Symbolic Hazard-Free Minimization and Encoding of Asynchronous Finite State Machines*

Robert M. Fuhrer
Dept. of Computer Science
Columbia University
New York, NY 10027

Bill Lin
IMEC Laboratory
Kapeldreef 75
B-3001 Leuven, Belgium

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

Abstract — This paper presents an automated method for the synthesis of multiple-input-change (MIC) asynchronous state machines. Asynchronous state machine design is subtle since, unlike synchronous synthesis, logic must be implemented without *hazards*, and state codes must be chosen carefully to avoid *critical races*. We formulate and solve an *optimal hazard-free and critical race-free encoding* problem for a class of MIC asynchronous state machines called *burst-mode*. Analogous to a paradigm successfully used for the optimal encoding of synchronous machines, the problem is formulated as an *input encoding problem*. Implementations are targeted to sum-of-product realizations. We believe this is the first general method for the optimal encoding of hazard-free MIC asynchronous state machines under a generalized fundamental mode of operation. Results indicate that improved solutions are produced, ranging up to 17% improvement.

1 Introduction

There has been a renewed interest in asynchronous design, because of their potential for high-performance, modularity and avoidance of clock skew. This paper focuses on one class of asynchronous designs: asynchronous state machines.

Several methods have recently been introduced for the synthesis of asynchronous state machines [9, 17, 8]. These methods have been automated and produce low-latency machines which are guaranteed hazard-free at the gate-level. The design tools have benefited from a number of hazard-free optimization algorithms: exact two-level logic minimization [10], multi-level logic optimization [15, 3, 4], and technology mapping [13]. However, none of these methods includes algorithms for optimal state assignment. *The contribution of this paper is a general method for the optimal state assignment of asynchronous state machines.*

Optimal state assignment of synchronous machines has been an active area of research. De Micheli [7] formulated and solved an *input encoding problem*, which approximates an optimal state assignment for PLA-based state machines. Other formulations as an *output encoding* or *input/output encoding problem* have also been developed [6, 16, 12, 1].

Synchronous state assignment methods are inadequate for asynchronous designs, since the resulting machines may have critical races and logic hazards. In this paper, we consider two related problems in the synthesis of asynchronous state machines: *critical race-free state encoding* and *hazard-free logic minimization.* In existing synthesis trajectories [17, 8], these problems are solved separately, where state assignment is typically performed without regard to the optimality of the eventual logic implementation, which may lead to unnecessarily expensive solutions.

Recently, we introduced algorithms to solve two constrained optimal state assignment problems for asynchronous state machines [2]. The first solved an optimal critical race-free assignment problem, but ignored hazard issues. The second solved a combined hazard-free/critical race-free assignment problem limited to *single-input change (SIC)* asynchronous state machines. In this paper, we generalize this work, and solve a combined hazard-free/critical race-free assignment problem for a class of *multiple-input change (MIC)* state machines, called *burst-mode* [9, 17, 8].

Analogous to a paradigm successfully used for the optimal state assignment of synchronous machines, such as KISS [7], the problem is formulated as an *input encoding problem*. In particular, we solve the combined problem by formulating a **symbolic hazard-free minimization problem** for asynchronous synthesis. In this formulation, a symbolic logic specification, where states are represented as a multiple-valued variable, is first minimized to obtain a minimal multi-valued two-level representation. As in KISS, we assume each output and symbolic next-state is treated as a *binary* output function, where the co-domain has only the values 0 and 1. Unlike KISS, however, we introduce an exact *hazard-free* multi-valued logic minimization procedure.

After symbolic minimization, a **constrained encoding step** is performed. Encoding constraints in the form of *dichotomies* [14, 12] are introduced, which must be satisfied in the context of MIC asynchronous state machines. These constraints are related to the critical race-free constraints introduced by Tracey [14] and the *face-embedding* constraints introduced by De Micheli [7], but *subsume both*.

Finally, **encoding constraints are solved** using exact and heuristic techniques (our previous work used only exact techniques [2]). The exact procedure makes use of an existing tool, *dichot* [12], and the heuristic procedure uses the simulated annealing mode of *nova* [16]. For the heuristic problem, we propose a novel partitioning of constraints into *compulsory* and *non-compulsory* constraints; a weighted annealing algorithm is used to ensure that compulsory constraints are solved.

A key contribution of our method is that it produces *exactly minimal hazard-free (two-level) output logic*, over *all* possible critical race-free assignments. This result is significant since the latency of an asynchronous machine is determined by its output logic: there are no clock or latches. For next-state logic, our approach leads only to an approximate solution. However, in practice, high quality solutions are produced for next-state logic as well, ranging up to 17% overall improvement. We believe this is the first general method for the optimal state assignment of hazard-free MIC asynchronous state machines.

*Supported in part by NSF under Grant no. MIP-9308810, by a grant from the IBM Corporation, and by the E.C. under Grant no. ESPRIT-6143 (EXACT).

