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

1   Introduction

There has been a renewed interest in asynchronous design, because of their potential for high-performance, modularity and avoidance of clock skew [43, 33, 22, 4, 16, 19]. This paper focuses on one class of asynchronous designs: asynchronous state machines. The design of asynchronous state machines has been an active area of research for the last 40 years. However, asynchronous state machine design remains a subtle problem, since to ensure correct dynamic behavior, *hazards* and *critical races* [42] must be eliminated.

Several methods have recently been introduced for the synthesis of asynchronous state machines [33, 29, 47, 7, 28]. 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 [30], multi-level logic optimization [42, 14, 17], technology mapping [39] and synthesis-for-testability [13, 31]. 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 [26] formulated and solved an *input encoding problem*, which approximates an optimal state assignment for PLA-based state machines. His program, *KISS*, performs symbolic logic minimization, and then solves a resulting set of encoding constraints to produce a state assignment. Other formulations as an *output encoding* or *input/output encoding problem* have also been developed [25, 44, 36, 5]. Other methods have targeted *multi-level* [18, 8] implementations.

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 [46, 7, 28], these problems are solved separately. First, state encoding is performed to ensure critical race-free operation [41]. This step is typically performed without regard to the optimality of the eventual logic implementation, which may lead to unnecessarily expensive solutions. For the second step, techniques are used to synthesize combinational logic that avoids all combinational hazards for a specified set of multiple-input changes [30, 14, 17]. This work goes beyond Eichelberger's early work on static hazards [9], to address the more complex problem of dynamic hazards [2]. Although a recent exact hazard-free two-level minimization algorithm [30] can be used, the quality of an implementation still depends heavily on the choice of state encoding.

Recently, we introduced algorithms to solve two constrained optimal state assignment problems for asynchronous state machines [12]. 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* [27, 46, 7, 28].

Analogous to a paradigm successfully used for the optimal state assignment of synchronous machines, such as KISS [26], 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 state variables are encoded as multiple-valued variables, 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* [41, 36] 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 [41] and the *face-embedding* constraints introduced by De Micheli [26], but *subsume both*. In particular, we extend the KISS *n-to-1* dichotomy constraints to *n-to-2* constraints.

Finally, **encoding constraints are solved** using exact and heuristic techniques (our previous work used only exact techniques [12]). The exact procedure makes use of an existing tool, *dichot* [36], and the heuristic procedure uses the simulated annealing mode of *nova* [45]. 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 output logic* (two-level), 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% improvement. We believe this is the first general method for the optimal state assignment of hazard-free MIC asynchronous state machines.

1.1 Organization of the Paper

The paper is organized as follows. Section 2 reviews an existing synchronous optimal state assignment method, KISS, and gives background on burst-mode asynchronous state machines and related work. Section 3 gives an overview of our new asynchronous state assignment algorithm. Section 4 gives definitions and theorems on hazard-free multi-valued logic, and Section 5 describes an exact multi-valued hazard-free two-level minimization algorithm. In Section 6, the symbolic minimization procedure is applied to the asynchronous optimal state assignment problem. Encoding constraints are presented in this section along with methods for solving them. Section 7 presents theoretical results, Section 8 describes the program implementation, and Section 9 presents the application of our new method to a number of examples.

2 Background

2.1 Optimal State Assignment for Synchronous Machines

In KISS [26], 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 [26]. 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 multiple-output multiple-valued-input minimization problem and solved using *espresso-mv* [26]. 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 state 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				state codes
	00	01	11	10	
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.* [26]

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 [26]).

The above encoding constraints can be described using *dichotomies* [5, 41]. 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) [42, 41]. 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. Exact dichotomy solvers have been developed which produce minimum-length assignments [5, 36].

A *1-hot encoding* [42, 26] always satisfies the above constraints, and can be used to implement the symbolic cover. This canonical state assignment has an important property:

> **Property #1:** The instantiated binary cover using a 1-hot code has the same cardinality as the original symbolic cover.

