit has stabilized. In this paper, we consider only transitions where f is fully defined; that is, for every $X \in [A, B]$, $f(X) \in \{0, 1\}$.

2.3 Function Hazards

A function f which does not change monotonically during an input transition is said to have a *function hazard* in the transition.

Definition 2.2 *A 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 2.3 *A function f contains a dynamic function hazard for the input transition from A to D if and only if: (1) $f(A) \neq f(D)$; and (2) there exist a pair of input states, B and C , 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 a glitch during the transition, assuming arbitrary gate and wire delays [16, 22]. Therefore, we consider only transitions which are free of function hazards².

2.4 Logic Hazards

If f is free of function hazards for a transition from input A to B , an implementation may still have hazards due to possible delays in the logic realization.

Definition 2.4 *A circuit implementing function f contains a static (dynamic) logic hazard for the input transition from minterm A to minterm B if and only if: (1) $f(A) = f(B)$ ($f(A) \neq f(B)$), and (2) for some assignment of delays to gates and wires, the circuit's output is not monotonic during the transition interval.*

That is, a static logic hazard occurs if $f(A) = f(B) = 1$ (0), but the circuit's output makes an unexpected $1 \rightarrow 0 \rightarrow 1$ ($0 \rightarrow 1 \rightarrow 0$) transition. A dynamic logic hazard occurs if $f(A) = 1$ and $f(B) = 0$ ($f(A) = 0$ and $f(B) = 1$), but the circuit's output makes an unexpected $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$ ($0 \rightarrow 1 \rightarrow 0 \rightarrow 1$) transition.

²Sequential synthesis methods, which use hazard-free minimization as a substep, typically include constraints in their algorithms such that no transitions with function hazards are generated [13, 25].

2.5 Conditions for a Hazard-Free Transition

We now review conditions to ensure that a sum-of-products implementation, F , is hazard-free for a given input transition (for details, see [16]). Assume that $[A, B]$ is the transition cube corresponding to a *function-hazard-free* transition from input state A to B for a function f . We say that f has a $f(A) \rightarrow f(B)$ transition in cube $[A, B]$.

Lemma 2.5 *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 2.6 *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 F (i.e., some product must hold its value at 1 throughout the transition).*

The conditions for the $0 \rightarrow 1$ and $1 \rightarrow 0$ cases are symmetric. Without loss of generality, we consider only a $1 \rightarrow 0$ transition³.

Lemma 2.7 *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 $c \in F$ intersecting $[A, B]$ also contains A (i.e., no product may glitch in the middle of a $1 \rightarrow 0$ transition).*

Lemma 2.8 *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 F (i.e., every $1 \rightarrow 1$ sub-transition must be free of logic hazards).*

$1 \rightarrow 1$ transitions and $0 \rightarrow 0$ transitions are called *static* transitions. $1 \rightarrow 0$ transitions and $0 \rightarrow 1$ transitions are called *dynamic* transitions.

2.6 Required and Privileged Cubes

The cube $[A, B]$ in Lemma 2.6 and the *maximal* subcubes $[A, X]$ in Lemma 2.8 are called *required* cubes. Each required cube *must* be contained in some cube of cover F to ensure a hazard-free implementation. More formally:

Definition 2.9 *Given a function f , and a set, T , of specified function-hazard-free input transitions of f , every cube $[A, B] \in T$ corresponding to a $1 \rightarrow 1$ transition, and every maximal subcube $[A, X] \subset [A, B]$ where f is 1 and $[A, B] \in T$ is a $1 \rightarrow 0$ transition, is called a **required cube**.*

³A $0 \rightarrow 1$ transition from A to B has the same hazards as a $1 \rightarrow 0$ transition from B to A .

Lemma 2.7 constrains the products which may be included in a cover F . Each $1 \rightarrow 0$ transition cube is called a *privileged cube*, since no product c in the cover may intersect it unless c also contains its *start point*. If a product 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. More formally:

Definition 2.10 *Given a function f , and a set, T , of specified function-hazard-free input transitions of f , every cube $[A, B] \in T$ corresponding to a $1 \rightarrow 0$ transition is called a **privileged cube**.*

Finally, we define a useful special case. For certain privileged cubes the function is only 1 at the start point and is 0 for all other minterms included in the transition cube. In this case, any product that intersects such a privileged cube always covers the start point, since the cube contains no other ON-set minterms. We call such a privileged cube **trivial**. All trivial privileged cubes can safely be removed from consideration without loss of information.

2.7 Hazard-Free Covers

A *hazard-free cover* of function f is a cover (i.e., set of implicants) of f whose AND-OR implementation is hazard-free for a *given set, T* , of specified input transitions. (It is assumed below that the function is defined for all specified transitions; the function is undefined for all other input states.)

Theorem 2.11 (Hazard-Free Covering [14, 16]) *A sum-of-products F is a hazard-free cover for function f for the set T of specified input transitions if and only if:*

- (a.) *No product of F intersects the OFF-set of f ;*
- (b.) *Each required cube of f is contained in some product of F ; and*
- (c.) *No product of F intersects any (non-trivial) privileged cube illegally.*

Theorem 2.11(a) and (c) determine the implicants which may appear in a hazard-free cover of a function f , called *dynamic-hazard-free (dhf-) implicants*.

Definition 2.12 *A dhf-implicant is an implicant which does not intersect any privileged cube of f illegally. 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.*

Theorem 2.11(b) defines 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. Thus, the **two-level hazard-free logic minimization problem** is to find a minimum cost cover of a

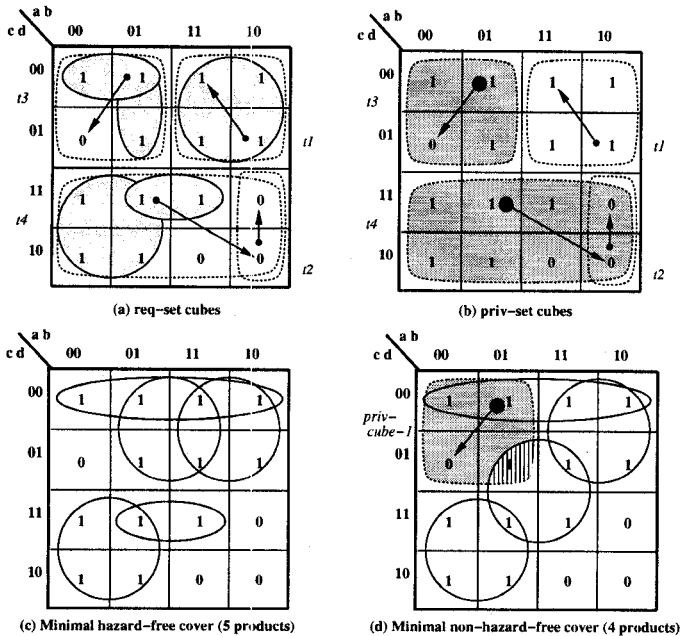


Figure 1: 2-Level Hazard-Free Minimization Example

function using only dhf-prime implicants where every required cube is covered.