2 BACKGROUND

2.1 Optimal State Assignment for Synchronous Machines

In KISS [7], De Micheli formulated the optimal state assignment problem as an *input encoding problem*. The goal is to find a binary encoding of symbolic inputs to ensure an optimal sum-of-products implementation. The algorithm has three steps:

1. Generate a minimal symbolic cover
2. Generate a set of encoding constraints
3. Solve these constraints to produce a state assignment

The first step is symbolic logic minimization. The next-state function is effectively treated as a *set* of functions, one for each possible next-state value, since no information is yet available as to the relation of the various next-state values to one another. As a result, the symbolic minimization problem can be formulated as a multi-output multiple-valued-input minimization problem and solved using *espresso-mv* [11]. A minimal symbolic cover is formed, consisting of a set of symbolic implicants. Each implicant has four parts: binary inputs, symbolic present state, symbolic next state, and binary outputs. Present and next state can be represented using either symbolic or positional-cube notation.

A key goal in this approach is to ensure the correctness of the symbolic cover after it is instantiated with binary state codes. To understand the problem, consider the state table of Figure 1, having 2 inputs, 4 symbolic states, and 1 output, and the given 2-variable state assignment. A minimal symbolic cover for the output consists of 2 symbolic implicants: p_1 =`<0* {D}>` and p_2 =`<*1 {B,C}>`. [1] Implicant p_1 contains a single symbolic state, D, and therefore can be instantiated as binary product `<0* 11>`. However, implicant p_2 contains a pair of symbolic states, B and C, forming a *state group*. The smallest single binary cube, or *group face*, which contains the given codes for B and C is the *supercube* of the two codes: $**$. In this case, the resulting binary product, `<*1 **>`, is *invalid*, since it also contains an OFF-set minterm `<11 00>` corresponding to symbolic minterm `<11 {A}>`.

	inputs				
	00	01	11	10	state codes
A	A,0	A,0	D,0	A,0	00
B	B,0	B,1	B,1	A,0	01
C	A,0	B,1	C,1	C,0	10
D	D,1	D,1	D,0	C,0	11

Figure 1: Example state table with state assignment

To avoid this problem, in the second step, *face embedding constraints* are imposed:

For each symbolic implicant p, with state group S_p, the corresponding group face must not intersect the code of any state s not in S_p. [7]

The above encoding constraints can be described using *dichotomies* [1, 14]. Given a set of states S, a dichotomy is a bipartition (U, V) of a subset T of states of S. In a given state assignment, a binary state variable y_i *covers* the dichotomy (U, V) if $y_i = 0$ for every state in U and $y_i = 1$ for every state in V (or vice-versa) [15, 14]. For the given problem, a set of *n-to-1 dichotomies* is formed, *i.e.*, between each state group S_p (containing n states) and each single disjoint state $s \notin S_p$. In the above example, dichotomies $(BC; A)$ and $(BC; D)$ are generated to prevent invalid state assignments with respect to the output implementation. Exact dichotomy solvers have been developed which produce minimum-length assignments [1, 12].

The third step is to find a state assignment satisfying these encoding constraints. A final step, after state assignment, is to produce a binary logic implementation. Typically, *espresso* or *espresso-exact* are used, since the resulting cover may have *smaller* cardinality than the symbolic cover (see [7]).

2.2 Burst-Mode Asynchronous State Machines

In this subsection, we give an overview of burst-mode machines, a class of multiple-input change asynchronous state machines.

Burst-Mode Specifications

An asynchronous state machine allowing multiple-input changes can be specified by a form of state diagram, called a *burst-mode specification* [9] (see example in Figure 2). A burst-mode specification contains a finite number of states, a number of labelled arcs connecting pairs of states, and a distinguished start state (initial wire values are either specified or assumed 0). Burst-mode specifications, and variants, have been used for several recent asynchronous design methods [9, 17, 8].Arcs are labelled with possible transitions, taking the system from one state to another. Each transition consists of a non-empty set of input changes (an *input burst*) and a set of output changes (an *output burst*). Note that every input burst must be non-empty; if no inputs change, the system is stable.

In a given state, when all the inputs in some input burst have changed value, the system generates the corresponding output burst and moves to a new state. Inputs in a given input burst may arrive in any order and at arbitrary times. However, once an input burst is complete, no further input changes may occur until the resulting output changes have occurred (see next subsection for details). There are two further restrictions on specifications. First, no input burst in a given state can be a subset of another, since otherwise the behavior may be ambiguous. This restriction is called the *maximal set property*. Second, a given state is always entered with the same set of input values; that is, each state has a *unique entry point.*

Target Implementation

A burst-mode specification can be realized as a Huffman machine, as shown in Figure 3. The machine consists of combinational logic with primary inputs, primary outputs and fed-back state variables [15]. State is stored on the feedback loops, which may have attached delay elements.

The machine behaves as follows. Initially, the machine is stable in some state. Inputs in a specified input burst may then change value in any order and at any time. Throughout this input burst, the machine outputs and state remain unchanged. When the input burst is complete, the outputs change value monotonically as specified. A state change may also occur concurrently with the output change. In this case, the machine will be driven to a new stable state. Only a single feedback cycle occurs. Alternatively, no state change may occur. In either case, no further inputs may arrive until the machine is stable. That is, the machine operates in *fundamental mode* [15]. When the machine is stable, the cycle is complete and the machine is ready to receive new inputs. Throughout the entire cycle, outputs and state variables must be glitch-free.

[1] For simplicity, we consider only single-output implicants in this example, though in general the method produces multiple-output implicants.

