Fast Heuristic and Exact Algorithms for Two-Level Hazard-Free Logic Minimization

Michael Theobald and Steven M. Nowick

Abstract— *None of the available minimizers for 2-level hazard-free logic minimization can synthesize very large circuits. This limitation has forced researchers to resort to manual and automated circuit partitioning techniques.*

This paper introduces two new 2-level hazard-free logic minimizers: Espresso-HF, *a heuristic method which is loosely based on* Espresso-II, *and* Impymin, *an exact method based on implicit data structures. Both minimizers can solve all currently-available examples, which range up to 32 inputs and 33 outputs. These include examples that have never been solved before. For the more difficult examples that can be solved by other minimizers, our methods are several orders of magnitude faster.*

As by-products of these algorithms, we also present two additional results. First, we propose a fast new method to check if a hazard-free covering problem can feasibly be solved. Second, we introduce a novel reformulation of the 2-level hazard-free logic minimization problem, by capturing hazard-freedom constraints within a synchronous function through the addition of new variables.

I. Introduction

Asynchronous design has been the focus of much recent research activity [10]. In fact, asynchronous design has been applied to several large-scale control- and datapath circuits and microprocessors [14], [22], [15], [23], [35], [39], [20], [2].

A number of methods have been developed for the design of asynchronous controllers. Much of the recent work has focused on *Petri Net-based methods* [18], [1], [16], [5], [40], [6], [7] and *burst-mode methods* [27], [9], [25], [43], [17], [31], [13]. These two classes of design methods differ in fundamental aspects: the *delay model*, and how the circuit interacts with its *environment* [10]. Petri Net-based methods typically synthesize circuits to work correctly regardless of gate delays (*speed-independent delay model*), and the environment is allowed to respond to the circuit's outputs without timing constraints (*input/output mode*). In contrast, burst-mode methods synthesize combinational logic to work correctly regardless of gate and wire delays, but the correct sequential operation depends on timing constraints. In this case, the environment must wait for a circuit to stabilize before responding with new inputs (*fundamental mode*).

The focus of this paper is on fundamental-mode asynchronous circuits, such as burst-mode machines. Burst-mode methods have recently been applied to several large and realistic design examples, including a low-power infrared communications chip [19], a second-level cache-controller [26], a SCSI controller [41], a differential equation solver [42], and an instruction length decoder [4].

This work was supported by NSF under Grant no. MIP-9501880 and by an Alfred P. Sloan Research Fellowship. The presented work is an extended version of two recent conference papers [37], [36] .

Michael Theobald and Steven M. Nowick are with the Department of Computer Science, Columbia University, New York, NY 10027. Email: {theobald,nowick}@cs.columbia.edu.

An important challenge for any asynchronous synthesis method 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, however, a key constraint is to provide *hazard-free* logic, i.e. to guarantee the absence of glitches [38]. Much progress has been made in developing hazard-free synthesis methods, including tools for exact two-level hazard-free logic minimization [29], optimal state assignment [12], [31], synthesis-for-testability [28] and low-power logic synthesis [24]. However, these tools have been limited in handling large-scale designs.

In particular, hazard-free 2-level logic minimization is a bottleneck in most asynchronous CAD packages. While the currently used Quine-McCluskey-like exact hazard-free minimization algorithm, Hfmin [12], has been effective on small- and medium-sized examples, and is used in several existing CAD packages [27], [9], [25], [43], [17], [13], it has been unable to produce solutions for several large design problems [17], [31]. 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 [17].

Contributions of This Paper

This paper introduces two very efficient 2-level logic minimizers for hazard-free multi-output minimization: Espresso-HF and Impymin.

Espresso-HF is an algorithm to solve the heuristic hazard-free two-level logic minimization problem. The method is heuristic solely in terms of the cardinality of solution. In all cases, it guarantees a hazard-free solution. The algorithm is based on Espresso-II [30], [11], but with a number of significant modifications to handle hazard-freedom constraints. It is the first heuristic method based on Espresso-II to solve the hazard-free minimization problem. Espresso-HF also includes a new and much more efficient algorithm to check for existence of a hazard-free solution, without generating all prime implicants.

Impymin is an algorithm to solve the exact hazard-free two-level logic minimization problem. The algorithm uses an implicit approach which makes use of data structures such as BDDs [3] and zero-suppressed BDDs [21]. The algorithm is based on a novel theoretical approach to hazard-free two-level logic minimization. We 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 to use an existing method for fast implicit generation of prime implicants. Moreover, our novel approach can be nicely incorporated into a very efficient implicit minimizer for hazard-free logic. In particular, the approach makes it possible to use the implicit set covering solver of Scherzo [8], the state-of-the-art minimization method for synchronous two-level logic, as a black box.

Both Espresso-HF and Impymin can solve all currently available examples, which range up to 32 inputs and 33 outputs. These include examples that have never been previously solved. For examples that can be solved by the currently fastest minimizer Hfmin, our two minimizers are typically several orders of magnitude faster. In particular, Impymin can find a minimum-size cover for all benchmark examples in less than 813 seconds, and Espresso-HF can find very good covers – at most 3% larger than a minimum-size cover – in less than 105 seconds.

Espresso-HF and Impymin are somewhat orthogonal. On the one hand Espresso-HF is typically faster than Impymin. On the other 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.

Paper Organization

Section 2 gives background on circuit models, hazards and hazard-free minimization. Section 3 describes the Espresso-HF algorithm for heuristic hazard-free minimization. Section 4 introduces a new approach to hazard-free minimization where hazard-freedom constraints are captured by a constructed synchronous function, leading to a new method for computing dynamic-hazard-free prime implicants. Based on the results of Section 4, Section 5 introduces our new implicit method for exact hazard-free minimization, called Impymin. Section 6 presents experimental results and compares our approaches with related work, and Section 7 gives conclusions.

II. Background

The material of this section focuses on hazards and hazard-free logic minimization, and is taken from [12] and [29]. For simplicity, we focus on single-output functions. A generalization of these definitions to multi-output functions is straightforward, and is described in [12].

A. Circuit Model

This paper considers combinational circuits having arbitrary finite gate and wire delays (an *unbounded wire delay model* [29]). A pure delay model is assumed as well (see [38]).

B. Multiple-Input Changes

Definition II.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, $[A, B]$ is the uniquely defined smallest cube that contains A and B: *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 $X \in [A, B]$, $f(X) \in \{0, 1\}$.

C. 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 II.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 II.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 [29], [38]. Therefore, we consider only transitions which are free of function hazards [1].

D. 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 II.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.

E. 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 [29]). Assume that $[A, B]$ is the transition cube corresponding to a *function-hazard-free*

[1]Sequential synthesis methods, which use hazard-free minimization as a substep, typically include constraints in their algorithms to insure that no transitions with function hazards are generated [27], [43].

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 II.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 II.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 $1 \rightarrow 0$ transition [2].

Lemma II.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 II.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.*

F. Required and Privileged Cubes

The cube $[A, B]$ in Lemma II.6 and the *maximal* subcubes $[A, X]$ in Lemma II.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 II.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**.
Lemma II.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 II.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. A privileged cube is called **trivial**, if 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. All trivial privileged cubes can safely be removed from consideration without loss of information.

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

G. 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 II.11 (Hazard-Free Covering [29])
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 II.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 II.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 II.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 function using only dhf-prime implicants where every required cube is covered.*

The difference between two-level hazard-free logic minimization and the well-know classic two-level logic minimization problem (e.g. solved by Quine-McCluskey algorithm) is that, in the hazard-free case, dhf-prime implicants replace prime implicants as the covering objects, and required cubes replace minterms as the objects-to-be-covered.

In general, the covering conditions of Theorem II.11 may not be satisfiable for an arbitrary Boolean function and set of transitions [38], [29]. This case occurs if conditions (b) and (c) cannot be satisfied simultaneously.

A hazard-free minimization example is shown in Figure 1. There are four specified transitions. t_1 is a $1 \rightarrow 1$ transition; it gives rise to one required cube (see part (a)). t_2 is a $0 \rightarrow 0$ transition; it gives rise neither to required cubes nor privileged cubes. t_3 and t_4 are $1 \rightarrow 0$ transitions. Each of the two gives rise to two required cubes (see (a)) and one privileged cube (see (b)). A minimum hazard-free cover is shown in part (c). Each required cube is covered, and no product in the cover illegally intersects any privileged cube. In contrast, the cover in part (d) is not hazard-free since *priv-cube-1* is intersected illegally (highlighted minterm) by product x_2x_4. In particular, this product may lead to a glitch during transition t_3.