In general, the covering conditions of Theorem 2.11 may not be satisfiable for an arbitrary Boolean function and set of transitions [22, 16]. This case occurs if conditions (b) and (c) cannot be satisfied simultaneously.

A hazard-free minimization example is shown in Figure 1.

2.8 Exact Hazard-Free Minimization Algorithm

A single-output exact hazard-free minimizer has been developed by Nowick and Dill [14, 16]. It has recently been extended to hazard-free multi-valued minimization⁴ by Fuhrer, Lin and Nowick [7]. The latter method, called HFMIN, has been the fastest minimizer for exact hazard-free minimization.

HFMIN makes use of ESPRESSO-II to generate all prime implicants, then transforms them into dhf-prime implicants, and finally employs ESPRESSO-II's MINCOV to solve the resulting unate covering problem. Each of the algorithms used in these three steps is critical, i.e. has a worst-case run-time that is exponential. As a result, HFMIN cannot solve several of the more difficult examples.

⁴It is well-known that multi-output minimization can be regarded as a special case of multi-valued minimization [17].

3 A Novel Approach of Incorporating Hazard-Freedom Constraints Within a Synchronous Function

In this section, we present a novel technique which recasts the dhf-prime implicant generation problem into a prime generation problem for a new *synchronous* function, with extra inputs. Based on this approach, we present a new implicit method for exact 2-level hazard-free logic minimization in Section 5.

3.1 Overview and Intuition

In this subsection, we first give a simple overview of our entire method. Details and formal definitions are provided in the remaining subsections.

Our approach is to recast the generation of dhf-prime implicants of an asynchronous function (f, T) into the generation of prime implicants of a synchronous function g . Here, hazard-freedom constraints are incorporated into the function g by adding extra inputs. An overview of the method is best illustrated by a simple example.

Example 3.1 Consider Figure 2. The Karnaugh map in part A represents a function (f, T) defined over the set of 3 variables $\{x_1, x_2, x_3\}$. The shaded area corresponds to the only non-trivial privileged cube of f (the second privileged cube $[101, 100]$ is trivial, cf. Section 2.6). We now define a new *synchronous* function g , shown in part B. g is obtained from f by adding a single new variable z_1 . That is, g is defined over 4 variables: $\{x_1, x_2, x_3, z_1\}$. In general, to generate g , one new z -variable is added for each non-trivial privileged cube. Next, the prime implicants of the synchronous function g are computed (shown in part B as ovals). Finally, we use a simple filtering procedure to filter out those prime implicants that correspond to those in f which intersect the privileged cube illegally. The remaining prime implicants of g are shown in part C. We then “delete” the z_1 -dimension from the prime implicants, and obtain the entire set of dhf-prime implicants of (f, T) (part D). \square

Our approach is motivated by the fact that dhf-prime-implicants are more constrained than prime implicants of the same function. While prime implicants are maximal implicants that do not intersect the OFF-set of the given function, dhf-prime-implicants, in addition, must also not intersect privileged cubes illegally. This means that there are two different kinds of constraints for dhf-prime-implicants: “maximality” constraints and “avoidance of illegal intersections” constraints. Our idea is now to unify these two types of constraints, i.e. to transform the avoidance constraints into maximality constraints so that dhf-primes can be generated in a uniform way. Intuitively, this can be achieved by adding auxiliary variables, i.e. by lifting the problem into a higher-dimensional Boolean space.

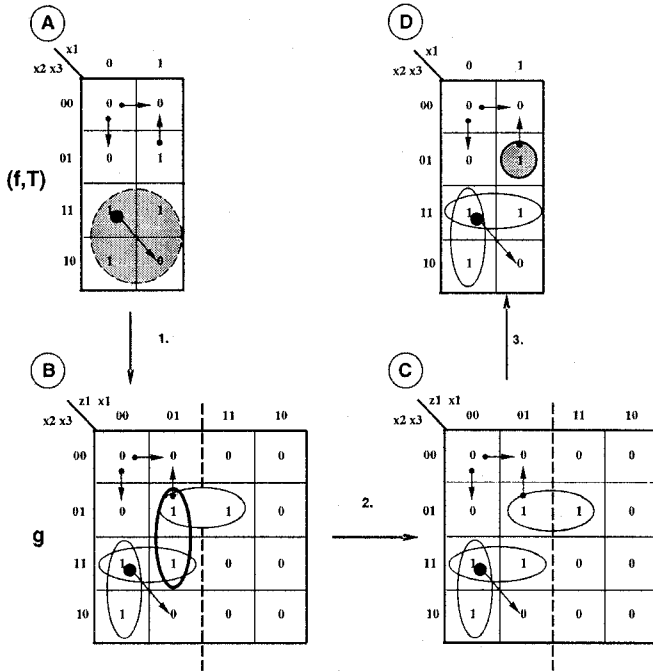


Figure 2: Example for recasting prime generation. A) shows the function (f, T) whose dhf-primes are to be computed. B) shows the auxiliary synchronous function g and its primes. C) shows primes of g that do not intersect illegally. D) shows the final dhf-primes of f , after deleting the z_1 variable.

In summary, the big picture is as follows. The definition of g ensures that all dhf-prime implicants of f ($dhf\text{-Prime}(f, T)$) can be easily obtained from the set of prime implicants of g ($Prime(g)$). While $Prime(g)$ may also include certain products which are non-hazard-free, these are filtered out easily, using a post-processing step.

3.2 The auxiliary synchronous function g

In this subsection, we explain how the synchronous function g is derived. For simplicity, assume for now that f is a single-output function.