This property indicates that the cardinality of the symbolic cover is an upper bound on the size of a binary solution.

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 is specified by a form of state diagram, called a *burst-mode specification* [29]. A state diagram 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

---

[1]For simplicity, we consider only single-output implicants in this example, though in general the method produces multiple-output implicants.
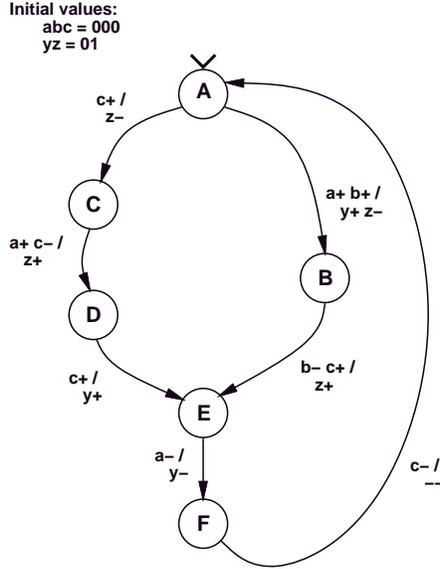
Figure 2: Example burst-mode specification.

0). Burst-mode specifications, and variants, have been used for several recent asynchronous design methods [29, 46, 28, 7]. 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 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 to specifications. First, no input burst in a given state can be a subset of another, since otherwise the behavior may be ambigorous. 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*.

An example of a burst-mode specification is shown in Figure 2. Each transition is labelled with an input burst followed by an output burst. Input and output bursts are separated by a slash, /. A rising transition is indicated by a "+" and a falling transition is indicated by a "−". This specification describes a simple controller having 3 inputs (a, b, c) and 2 outputs (y, z). Note that, in a burst-mode specification, only *specified* input changes may occur. For example, in Figure 2, input change $a-$ is disallowed in state B, since it is not described by the specification.

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 [42]. 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* [42]. When the machine is stable, the cycle is complete and the machine is ready to receive new inputs. Throughout the entire machine cycle, outputs and state variables must be free

4
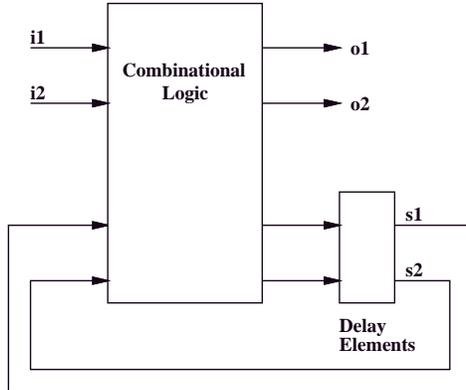
Figure 3: Block diagram of Huffman machine.

of glitches.

2.3   Previous Work

The state encoding problem for asynchronous machines has been studied by researchers for over 30 years. Several methods have been proposed for minimal-length critical race-free assignment, which ignore logic complexity and hazards: Liu [20] and Tracey [41] targeted USTT codes, while Saucier [37] and Datta et al. [6] achieved shorter codes with non-STT assignments. Tan [40] and Saucier [38] proposed race-free STT encoding algorithms which heuristically minimize next-state logic; however, the former only provides a hazard-free implementation for SIC operation, and both ignored output logic.

More recent work by Fisher et al. [10] seeks race-free (but non-STT) assignments for large machines, again heuristically minimizing code length while ignoring logic complexity. Finally, Lam et al. [15] present a greedy algorithm which reduces the number of state bits, but only for a limited class of delay-insensitive circuits.

3   Overview of Synthesis Strategy

3.1   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.*

Optimal synchronous assignment methods are inadequate, not only because they do not consider critical races and transient behavior, but because they do not target a hazard-free implementation.

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 (Step 3), a binary hazard-free minimizer is used to find a hazard-free implementation. A sketch of this formulation is described next.