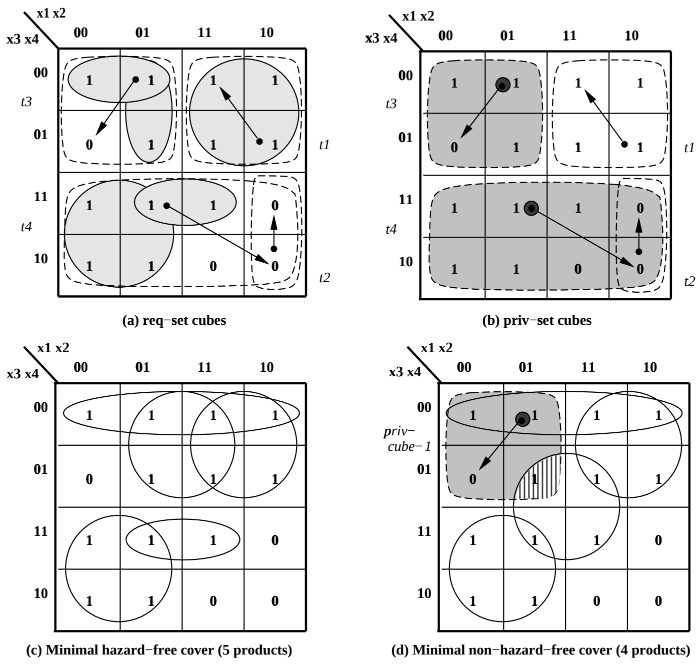
x1 x2
x3 x4
00
01
11
10
t1
t2
t3
t4
priv–cube–1
(a) req–set cubes
(b) priv–set cubes
(c) Minimal hazard–free cover (5 products)
(d) Minimal non–hazard–free cover (4 products)

Fig. 1. Two-Level Hazard-Free Minimization Example: (a) shows the set of required cubes (shaded), and the set of transition cubes (dotted); (b) shows the set of privileged cubes (shaded); (c) shows a minimal hazard-free cover; (d) shows a minimum-cost cover that is not hazard-free, since it contains a logic hazard.

H. Exact Hazard-Free Minimization Algorithms

A single-output exact hazard-free minimizer has been developed by Nowick and Dill [29]. It has recently been extended to hazard-free multi-valued minimization[3] by Fuhrer, Lin and Nowick [12]. 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 the above 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.

Very recently, Rutten [33], [32] has proposed an alternative exact method. However, his method has yet to be evaluated on difficult examples, e.g. on those that cannot be easily solved by HFMIN (see Section VI-C for more details).

III. HEURISTIC HAZARD-FREE MINIMIZATION: ESPRESSO-HF

A. Overview

The goal of heuristic hazard-free minimization is to find a very good (but not necessarily exactly minimum) solution to the hazard-free covering problem. The basic minimization strategy of ESPRESSO-HF for hazard-free minimization is similar to the one used by ESPRESSO-II. However, we use additional constraints to ensure that the resulting cover is hazard-free, and the algorithms are significantly different.

One key distinction is in the use of the *unate recursive paradigm* in ESPRESSO-II, i.e. to decompose operations recursively leading to efficiently solvable sub-operations on unate functions. To the best knowledge of the authors, the unate recursive paradigm cannot be applied directly to ESPRESSO-II-like heuristic hazard-free minimization. (In [33], [32], a unate recursive method was proposed, but only for use in the dhf-prime generation step for exact hazard-free minimization; see Section VI-C.) The intuitive reason for this is that the operators in ESPRESSO-II manipulate covers. For example, the "many" ON-set minterms (objects-to-be-covered) can typically be stored compactly and manipulated efficiently as an ON-set cover of "a few" cubes. In contrast, required cubes cannot be combined into larger cubes without loss of information, which means that the basis for the unate recursive paradigm, i.e. the concept of covers, becomes obsolete.

We therefore follow the basic steps of ESPRESSO-II, modified to incorporate hazard-freedom constraints, but without the use of unate recursive algorithms. However, because of the constraints and granularity of the hazard-free minimization problem, high-quality results are still obtained even for large examples.

In this subsection, we describe the basic steps of the algorithm, concentrating on the new constraints that must be incorporated to guarantee a cover to be hazard-free. We then describe the individual steps in detail in later subsections.

As in ESPRESSO-II, the size of the cover is never increased in size. In addition, after an initial phase, the cover always represents a valid solution, i.e. a cover of f that is also hazard-free. Pseudocode for the algorithm is shown in Figure 2.

The first step of ESPRESSO-HF is to read in PLA files specifying a Boolean function, f, and a set of specified function-hazard-free transitions, T. These inputs are used to generate the set of required cubes Q, the set of privileged cubes P and their corresponding start points S, and the OFF-set R. Generation of these sets is immediate from the earlier lemmas (see also [29]) [4].

The set Q can be regarded both as an initial cover F of the function, and as a set of objects to be covered. Unlike ESPRESSO-II, however, the given initial cover Q does *not* in general represent a valid solution: while Q is a cover of f, it is not necessarily hazard-free. Therefore, processing begins by first expanding each required cube into the *uniquely defined* minimum dhf-implicant covering it, or the detection that this is impossible, denoted by "undefined". The latter case indicates that the hazard-free minimization problem has no solution (see Section III-J). Otherwise, the result is an initial hazard-free cover, F, and set of objects to be covered, Q^f.

[3]It is well-known that multi-output minimization can be regarded as a special case of multi-valued minimization [30].

[4]The algorithm does not need an explicit cover for the don't-care set because the operations only require the OFF-set to check if a cube is valid.

```
Espresso-HF(f,T)
  $Q$ = generate_set_of_required-cubes(f,T)
  $P$ = generate_set_of_privileged-cubes(f,T)
  $S$ = generate_set_of_start-points(f,T)
  $R$ = OFF-set(f)
  $Q^f = \{supercube_{dhf}(q) | q \in Q\}$
  If "undefined" $\in Q^f$ then no solution is possible; exit
  Minimize $Q^f$ with respect to single cube containment
  $F = Q^f$
  $(F, E)$ = expand_and_compute_essentials($F$)
  Remove all cubes from $Q^f$ that are already covered by $E$
  $F = F$ - $E$
  $F$ = irredundant($F$)
  do
      $\phi_2 = |F|$
      do
          $\phi_1 = |F|$
          $F$ = reduce($F$)
          $F$ = expand($F$)
          $F$ = irredundant($F$)
      while ($|F| < \phi_1$)
      $F$ = last_gasp($F$)
  while ($|F| < \phi_2$)
  $F = F \cup E$
  $F$ = make_dhf_prime($F$)
```

Fig. 2. The ESPRESSO-HF algorithm.

The next step is to identify *essential dhf-implicants*, using a modified EXPAND step. This algorithm uses a novel approach to identifying *equivalence classes* of implicants, each of which is treated as a single implicant. Essential implicants, as well as all required cubes covered by them, are then removed from F and Q^f, respectively, resulting in a smaller problem to be solved by the main loop. Before the main loop, the current cover is also made irredundant.

Next, as in ESPRESSO-II, ESPRESSO-HF applies the three operators REDUCE, EXPAND, and IRREDUNDANT to the current cover until no further improvement in the size of the cover is possible. Since the result may be a local minimum, the operator LAST_GASP is then applied to find a better solution using a different method. EXPAND uses new hazard-free notions of *essential parts* and *feasible expansion.* The other steps differ from ESPRESSO-II as well.

At the end, there is an additional step to make the resulting implicants dhf-prime, MAKE_DHF_PRIME, since it is desirable to obtain a cover that consists of dhf-prime implicants. The motivation for this step will be made clear in the sequel.

In addition to the steps shown in Figure 2, our implementation has several optional pre- and postprocessing steps.

B. Dhf-Canonicalization of Initial Cover

In ESPRESSO-II, the initial cover of a function is provided by its ON-set, F^{ON}. This cover is a seed solution, which is iteratively improved by the algorithm. By analogy, in ESPRESSO-HF, the initial cover is provided by the set of required cubes, Q. However, *unlike* ESPRESSO-II, our initial specification does not in general represent a solution: though Q is a cover, it is not necessarily hazard-free. Therefore, processing begins by expanding each required cube into the *uniquely defined* minimum dhf-implicant containing it. This expansion represents a *canonicalization step*, transforming a potentially hazardous initial cover Q into a hazard-free initial cover Q^f.

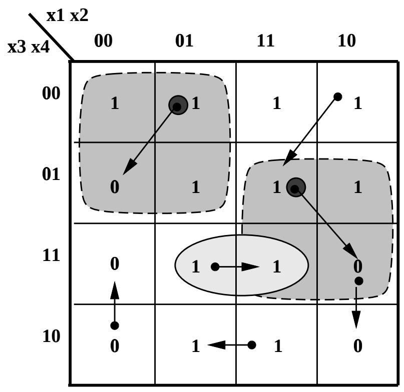

Fig. 3. Canonicalization Example

Example. Consider the function f in the Karnaugh map of Figure 3. A set T of specified multiple-input transitions is indicated by arrows. There are two $1 \rightarrow 0$ transitions, each corresponding to a privileged cube: $p1 = \overline{x}_1\overline{x}_3$ (start point $p1_{start} = \overline{x}_1 x_2 \overline{x}_3 \overline{x}_4$) and $p2 = x_1 x_4$ (start point $p2_{start} = x_1 x_2 \overline{x}_3 x_4$). The initial cover is given by the set Q of required cubes: $\{\overline{x}_1\overline{x}_3\overline{x}_4,\ \overline{x}_1 x_2 \overline{x}_3,\ x_1\overline{x}_3,\ x_1\overline{x}_3 x_4,\ x_1 x_2 x_4,\ x_2 x_3 x_4,\ x_2 x_3 \overline{x}_4\}$. This cover is hazardous. In particular, consider the required cube $r = x_2 x_3 x_4$, corresponding to the $1 \rightarrow 1$ transition from $x_1 x_2 x_3 x_4 = 0111$ to 1111. Required cube r illegally intersects privileged cube $p2$, since it intersects $p2$ but does not contain $p2_{start}$. To avoid illegal intersection, r must be expanded to the smallest cube which also contains $p2_{start}$: $r^{(1)} = \text{supercube}(\{r, p2_{start}\})$. However, this new cube $r^{(1)} = x_2 x_4$ now illegally intersects privileged cube $p1$, since it does not contain $p1_{start}$. Therefore, cube $r^{(1)}$ in turn must be expanded to the smallest cube containing $p1_{start}$: $r^{(2)} = \text{supercube}(\{r^{(1)}, p1_{start}\})$. The resulting expanded cube, $r^{(2)} = x_2$, has no illegal intersections and is therefore a dhf-implicant. □

In this example, $r^{(2)}$ is a hazard-free expansion of r, called a **canonical required cube**; it can therefore replace r in the initial cover. (Note that such a canonicalization is feasible if and only if the hazard-free covering problem has a solution; see Section III-J.)