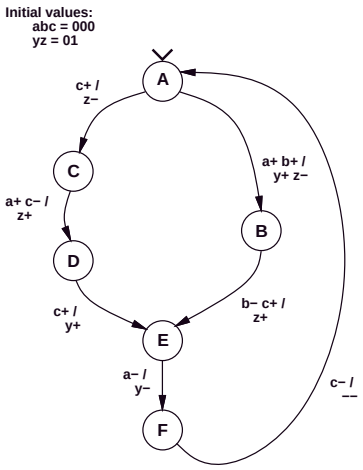


Figure 2: Example burst-mode specification.

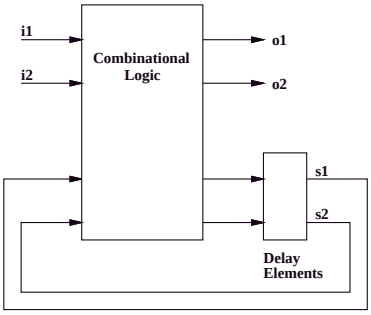


Figure 3: Block diagram of Huffman machine.

3 Problem Statement

We can now define the synthesis problem:

> **Problem: Optimal Hazard-Free/Critical Race-Free Assignment for Burst-Mode (MIC) Asynchronous State Machines.** *Find a critical race-free assignment for a burst-mode flow table having a* **hazard-free** *sum-of-products implementation of minimal cost.*

Our synthesis method follows the 3 basic steps of the KISS algorithm, but with modifications. In the first step, it formulates a *hazard-free* symbolic covering problem. In the second step, modified encoding constraints are generated. These constraints are *not* the union of the KISS and Tracey constraints, but subsume both. After solving these encoding constraints in step 3, a binary hazard-free minimizer is used to find a hazard-free logic implementation.

4 Multiple-Valued Functions and Hazards

For the following, we assume basic familiarity with the terminology of multi-valued logic minimization (see [11]).

4.1 Circuit Model

This paper considers combinational circuits having arbitrary finite gate and wire delays (*unbounded wire delay model* [10]).A pure delay model is assumed (see [15]).

4.2 Multiple-Valued Multiple-Input Changes

In this section, we generalize the notions of multiple-input changes and transition cubes from the binary domain [10] to the multiple-valued domain.

Definition 4.1 (Multiple-valued transition cube) *A* **multiple-valued transition cube** *is a cube with a* **start point** *and an* **end point***. Let A and B be two minterms in a multiple-valued space D. The multiple-valued transition cube, denoted as $[A, B]$, from A to B has start point A and end point B and contains all minterms that can be reached during a transition from A to B. More formally, if A and B are described by products, with i-th literals $A_i^{S_{A_i}}$ and $B_i^{S_{B_i}}$, respectively, then the i-th literal for the product of $T = [A, B]$ is the literal $T_i^{S_{A_i} \cup S_{B_i}}$.*

Definition 4.2 (Multiple-valued open transition cube) *The* **(multiple-valued) open transition cube** *$[A, B)$ from A to B is defined as: $[A, B] - \{B\}$.*

Definition 4.3 (Multiple-valued input transition) *A* **(multiple-valued) input transition** *or* **(multiple-valued) multiple-input change** *from input state A to B is described by transition cube $[A, B]$.*

An *intermediate state* $X \in [A, B]$ is potentially reachable during the input transition from A to B if for all variables X_i, the corresponding literal X_i is either equal to A_i or B_i. A multiple-input change specifies what variables are permitted to 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. An input transition occurs during a *transition interval*, $t_I \leq t \leq t_F$, where inputs change at time t_I and the circuit returns to a steady state at time t_F.

Definition 4.4 (Static and dynamic transitions) *An input transition from input state A to B for a multiple-valued function f is a* **static transition** *if $f(A) = f(B)$; it is a* **dynamic transition** *if $f(A) \neq f(B)$.*

In this paper, we consider only static and dynamic transitions where f is fully defined; that is, for every $X \in [A, B]$, $f(X) \in \{0, 1\}$.

4.3 Multiple-Valued 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 4.5 (Static function hazard) *A multiple-valued 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 4.6 (Dynamic function hazard) *A multiple-valued 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 multiple-valued implementation of the function is guaranteed to avoid a glitch during the transition, assuming arbitrary gate and wire delays. Therefore, we consider only transitions which are free of function hazards (cf. [10]).

4.4 Multiple-Valued 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. Here, we extend notions of static and dynamic logic hazards to multiple-valued functions. To do so, we will provide these definitions in terms of an abstract **multiple-valued sum-of-products implementation**. That is, each multiple-valued product term in the multiple-valued cover is implemented as a single *multiple-valued AND gate*. The circuit output is implemented as a *Boolean OR gate* that combines the AND gates.

Definition 4.7 (Static (Dynamic) logic hazard) *A multiple-valued cover circuit implementing multiple-valued 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, 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.

4.5 Problem Abstraction

The hazard-free multiple-valued minimization problem can now be stated as follows. Given a multiple-valued function f, and a set, T, of *specified* function-hazard-free multiple-valued (static and dynamic) input transitions of f, find a minimal-cost *multiple-valued cover* of f that is free of logic hazards for every specified input transition $t \in T$.