3.2   Problem Formulation

A burst-mode specification is first transformed into an unminimized, or *primitive*, flow table [28]. An example primitive flow table, indicating *specified transitions*, is shown in Figure 4. In [28], the problem of state minimization is then addressed (an alternative method is described in [46]). We assume the same state
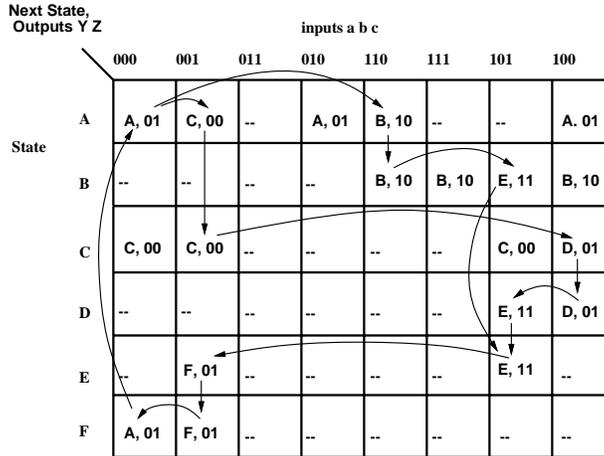
Figure 4: Example flow table

minimization strategy to obtain a *minimized* flow table that is guaranteed to have a hazard-free two-level logic realization. The interested reader is referred to these literature [42, 46, 28] for details.

Starting from the minimized flow table, we are confronted with the problem of encoding the minimized states with unique state codes that avoids *critical race* problems [41]. To ensure that an unstable state leads directly to a new stable state, a *unicode single-transition time (USTT)* assignment is used [42]. Such an assignment ensures that only one feedback cycle is required during a state change. In our target implementation, delays are added along the feedback path (cf. Figure 3) as needed to avoid *essential hazards* [42]. These delays separate an *input burst* (where the machine receives input changes) from a *state burst* (where the machine receives changes in its current state). As a result, the sequential synthesis problem is transformed into a combinational logic problem, for which there exists efficient hazard-free two-level minimizers [2, 30].

We now intuitively describe how the symbolic minimization and encoding problems are formulated and depict the overall trajectory. First, the minimized flow table is translated to a (multiple-output) *multiple-valued function,* $\mathbf{f}: B_m \times X \mapsto B^r$, with m binary input variables that correspond to the external inputs of the flow table and a single multiple-valued input (*mvi*) variable X that represents the possible states. The mvi-variable X can take on $n = |X|$ possible values corresponding to the number of possible states. The multiple-valued function can have only *binary output* values. This translation is analogous to the synchronous approach proposed by De Micheli [26] where the "next-state-output" is temporarily "one-hot" encoded to produce a binary-output multiple-valued function for symbolic minimization. Thus, the multiple-valued function $\mathbf{f}$ has $r = n + p$ binary outputs corresponding to the n possible next-states and the p external outputs of the flow table.

An important difference between a synchronous and asynchronous flow table is that in the latter, the dynamic behavior is specified. In particular, an asynchronous flow table has a set of *specified* input and state changes (implicit or explicit), for which its operation must be hazard-free (see Figure 4).

After the minimized flow table has been translated to a multiple-valued function, a hazard-free multiple-valued minimization step (*cf.* Section 5) is carried out to ensure that the cover for this multiple-valued function is free of all static and dynamic hazards for all specified transitions. When the multiple-valued function is extracted from a burst-mode specification, the possible (multiple-input change) transitions correspond exactly to the *input bursts* and *state bursts*. A further property inherited from a burst-mode specification is that the state burst transitions correspond to *static transitions*, meaning all outputs and the next-state remain unchanged. However, the input burst transitions may correspond to static or *dynamic transitions*, meaning that some external outputs and the next-state can change value. However, from the point of view of the hazard-free multiple-valued minimizer, this distinction is not important.