Suppose f is defined over the set of variables $\{x_1, \dots, x_n\}$, and that the set of transitions T gives rise to the set of non-trivial privileged cubes $PRIV(f, T) = \{p_1, \dots, p_l\}$. The idea is to define a function g over $\{x_1, \dots, x_n, z_1, \dots, z_l\}$; that is, *one new variable is added per privileged cube*. Formally, g is defined as follows:

$$g(x_1, \dots, x_n, z_1, \dots, z_l) = f \cdot \prod_{1 \leq i \leq l} (\bar{z}_i + \bar{p}_i)$$

That is, the function g is the product of f and some function which depends on the added inputs. The intuition behind the definition of g is that in the $z_i = 0$

half of the domain g is defined as f , while in the $z_i = 1$ half of the domain g is defined as f but with the i -th privileged cube p_i "filled in" with all 0's (i.e., p_i is "masked out").

Example 3.2 As an example, Figure 2A shows a Boolean function (f, T) with privileged cube x_2 (highlighted in gray). Figure 2B shows the corresponding new function g , with added variable z_1 . In the $z_1 = 0$ half, function g is identical to f . In the $z_1 = 1$ half, g is identical to f except that g is 0 throughout the cube $z_1 x_2$, which corresponds to the privileged cube in the original function f . In particular, function g is defined as $g = f \cdot (\bar{z}_1 + \bar{p}_1)$, where $p_1 = x_2$. \square

3.3 Prime implicants of function g

To understand the role of function g , we consider its prime implicants $Prime(g)$.

We start by considering a function (f, T) that has only *one* privileged cube p_1 . Let q be any implicant of the function g that is contained in the $z_1 = 0$ plane of g . Since the $z_1 = 0$ plane is defined as f , q also corresponds to an implicant of f . Now, consider the expansion of q into the $z_1 = 1$ plane of function g . There are 2 possibilities: either (i) q can expand into $z_1 = 1$ plane, or (ii) q cannot expand into the $z_1 = 1$ plane. In case (i), expansion of q into the $z_1 = 1$ plane means that g is identical to f in the expanded region. Therefore, q does not intersect privileged cube p_1 in the original function f (if it did, g would have all 0's in p_1 in the $z_1 = 1$ plane, and expansion would be impossible). In case (ii), expansion into the $z_1 = 1$ plane is impossible. In this case, q must intersect p_1 in function f (g has all 0's in p_1).

In summary, q may or may not be able to expand from $z_1 = 0$ into $z_1 = 1$ planes. Expansion can occur precisely if q does not intersect the privileged cube p_1 in the original function.

Example 3.3 Consider the minterm $q_1 = \bar{z}_1 x_1 \bar{x}_2 x_3$ of g in Figure 2B, which corresponds to the minterm $x_1 \bar{x}_2 x_3$ of f . q_1 can be expanded into the $z_1 = 1$ plane into the prime implicant of g : $x_1 \bar{x}_2 x_3$ (shaded oval). Intuitively, the expansion is possible since q_1 does not intersect the privileged cube, i.e. the cube $\bar{z}_1 x_2$, which corresponds to the privileged cube x_2 of the original function f . However, the implicant $q_2 = \bar{z}_1 x_1 x_3$ (oval with thick dark border) of g *cannot* be expanded into the $z_1 = 1$ plane: it intersects the privileged cube, and therefore the corresponding region in the $z_1 = 1$ plane is filled with 0's. Note that prime generation is an expansion process until no further expansion is possible. \square

Let us now consider the general case, i.e. where (f, T) may have more than one privileged cube. We show that the support variables of each prime of g *precisely* indicate which privileged cubes are intersected

by the prime's corresponding implicant in f . Let q be any prime implicant of g :

$$q = x_{i_1} \cdots x_{i_n} z_{j_1} \cdots z_{j_l}$$

Here, x_{i_k} is a positive or negative x -literal⁵. However, z_{j_k} can *only* be a negative z -literal. The reason is that g is a negative unate function in z -variables (see definition of g), and therefore prime implicants of g will not include positive z -literals.

We indicate by q^x the *restriction of q to the x -literals*, i.e. $q^x = x_{i_1} \cdots x_{i_n}$. Note that q^x is an implicant of f by the definition of g . If q includes the literal \bar{z}_i , then q^x intersects p_i . The reason is that the primality of q indicates that q cannot be expanded into the $z_i = 1$ plane. As explained above, this is equivalent to the intersection of p_i in the original function f . On the other hand, if q does not include \bar{z}_i , then q^x does not intersect p_i . Intuitively, the primes, $\text{Prime}(g)$, are maximal in two senses: they are maximally expanded in f , or maximally non-intersecting of privileged cubes, in some combination, which is indicated by the set of support of the primes.

Therefore, the key observation is that the set of support of a prime implicant q of g *precisely* indicates which privileged cubes are intersected by the corresponding implicant q^x in f . This observation will be critical in obtaining the final set of dhf-prime implicants of f , $\text{dhf-Prime}(f, T)$.

3.4 Transforming Prime(g) into dhf-Prime(f, T)

Once $\text{Prime}(g)$ is computed, $\text{dhf-Prime}(f, T)$ can be directly computed. The key insight for this computation is that the prime implicants of $\text{Prime}(g)$ fall into 3 classes with respect to a specific privileged cube p_i . Each prime q is distinguished based on *if* and *how* it intersects the privileged cube p_i in f , i.e. based on the intersection of q^x with p_i :

- Class 1: Prime implicants q that do not intersect the privileged cube, i.e. q^x does not intersect p_i .
- Class 2: Prime implicants q that intersect the privileged cube legally, i.e. q^x intersects p_i and contains its start point.
- Class 3: Prime implicants q that intersect the privileged cube illegally, i.e. q^x intersects p_i but does not contain the start point.

$\text{Dhf-Prime}(f, T)$ can now be computed as follows. Start with $\text{Prime}(g)$. Filter out all prime implicants that fall in Class 3 with respect to the first privileged cube. Then, filter out all prime implicants that fall in Class 3 with respect to the second privileged cube, and so on. Finally, we obtain a set such that each

⁵Note that q may not depend on all x -variables.

of its elements is a valid dhf-implicant of (f, T) if restricted to the x -variables. The reason is, first, that all primes of g are implicants of f if restricted to x -variables. Second, the filtering removed any element that intersected any privileged cube illegally. Therefore, the set only includes dhf-implicants. In fact, it contains *all* dhf-prime-implicants of (f, T) . This will be proven in the next subsection.