It is easy to see that each required cube has a *unique* corresponding canonical required cube. Suppose there were two distinct minimal dhf-implicants, q_1 and q_2, which contain some required cube r. In this case, we now show that we can construct a dhf-implicant which is smaller than either cube: the intersection of q_1 and q_2, $q_{12} = q_1 \cdot q_2$. Clearly, implicant q_{12} contains r. Furthermore, if q_{12} were not a dhf-implicant, then it would intersect some privileged cube p illegally, i.e. intersect p but not contain its start point p_{start}. However, this would mean that both original implicants, q_1 and q_2, intersected p, but at least one of them (say q_1) did not contain p_{start}. As a result, q_1 would not be a dhf-implicant, since it would illegally intersect p, thus contradicting our assumption. Therefore, q_{12} is a dhf-implicant which contains r, and so q_1 and q_2

```
$supercube_{dhf}$ (set of cubes  $C = \{c_1, \ldots, c_n\}$)
    $r = supercube(\{c_1, \ldots, c_n\})$
    while ($r$ intersects some privileged cube $p_i$ illegally)
        $r = supercube(\{r, s_i\})$ where $s_i$ is the start point of $p_i$
    if $r$ intersects the OFF-set then return "undefined" else return $r$
```

Fig. 4. *Supercube*$_{dhf}$ computation

could not have been minimal. In sum, each required cube has a unique corresponding canonical required cube, which contains it.

Based on the above discussion, an initial set Q of required cubes is replaced by the corresponding set Q^f of canonical required cubes. This set is then minimized with respect to single-cube containment. Q^f is a valid hazard-free cover of the function to be minimized, and is used as an initial cover for the minimization process. Interestingly, Q^f has a second role as well: it is used to simplify the covering problem. In particular, Q^f defines a new covering problem: each cube of Q^f (*not* Q) must be contained in some dhf-implicant. It is straightforward to show that the two covering problems are equivalent, i.e. a dhf-implicant p contains a required cube r in Q *if and only if* p also contains the corresponding canonical required cube of r in Q^f. To see this, suppose that p contained r but did not contain the canonical required cube of r. In this case, p could not be a dhf-implicant, since it must illegally intersect at least one of those privileged cubes that caused r to be expanded into its canonical required cube.

In the above example, any dhf-implicant which contains required cube $r = x_2x_3x_4$ must also contain canonical required cube $r^{(2)} = x_2$. Therefore, the hazard-free minimization problem is unchanged, but canonical required cubes are used. An advantage of using Q^f is that it may have smaller size than Q, i.e. being a more efficient representation of the problem. Also, since the cubes in Q^f are in general larger than the corresponding ones in Q, the EXPAND operation may be sped up.

To conclude, the new set of canonical required cubes Q^f replaces the original set of required cubes Q as both (i) the initial cover, and (ii) the set of objects to be covered. Henceforth, the term "set of required cubes" will be used to refer to set Q^f.

We formalize the notion of canonicalization below.

Definition III.1: The **dhf-supercube** of a set of cubes C with respect to function f and transitions T, indicated as $supercube^{(f,T)}_{dhf}(C)$, is the smallest dhf-implicant containing the cubes of C.

The superscript (f, T) is omitted when it is clear from the context. $supercube_{dhf}(C)$ is computed by the simple algorithm shown in Figure 4.

The *canonical required cube* of a required cube r can now be defined as the *dhf-supercube* of the set $C = \{r\}$. The computation of dhf-supercubes for larger sets will be needed to implement some of the operators presented in the sequel.

```
Expand_cube(cube $a$, req-set $Q^f$, priv-set $P$, cover-set $F$, OFF-set $R$)
    $F_a = F - a$
    $Q_a = Q^f$
    $P_a = P$
    $R_a = R$
    $free\_entries$ =complement_pos_cube_notation($a$)
    while ($F_a \neq \oslash$)
        update($a, free\_entries, F_a, Q_a, P_a, R_a$)
        $F_a = \{c \in F_a | supercube_{dhf}(\{a, c\})$ is defined $\}$
        Let $c_b$ be the best candidate in $F_a$
        $a = supercube_{dhf}(\{a, c_b\})$
        $F_a = single\text{-}cube\text{-}contain\{F_a \cup \{a\}\} - \{a\}$
    while ($Q_a \neq \oslash$)
        update($a, free\_entries, F_a, Q_a, P_a, R_a$)
        $Q_a = \{q \in Q_a | supercube_{dhf}(\{a, q\})$ is defined $\}$
        Let $q_b$ be the best candidate in $Q_a$
        $a = supercube_{dhf}(\{a, q_b\})$
        $Q_a = single\text{-}cube\text{-}contain\{Q_a \cup \{a\}\} - \{a\}$
```

Fig. 5. Expand (for a cube a)

C. Expand

In Espresso-II, the goal of EXPAND is to enlarge each implicant of the current cover in turn into a prime implicant. As an implicant is expanded, it may contain other implicants of the cover which can be removed, hence the cover cardinality is reduced. If the current implicant cannot be expanded to contain another implicant completely, then, as a secondary goal, the implicant is expanded to overlap as many other implicants of the current cover as possible.

In Espresso-HF, the primary goal is similar: to expand a dhf-implicant of the current cover to contain as many other dhf-implicants of the cover as possible. However, EXPAND in Espresso-HF has two major differences. First, unlike Espresso-II, expansion in some literal (*i.e.,* "raising of entries") may *imply* that other expansions be performed. That is, raising of entries is now a *binate problem,* not a unate problem. In addition, Espresso-HF's EXPAND uses a different strategy for its secondary goal. By the Hazard-Free Covering Theorem, each required cube needs to be contained in some cube of the cover. Therefore, as a secondary goal, an implicant is expanded to contain as many required cubes as possible.

We now describe the implementation of EXPAND in Espresso-HF. Pseudocode for the expansion of a single cube is shown in Figure 5.

C.1 Determination of Essential Parts and Update of Local Sets

As in Espresso-II, *free entries* are maintained, to accelerate the expansion [30]. Free entries indicate which literals of an implicant are candidates for removal, during the expansion process.

To explain this concept in a unified way, the current implicant and its free entries are represented in positional cube notation [30]. As an example, $x_1\overline{x_2}x_4$ is represented in positional cube notation as 01 10 11 01, i.e. where literal $x_i(\overline{x_i})$ is encoded as 01 (10). Thus, each literal in $x_1\overline{x_2}x_4$ has a corresponding 0 in the positional cube notation, and changing, or *raising*, the 0 to 1 corresponds to removing this literal from the implicant.

Initially, a free entry is assigned a 1 (0) if the current implicant to be expanded, a, has a 0 (1) in the corresponding position. For the above example, the free entries are: 10 01 00 10. An *overexpanded cube* is defined as the cube a where all its free entries have been raised simultaneously.

As in [30], an *essential part* is one which can never, or always, be raised. However, our definition of essential parts is different from ESPRESSO-II, since a hazard-free cover must be maintained. We determine essential parts in procedure *update*, described below.

First, we determine which entries can *never be raised* and remove them from *free_entries*. This is achieved by searching for any cube in the OFF-set R that has distance 1 from a, in this entry, using the same approach as in ESPRESSO-II.

Next, we determine which parts can *always be raised*, raise them and remove them from *free_entries*. This step differs from ESPRESSO-II. In ESPRESSO-II, a part can always be raised if it is 0 in all cubes of the OFF-set, R. That is, it is guaranteed that the expanded cube will never intersect the OFF-set. In contrast, in ESPRESSO-HF, we must ensure that an implicant is also *hazard-free*: it cannot intersect the OFF-set, nor can it illegally intersect a privileged cube. Unlike ESPRESSO-II, this is achieved by searching for any column that (i) has only 0s in R *and* (ii) where, for each privileged cube p in P having a 1 in this a column, the corresponding start point p_{start} will be contained by the expanded cube a.

Example. Figure 1(a) indicates the set of required cubes, which forms an initial hazard-free cover. Consider the cube $x_2x_3x_4$ (11010101, in positional cube notation). As in ESPRESSO-II, the 0-entries for literals $\overline{x}_2$ and $\overline{x}_4$ can never be raised, since the cube would intersect the OFF-set. However, after updating the free entries, ESPRESSO-II indicates that literal $\overline{x}_3$ can *always* be raised, since the resulting cube will never intersect the OFF-set. In contrast, in ESPRESSO-HF, raising $\overline{x}_3$ results in an illegal intersection with privileged cube $\overline{x}_1\overline{x}_3$, so it cannot "always be raised". □

Since hazard-free minimization is somewhat more constrained, the expansion of a cube a can be accelerated by the following new operations on 2 local sets: P_a, R_a. These sets are associated with cube a, and P_a (R_a) is initially assigned the set of privileged (OFF-set) cubes. Both sets are *updated* as expansion proceeds (in procedure *update*). (1) Remove privileged cubes from P_a where the corresponding start point is already covered by a (since no further checking for illegal intersection is required). (2) Move privileged cubes from set P_a to the local OFF-set R_a if the overexpanded cube does not include the corresponding start points (since a can never be expanded to include these start points, therefore one must avoid intersection with these privileged cubes entirely). (3) Move privileged cubes from P_a to the local OFF-set R_a where $supercube_{dhf}(\{$a, start point$\})$ intersects the OFF-set (a can never be expanded to include these start points, therefore one must avoid intersection with the cubes entirely).