5 Symbolic Hazard-Free Minimization

In this section, we present an exact minimization algorithm for producing a hazard-free multiple-valued cover. While the standard multiple-valued minimization problem *without* considerations for hazards has been adequately addressed before [11], the corresponding problem in the context of asynchronous synthesis and hazard-free synthesis has not yet been addressed. We first state the conditions that the cover must satisfy in order to ensure hazard-freeness. These conditions will lead to a notion of **multiple-valued dynamic-hazard-free (DHF-) prime implicants**. Using these *prime implicants*, a *constrained* covering step must be solved to select a hazard-free cover. These issues are addressed in the sequel.

5.1 Conditions for a Hazard-Free Transition

We now describe conditions to ensure that a sum-of-products implementation is hazard-free for a given input transition. Assume that $[A, B]$ is the transition cube corresponding to a *function-hazard-free* transition from input state A to B for a multi-valued combinational function f. In the following discussion, we assume that C is any multi-valued cover of f. The following lemmas present necessary and sufficient conditions to ensure that a multi-valued AND-OR implementation of f has *no logic hazards* for the given transition. The following results are extensions from the binary case [10].

Lemma 5.1 *If f has a $0 \rightarrow 0$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B.*

Lemma 5.2 *If f has a $1 \rightarrow 1$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B* if and only if *$[A, B]$ is contained in some cube of cover C.*

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

Lemma 5.3 *If f has a $1 \rightarrow 0$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B* if and only if *every cube $c \in C$ intersecting $[A, B]$ also contains A.*

Lemma 5.2 requires that in a $1 \rightarrow 1$ transition, *some* product holds its value at 1 throughout the transition. Lemma 5.3 ensures that no product will glitch *in the middle* of a $1 \rightarrow 0$ transition: all products change value monotonically during the transition. In each case, the implementation will be free of hazards for the given transition.

An immediate consequence of Lemma 5.3 is that, if a dynamic transition is free of logic hazards, then every static sub-transition will be free of logic hazards as well:

Lemma 5.4 *If f has a $1 \rightarrow 0$ transition from input state A to B which is hazard-free in the implementation, then, for every input state $X \in [A, B)$ where $f(X) = 1$, the transition subcube $[A, X]$ is contained in some cube of cover C.*

Lemma 5.5 *If f has a $1 \rightarrow 0$ transition from input state A to B which is hazard-free in the implementation, then for every input state $X \in [A, B)$ where $f(X) = 1$, the static $1 \rightarrow 1$ transition from input state A to X is free of logic hazards.*

Lemmas 5.2 and 5.4 are used to define the covering requirement for a hazard-free transition. The cube $[A, B]$ in Lemma 5.2 and the *maximal* subcubes $[A, X]$ in Lemma 5.4 are called *required cubes*. These cubes define the ON-set of the function in a transition. Each required cube *must* be contained in some cube of cover C to ensure a hazard-free implementation. This property can be more formally stated as follows.

Definition 5.1 (Required cube) *Given a multiple-valued function f, and a set, T, of specified function-hazard-free multiple-valued input transitions of f, every cube $[A, B] \in T$ corresponding to a static $1 \rightarrow 1$ transition, and every maximal subcube $[A, X] \subset [A, B]$ where f is 1 and $[A, B] \in T$ is a dynamic $1 \rightarrow 0$ transition, is called a* **required cube**.

Lemma 5.3 constrains the cubes which may be included in a cover C. Each $1 \rightarrow 0$ transition cube is called a *privileged cube*, since no cube c in the cover may intersect it unless c contains its *start point*. If a cube intersects a privileged cube but does not contain its start point, it *illegally intersects* the privileged cube and may not be included in the cover. This property can be more formally stated as follows.

Definition 5.2 (Privileged cube) *Given a multiple-valued function f, and a set, T, of specified function-hazard-free multiple-valued input transitions of f, every cube $[A, B] \in T$ corresponding to a dynamic $1 \rightarrow 0$ transition is called a* **privileged cube**.

5.2 Hazard-Free Covers

A *hazard-free cover* of function f is a cover of f whose multi-valued AND-OR implementation is hazard-free for a *given set* of specified input transitions. The following theorem describes all hazard-free covers for function f for a set of multiple-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 5.1 (Hazard-free covering) *A sum-of-products C is a hazard-free cover for function f for all specified input transitions if and only if:*
(a.) No cube of C intersects the OFF-set of f;
(b.) Each required cube *of f is contained in some cube of C; and*
(c.) No cube of C intersects any privileged cube *illegally.*

Conditions (a) and (c) in Theorem 5.1 determine the implicants which may appear in a hazard-free cover of a Boolean function f. Condition (b) determines the covering requirement for these implicants in a hazard-free cover. Therefore, Theorem 5.1 precisely characterizes the covering problem for hazard-free two-level logic.

In general, the covering conditions of Theorem 5.1 may not be satisfiable for an arbitrary Boolean function and set of transitions (cf. [15, 10]). This case occurs if conditions (b) and (c) cannot be satisfied simultaneously.

5.3 Exact Hazard-Free Multiple-Valued Minimization

Many exact logic minimization algorithms, such as Espresso-MV-Exact [11], are based on the Quine-McCluskey algorithm [5]. The Espresso-MV-Exact algorithm solves the two-level multiple-valued minimization problem in three steps:

1. Generate multiple-valued prime implicants;
2. Construct prime implicant table;
3. Generate minimum cover of this table.