Example 3.4

Figure 2B shows function g and its prime implicants, $\text{Prime}(g) = \{x_1\bar{x}_2x_3, \bar{z}_1x_1x_3, \bar{z}_1x_2x_3, \bar{z}_1\bar{x}_1x_2\}$. Part C shows the result of filtering out primes that illegally intersect regions corresponding to privileged cubes in f . In this case, $\bar{z}_1x_1x_3$ (oval with thick dark border) falls into Class 3 with respect to p_1 : it is deleted since it has a \bar{z}_1 -literal, i.e. intersects the region corresponding to privileged cube p_1 and does not contain the start point $\bar{z}_1\bar{x}_1x_2x_3$. However, $x_1\bar{x}_2x_3$ (shaded oval) falls into Class 1: it is not deleted since it does not have a \bar{z}_1 -literal and therefore does not intersect the region corresponding to the privileged cube p_1 . The remaining two primes $\bar{z}_1x_2x_3$ and $\bar{z}_1\bar{x}_1x_2$ fall into Class 2: they intersect the region corresponding to p_1 and contain the start point. Part D shows the result of step 3 which deletes the z -literals in each cube. We obtain $\{x_1\bar{x}_2x_3, x_2x_3, \bar{x}_1x_2\}$, which is $\text{dhf-Prime}(f, T)$. Note that the introduction of the z_1 -variable ensures that the dhf-implicant of f , $x_1\bar{x}_2x_3$, which is not a prime implicant of f , since it is contained by the prime implicant, x_1x_3 , is nevertheless generated. \square

3.5 Formal characterization of dhf-Prime(f, T) in terms of auxiliary function g

In this subsection, based on above discussion, we present the main result of this section: a new formal characterization of $\text{dhf-Prime}(f, T)$. We use the following notations. g_{z_i} and $g_{\bar{z}_i}$ denote the positive and negative cofactors of g with respect to variable z_i , respectively. RemZ denotes an operator on a set of cubes which removes all z -literals of each cube. As an example, $\text{RemZ}(\{x_1x_2z_1, x_1x_3\bar{z}_2, x_1x_3z_1z_3\}) = \{x_1x_2, x_1x_3\}$. The SCC -operator on a set of cubes (single-cube-containment) removes those cubes contained in other cubes.

Theorem 3.5 *Given (f, T) . Let $\text{PRIV}(f, T) = \{p_1, \dots, p_l\}$ be the set of non-trivial privileged cubes, and $\text{START}(f, T) = \{s_1, \dots, s_l\}$ be the set of corresponding start points. Define*

$$g(x_1, \dots, x_n, z_1, \dots, z_l) = f \cdot \prod_{1 \leq i \leq l} (\bar{z}_i + \bar{p}_i)$$

Then the set $\text{dhf-Prime}(f, T)$ can be expressed as follows:

$$SCC \left(\bigcap_{1 \leq i \leq l} \left[\begin{array}{l} RemZ(Prime(g_{z_i})) \\ \cup \{q \in RemZ(Prime(g_{\bar{z}_i}) | q \supseteq s_i\} \end{array} \right] \right)$$

Intuition: $RemZ(Prime(g_{z_i}))$ includes implicants of f that do not intersect the privileged cube p_i . $\{q \in RemZ(Prime(g_{\bar{z}_i}) | q \supseteq s_i\}$ includes implicants of f that legally intersect p_i , i.e. contain the corresponding start point s_i . The \bigcap ensures that only those implicants remain that are legal with respect to all privileged cubes, i.e. that are dhf-implicants. The SCC removes implicants contained in other implicants to yield the final set of dhf-prime-implicants.

Proof: “ \subseteq ” (any product in $dhf\text{-Prime}(f, T)$ is also contained in the SCC -expression⁶):

Let $q \in dhf\text{-Prime}(f, T)$, then q does not intersect any privileged cube illegally, i.e. for each privileged cube it holds that q either contains the corresponding start point or does not intersect the privileged cube at all.

Suppose q intersects legally p_1, \dots, p_i , and q does not intersect p_{i+1}, \dots, p_l - i.e. q is an implicant of $\bar{p}_{i+1}, \dots, \bar{p}_l$, then $q\bar{z}_1 \dots \bar{z}_i$ is an implicant of g .

$q\bar{z}_1 \dots \bar{z}_i$ is a prime implicant of g because:

(i) Removing (any) \bar{z}_i results in a cube which is not an implicant of $\bar{z}_i + \bar{p}_i$, and hence not an implicant of g .

(ii) Removing (any) positive or negative x_j literal (of q) results in a cube such that its restriction to the x -literals, q_{new} , either intersects the OFF-set of f , or intersects for some i privileged cube p_i , $i \in \{i+1, \dots, l\}$ and is therefore no longer an implicant of $\bar{z}_i + \bar{p}_i$. In either case q_{new} is not an implicant of g .

Thus, for each i , q is by construction in at least one of $RemZ(Prime(g_{z_i}))$ or $\{q \in RemZ(Prime(g_{\bar{z}_i}) | q \supseteq s_i\}$. Therefore, q is contained in the intersection of those l sets. Also, q cannot be filtered out by the SCC -operator since by construction all cubes contained in the SCC -expression are dhf-implicants. Thus, q is contained in the SCC -expression.

“ \supseteq ” (any product contained in the SCC -expression is also contained in $dhf\text{-Prime}(f)$):

Let $q \notin dhf\text{-Prime}(f, T)$. We show that q is not contained in the SCC -expression.

Case (i): q is a dhf-implicant that is strictly contained in some dhf-prime implicant. Then q is filtered

⁶“ SCC -expression” refers to the entire expression:

$$SCC \left(\bigcap_{1 \leq i \leq l} \left[RemZ(Prime(g_{z_i})) \cup \{q \in RemZ(Prime(g_{\bar{z}_i}) | q \supseteq s_i\} \right] \right)$$

out because of the SCC -operator and therefore not contained in the SCC -expression.

Case (ii): q is not a dhf-implicant. Since by construction all cubes contained in the SCC -expression are dhf-implicants, q cannot be contained in the SCC -expression. \square

3.6 Multi-output Case

For simplicity of presentation only, it was assumed that f is a single-output function. However, it is well-known [20] that multi-output logic minimization can be reduced to single-output minimization. Based on this theorem, the above characterization carries over in a straightforward way to multi-output functions. All examples given later in the experimental results section are multi-output functions.