C.2 Detection of Feasibly Covered Cubes of F

In ESPRESSO-II, a cube in F is expanded through a *supercube* operation. A cube d in F is said to be *feasibly covered* by a if supercube($\{a,d\}$) is an implicant. In ESPRESSO-HF, this definition needs to be modified to insure *hazard-free covering*, after expansion of cube a.

Definition III.2: A cube d in F is **dhf-feasibly covered** by a if $supercube_{dhf}(\{a,d\})$ is defined.
This definition insures that the resulting expanded cube, $supercube_{dhf}(\{a,d\})$, is (i) an implicant (does not intersect OFF-set), and (ii) is also a dhf-implicant (does not intersect any privileged cube illegally). Effectively, this definition canonicalizes the resulting supercube to produce a dhf-implicant. That is, $supercube_{dhf}(\{a,d\})$ may properly contain supercube($\{a,d\}$), since the former may be expanded through a series of implications in order to reach the minimum dhf-implicant which contains both a and d. Using this definition, the following is an algorithm to find dhf-feasibly covered cubes of F.

While there are cubes in F that are dhf-feasibly covered by cube a, iterate the following:
Replace a by $supercube_{dhf}(\{a, d\})$, where d is a dhf-feasibly covered cube such that the resulting cube will cover as many cubes of the cover as possible. Covered cubes are then removed, using the "single-cube-containment" operator, thus reducing the cover cardinality. Determine essential parts and update local sets (see above).

C.3 Detection of Feasibly Covered Cubes of Q^f

Once cube a can no longer be feasibly expanded to cover any other cube, d, of F, we still continue to expand it. This is motivated by the Hazard-Free Covering Theorem, which states that each required cube needs to be contained in some cube of the cover. Therefore, as a secondary goal, cube a is expanded to contain as many required cubes as possible. The strategy used in this sub-step is similar to the one used in the preceding one, i.e. while there are cubes in Q^f that are dhf-feasibly covered by cube a, iterate the following:
Replace a by $supercube_{dhf}(\{a, q\})$, where q is a dhf-feasibly covered *required cube* such that the resulting cube will cover as many required cubes not already contained in a as possible. Covered required cubes are then removed, using the "single-cube-containment" operator. Determine essential parts and update local sets (see above).

C.4 Constraints on Hazard-Free Expansion

In ESPRESSO-II, an implicant is expanded until no further expansion is possible, i.e. until the implicant is prime. Two steps are used: (i) expansion to overlap a maximum number of cubes still covered by the overexpanded cube; and (ii) raising of entries to find the largest prime implicant covering the cube.

In ESPRESSO-HF, however, we do not implement these remaining EXPAND steps, based on the following observation. The result of our EXPAND steps (cf. III-C.2 and

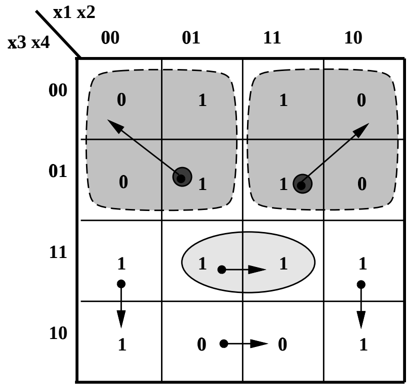

Fig. 6. Essential Example

III-C.3) guarantees that a dhf-implicant can never be further expanded to contain additional required cubes. Therefore, by the Hazard-Free Covering Theorem, no additional objects (required cubes) can possibly be covered through further expansion. In contrast, in ESPRESSO-II, expansion steps (i) and (ii) may result in covering of additional ON-set minterms. Because of this distinction, the benefit of further expansion is mitigated. Therefore, in general, our EXPAND algorithm makes no attempt to transform dhf-implicants into dhf-prime implicants. However, since expansion to dhf-primes is important for literal reduction and testability, it is included as a final post-processing step: MAKE_DHF_PRIME (see Figure 2).

D. Essentials

Essential prime implicants are prime implicants that need to be included in any cover of prime implicants. Therefore, it is desirable to identify them as soon as possible to make the resulting problem size smaller. On the one hand, we know of no efficient solution for identifying the essential dhf-primes using the unate recursion paradigm of ESPRESSO-II. On the other hand, the hazard-free minimization problem is highly constrained by the notion of covering of *required cubes*, allowing a powerful new method to classify essentials as equivalence classes.

Example. Consider Figure 6. The required cube, $r = x_2x_3x_4$, is covered by precisely two dhf-prime implicants: $p1 = x_2x_4$ and $p2 = x_3x_4$, which cover no other required cubes. Neither $p1$ nor $p2$ is an essential dhf-prime, since r is covered by both. And yet, clearly, either $p1$ or $p2$ (not both) *must* be included in any cover of dhf-primes. Also, if we assume the standard cost function of cover cardinality, $p1$ and $p2$ are of equal cost. □

Definition III.3: Two dhf-prime implicants are **equivalent** if they cover the same set of required cubes. An equivalence class of dhf-prime implicants is **maximal** if it covers a set of required cubes which is not covered by any other equivalence class. A maximal equivalence class of dhf-prime implicants is **essential** if they cover at least one required cube which is not covered by any other maximal equivalence class.

In the above example, the set $\{x_2x_4, x_3x_4\}$ of dhf-primes is an *equivalence class*, since both dhf-primes cover the same set of required cubes $\{x_2x_3x_4\}$. In fact, the class is an *essential equivalence class*, since it is the only equivalence class that covers the required cube $x_2x_3x_4$.

ESPRESSO-II computes essentials after an initial EXPAND and IRREDUNDANT. In contrast, ESPRESSO-HF computes essentials as part of a modified EXPAND-step. The algorithm is outlined as follows:

The algorithm starts with the initial hazard-free cover, Q^f, of required cubes. To simplify the presentation, assume that one seed cube is selected and expanded greedily, using EXPAND, to a dhf-implicant p. This implicant is characterized by the set, Q^p, of required cubes which it contains. Dhf-implicant p thus corresponds to the equivalence class of dhf-primes that cover Q^p. Since EXPAND guarantees that p covers a maximal number of required cubes, this equivalence class is also maximal. Moreover, this class is an *essential equivalence class* if Q^p contains some required cube, q^f, which cannot be expanded into any other maximal equivalence class.

To check if q^f can be expanded into a different maximal equivalence class, a simple pairwise check is used: for each required cube s^f not covered by p, determine if $supercube_{dhf}(\{q^f, s^f\})$ is feasible. If no such feasible expansion exists for q^f, q^f is called a **distinguished required cube**, and therefore the equivalence class corresponding to p is essential. Otherwise, the process is repeated for every required cube q^f contained in Q^p. If p corresponds to an essential equivalence class, then p is removed from the cover. In addition, all required cubes covered by p are removed, since it is ensured that they will be covered. This step can result in "secondary essential" equivalence classes. In fact, due to the removal of required cubes, more dhf-prime implicants become equivalent to each other. As a consequence, further equivalence classes may become essential.

The procedure iterates until all essentials are identified.

The above discussion seems to imply that the essentials step is more or less quadratic in the number of required cubes, i.e. very inefficient. However, by making use of techniques similar to the ones described in the EXPAND section III-C, e.g. by using an overexpanded cube, the number of necessary $supercube_{dhf}$-calls can be reduced dramatically. Therefore, in practice, essentials can be identified efficiently and the problem size is usually significantly reduced (see Section VI).

E. Reduce

The goal of the REDUCE operator is to set up a cover that is likely to be made smaller by the following EXPAND step. To achieve this goal, each cube c in a cover F is maximally reduced in turn to a cube $\tilde{c}$, such that the resulting set of cubes, $\{F - c\} \cup \tilde{c}$ is still a cover.

ESPRESSO-II uses the unate recursive paradigm to maximally reduce each cube. Since ESPRESSO-HF is a required-cube covering algorithm, there is no obvious way to use this paradigm. Fortunately, the hazard-free problem is more constrained, making it possible to use an efficient enumerative approach based on required cubes.

Our REDUCE algorithm is as follows. The algorithm reduces each cube c in the cover in order. In particular, a cube c is reduced to the smallest dhf-implicant $\tilde{c}$ that covers all required cubes that are *uniquely covered* by c (i.e. contained in no other cube of the cover F). That is, if $r_1, \ldots, r_l$ is the set of required cubes that are uniquely covered by c, then c is replaced by $\tilde{c} = \mathit{supercube}_{dhf}(\{r_1, \ldots, r_l\})$.

Note that the outcome of this algorithm depends on the order in which the cubes c of the cover F are processed. Suppose c_i is reduced before c_j, and that c_i and c_j cover some required cube r but no other cube of F covers r. If c_i is reduced to a cube $\tilde{c_i}$ that does not cover r, then c_j cannot be reduced to a cube that does not cover r.

F. Irredundant

Espresso-II uses the unate recursive paradigm to find an irredundant cover. However, in our case, we cannot employ the same algorithm, since a "redundant cover" (according to covering of minterms) may in fact be irredundant with respect to covering of required cubes.