After the symbolic minimization step, a constrained encoding step (*cf.* Section 6) is performed to ensure the following:

6

1. critical race-free operation [41];

2. the hazard-free properties are preserved; and

3. the cardinality of the solution is preserved.

For the output logic, we show in Section 7 that it is always possible to find an encoding that ensures all three above properties. This means that the exact symbolic solution achieved in the symbolic minimization step can be directly translated to a binary cover of the same cardinality. Therefore, our solution framework indeed achieves an exact solution to the hazard-free minimization problem for the output logic *under all possible critical race-free encodings*. This is crucial since, for performance-critical applications, output logic usually determines the latency of an asynchronous machine. Also, it often represents a substantial part of the overall area. For the next-state logic, our approach only leads to an approximate solution for two reasons: First, we have made an approximation of the "next-state-output encoding" problem by assuming a "one-hot" next-state output. Second, some modifications are required to the cover *after* the expansion of encodings to ensure that all hazard-free properties are preserved. Both of these issues are discussed in later sections. The resulting instantiated hazard-free cover is then run through an exact hazard-free minimizer (with now only binary variables) in the final stage. Though the solution achieved for the next-state logic is not exact, in practice our method leads to high quality solutions.

4 Multiple-Valued Functions and Hazards

For the following, we assume basic familiarity with the terminology of multi-valued logic minimization (see [34]).

4.1 Circuit Model

This paper considers combinational circuits having arbitrary finite gate and wire delays (*unbounded wire delay model* [24]). A pure delay model is assumed [1].

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 [30] 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 $(A \neq B, C \neq D)$ such that (a) $B \in [A, D]$ and $C \in [B, D]$, and (b)$f(B) = f(D)$ and $f(A) = f(C)$.*

If a transition has a function hazard, no multiple-valued implementation of the function is guaranteed to avoid glitch on the transition, assuming arbitrary gate and wire delays. This can easily be seen by generalizing of the result shown by [9, 3]. Therefore, we consider only transitions which are free of function hazards.

4.4 Multiple-Valued Logic Hazards

If f is free of function hazards for a transition from input A to B, a corresponding *encoded implementation* may still have hazards due to possible delays in the actual logic realization [42, 3, 1]. 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*. This multiple-valued AND gate will become an ordinary *binary-valued AND gate* after a *constrained encoding step*, as described in Section 6. 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 $1 \to 0 \to 1$ ($0 \to 1 \to 0$) 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 $1 \to 0 \to 1 \to 0$ ($0 \to 1 \to 0 \to 1$) 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 $t \in T$.

8

5   Symbolic Hazard-Free Multiple-Valued 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 [34], 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 DHF-prime implicants** (DHF stands for *dynamic-hazard-free*). 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 [30].

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

Proof: In an AND-OR implementation, all gates are stable at 0 in a $0 \to 0$ transition. Thus, no spurious transitions are possible. □

Lemma 5.2 *If f has a $1 \to 1$ 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.*

Proof: $\Rightarrow$ If there is a cube in cover C that is ON during the entire transition $[A, B]$, then the output of the OR gate will be stable at 1 during the entire $1 \to 1$ transition.
$\Leftarrow$ Suppose it is possible for the circuit output to make a spurious $1 \to 0 \to 1$. Then it must be the case that at some moment during the $[A, B]$ transition all cubes have the value 0. This contradicts the statement that $[A, B]$ is contained in some cube of cover C. □

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

Lemma 5.3 *If f has a $1 \to 0$ 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.*