Here, we extend an existing exact *hazard-free* two-level minimizer [10] to multi-valued functions. Theorem 5.1(a) and (c) determine the implicants which may appear in a hazard-free cover of a multiple-valued function f. Such implicants will be called *multiple-valued dynamic-hazard-free (DHF-) implicants.* They are defined as follows:

Definition 5.3 (Multiple-valued DHF-implicants) *A* **multiple-valued DHF-implicant** *is an implicant which does not intersect any privileged cube of f illegally. A* **multiple-valued DHF-prime implicant** *is a multiple-valued DHF-implicant contained in no other multiple-valued DHF-implicant. An* **essential multiple-valued DHF-prime implicant** *is a multiple-valued DHF-prime implicant which contains a required cube contained in no other multiple-valued DHF-prime implicant.*

By Theorem 5.1(c), **only multiple-valued DHF-implicants may appear in a hazard-free cover**. Theorem 5.1(b) determines the covering requirement for a hazard-free cover of f: **every required cube of f must be covered**, that is, contained in some cube of the cover. Thus, the *two-level hazard-free logic minimization problem* is *to find a minimum cost cover of a function using only multiple-valued DHF-prime implicants where every required cube is covered.*

The modified hazard-free multiple-valued minimization algorithm is as follows:

1. Generate multiple-valued DHF-prime implicants;
2. Construct multiple-valued DHF-prime implicant table;
3. Generate minimum cover of this table.

These steps are detailed below.

5.4 Generation of Multiple-Valued DHF-Prime Implicants

Multiple-valued DHF-prime implicants for function f are generated in two steps. The new algorithm follows the approach described in [10], but extended to multiple-valued functions. First, multiple-valued prime implicants of f are generated from the required cubes (which defines the on-set) and the off-set, using existing algorithms [11]. Second, these prime implicants are transformed into multiple-valued DHF-prime implicants by iterative refinement. The new algorithm, *MVI-PI-to-DHF-PI,* checks each implicant p for illegal intersection with any multiple-valued privileged cube, q. If such an intersection occurs, the implicant is reduced in all possible ways to avoid intersection. In particular, p is replaced by the set $\{p_1, \ldots, p_n\}$ of maximal subcubes of p which do not intersect q (*i.e.,* $\forall i \in \{1, \ldots, n\}, p_i \cap q = \phi$). Note that, in the multi-valued framework, reduction is uniformly performed across both input and output spaces. The reduced implicants may have remaining, or new, illegal intersections with other privileged cubes. The process continues until only dhf-implicants remain. Non-prime dhf-implicants are removed by a check for single-cube containment.

5.5 Generation of DHF-Prime Implicant Table

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

5.6 Generation of a Minimum Cover

The multiple-valued DHF-prime implicant table describes a standard unate covering problem. It is solved using an existing algorithm, *minimum-cover* [11].

5.7 Handling Multiple-Output Minimization

As in Espresso-MV-Exact [11], *multiple-output functions* are handled by making the output parts into a single N*-valued MV variable*, where N is the number of outputs. The transformation is straightforward and is described in [11]. Using this transformation, the symbolic hazard-free multiple-valued minimization procedure can be used to minimize multiple-output functions.

6 Constrained Encoding

We now consider the constrained encoding problem that must be solved to produce a correct binary logic implementation for an asynchronous flow table.

6.1 Encoding Constraints

In this step, encoding constraints are generated based on the symbolic cover. These constraints ensure that the cover will be correctly instantiated.

The face embedding constraints used by KISS for synchronous machines are insufficient for asynchronous machines for two reasons: (1) they do not consider the transient behavior of an asynchronous state machine, and (2) they do not consider hazard-free logic requirements. Therefore, face embedding constraints must be generalized. We consider these two problems in turn.

A new condition concerns the *functional correctness* of the output and next-state implementations in the presence of state transitions. During a transition of 2 or more state variables, transient points in the total input state space are reached which do not correspond to any valid encoded state. The possibility arises that the group face for some symbolic product term implementing a binary output may intersect such a transient point, thus inadvertently turning on the product term *during* the state transition. If

the intended value of that output during the state transition is `0`, the output function will be incorrectly implemented.

Example. Consider a binary output symbolic implicant `<I1 {S0,S1,S2}>` for some output function z. Suppose there is a state transition `S3` $\rightarrow$ `S4` in input column `I1` during which z should be held at `0`. Assume the following state assignment:

S0	0000
S1	1000
S2	1100
S3	0110
S4	0101

Using this assignment, the corresponding binary implicant is `<I1 **00>`. As a result, during the `S3` $\rightarrow$ `S4` transition, the state variables can reach the transient value `0100` which would turn on the given implicant, incorrectly forcing the output value to `1`. This problem occurs even though the face embedding constraints for state group $\{S0, S1, S2\}$ are satisfied.□

A similar problem occurs for the next-state function. This case requires a trivial generalization of the condition: if the value of the symbolic function (i.e. the destination state) during the transition differs from that which the product term implements, the machine will be incorrectly realized.