Therefore, as in REDUCE, our approach is required-cube based. Considering the Hazard-Free Covering Theorem, it is straightforward that IRREDUNDANT can be reduced to a covering problem of the cubes in Q^f by the cubes in F. That is, the problem reduces to a minimum-covering problem of (i) required cubes, using (ii) dhf-implicants in the current cover. In practice, the number of required cubes and cover cubes usually make the covering problem manageable. Espresso-II's Mincov can be used to solve this covering problem exactly, or heuristically (using its heuristic option).

G. Last Gasp

The inner loop of Espresso-HF may lead to a suboptimal local minimum. The goal of LAST_GASP is to use a different approach to attempt to reduce the cover size. In Espresso-II, each cube $c \in F$ is independently reduced to the smallest cube containing all minterms not covered by any other cube of F. In contrast, Espresso-HF computes, for each $c \in F$, the smallest dhf-implicant containing all *required cubes* that are not covered by any other cube in F.

As in Espresso-II, cubes that can actually be reduced by this process are added to an initially empty set G. Each such $g \in G$ is then expanded in turn with the goal to cover at least one other cube of G, using the $\mathit{supercube}_{dhf}$ operator, and, if achieved, the expanded cube is added to F. Finally, the IRREDUNDANT operator is applied to F with the hope to escape the above-mentioned local minimum.

H. Make-dhf-prime

The cover being constructed so far does not necessarily consist of dhf-primes. It is usually desirable to expand each dhf-implicant of the cover to make it dhf-prime as a last step. This can be achieved by a modified EXPAND step. A simple greedy algorithm will expand an implicant c to a dhf-prime: While dhf-feasible, raise a single entry of c.

I. Pre- and postprocessing steps

Espresso-HF includes optional pre- and postprocessing steps. In particular, the efficiency of Espresso-HF depends very much on the size of the ON-set and OFF-set covers that are given to it. Thus, Espresso-HF includes an optional *preprocessing* step which uses Espresso-II to find covers of smaller size for the initial ON-set and OFF-set [5]. Espresso-HF also includes a *postprocessing* step to reduce the literal count of a cover, similar to Espresso-II's MAKE_SPARSE.

J. Existence of a hazard-free solution

As indicated earlier, for certain Boolean functions and sets of transitions, no hazard-free cover exists [29]. The currently used exact hazard-free minimization method Hfmin is only able to decide if a hazard-free solution exists after generating all dhf-prime implicants. A hazard-free solution does not exist if and only if the dhf-prime implicant table includes at least one required cube not covered by any dhf-prime implicant. However, since the generation of all dhf-primes may very well be infeasible [6] for even medium-sized examples, it is important to find an alternative approach.

We now introduce a new theorem to check for the existence of a hazard-free solution, without the need to generate all dhf-prime implicants. This theorem leads directly to a fast and simple algorithm that is incorporated into Espresso-HF.

Theorem III.4: Given a function f and a set, T, of specified function-hazard-free input transitions of f, a solution of the two-level hazard-free logic minimization problem exists if and only if $\mathit{supercube}_{dhf}(q)$ is defined for each required cube q.

The proof is immediate from the discussion in Section III-B.

Example. Consider the Boolean function in Figure 7, with four specified input transitions. To check for existence of a hazard-free solution, we compute $\mathit{supercube}_{dhf}(q)$ for each required cube q. Except for $x_1x_2x_4$, it holds that $q = \mathit{supercube}_{dhf}(q)$ since no privileged cube is intersected illegally. To compute $\mathit{supercube}_{dhf}(x_1x_2x_4)$, note that privileged cube x_3 is intersected illegally, i.e. $\mathit{supercube}_{dhf}(x_1x_2x_4) = \mathit{supercube}_{dhf}(x_2x_4)$. Since x_2x_4 now intersects privileged cube $\overline{x}_1\overline{x}_3$, we get $\mathit{supercube}_{dhf}(x_1x_2x_4) = \mathit{supercube}_{dhf}(x_2)$ leading directly to the fact that $\mathit{supercube}_{dhf}(x_1x_2x_4)$ does not exist because x_2 intersects the OFF-set. Thus, a hazard-free cover does not exist for this example. □

[5]ON-set and OFF-set are necessary to form the initial set of required cubes, Q. More importantly, the OFF-set is used to check if a cube expansion is valid, see Figure 4.

[6]This refers to "explicit representations"; we will show later that "implicit representations" very often are feasible.

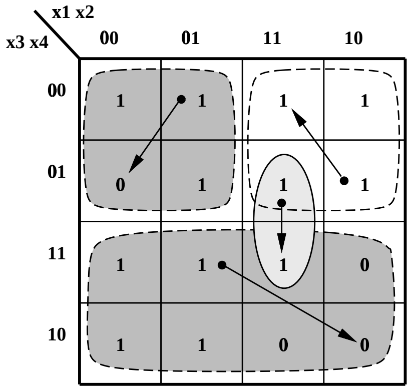

Fig. 7. Existence Example

IV. A Novel Approach to Incorporating Hazard-Freedom Constraints Within a Synchronous Function

After having discussed the *heuristic hazard-free minimization problem* in the previous section, we will now shift our discussion to the *exact hazard-free minimization problem*.

We begin by presenting, in this section, 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 V.

A. Overview and Intuition

We first give a simple overview of our entire method. Details and formal definitions are provided in the remaining sections. 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. (The exact definition of g is given in IV-B.) An overview of the method is best illustrated by a simple example.

Example. Consider Figure 8. 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 II-F). 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). □

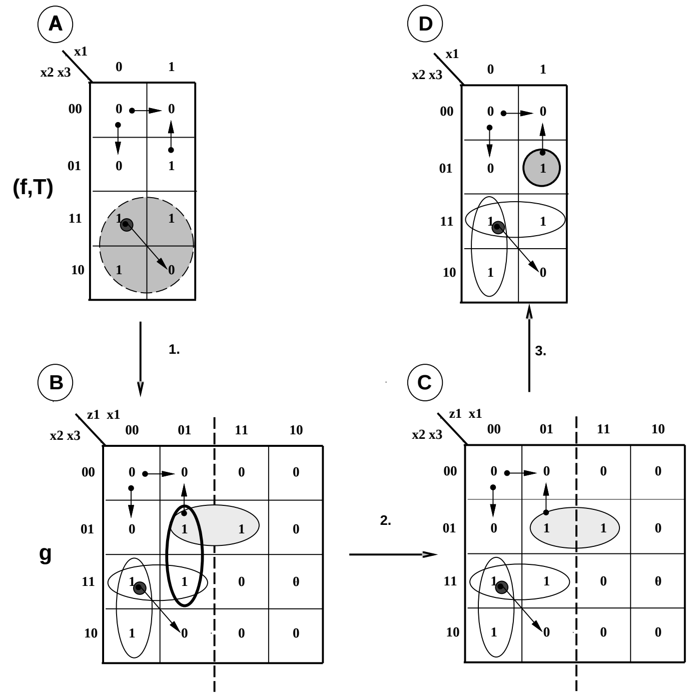

Fig. 8. Example for recasting prime generation. A) shows the function (f, T) whose dhf-primes are to be computed. B) shows the auxiliary synchronous function g and its primes. C) shows primes of g that do not intersect illegally. D) shows the final dhf-primes of f, after deleting the z_1 variable.

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 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 unification can be achieved by adding auxiliary variables, i.e. by lifting the problem into a higher-dimensional Boolean space.

In summary, the big picture is as follows. The definition of g ensures that all dhf-prime implicants of f (*dhf-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.

B. The auxiliary synchronous function g

We now 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 $\{x_1, \ldots, x_n\}$, and that the set of transitions T gives rise to the set of non-trivial privileged cubes $PRIV(f, T) = \{p_1, \ldots, p_l\}$. The idea is to define a function g over

$\{x_1, \ldots, x_n, z_1, \ldots z_l\}$; that is, *one new variable is added per privileged cube.* Formally, g is defined as follows:

$$g(x_1, \ldots, x_n, z_1, \ldots, z_l) = f \cdot \prod_{1 \leq i \leq l} (\overline{z_i} + \overline{p_i})$$

Function g is the product of f and some function which depends on the new 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. As an example, Figure 8A 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 entire cube $z_1 x_2$, which corresponds to the privileged cube in the original function f. In particular, function g is defined as $g = f \cdot (\overline{z_1} + \overline{p_1})$, where $p_1 = x_2$. □

C. 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, since function g is identically defined as f in both planes "outside" the privileged cube. Expansion cannot occur if q intersects the privileged cube p_1, because in the $z_1 = 1$ plane, the privileged cube is filled in entirely with 0's.

Example. Consider the minterm $q_1 = \overline{z_1} x_1 \overline{x_2} x_3$ of g in Figure 8B, which corresponds to the minterm $x_1 \overline{x_2} x_3$ of f. q can be expanded into the $z_1 = 1$ plane into the prime implicant of g: $x_1 \overline{x_2} x_3$ (shaded oval). Intuitively, the expansion is possible since q_1 does not intersect the privileged cube, i.e. the cube $\overline{z_1} x_2$, which corresponds to the privileged cube x_2 of the original function f. However, the implicant $q_2 = \overline{z_1} x_1 x_3$ (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. □

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* define which privileged cubes are intersected by the corresponding implicant in f.

Let q be any prime implicant of g:

$$q = x_{i_1} \cdots x_{i_{\hat{n}}} z_{j_1} \cdots z_{j_{\hat{l}}}$$

Here, x_{i_k} is a positive or negative x-literal [7]. However, z_{j_k} can *only* be a negative z-literal. The reason is that g is a negative unate function in z-variables (by the definition of g), and therefore prime implicants of g will never include positive z-literals. We indicate by q^x the *restriction of q to the x-literals*, i.e. $q^x = x_{i_1} \cdots x_{i_{\hat{n}}}$. Note that q^x is an implicant of f by the definition of g.