4 Implicit 2-Level Logic Minimization: SCHERZO

Before proceeding to our new hazard-free implicit minimization algorithm, this section briefly reviews the state-of-the-art synchronous exact two-level logic minimization algorithm, called SCHERZO [6, 4, 3, 5], which forms a basis of our new hazard-free implicit minimization method. Using *implicit* minimization techniques, SCHERZO is 10 to more than 100 times faster than the best previous minimization methods.

SCHERZO has two significant differences from classic minimization algorithms like the well-known Quine-McCluskey algorithm:

- SCHERZO uses data structures like BDDs and ZBDDs to represent Boolean functions and sets of products very efficiently (see the Appendix for a review of BDDs and ZBDDs). Thus, the complexity of the minimization problem is shifted, and the cost of the cyclic core computation⁷ is independent of the number of products (e.g. the number of prime implicants) that are manipulated.
- SCHERZO includes new algorithms that operate on these data structures. The motivation is that the logic minimization problem can be considered as a set covering problem over a lattice. More specifically, both the *covering objects*, P , and the *objects-to-be-covered*, Q , are subsets of the lattice \mathcal{P} of all Boolean products (over the set of literals). A new cyclic core computation algorithm (see below) uses then two endomorphisms τ_P and τ_Q , which operate on Q and P respectively, to capture dominance relations and to compute the fixpoint C , which can be shown to be isomorphic to the cyclic core.

⁷A set covering problem can be reduced in size by repeated elimination of essential elements and application of dominance relations. The remaining set covering problem (if any) is called the cyclic core.

Below is a short description of SCHERZO's algorithmic approach⁸. Note that for the understanding of this paper the actual implementation of algorithms is not important. Rather it is of interest which data structures they manipulate and that the algorithms have been very effective in practice.

Algorithm: SCHERZO
Input: Boolean function f .
Output: All minimum covers of f .

1. Compute the ZBDD $P^{(init)}$ of Prime(f) (the set of all prime implicants of f , or covering objects). Here, f is given as a BDD.
2. Compute the ZBDD $Q^{(init)}$ of the set of ON-set minterms of f , (i.e., the objects to be covered).
3. Solve the implicit set covering problem $\langle Q^{(init)}, P^{(init)}, \subseteq \rangle$
 (Note that " \subseteq " replaces " \in ", usually used to describe the relation between the two sorts of objects of a covering problem, since our set covering problem is considered over a lattice, as explained above.)

- (a) Determining the cyclic core:
 Compute the fixpoint C , which is isomorphic to the cyclic core, produced by the following *rewriting rules* on the implicit set covering problem $\langle Q, P, \subseteq \rangle := \langle Q^{(init)}, P^{(init)}, \subseteq \rangle$,

$$\begin{aligned} \langle Q, P, \subseteq \rangle &\rightarrow \langle \max_{\subseteq} \tau_P(Q), \max_{\subseteq} \tau_Q(P), \subseteq \rangle \\ \langle Q, P, \subseteq \rangle &\rightarrow \langle Q - E, P - E, \subseteq \rangle, \\ &\text{with } E = Q \cap P \end{aligned}$$

where τ_P and τ_Q are defined from \mathcal{P} into \mathcal{P} by:

$$\begin{aligned} \tau_Q(r) &= \sup_{\subseteq} \{q \in Q \mid q \subseteq r\} \\ \tau_P(r) &= \inf_{\subseteq} \{p \in P \mid r \subseteq p\} \end{aligned}$$

Intuition for τ -operators

To understand the rewriting rules consider first the following examples for *sup* (supremum) and *inf* (infimum):

$$\begin{aligned} \sup_{\subseteq} \{x_1 x_2, x_2 \bar{x}_3\} &= x_2 \text{ and} \\ \inf_{\subseteq} \{x_1 x_2, x_2 \bar{x}_3\} &= x_1 x_2 \bar{x}_3. \end{aligned}$$

Operator τ_Q maps each product r of the covering objects (initially a prime implicant)

onto the supremum of all products (initially on-set minterms) that it covers. Basically, r is mapped onto the smallest cube r' such that r' still covers the same set of products as r . This process often reduces product r .

Operator τ_P maps each product r of the objects-to-be-covered (initially an on-set minterm) onto the infimum of all products (initially prime implicants) by which it is covered. Basically, r is mapped onto the largest cube r' such that r' is still covered by the same set of products as r . This process often enlarges product r .

max removes cubes contained in other cubes. Each non-maximal covering object can be removed since it is included in a "better" cube, i.e. one that covers more. Each non-maximal object-to-be-covered can be removed since the containment in another larger object-to-be-covered ensures its covering.

The intuition behind the τ -operators (together with *max*) is that they are very often not injective, that is, they may reduce the size of the covering problem. Basically, the τ operators capture dominance relations. Also, it can be shown that the *essential elements*⁹ (above denoted by E) are those elements that are present in the intersection of P and Q at any iteration.

The rewriting rules for $\langle Q, P, \subseteq \rangle$ are iterated until no change: the fixpoint C is computed, which means that the cyclic core is determined and implicitly represented by Q and P .

- (b) Solving the cyclic core:
 The resulting fixpoint C is solved by using a branch-and-bound method, modified to generate all minimum-cost solutions, and step 3(a).
- (c) Solutions to covering problem:
 Let F be the union of the sets E found during the computation of the fixpoint C in step 3(a). Let $Sol(C)$ be the set of solutions to C . Then the set of *all* solutions of the 2-level logic minimization of f is:

$$\bigcup_{S \in Sol(C)} \times_{r \in S \cup F} \{p \in P^{(init)} \mid r \subseteq p\}$$

Intuition: Each $r \in S \cup F$ represents an equivalence class of primes, which is the set

⁸The ZBDD based recursive algorithms that implement the steps efficiently can be found in [3].

⁹Essential elements are products that are in every minimum solution.

of primes that cover r . Each solution includes exactly one of these primes. The Cartesian product therefore gives rise to the set of all solutions.

5 A new implicit minimizer for 2-level hazard-free logic: IMPYMIN

Based on the ideas of the previous two sections, we are now able to present a new exact minimization algorithm for multi-output 2-level hazard-free logic. We will show in the next section that our implicit method outperforms existing minimizers by a large factor.