Sketch of Proof: This is an extension of results for the binary case from [42, 11, 30].
$\Rightarrow$ If every cube $c \in C$ intersecting $[A, B]$ also contains A, then all cubes that can be ON during the transition $[A, B]$ must all be *monotonic* in that they can make at most one $1 \to 0$ transition. If all cubes are monotonic, then the output of the OR gate will also be monotonic.
$\Leftarrow$ Suppose there is some cube $c \in C$ that intersects $[A, B]$ but does not contain A. Then this cube can make a $0 \to 1 \to 0$ transition during the transition $[A, B]$ under some delay assignment. Under some delay assignment, the other cubes making $1 \to 0$ transitions can all be turned OFF before c is turned ON. Therefore, the circuit output can make a spurious $1 \to 0 \to 1 \to 0$ transition. □

Lemma 5.2 requires that in a $1 \to 1$ 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 $1 \to 0$ 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 $1 \to 0$ transition from input state A to B which is hazard-free in the implementation, then, for every input state $X \in [A, B)$ where $f(X) = 1$, the transition subcube $[A, X]$ is contained in some cube of cover C.*

Proof: Since C is a cover of function f, there exists some cube $c \in C$ which contains X. Since f is hazard-free in the transition from A to B, then, by Lemma 5.3, cube c contains A as well; therefore c contains $[A, X]$. □

Lemma 5.5 *If f has a $1 \to 0$ 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 $1 \to 1$ transition from input state A to X is free of logic hazards.*

Proof: Immediate from Lemmas 5.2 and 5.4. □

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. It 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 $1 \to 1$ transition, and every maximal subcube $[A, X] \subset [A, B]$ where f is 1 and $[A, B] \in T$ is a dynamic $1 \to 0$ transition, is called a **required cube**.*

Lemma 5.3 constrains the cubes which may be included in a cover C. Each $1 \to 0$ 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. It 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 $1 \to 0$ 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 the cover, C; and*

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

Sketch of Proof: The result follows immediately from Lemmas 5.1– 5.5, and the definitions of hazard-free cover, required cube and privileged cube. □

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. [42, 1, 11]). 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 [35, 34], are based on the Quine-McCluskey algorithm [23]. The Espresso-MV-Exact algorithm solves the two-level multiple-valued minimization problem in three steps:

1. Generate the multiple-valued prime implicants of a function;

2. Construct a prime implicant table; and

3. Generate a minimum cover of this table.

Here, we extend an existing exact *hazard-free* two-level minimizer [30] 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 a *multiple-valued dynamic-hazard-free implicant* (or *multiple-valued DHF-implicant* for short). It is 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.*

Only multiple-valued DHF-implicants may appear in a hazard-free cover, by Theorem 5.1(c). 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 determine 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 the multiple-valued DHF-prime implicants of a function;

2. Construct a multiple-valued DHF-prime implicant table; and

3. Generate a minimum cover of this table.

These steps are detailed below.

5.4 Generation of Initial Sets

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

The sets are generated by a simple iteration through every specified transition of the given multi-output multiple-valued function using Algorithm **MVI-Make-Sets**. The algorithm is a generalization of an existing binary *Make-Sets* procedure [30] to the multi-valued case.

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

5.5 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 [30], but extended to multiple-value functions. First, multiple-valued prime implicants of f are generated from the req-set (which defines the on-set) and the off-set, using existing algorithms [34]. Second, the multiple-valued 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.

5.6 Generation of the Multiple-Valued 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* used to cover the columns. The columns are labelled with the *required cubes* which must be covered. The table sets up the two-level hazard-free logic minimization problem.

5.7 Generation of a Minimum Cover

The multiple-valued DHF-prime implicant table describes a standard unate covering problem. It can be be solved using an existing algorithm, *minimum-cover* [34].

5.8 Handling Multiple-Output Minimization

As in Espresso-MV-Exact [35, 34], *multiple-output functions* are handled by making the output parts into a single *N-value MV variable*, where N is the number of outputs. The transformation is straightforward and is described in [35, 34]. Using this transformation, the symbolic hazard-free multiple-valued minimization procedure can be used to minimize multiple-output functions.