We show that the presence, or absence, of $\overline{z_i}$ literals in prime implicant q, indicates which privileged cubes are intersected by q^x. If q includes literal $\overline{z_i}$, then q^x intersects privileged cube p_i in function f. To see this, note that since q is prime, clearly q cannot be expanded into the $z_i = 1$ plane. As a result, as explained above, q^x must intersect privileged cube p_i in the original function f. On the other hand, if q does not include $\overline{z_i}$, then q^x does not intersect p_i. Intuitively, the primes, $Prime(g)$, are maximal in two senses: they are maximally expanded in f, or maximally non-intersecting of privileged cubes, in some combination, which is explicitly indicated by the set of support of the primes.

In sum, the key observation is that the set of support of a prime implicant q of g *immediately* 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, *dhf-Prime*(f, T).

D. Transforming Prime(g) into dhf-Prime(f,T)

Once $Prime(g)$ is computed, *dhf-Prime*(f, T) can be directly computed. The key insight for this computation is that the prime implicants of $Prime(g)$ 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.

Prime implicants q that fall into Classes 2 and 3 (i.e. q^x intersects some privileged cube) can be immediately identified by the observation of the previous subsection. Those

[7]Note that q may not depend on all of the x-variables.

which fall into Class 3 can then be identified, and removed, using a simple containment check (i.e. determine if q^x contains the start point of each intersected privileged cube).

The set $dhf\text{-}Prime(f,T)$ can therefore be computed as follows. Start with $Prime(g)$. 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 of its elements is a valid dhf-implicant of (f,T) if restricted to the x-variables. The reason is that, first, all primes of g are implicants of f if restricted to x-variables, and second, the filtering removed any element that intersected any privileged cube illegally. Therefore, the set only includes dhf-implicants. In fact, it also contains *all* dhf-prime-implicants of (f,T). This will be proved in the next subsection.

Example. Figure 8B shows function g and its prime implicants, $Prime(g) = \{x_1\overline{x_2}x_3, \overline{z_1}x_1x_3, \overline{z_1}x_2x_3, \overline{z}_1\overline{x}_1x_2\}$. Part C shows the result of filtering out primes that illegally intersect regions corresponding to privileged cubes in f. In this case, $\overline{z_1}x_1x_3$ (oval with thick dark border) falls into Class 3 with respect to p_1: it is deleted since it has a $\overline{z_1}$-literal, i.e. intersects the region corresponding to privileged cube p_1 and, in addition, does not contain the start point $\overline{z}_1\overline{x}_1x_2x_3$. However, $x_1\overline{x_2}x_3$ (shaded oval) falls into Class 1: it is not deleted since it does not have a $\overline{z_1}$-literal and therefore does not intersect the region corresponding to the privileged cube p_1. The remaining two primes $\overline{z_1}x_2x_3$ and $\overline{z}_1\overline{x}_1x_2$ fall into Class 2: they intersect the region corresponding to p_1 and also contain the start point. Part D shows the result of step 3 which deletes the z-literals in each cube. We obtain $\{x_1\overline{x_2}x_3, x_2x_3, \overline{x_1}x_2\}$, which is $dhf\text{-}Prime(f,T)$. Note that the introduction of the z_1-variable ensures that the dhf-implicant of f, $x_1\overline{x_2}x_3$, which is not a prime implicant of f, since it is contained by the prime implicant, x_1x_3, is nevertheless generated. □

E. Formal characterization of dhf-Prime(f,T) in terms of function g

In this subsection, based on above discussion, we present the main result of this section: a new formal characterization of $dhf\text{-}Prime(f,T)$. We use the following notations. g_{z_i} and $g_{\overline{z_i}}$ 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, $RemZ\left(\{x_1x_2z_1, x_1x_3\overline{z_2}, x_1x_3z_1z_3\}\right) = \{x_1x_2, x_1x_3\}$.[8] The SCC-operator on a set of cubes (single-cube-containment) removes those cubes contained in other cubes.

Theorem IV.1: Given (f,T). Let $PRIV(f,T) = \{p_1,\ldots,p_l\}$ be the set of non-trivial privileged cubes [9], and $START(f,T) = \{s_1,\ldots,s_l\}$ be the set of corresponding start points. Define

$$g(x_1,\ldots,x_n,z_1,\ldots,z_l) = f\cdot\prod_{1\le i\le l}(\overline{z_i}+\overline{p_i})$$

Then the set dhf-Prime(f,T) can be expressed as follows:

$$SCC\left(\bigcap_{1\le i\le l}\left[RemZ\left(Prime\left(g_{z_i}\right)\right)\cup\left\{q\in RemZ\left(Prime\left(g_{\overline{z_i}}\right)\right)|q\supseteq s_i\right\}\right]\right)\quad(1)$$

Intuition: $RemZ\left(Prime\left(g_{z_i}\right)\right)$ includes implicants of f that do not intersect the privileged cube p_i. $\{q\in RemZ\left(Prime\left(g_{\overline{z_i}}\right)\right)|q\supseteq s_i\}$ 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−Prime(f,T) is also contained in (1)):
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 $p_1,\ldots,p_{\hat{l}}$, and q does not intersect $p_{\hat{l}+1},\ldots,p_l$ - i.e. q is an implicant of $\overline{p_{\hat{l}+1}},\ldots,\overline{p_l}$ -, then $q\overline{z_1}\cdots\overline{z_{\hat{l}}}$ is an implicant of g.

$q\overline{z_1}\cdots\overline{z_{\hat{l}}}$ is a prime implicant of g because:

(i) Removing (any) $\overline{z_i}$ results in a cube which is not an implicant of $\overline{z_i}+\overline{p_i}$, 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} is not a dhf-prime implicant. Thus q_{new} either intersects the OFF-set of f, or intersects for some i privileged cube p_i, $i\in\{\hat{l}+1,\ldots,l\}$ and is therefore no longer an implicant of $\overline{z_i}+\overline{p_i}$. In either case q_{new} is not an implicant of g.

Thus, for each i, q is by construction in at least one of $RemZ\left(Prime\left(g_{z_i}\right)\right)$ or $\{q\in RemZ\left(Prime\left(g_{\overline{z_i}}\right)\right)|q\supseteq s_i\}$). 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 (1) are dhf-implicants. Thus, q is contained in (1).

"$\supseteq$" (any product contained in (1) is also contained in dhf-Prime(f)):
Let $q\notin dhf\text{-}Prime(f,T)$. We show that q is not contained in (1).

Case (i): q is a dhf-implicant that is strictly contained in some dhf-prime implicant. Then q is filtered out because of the SCC-operator and therefore not contained in (1).

Case (ii): q is not a dhf-implicant. By construction all cubes contained in (1) are dhf-implicants: the intersection ensures that each cube is valid with respect to all privileged cubes, i.e. the cube either does not intersect or contains the startpoint. Thus, q cannot be contained in (1). ■

[8] $RemZ$ can formally be expressed by existential quantification over z-variables, i.e. $RemZ(P) = \{x\in\{x_1,\overline{x}_1,x_2,\overline{x}_2,\ldots,x_n,\overline{x}_n\}^*\ |\exists z\in\{z_1,\overline{z}_1,z_2,\overline{z}_2,\ldots,z_l,\overline{z}_l\}^* : xz\in P\}$.

[9] In the theorem, $\overline{p_i}$ denotes the complement function of p_i. Example: $p_1 = x_1x_2\overline{x}_4$. Then, $\overline{p}_1 = \overline{x_1x_2\overline{x}_4} = \overline{x}_1+\overline{x}_2+x_4$.

F. Multi-output Case

For simplicity of presentation only, it was assumed that f is a single-output function. However, it is well-known [34] 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.

V. Exact hazard-free minimization: Impymin

Based on the ideas of the previous section, we are now able to present a new exact implicit minimization algorithm for multi-output 2-level hazard-free logic.

A. Implicit 2-Level Logic Minimization: Scherzo

We first briefly review the state-of-the-art *synchronous* exact two-level logic minimization algorithm, called Scherzo [8], which forms a basis of our new hazard-free implicit minimization method.

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

- Scherzo uses data structures like BDDs [3] and ZBDDs [21] to represent Boolean functions and sets of products very efficiently. Thus, the complexity of the minimization problem is shifted, and the cost of the cyclic core computation [10] is now 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 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.

Algorithm:	Scherzo
Input:	Boolean function f.
Output:	All minimum 2-level implementations 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.)

[10]A 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.

B. Implicit 2-Level Hazard-fee Logic Minimization: Impymin

Nowick/Dill reduced 2-level hazard-free logic minimization to a unate covering problem (see Section II) where each required cube must 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 from synchronous two-level logic minimization is the setting up of the covering problem. In particular, a method is needed that computes the set dhf-Prime(f,T) efficiently, preferably in an implicit manner. To do so, we use the new characterization of dhf-Prime(f,T) of Section IV. Our algorithm is as follows.

Algorithm:	Impymin
Input:	Boolean function f, set of input transitions T.
Output:	All minimum hazard-free 2-level implementations of (f, T).

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

We now explain each of the steps in detail.