Nowick/Dill reduced 2-level hazard-free optimization to a unate covering problem (see Section 2) where each required cube has to be covered by at least one dhf-prime implicant. As with synchronous logic minimization in SCHERZO, hazard-free logic minimization can also be considered over the lattice of the set of products (over the set of literals). The major difference to synchronous two-level logic minimization is the setting up of the covering problem, i.e. we need to find a method that computes the set $\text{dhf-Prime}(f, T)$ efficiently, i.e. preferably in an implicit manner. Fortunately, this can be done using the new characterization of $\text{dhf-Prime}(f, T)$ of Section 3. Our algorithm is as follows.

Algorithm: Implicit hazard-free logic minimization

Input: Boolean function f , set of transitions T .

Output: All minimum hazard-free covers of (f, T) .

1. Compute the ZBDD $P^{(init)}$ of $\text{dhf-Prime}(f, T)$.
2. Compute the ZBDD $Q^{(init)}$ of $\text{REQ}(f, T)$ (set of required cubes of (f, T)).
3. Solve the implicit unate set covering problem $(Q^{(init)}, P^{(init)}, \subseteq)$.

We now explain each of the steps in detail.

5.1 Computation of the ZBDD of $\text{dhf-Prime}(f, T)$

Suppose that f is given as a BDD (if f is given as a set of cubes, we first compute its BDD). From the BDD representing f , we can easily compute the BDD representing g , and then the ZBDD of $\text{Prime}(g)$ using an existing recursive algorithm [3]. From the ZBDD of $\text{Prime}(g)$, we compute the ZBDD of $\text{dhf-Prime}(f, T)$ using Theorem 3.5. It remains to show that the necessary operations $\text{Prime}(g_{z_i})$, $\text{Prime}(g_{\bar{z}_i})$, RemZ , and SCC can be implemented efficiently on ZBDDs:

- *Computing $\text{Prime}(g_{z_i})$:* Assuming that positive and negative literal nodes of the same variable are always adjacent in the ZBDD, we only need to traverse the ZBDD of $\text{Prime}(g)$. We apply at each z_i variable the following operation. We

compute the set union of the two successors corresponding to those products that include positive literal z_i and to those products that do not depend on z_i .

- *Computing the ZBDD of $\text{Prime}(g_{\bar{z}_i})$:* Analogously.
- *Computing the ZBDD of RemZ :* RemZ deletes all z -literals in the ZBDD. We traverse the ZBDD, and at each z_i - or \bar{z}_i -literal, we replace the corresponding node with the ZBDD corresponding to the union of the two successors.
- *SCC (Single-Cube Containment):* The last task, the application of the SCC -operator, which removes cubes contained in other cubes, is actually not done in this step, since it is automatically taken care of in step 3.

To summarize, based on Theorem 3.5 we can compute the covering objects, $\text{dhf-Prime}(f, T)$, in an implicit manner.

5.2 Computation of the ZBDD of $\text{REQ}(f, T)$

From the set of input transitions, T , the set of required cubes can be easily computed (see [16]). The set of required cubes can then be stored as a ZBDD.

5.3 Solving the Implicit Covering Problem

The implicit set covering problem $(Q^{(init)}, P^{(init)}, \subseteq)$ can be solved analogously to Step 3 of SCHERZO (i.e. passed to the unate set covering solver of SCHERZO).

One subtle difference regarding the correctness is worth considering. SCHERZO's τ operators map products onto other products (for details, see the Appendix). It is possible that a product which is a dhf-implicant is mapped, by τ , onto a non-dhf implicant. This does *not* do any harm because we are ensured that all products of the final solution produced by the solver are products that were originally given to the solver, i.e. dynamic-hazard-free, through a re-mapping operation (see Step 3(c) in Section 4). Hence, it is fine to use SCHERZO's set covering solver as a black box.

5.4 A Note on the Efficiency of IMPYMIN

It is worth pointing out that appending z -variables for dhf-prime generation is only a small change to the corresponding synchronous problem. In particular, the BDD for g is not much larger than the BDD for f . Thus, the generation of $\text{dhf-Prime}(f, T)$ can be done nearly as fast as the generation of primes *without* hazard-freedom considerations. Moreover, the resulting covering problem is unlikely to be much harder than the corresponding synchronous problem. To

summarize, the proposed method performs hazard-free logic minimization nearly as efficient as synchronous logic minimization by incorporating state-of-the-art techniques for implicit prime generation and implicit set covering solving. However, note that this could only be achieved based on the presented new and non-trivial formulation of the set of dhf-prime implicants, presented in Section 3.

6 Experimental Results and Comparison with Related Work

A prototype version of our new minimizer for exact 2-level hazard-free logic minimization was run on several well-known benchmark circuits [7, 21] on an ULTRA-SPARC 140 workstation (Memory: 89 MB real/ 230 MB virtual).

6.1 Comparison of exact minimizers: IMPYMIN vs. HFMIN

The table in Figure 3 compares our new exact minimizer IMPYMIN with the currently fastest available exact minimizer, HFMIN, by Robert Fuhrer et al. [7].

For the smaller problems, HFMIN is faster, since our implementation is not yet optimized¹⁰. However, the bottleneck of HFMIN becomes clearly visible already for medium-sized examples. For examples *sd-control* and *stetson-p2*, IMPYMIN is more than three times faster; for the benchmark *pscsi-pscsi* even more than fifteen times.

For very large examples, IMPYMIN outperforms HFMIN by a large factor. While HFMIN cannot solve *stetson-p1* within 20 hours, we can solve it in just 813 seconds. The superiority of implicit techniques becomes very apparent for the benchmark *cache-ctrl*. While HFMIN gives up (after many minutes of run-time) because the 230MB of virtual memory are exceeded, our method can minimize the benchmark in just 301 seconds.

6.2 Comparison of IMPYMIN with the heuristic minimizer ESPRESSO-HF

Figure 4 compares IMPYMIN with an improved version of ESPRESSO-HF, the heuristic minimizer presented in [21]. Note that the reported run-times for both IMPYMIN and ESPRESSO-HF are much faster than those run-times for an older version of ESPRESSO-HF reported in [21]. Besides run-time and size of solution, the table also reports the number of variables that need to be added (for IMPYMIN).

The two minimizers are somewhat orthogonal.