6 Constrained Encoding for Asynchronous State Machines

We now consider the constrained encoding problem that must be solved to produce a realizable binary logic implementation for the minimized 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 our purposes for two reasons: (1) they do not consider the transient behavior of an asynchronous state machine, and (2) they do not consider hazard-free requirements. For the asynchronous case, the face embedding constraints must be generalized. We consider these two problems in turn.

A new condition concerns the correctness of the implementation of output and next-state functions 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` → `S4` in input column `I1` during which the binary output 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` → `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`. □

A similar problem occurs for the next-state function. In this case, the function 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 could have an incorrect implementation.

The proposed solution is to add dichotomy constraints to avoid such problems resulting from 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 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 relates to the need to avoid hazards. In KISS, face embedding constraints ensure that an implicant does not intersect an OFF-set minterm. 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.

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 the 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* [36]) and heuristic solution (using *nova*'s simulated annealing mode [45]). 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 [26], 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 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 [42], and therefore cannot be used. In fact, *no single encoding suffices for all asynchronous symbolic covers of N states*; that is, there is no natural canonical state assignment to use for cover instantiation. As an alternative, to demonstrate theoretical results, we propose the following: solve the encoding constraints and produce any critical race-free assignment. 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) $\langle input, present \rangle$ maps to the binary startpoint (endpoint) $\langle input, encoding(present) \rangle$.

---

[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.

Viewing the functional specification as a set of ON-set and OFF-set cubes, a symbolic product p (a 4-tuple $\langle input, present, next, output \rangle$), maps onto a binary product $\tilde{p}$, as follows:

$$
\begin{array}{ccccc}
p: & input & stategroup & next & output \\
& \Downarrow & \Downarrow & \Downarrow & \Downarrow \\
\tilde{p}: & input & supercube(encodings(stategroup)) & encoding(next) & output
\end{array}
$$

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>`.

In 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 filled in as well. This latter property can easily be guaranteed by the following constraint on the symbolic specification: a *single product* must be used to specify function values for each column transition. This constraint ensures that, for each column transition (*i.e.,* state change), this product will be instantiated to define 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 same 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:
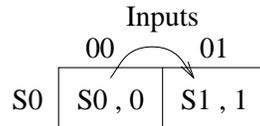
No implicant in the symbolic cover covers the entire transition, since both output and next-state undergo dynamic transitions. In particular, the next-state function S_0 has a $1 \rightarrow 0$ transition and the next-state function S_1 has a $0 \rightarrow 1$ transition (as does the output). However, 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 for next-state. 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 specified input 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 [26]. In practice, shorter codes are sought, and the instantiated cover is likewise re-run through a binary minimizer to improve results [26, 44].

In both KISS and our method, an input encoding formulation and solution yield only approximations to optimal state assignment. In practice, though, both methods can result in significant improvements (see Section 9).

7.2   Correctness of Binary Cover

Due to space limitations, some of the following proofs are only sketched. In what follows, we denote the symbolic flow table by M_S, and the encoded flow table by M_E. Let ns_i denote the symbolic next-state function i, and es_j the encoding of present-state state j. Likewise, $es_j[i]$ is bit i of that encoded state. A transition t is sometimes described by a product (i.e. the supercube of its endpoints); $input(t)$ and $present(t)$ refer to the input and present state fields, respectively, of the transition cube.

We first prove that the binary cover is a functional implementation of the encoded flow table.

Lemma 7.1 *Cube containment is preserved by instantiation. That is, if symbolic cube c_1 contains c_2, then mapped cube $\tilde{c}_1$ contains $\tilde{c}_2$.*

Theorem 7.1 *The binary implicants of cover C contain the entire ON-set of encoded machine M_E.*

Proof Sketch. By definition, all of M_E's ON-set points lie within the specified transitions for M_E, which are simply M_S's transitions, mapped 1-for-1. In fact, they lie within the required cubes of M_E. We proceed by treating binary outputs and state bits separately.