B.1 Computation of the ZBDD of 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 a BDD for the auxiliary synchronous function g, and then the ZBDD of $Prime(g)$ using an existing recursive algorithm [8]. From the ZBDD of $Prime(g)$, we can then compute the final ZBDD of $dhf\text{-}Prime(f, T)$ using Theorem IV.1. It remains to show that the necessary operations, $Prime(g_{z_i}), Prime(g_{\overline{z_i}}), RemZ$, and SCC, for these steps, can be implemented efficiently on ZBDDs:

- *Computing $Prime(g_{z_i})$*: 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 $Prime(g)$. We replace each node labeled with a z_i variable by the result of 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. The resulting ZBDD may actually include non-primes, i.e. cubes contained in other cubes. However, these cubes are filtered out by SCC (see below).
- *Computing the ZBDD of $Prime(g_{\overline{z_i}})$*: Analogously.
- *Computing the ZBDD of $RemZ$*: $RemZ$ deletes all z-literals in the ZBDD. We traverse the ZBDD, and at each z_i- or $\overline{z_i}$-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 performed in this step, since it is automatically handled in Step 3 of the algorithm.

To summarize, based on Theorem IV.1 we can compute the covering objects, $dhf\text{-}Prime(f,T)$, in an implicit manner.

B.2 Computation of the ZBDD of REQ(f,T)

From the set of input transitions, T, the set of required cubes can easily be computed (see [29]). This set can then be stored as a ZBDD.

B.3 Solving the Implicit Covering Problem

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

C. A Note on the Efficiency of Impymin

Impymin appends z-variables in dhf-prime generation during the construction of the synchronous function g. It is worth pointing out that the algorithm does not become unattractive even in cases where many z-variables are necessary. Such cases typically arise when there are many dynamic transitions, and hence many privileged cubes. In practice, the addition of many z-variables does not necessarily imply that the BDD for g will be much larger than the BDD for f (see Subsection VI-D).

Experimental results also indicate that Impymin has significantly better runtime than existing asynchronous methods on large examples. It also performs hazard-free logic minimization nearly as efficiently as synchronous logic minimization for many examples. One reason is that the new characterization of the set of dhf-prime implicants, presented in Section IV, makes it possible to use state-of-the-art synchronous techniques for implicit prime generation and implicit set covering solving (see Subsection VI-D for a detailed discussion).

VI. Experimental Results and Comparison with Related Work

Prototype versions of our two new minimizers Espresso-HF[11] and Impymin were run on several well-known benchmark circuits [12], [37] on an ULTRA-SPARC 140 workstation (Memory: 89 MB real/ 230 MB virtual).

A. Comparison of exact minimizers: Impymin *vs.* Hfmin

The table in Figure 9 compares our new exact minimizer Impymin with the currently fastest available exact minimizer, Hfmin, by Fuhrer et al. [12].

For smaller problems, Hfmin is faster. It should be noted, though, that our implementation is not yet optimized [12]. However, the bottleneck of Hfmin becomes clearly visible already for medium-sized examples. For *sd-control* and *stetson-p2*, Impymin is more than three times faster; for the benchmark *pscsi-pscsi* more than fifteen times.

			Hfmin **[FLN]**	Impymin
name	*i/o*	*#c*	*time(s)*	*time(s)*
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

Fig. 9. Comparison of exact hazard-free minimizers (#c - number of cubes in minimum solution, time - run-time in seconds)

For very large examples, Impymin outperforms Hfmin by a large factor. While Hfmin cannot solve *stetson-p1* within 20 hours, Impymin 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.

B. Comparison of our new methods: Impymin *vs.* Espresso-HF

Figure 10 compares our two new minimizers Espresso-HF and Impymin. Besides run-time and size of solution, the table also reports the number of essentials (for Espresso-HF) and the number of variables that need to be added (for Impymin).

The two minimizers are somewhat orthogonal.

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. In particular, Espresso-HF finds always a cover that is at most 3% larger than the minimum cover size. It is worth pointing out that many examples were very positively influenced by our new notion of essentials. Quite a few examples can be minimized by *just* the essentials step, resulting in a guaranteed minimum solution; e.g. *dram-ctrl* and *pe-send-ifc*.

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 im-

[11]Our implementation is not a simple modification of the Espresso-II code. We do not re-use any Espresso-II code. The reason is that while we use the same set of main operators - EXPAND, REDUCE, IRREDUNDANT - the algorithms that implement these operators, as explained in detail in Section 3, are actually very different from Espresso-II.

[12]Our BDD package is still very inefficient. In particular, it includes a static (i.e. not a dynamic) hashtable. The hashtable for small examples is unnecessarily large. In fact, the run-time is completely dominated by initializing the hashtables. If we use an appropriate-sized hashtable for smaller examples, experiments indicate that Impymin can solve the small examples as fast as Hfmin.

plicants, 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 these bottlenecks is very different. Whereas IMPYMIN uses implicit data structures (but still follows some of the same basic steps as HFMIN), ESPRESSO-HF follows a very different approach. Thus, the two methods are orthogonal in their approach to overcome these 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.

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 discussion in [8]). In particular, it is not easily possible to simply augment ESPRESSO-HF or HFMIN with BDDs to obtain a high-quality tool. Instead, a new theoretical formulation was needed on the characterization of dhf-prime implicants (cf. Section IV-E), on which the new exact implicit minimizer could be based.

C. Comparison with Rutten's Work

An interesting alternative approach to our new characterization of dhf-prime implicants (cf. Section IV-E) was recently proposed by Rutten et al. [33], [32], as part of an exact hazard-free minimization algorithm. His new algorithm to computing dhf-prime implicants is very different from ours. His approach follows a divide-and-conquer paradigm. In particular, the dhf-prime generation 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 [33] 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 then 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 ours is. Furthermore, while Rutten's work is promising, it has not been fully evaluated so far. In particular, he only presented run-times for functions that are *significantly smaller* than those that can be handled by our two new methods. To be precise, on the examples he reports, his own re-implementation of the existing HFMIN tool never takes more than a few seconds. Thus, Rutten evaluates his approach (and admittedly shows improvement) only on examples that can already easily be solved by existing algorithms. In contrast, as shown in the previous subsection, our new methods are more powerful, since they can solve examples efficiently that cannot be solved by HFMIN within several hours of run-time.

D. Comparison of synchronous vs. asynchronous minimization

We now compare our two new tools for 2-level hazard-free minimization, ESPRESSO-HF and IMPYMIN, with the two corresponding state-of-the-art tools for 2-level non-hazard-free minimization, ESPRESSO-II and SCHERZO. The table in Figure 10 compares both *run-time* and *cardinality* of solution for all four minimizers. In addition, the table indicates the number of identified *essentials* for the two heuristic minimizers, ESPRESSO-II and ESPRESSO-HF. Finally, for IMPYMIN, it reports the *number of added variables* and their impact on *BDD size*.

The run-time comparison indicates that, although our tools are not implemented as efficiently as their synchronous counterparts, they are comparably fast. Interestingly, our tools are actually faster than the synchronous tools for the two largest examples, *cache-ctrl* and *stetson-p1*. For our set of benchmarks, this seems to indicate that the *more constrained* asynchronous problem, which is to minimize a function f without hazards for a set of transitions T, may be easier than the corresponding synchronous problem, which is to minimize the same function f without any specified input transitions and without hazard-free constraints.

The comparison in terms of cardinality of solution indicates an increase in the asynchronous case compared with the synchronous case. In an earlier comparison [29], it was observed that the logic overhead for the asynchronous case was never greater than 6%. In contrast, in our table, there is a large variation in overhead, ranging from 0% (*stetson-p3*) to 60% (*ssci-trcv-bm*). The increase in overhead is due to the fact that we report on significantly more complex problems: while [29] only performed single-output minimization, we do multi-output minimization (on many of the same circuit examples), including for functions ranging up to 32 inputs and 33 outputs.

However, it is important to note that this table should not be used to draw general conclusions regarding how much logic overhead asynchronous designs incur due to the necessity to avoid hazards. Our benchmark functions have been generated by asynchronous synthesis methods, i.e. these functions do not really make much sense in a synchronous system. On the one hand, functions derived from asynchronous FSMs must have function-hazard-free input changes and critical race-free state changes, unlike those derived from synchronous FSMs. On the other hand, asynchronous FSMs are typically specified in a more controlled environment, with more don't cares. A truly fair comparison on this interesting point is much beyond the scope of paper.

Figure 10 also compares the number of identified essentials, using both hazard-free and non-hazard-free algorithms. ESPRESSO-HF's new formulation of essential equivalence classes typically allows many more essentials to be identified than in ESPRESSO-II. For example, in *cache-*

		Espresso-HF			Impymin				Espresso-II			Scherzo	
name	*i/o*	*#c*	*time*	*#e*	*#c*	*time*	*#v*	*BDDf/g*	*#c*	*time*	*#e*	*#c*	*time*
cache-ctrl	20/23	99	105	50	97	301	39	795/1813	89	217	7	80	756
dram-ctrl	9/8	22	1	22	22	13	6	91/140	19	1	6	18	1
pe-send-ifc	12/10	27	1	27	27	16	5	158/299	20	1	1	20	1
pscsi-ircv	8/7	12	1	12	12	10	3	45/110	10	1	4	10	1
pscsi-isend	11/10	23	1	23	23	15	6	115/264	16	1	1	16	1
pscsi-pscsi	16/11	78	11	55	77	105	23	319/852	66	5	10	63	14
pscsi-tsend	11/10	22	1	22	22	13	4	113/223	16	1	3	16	1
pscsi-tsend-bm	11/11	23	1	23	23	13	4	112/231	16	1	3	16	1
sd-control	18/22	35	3	23	34	52	0	448/448	25	2	4	24	14
sscsi-isend-bm	10/9	22	1	22	22	11	3	98/153	15	1	5	15	1
sscsi-trcv-bm	10/9	24	1	21	24	13	5	96/189	15	1	3	15	1
sscsi-tsend-bm	11/10	20	1	20	20	13	4	123/214	15	1	2	15	1
stetson-p1	32/33	60	21	34	60	813	9	1463/1933	45	33	1	?	> 72000
stetson-p2	18/22	37	2	26	37	49	0	457/457	25	1	6	25	17
stetson-p3	6/4	7	1	7	7	8	1	30/41	7	1	6	7	1

Fig. 10. Comparison of the heuristic hazard-free minimizer Espresso-HF, the exact hazard-free minimizer Impymin, the heuristic minimizer Espresso-II, and the exact minimizer Scherzo. (#c - number of cubes in solution, time - run-time in seconds, #e - number of essentials, #v - number of added variables, BDD f/g - BDD sizes without/with added variables)

ctrl, Espresso-HF identifies 50 essentials (out of an exact minimum cover of 97 cubes), while Espresso-II identifies only 7 essentials (out of an exact minimum cover of 80 cubes). Thus, Espresso-HF makes positive use of hazard-freedom constraints to obtain a very strong formulation of essentials, which has positive impact on both run-time and quality of solution.