¹⁰Our BDD package is still very inefficient. In particular, it includes a static (i.e. not a dynamic) hash table. The hash table for small examples is unnecessarily large. In fact, the run-time is completely dominated by initializing the hash tables. If we use an appropriate-sized hash table for smaller examples, experiments indicate that IMPYMIN can actually solve the small examples as fast as HFMIN.

name	i/o	#c	HFMIN	IMPYMIN
			time	time
cache-ctrl	20/23	97	impossible	301
dram-ctrl	9/8	22	1	13
pe-send-ifc	12/10	27	9	16
pscsi-ircv	8/7	12	1	10
pscsi-isend	11/10	23	3	15
pscsi-pscsi	16/11	77	1656	105
pscsi-tsend	11/10	22	3	13
pscsi-tsend-bm	11/11	23	3	13
sd-control	18/22	34	172	52
sscsi-isend-bm	10/9	22	1	11
sscsi-trcv-bm	10/9	24	1	13
sscsi-tsend-bm	11/10	20	2	13
stetson-p1	32/33	60	> 72000	813
stetson-p2	18/22	37	151	49
stetson-p3	6/4	7	1	8

Figure 3: Comparison of exact hazard-free minimizers (#c - number of cubes in minimum solution, time - run-time in seconds)

On the one hand, IMPYMIN computes a cover of minimum size, whereas ESPRESSO-HF is not guaranteed to find a minimum cover, but typically does find a cover of very good quality.

On the other hand, ESPRESSO-HF is typically faster than IMPYMIN. However, since neither tool has been highly optimized for speed, we think it is very important to analyze the intrinsic advantages and disadvantages of the two methods. Intuitively, both methods overcome the three bottlenecks of HFMIN—prime implicant generation, transformation of prime implicants to dhf-prime implicants, and solution of the covering problem—each of which being solved by an algorithm with exponential worst-case behavior. However, the way in which ESPRESSO-HF and IMPYMIN overcome the bottlenecks is very different. Whereas IMPYMIN uses implicit data structures (but still follows the same steps as HFMIN), ESPRESSO-HF follows a very different approach. Thus, the two methods are orthogonal in its approach to overcome the bottlenecks. Moreover, while ESPRESSO-HF is faster than IMPYMIN on all of our examples, this does not mean that this is necessarily true for other examples. The reason is that the complexity of ESPRESSO-HF's algorithms depend on the the number of required cubes, whereas IMPYMIN benefits from a more compact representation of the set of required cubes as a ZBDD.

In this context, it is important to note that very often the role data structures like BDDs play in obtaining efficient implementations of CAD algorithms is misunderstood. Using BDDs, many CAD problems can now be solved much faster than before the inception of BDDs. However, the naive approach of taking an existing CAD algorithm and augmenting it with BDDs does not necessarily lead to a good tool (see

name	i/o	ESPRESSO-HF		IMPYMIN		
		#c	time	#c	time	#v
cache-ctrl	20/23	99	105	97	301	39
dram-ctrl	9/8	22	1	22	13	6
pe-send-ifc	12/10	27	1	27	16	5
pscsi-ircv	8/7	12	1	12	10	3
pscsi-isend	11/10	23	1	23	15	6
pscsi-pscsi	16/11	78	11	77	105	23
pscsi-tsend	11/10	22	1	22	13	4
pscsi-tsend-bm	11/11	23	1	23	13	4
sd-control	18/22	35	3	34	52	0
sscsi-isend-bm	10/9	22	1	22	11	3
sscsi-trcv-bm	10/9	24	1	24	13	5
sscsi-tsend-bm	11/10	20	1	20	13	4
stetson-p1	32/33	60	21	60	813	9
stetson-p2	18/22	37	2	37	49	0
stetson-p3	6/4	7	1	7	8	1

Figure 4: Comparison of the heuristic minimizer ESPRESSO-HF with the exact minimizer IMPYMIN (#c - number of cubes in solution, time - run-time in seconds, #v - number of added variables)

discussion in [3]). In particular, it is impossible to just augment ESPRESSO-HF or HFMIN with BDDs and get a superb tool. That is why we needed a new theoretical result on the characterization of dhf-prime implicants (cf. Section 3.5) on which our new exact implicit minimizer is based.

6.3 Comparison with Rutten's Work

An interesting alternative approach to our new characterization of dhf-prime implicants (cf. Section 3.5) was recently presented by Rutten et al. [19]. His new algorithm for computing dhf-prime implicants is very different from ours. His approach follows a divide-and-conquer paradigm. In particular, the problem is split into three sub-problems with respect to a splitting variable. The first (second, third) sub-problem generates those dhf-prime implicants that have a positive literal (negative literal, don't care-literal) for the splitting variable. The underlying idea why this approach may be efficient is that it allows to determine illegal intersections of privileged cubes already during the splitting phase (see [19] for details), which can significantly reduce the recursion tree and lead fast to terminal cases. In the merging phase of the divide-and-conquer approach, the solutions to the sub-problems are combined.

However, it is worth pointing out that a major difference of our work to Rutten's work is that his approach is *not* based on implicit representations. While Rutten's work is nevertheless very promising, it has not been fully evaluated so far. In particular, he only presented run-times for the computation of dhf-prime implicants of *single-output functions*, i.e. only for functions that are *significantly smaller* than those

that can be handled by our method (cf. Section 6.1). Moreover, no results for hazard-free 2-level logic minimization, based on his new approach to computing dhf-prime implicants, were presented.

7 Conclusions

We have presented IMPYMIN, a new and very efficient implicit exact minimization method for multi-output 2-level hazard-free logic.

IMPYMIN performs hazard-free logic minimization nearly as efficiently as synchronous logic minimization by incorporating state-of-the-art techniques for implicit prime generation and implicit set covering solving.

IMPYMIN is able to solve all examples, ranging up to 32 inputs and 33 outputs, in less than 813 seconds. These include several large examples that could not be exactly minimized by previous methods¹¹. For medium-sized examples our minimizer outperforms existing techniques by a large factor.

IMPYMIN is based on the new idea of incorporating hazard-freeness constraints within a synchronous function by adding extra inputs. We expect that the proposed technique may very well be applicable to other hazard-free optimization problems, too.

Acknowledgment

The authors would like to thank Olivier Coudert for very helpful discussions and for his immense help with the experiments. The authors would also like to thank Bob Fuhrer for providing his tool HFMIN, and Montek Singh for interesting discussions.