The proposed solution is to add dichotomy constraints to avoid problems resulting from such state transitions. Unlike the face embedding constraints, these dichotomies are *n-to-2*: between (i) a state group of an symbolic product (*e.g.*, $\{S0, S1, S2\}$ in the preceding example) and (ii) a pair of states defining a state transition (*e.g.*, $\{S3, S4\}$). The resulting generalized embedding constraint, ($\{S0, S1, S2\}$,$\{S3, S4\}$), ensures that the output will be correctly implemented after instantiation.

The above discussion only addresses constraints derived from a symbolic cover. It does not consider critical race-free encoding constraints. In Section 7, however, it will be shown that the above constraints in fact subsume all Tracey constraints, and therefore ensure a critical race-free assignment.

In summary, asynchronous designs differ from synchronous designs, since state changes may pass through intermediary states. While face embedding constraints ensure that an implicant does not intersect an OFF-set *minterm*, generalized constraints are needed for asynchronous machines to ensure that an implicant does not intersect a *set* of OFF-set minterms that may be traversed during a state change.

The second difference between the original face embedding constraints and asynchronous constraints concerns the need to avoid *logic hazards*. In KISS, face embedding constraints ensure that an implicant does not intersect the OFF-set. However, in asynchronous synthesis, a non-prime *DHF-prime implicant* may not illegally intersect a privileged cube as well. Encoding constraints must be added to ensure that, if a symbolic implicant has no illegal intersections, the encoded implicant will not either.

For the given class of burst-mode machines, though, such hazard-free constraints are degenerate. As indicated earlier, in a burst-mode flow table, dynamic transitions only occur during input bursts: that is, within a given state. Therefore, each privileged cube has a singleton state group. If a DHF-prime implicant has state group $\{S0, S1, S2\}$ and it must avoid intersection with a privileged cube in state $S3$, a simple *n-to-1* dichotomy must be generated. However, such a dichotomy is already generated as a face-embedding constraint. Therefore, no further constraints need to be generated for this class of machines to avoid dynamic hazards.

Constraint Generation Algorithm

In addition to the KISS face embedding constraints, we use the following algorithm:

```
for each implicant p in the symbolic cover {
  for each state transition t {
    if p intersects the input column of t {
      if some output o that p implements has value 0 during t {
        generate dichotomy { stategroup(p); states(t) }   }}}}
```

This algorithm generates *n-to-2* dichotomies, where t is a state transition from an unstable to a stable state.

6.2 Solving Constraints and Hazard-Free Logic Minimization

Since all constraints are described as dichotomies, they can be solved using a dichotomy solver. The resulting constraints ensure that products can be safely instantiated with respect to both stable and transient points in the symbolic flow table.

Constraints are solved using two methods: exact solution (using *dichot* [12]) and heuristic solution (using *nova*'s simulated annealing mode [16]). The goal of the heuristic method is to solve as many constraints as possible given a fixed code-length.

However, a problem arises in the straightforward application of the heuristic method. Unlike synchronous applications, a heuristic solution of our asynchronous constraints may result in an *incorrect* implementation. In particular, as a bare minimum, we require that every state assignment be critical race-free. These critical race-free constraints are described by dichotomies, which are subsumed by our optimality constraints (see Section 7). Since a partial constraint solver may not satisfy all dichotomies, the resulting state assignment may have critical races.

Our solution is to partition dichotomies into two classes: *compulsory* and *non-compulsory*. Critical race-free constraints are compulsory, and must be satisfied. Remaining constraints are concerned with logic optimality; these are non-compulsory, or optional. Different weights are assigned to the dichotomies in the two classes, to ensure that all compulsory constraints are satisfied. In practice, such an approach has worked well on a number of examples. [2]

Finally, once a state assignment is produced, the symbolic machine is instantiated with the resulting encoding. The resulting binary-valued function is then passed through a multi-output binary-valued hazard-free logic minimizer to produce a final machine implementation.

7 Theoretical Results

We now sketch the basic theoretical results for our synthesis algorithm. First, we define a "pseudo-canonical" state assignment, roughly analogous to the use of a "canonical" 1-hot assignment in KISS. We then formally define the instantiated asynchronous machine specification (encoded flow table) and binary implementation (cover) under this assignment. Second, we summarize results on the correctness and cardinality of the binary cover. Finally, we present results on the optimality of the binary cover.

7.1 Machine Instantiation

Pseudo-Canonical State Assignment

In [7], DeMicheli indicates that, for synchronous machines, any symbolic minimized cover can be assigned a 1-hot canonical encoding. The result is a 1 $\rightarrow$ 1 mapping of symbolic to binary

[2] It is possible that a solution will not satisfy all compulsory constraints. If this occurs, the weights can be modified, the run can be repeated to randomly explore another portion of the solution space, or the code length limit can be raised.

implicants, yielding a canonical cover whose cardinality is identical to that of the symbolic cover. For asynchronous machines, however, a 1-hot encoding is not in general critical race-free [15]; furthermore, it will not generally satisfy the encoding constraints of Section 6. As a simple alternative, to demonstrate theoretical results, we propose the following: solve the encoding constraints to produce an assignment.[3] This assignment will be called *pseudo-canonical* for the given machine.