Finally, the table indicates that adding (sometimes many) variables in Impymin does not lead to an explosion in terms of BDD size. To incorporate hazard-freedom constraints, Impymin (unlike Scherzo) transforms the BDD of f into the BDD of auxiliary function g. The table, which compares the corresponding BDD sizes for the same BDD package and variable ordering, indicates that adding variables for this transformation increases the BDD size even for large examples only by a small factor, which is typically about 2. Thus, the BDD size of auxiliary function g is not much larger than the BDD size of f.

VII. Conclusions

We have presented two new minimization methods for multi-output 2-level hazard-free logic minimization: Espresso-HF, a heuristic method based on Espresso-II, and Impymin, an exact method based on implicit data structures.

Both tools can solve all examples that we have available. These include several large examples that could not be minimized by previous methods [13]. In particular both tools can solve examples that cannot be solved by the currently fastest minimizer Hfmin. On the more difficult examples that can be solved by Hfmin, Espresso-HF and Impymin are typically orders of magnitude faster.

Although Espresso-HF is a heuristic minimizer, it almost always obtains an exactly minimum-size cover. Espresso-HF also employs a new fast method to check for the existence of a hazard-free solution, which does not need to generate all prime implicants.

[13]In publications on the 3D method (see e.g. [43], [41]), note that several of these examples appear but only *single-output* minimization is performed.

Impymin performs exact 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 based on the new idea of incorporating hazard-freedom 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 the reviewers for insightful suggestions, Bob Fuhrer for providing his tool Hfmin, and Montek Singh for interesting discussions.

References

[1] Peter A. Beerel. *CAD Tools for the Synthesis, Verification, and Testability of Robust Asynchronous Circuits.* PhD thesis, Stanford University, 1994.

[2] M. Benes, S.M. Nowick, and A. Wolfe. A fast asynchronous huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1998.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] 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 *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1998.

[5] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, MIT Laboratory for Computer Science, June 1987.

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

[7] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, E80-D(3):315–325, March 1997.

[8] O. Coudert. Two-level logic minimization: an overview. *Integration, the VLSI journal*, 17:97–140, 1994.

[9] A. Davis, B. Coates, and K. Stevens. Automatic synthesis of fast compact asynchronous control circuits. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 193–207. Elsevier Science Publishers, 1993.

[10] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. In Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 38-Sup23, pages 231–286. Marcel Dekker, Inc., 1998.

[11] R.K. Brayton et al. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, 1984.

[12] R.M. Fuhrer, B. Lin, and S.M. Nowick. Symbolic hazard-free minimization and encoding of asynchronous finite state machines. In *1995 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society Press, November 1995.

[13] R.M. Fuhrer, S.M. Nowick, M. Theobald, N.K. Jha, and L. Plana. Minimalist: An environment for the synthesis and verification of burst-mode asynchronous machines. In *International Workshop on Logic Synthesis*, 1998.

[14] S.B. Furber, J.D. Garside, S. Temple, J. Liu, P. Day, and N.C. Paver. Amulet2e: An asynchronous embedded controller. In *Async97 Symposium*. ACM, April 1997.

[15] J. Kessels and P. Marston. Design asynchronous standby circuits for a low-power pager. In *Async97 Symposium*. ACM, April 1997.

[16] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. Series in Parallel Computing. John Wiley & Sons, 1994.

[17] P. Kudva, G. Gopalakrishnan, and H. Jacobson. A technique for synthesizing distributed burst-mode circuits. In *Proceedings of the 33rd Design Automation Conference*. ACM, 1996.

[18] Luciano Lavagno and Alberto Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, 1993.

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

[20] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. The design of an asynchronous microprocessor. In *1989 Caltech Conference on Very Large Scale Integration*, 1989.

[21] S. Minato. Zero-Suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th Design Automation Conference*. ACM, 1993.

[22] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: design of a quasi-delay-insensitive microprocessor. *IEEE Design and Test*, 11(2):50–63, Summer 1994.

[23] L.S. Nielsen and J. Sparso. A low-power asynchronous data path for a fir filter bank. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 197–207. IEEE Computer Society Press, November 1996.

[24] S. M. Nowick and M. Theobald. Synthesis of low-power asynchronous circuits in a specified environment. In *Proceedings of 1997 International Symposium on Low Power Electronics and Design*, pages 92–95, 1997.

[25] S.M. Nowick and B. Coates. UCLOCK: automated design of high-performance unclocked state machines. In *IEEE International Conference on Computer Design*, pages 434–441, October 1994.

[26] 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 *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 419–427. IEEE Computer Society Press, January 1993.

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

[28] S.M. Nowick, N.K. Jha, and F. Cheng. Synthesis of asynchronous circuits for stuck-at and robust path delay fault testability. *IEEE Transactions on CAD*, CAD-16(12):1514–1521, December 1997.

[29] Steven M. Nowick and David L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. *IEEE Transactions on CAD*, CAD-14(8):986–997, August 1995.

[30] R. Rudell and A. Sangiovanni Vincentelli. Multiple valued minimization for PLA optimization. *IEEE Transactions on CAD*, CAD-6(5):727–750, September 1987.

[31] J.W.J.M. Rutten and M.R.C.M. Berkelaar. Improved state assignments for burst mode finite state machines. In *Proceedings of the 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1997.

[32] J.W.J.M. Rutten, M.R.C.M. Berkelaar, C.A.J. van Eijk, and M.A.J. Kolsteren. An efficient divide and conquer algorithm for exact hazard free logic minimization. In *Proc. Design, Automation and Test in Europe (DATE)*. IEEE Computer Society Press, February 1998.

[33] 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.

[34] 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.

[35] R.F. Sproull, I.E. Sutherland, and C.E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, 1994.

[36] M. Theobald and S.M. Nowick. An implicit method for hazard-free two-level logic minimization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1998.

[37] M. Theobald, S.M. Nowick, and T. Wu. Espresso-HF: A heuristic hazard-free minimizer for two-level logic. In *Proceedings of the 33rd Design Automation Conference*. ACM, 1996.

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

[39] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design and Test of Computers*, 11(2):22–32, Summer 1994.

[40] Chantal Ykman-Couvreur, Bill Lin, and Hugo de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, September 1994. User and Tutorial manual.

[41] K. Yun and D.L. Dill. A high-performance asynchronous SCSI controller. In *IEEE International Conference on Computer Design*. IEEE Computer Society Press, October 1995.

[42] Kenneth Y. Yun, Ayoob E. Dooply, Julio Arceo, Peter A. Beerel, and Vida Vakilotojar. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

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

Michael Theobald is a Ph.D. student of Computer Science at Columbia University. He received the Diplom degree in Computer Science from Johann Wolfgang Goethe-Universität, Frankfurt/Main, Germany, in 1994.
His research interests include synchronous and asynchronous circuits, computer-aided digital design, logic synthesis, formal verification, efficient algorithms and data structures, and combinatorial optimization.
He received the Honorable Mention Award at the 1997 International Conference on VLSI Design, and was a Best Paper Finalist at the 1998 IEEE Async Symposium.

Steven M. Nowick is an Associate Professor of Computer Science at Columbia University. He received a Ph.D. in Computer Science from Stanford University in 1993, and a B.A. from Yale University. His Ph.D. dissertation introduced an automated synthesis method for locally-clocked asynchronous state machines, and he formalized the asynchronous specification style called "burst mode".

His research interests include asynchronous circuits, computer-aided digital design, low-power and high-performance digital systems, logic synthesis, and formal verification of finite-state concurrent systems.

Dr. Nowick received an NSF Faculty Early Career (CAREER) Award (1995), an Alfred P. Sloan Research Fellowship (1995) and an NSF Research Initiation Award (RIA) (1993). He received a Best Paper Award at the 1991 International Conference on Computer Design, and was a Best Paper Finalist at the 1993 Hawaii International Conference on System Sciences and at the 1998 Async Symposium.

He was Program Committee Co-Chair of IEEE Async-94 Symposium, and is Program Committee Co-Chair of the upcoming IEEE Async-99 Symposium. He is a member of several international program committees, including ICCAD, ICCD, ARVLSI, and Async. He is also Guest Editor of a forthcoming special issue of the journal, "Proceedings of the IEEE", on asynchronous circuits and systems.