References

- [1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677-691, August 1986.
- [2] Chou, Beerel, Ginosar, Kol, Myers, Rotem, Stevens, and Yun. Optimizing average-case delay in the technology mapping of domino dual-rail circuits: A case study of an asynchronous instruction length decoding pla. In *IEEE ASYNC-1998 Symp.*
- [3] O. Coudert. Two-level logic minimization: an overview. *Integration, the VLSI journal*, 17:97-140, 1994.
- [4] O. Coudert. Doing two-level logic minimization 100 times faster. In *ACM-SIAM Symposium on Discrete Algorithms*, 1995.
- [5] O. Coudert. On solving covering problems. In *DAC-1996*.
- [6] O. Coudert and J.C. Madre. New ideas for solving covering problems. In *DAC-1995*.
- [7] R.M. Fuhrer, B. Lin, and S.M. Nowick. Symbolic hazard-free minimization and encoding of asynchronous finite state machines. In *ICCAD-1995*.
- [8] P. Kudva, G. Gopalakrishnan, and H. Jacobson. A technique for synthesizing distributed burst-mode circuits. In *DAC-1996*.
- [9] A. Marshall, B. Coates, and P. Siegel. The design of an asynchronous communications chip. *Design and Test*, June 1994.

¹¹Note that in publications on the 3D method (see e.g. [25, 23]) several of these examples appear, but only *single-output* minimization was performed.

- [10] S. Minato. Zero-Suppressed BDDs for set manipulation in combinatorial problems. In *DAC-1993*.
- [11] S.M. Nowick and B. Coates. UCLOCK: automated design of high-performance unlocked state machines. In *ICCD-1994*.
- [12] 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. In *HICSS-1993*.
- [13] S.M. Nowick and D.L. Dill. Synthesis of asynchronous state machines using a local clock. In *ICCD-1991*.
- [14] S.M. Nowick and D.L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. In *ICCAD-1992*.
- [15] S.M. Nowick, N.K. Jha, and F. Cheng. Synthesis of asynchronous circuits for stuck-at and robust path delay fault testability. In *VLSI Design 1995*.
- [16] S. M. Nowick and D. L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. *IEEE TCAD*, CAD-14(8):986-997, August 1995.
- [17] R. Rudell and A. Sangiovanni Vincentelli. Multiple valued minimization for PLA optimization. *IEEE TCAD*, CAD-6(5):727-750, September 1987.
- [18] J.W.J.M. Rutten and M.R.C.M. Berkelaar. Improved state assignments for burst mode finite state machines. In *IEEE ASYNC-1997 Symp.*
- [19] J.W.J.M. Rutten and M.A.J. Kolsteren. A divide and conquer strategy for hazard free 2-level logic synthesis. In *International Workshop on Logic Synthesis*, 1997.
- [20] T. Sasao. An application of multiple-valued logic to a design of programmable logic arrays. In *Proceedings of Int. Symposium on Multiple-Valued Logic*, 1978.
- [21] M. Theobald, S.M. Nowick, and T. Wu. Espresso-HF: A heuristic hazard-free minimizer for two-level logic. In *DAC-1996*.
- [22] S.H. Unger. *Asynchronous Sequential Switching Circuits*. New York: Wiley-Interscience, 1969.
- [23] K. Yun and D.L. Dill. A high-performance asynchronous SCSI controller. In *ICCD-1995*.
- [24] K. Yun, A. Dooply, J. Arceo, P. Beerel, and V. Vakilotojar. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *IEEE ASYNC-1997 Symp.*
- [25] K.Y. Yun, D.L. Dill, and S.M. Nowick. Synthesis of 3D asynchronous state machines. In *ICCD-1992*.

Appendix

A BDDs

Binary Decision Diagrams (BDDs) [1] are used to efficiently represent Boolean functions. A BDD of a function f is obtained from the Shannon tree representation of f by reduction rules which (i) identify isomorphic subgraphs and (ii) delete each vertex that has the same left and right children.

Example A.1 In Figure 5a) the Shannon tree of the function $f = ab+c$ is shown. To find the function value for a specific assignment to the variables, one follows the path from the root node to a terminal node, taking the left (right) branch if the corresponding variable is assigned the value 0 (1). The corresponding BDD obtained by above reduction rules is shown in part

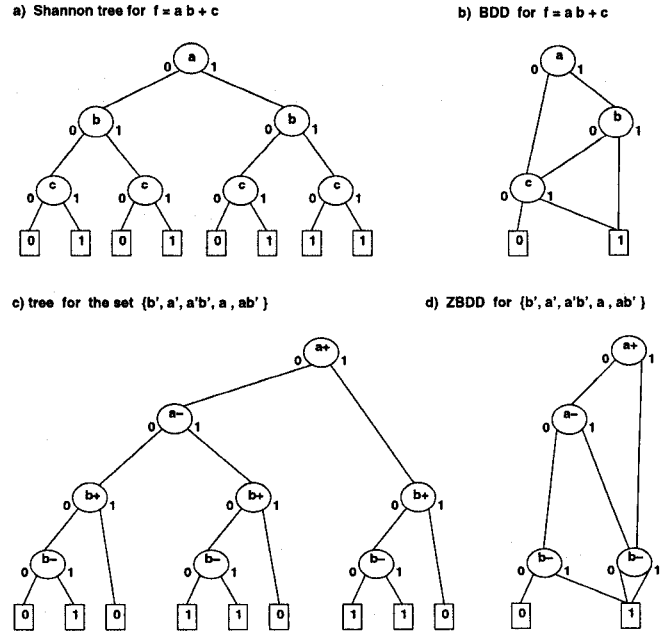


Figure 5: BDDs and ZBDDs

b) of the figure. Note that the BDD of f is just a compact representation of the Shannon tree of f . In particular, the same algorithm can be used to evaluate the function for an assignment to the variables. \square

B ZBDDs

Zero-suppressed BDDs [10] are a variant of BDDs which were introduced to efficiently represent sets of products, e.g. the set of prime implicants of a function f . A ZBDD of a set of products is obtained from a tree representation of the set of products by reduction rules which (i) identify isomorphic subgraphs and (ii) delete each vertex whose right children points to 0 (i.e. the empty set). Note that to achieve small representations for sparse sets, the second reduction rule differs from the second reduction rule for BDDs. Another difference from BDDs is that a ZBDD makes decisions based on *literals* instead of *variables*.

Example B.1 Consider the tree representation of the set of products $\{b', a', a'b', a, ab'\}$ in Figure 5c). Here each path from the root node to a terminal 1 node corresponds to a product in the set. The product consists of those *literals* encountered on taking right branches on the path. Here, positive (negative) literals are denoted by a '+' superscript ('-' superscript). The ZBDD for this set of products obtained by above reduction rules is shown in part d) of the figure. \square