Symbolic Machine Instantiation

An encoding defines a mapping from a symbolic machine specification to an equivalent binary one. There are two components of an asynchronous machine specification: its functional specification and a set of specified transitions. For the functional specification, it is assumed that both ON-set and OFF-set are explicitly defined. The transitions are mapped in the obvious way: each symbolic startpoint (endpoint) $\langle in, present \rangle$ maps to the binary startpoint (endpoint) $\langle in, code(present) \rangle$.

We can view the functional specification as a set of ON-set and OFF-set cubes. Each symbolic product p (a 4-tuple $\langle in, present, next, out \rangle$), maps onto a binary product $\tilde{p}$, as follows:

p:	in	$present$	$next$	out
	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
$\tilde{p}$:	in	$supercube(codes(present))$	$code(next)$	out

For example, under the state assignment $S_0 = 000$, $S_1 = 011$, $S_2 = 100$, and $S_3 = 101$, the symbolic product `<011| {S0,S2} S2 |100>` is mapped to the binary product `<011|-00 100|100>`.

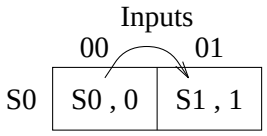
With the above view, mapping an asynchronous symbolic flow table to an encoded table, column transitions require special care. In a symbolic table, a column transition is defined only at its symbolic startpoint and endpoint. However, in an encoded table with a USTT critical race-free assignment, all *intermediate* (transient) entries for the transition must be defined as well. This latter property can easily be guaranteed by constraining the symbolic specification: a *single product* must be used to specify each column transition. This constraint ensures that, for each column transition (*i.e.*, state change), some product will be instantiated which defines all intermediate states in the encoded transition.

Symbolic Cover Instantiation

Given a symbolic hazard-free cover and a resulting state assignment, symbolic implicants can be instantiated by substituting binary codes using the mapping described above, yielding a binary cover C. Note that instantiating a symbolic implicant may produce an empty binary implicant, if its symbolic next-state is mapped to the binary 0-vector. Such an implicant can be dropped from the binary cover.

Unfortunately, the sharing of 1-bits by different state codes may cause *static* transitions for next-state to appear in the binary machine where only *dynamic* transitions appeared in the symbolic machine. To avoid hazards, extra terms must be added to the binary cover: static-1 transitions must each be completely covered by some implicant, while the symbolic dynamic transitions clearly would not have been.

Example. To understand the problem, consider an input transition in a machine with one output:



No implicant in the symbolic cover covers the entire transition, since both output and next-state undergo dynamic transitions. In particular, the next-state function S_0 has a $1 \rightarrow 0$ transition and the next-state function S_1 has a $0 \rightarrow 1$ transition (as does the output). However, suppose that S_0 is assigned code 011 and S_1 is assigned 110. In the instantiated machine, the second state bit will then make a $1 \rightarrow 1$ transition. However, since no symbolic cube covered the entire transition, no instantiated binary cube will either, and the second state bit will have a static-1 hazard. □

In sum, a naively instantiated cover will fail to properly implement certain static transitions of the next-state variables. A solution is to add one product term to the instantiated cover for each such static-1 transition. For the above transition, the implicant `<0- 011 010 0>` would be added, where 011 corresponds to state S_0. As a result, the canonical cover may have *greater* cardinality than the symbolic cover:

Property 7.1 (Opt-HFCRF Cardinality of Cover) *Let $|S|$ be the cardinality of the symbolic cover, $|C|$ be the cardinality of the binary instantiated cover, and k is the number of unstable state transitions in the flow table; then $|C| = O(|S| + k)$.*

Note that *this result is a theoretical upper bound only*. In practice, k additional products need not be added. Instead, the instantiated cover C is passed to a binary hazard-free minimizer and *re-run*, to improve results.

By analogy, KISS produces a theoretical upper bound on cardinality based a 1-hot-instantiated cover (although in KISS the upper bound is the cardinality $|S|$ of the symbolic cover; no added terms are required). This 1-hot-instantiated cover in KISS is *neither* guaranteed to have minimum number of products *nor* minimum code length [7]. In practice, shorter codes are sought, and the instantiated cover is likewise re-run through a binary minimizer to improve results [7, 16].

In both KISS and our method, the input encoding formulation and solution yield only approximations to optimal state assignment. In practice, though, both methods can result in significant improvements (see Section 8).

7.2 Correctness of Binary Cover

Due to space limitations, proofs are omitted for the following theorems. In the following, let C be the instantiated cover, derived using the pseudo-canonical assignment.

Theorem 7.1 *Cover C is a correct functional implementation of the encoded flow table.*

Furthermore, our encoding constraints *subsume* all critical race-free constraints [14], and the resulting implementation is hazard-free.

Theorem 7.2 *Any state assignment satisfying the encoding constraints of Section 6 is critical race-free.*

Theorem 7.3 *The cover C is hazard-free for every specified input and state transition.*

7.3 Optimality of Binary Cover

A final key result is that our algorithm produces state assignments and hazard-free realizations which are *exactly optimal* with respect to output logic (if outputs and next-state are minimized separately).

[3] In fact, a state assignment which satisfies all possible N-to-1 and N-to-2 constraints can always be found, for a given number N of states. However, such an assignment is prohibitively expensive; for simplicity, we consider a more practical assignment here.

Property 7.2 (Opt-HFCRF Optimality of Output Cover) *The binary instantiated* output *cover $\mathcal{OC}$ (where outputs are minimized separately from next-state) is* exactly *minimal.*

This result is especially important for asynchronous state machines. Since asynchronous machines have no clock or latches, the input-to-output latency is determined by output logic delay. Our algorithm finds a USTT state assignment which results in a *hazard-free output cover with smallest cardinality over all possible critical race-free assignments.*

8 Experimental Results

A preliminary set of experiments was run on industrial examples using our optimal encoding and logic minimization algorithms. Results appear in Figure 4. For each set of runs, the number of state variables (#b) and number of cubes (#c) in the final cover are reported. The column labelled *optimal* lists runs in which all constraints were solved. A parallel set of runs using a "random" (but minimal length) critical race-free encoding was done as well, labelled *base-crf*, for comparison with the optimal. Finally, a third set of runs, *opt-fixed*, was performed (for cases where *optimal* and *base-crf* differed in code length), using a fixed code length and partial constraint satisfaction. For this set, runs at or near the code length of the *base-crf* case were performed; the best of several iterations is reported.

The *opt-fixed* algorithm achieves results at least as good as the *optimal* and *base-crf* algorithms. As in KISS [7] and NOVA [16], this phenomenon occurs because input encoding is itself an approximate formulation. Hence, by using partial constraint satisfaction with restricted code lengths, a large percentage of optimality constraints can be satisfied with less overhead in the next-state implementation.

For all sets of runs, the hazard-free multi-output logic minimization algorithm was used for the binary implementation step. Improvements ranging up to 17% are observed.

Acknowledgments

The authors thank Giovanni De Micheli for suggesting that we consider the problem of optimal state assignment of hazard-free asynchronous machines.

		opt-fixed		*optimal*		*base-crf*	
DESIGN	**I/S/O**	#b	#c	#b	#c	#b	#c
sbuf-read-ctl	3/3/3	2	7	3	9	2	8
sbuf-send-ctl	3/4/3	2	11	4	12	2	11
rf-control	6/6/5	3	13	6	15	3	15
it-control	5/5/7	3	15	6	15	3	15
pe-send-ifc	5/5/3	3	18	7	27	3	21
sd-control	8/13/12	5	29	10	34	4	35
dram-ctrl	7/3/6	-	-	2	22	2	22
pscsi-ircv	4/4/3	2	9	4	12	2	10
pscsi-isend	4/6/3	3	17	7	23	3	19
pscsi-trcv	4/4/3	3	9	4	13	2	11
pscsi-trcv-bm	4/4/4	2	12	4	15	2	14
pscsi-tsend	4/7/3	3	18	7	22	3	18
sscsi-isend-bm	5/4/4	2	21	5	22	2	24
sscsi-isend-csm	5/3/4	-	-	2	12	2	12
sscsi-trcv-bm	5/4/4	2	18	5	24	2	18
sscsi-trcv-csm	5/3/4	2	12	3	12	2	12
sscsi-tsend-bm	5/5/4	3	17	6	20	3	18
sscsi-tsend-csm	5/4/4	2	14	5	15	2	14
stetson-p1	13/12/14	4	53	19	–*	4	55
stetson-p2	8/13/12	4	31	10	37	4	36

*Exact logic minimization failed due to insufficient virtual memory in prime generation.

Figure 4: Experimental Results

References

[1] M.J. Ciesielski, J.J. Shen, and M. Davio. A unified approach to input-output encoding for fsm state assignment. In *DAC-91*, June 1991.

[2] R.M. Fuhrer, B. Lin, and S.M. Nowick. Algorithms for the optimal state assignment of asynchronous state machines. In *1995 Conference on Advanced Research in VLSI*, pages 59–75. IEEE Computer Society Press, 1995.

[3] D.S. Kung. Hazard-non-increasing gate-level optimization algorithms. In *ICCAD-1992*.

[4] B. Lin and S. Devadas. Synthesis of hazard-free multilevel logic under multiple-input changes from binary decision diagrams. *IEEE Transactions on CAD*, 14(8):974–985, August 1995.

[5] E.J. McCluskey. *Logic Design Principles.* Prentice-Hall, 1986.

[6] G. De Micheli. Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros. *IEEE Transactions on CAD*, CAD-5(4):597–616, October 1986.

[7] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on CAD*, CAD-4(3):269–285, July 1985.

[8] S.M. Nowick and B. Coates. Automated design of high-performance unclocked state machines. In *ICCD-1994*.

[9] S.M. Nowick and D.L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *ICCAD-1991*.

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

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

[12] A. Saldanha, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. A framework for satisfying input and output encoding constraints. In *DAC-91*, June 1991.

[13] P. Siegel, G. De Micheli, and D. Dill. Technology mapping for generalized fundamental-mode asynchronous designs. In *30th ACM/IEEE Design Automation Conference*, June 1993.

[14] J.H. Tracey. Internal state assignments for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-15:551–560, August 1966.

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

[16] T. Villa and A. Sangiovanni-Vincentelli. NOVA: state assignment of finite state machines for optimal two-level logic implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(9):905–924, September 1990.

[17] K.Y. Yun, D.L. Dill, and S.M. Nowick. Synthesis of 3D asynchronous state machines. In *ICCD-1992*.