Part 1: Binary outputs. M_E's required cubes for binary outputs are precisely M_S's required cubes, mapped 1-for-1. Thus, the ON-set points for M_E's binary outputs lie within mapped symbolic required cubes. Now, the symbolic hazard-free cover contains implicants to cover all symbolic required cubes. Hence, by Lemma 7.1, all binary output ON-set points are covered.

Part 2: State bits s_i. We show that all s_i's required cubes are covered. First, note that the ON-set of s_i is $\bigcup_{j|es_j[i]=1} map(ONset(ns_j))$. M_E's required cubes for s_i include those cubes ($\mathcal{R}'$) added to cover static-1 hazards introduced by 1-bit sharing: $ReqCubes(s_i) = \bigcup_{j|es_j[i]=1} map(ReqCubes(ns_j)) \bigcup \mathcal{R}'$. The symbolic cover $\mathcal{S}$ contains implicants to cover all symbolic required cubes, including $\mathcal{R}'$. Since instantiation replaces the next-state field ns_j of implicant p with its encoding es_j, the binary implicant $\tilde{p}$ contributes to state bits s_i for which $es_j[i] = 1$. Thus all *mapped* required cubes for s_i are covered. Finally, we added products as needed to cover the added required cubes $\mathcal{R}'$. □

Theorem 7.2 *No binary implicant of C intersects the OFF-set of encoded machine M_E.*

Proof Sketch. We proceed by showing that no OFF-set minterm is contained. Note that all OFF-set points of M_E lie within specified transitions. Canonical cover C consists of: (i) instantiated symbolic products; and (ii) products to cover required cubes $\mathcal{R}'$ due to new static-1 state bit transitions in M_E. Recall that products in $\mathcal{R}'$ cover precisely and only ON-set points in M_E, and implement only the state bits involved in such transitions. In what follows, let m be any OFF-set minterm of some output o or state bit s_i, lying within a transition t. We show that m is not contained in any implicant $\tilde{p}$ of C.

Case 1: Binary outputs o. For each implicant p, either: (i) $o \notin output(\tilde{p})$ (i.e. $\tilde{p}$ does not implement o); or (ii) the input fields of $\tilde{p}$ and t do not intersect; or (iii) $\tilde{p}$ implements o and $input(\tilde{p}) \cap input(t)$. Now, either: $o = 1$ during t (a contradiction, since $m \in t$ is in the OFF-set of o), or $o = 0$ during t. If so, we generate an encoding constraint to prevent intersection of $present(t)$ and $present(\tilde{p})$ (Section 6). Therefore, $\tilde{p}$ does not contain m.

Case 2: State bits s_i. The OFF-set of s_i is $\bigcup_{j|es_j[i]=0} map(ONset(ns_j))$. Symbolic implicants implement only 1 next-state. Hence, instantiated implicants implement only state bits s_i for which $es_j[i] = 1$. For each implicant $\tilde{p} \in C$, either: (i) $s_i \notin output(\tilde{p})$ (i.e. $\tilde{p}$ does not implement s_i); or (ii) the input fields of $\tilde{p}$ and t do not intersect; or (iii) $\tilde{p}$ implements s_i and $input(\tilde{p}) \cap input(t)$. Now, since $\tilde{p}$ implements s_i, $next(p) = ns_j$, for some j such that $es_j[i] = 1$. The value of ns_j during t is either 0 or 1. If 1, we have a contradiction: $m \in t$ is in the OFF-set of s_i, but $ns_j = 1$ in t implies $s_i = 1$ there. If 0, then $present(\tilde{p})$ does not intersect

$present(t)$, since we generated an encoding constraint to avoid it (Section 6). Therefore, $\tilde{p}$ does not contain m. □

We next prove that our generated encoding constraints *subsume* all critical race-free constraints [41].

Theorem 7.3 *Any state assignment satisfying the encoding constraints of Section 6 is critical race-free.*

Proof. (See also [12].) In each flow-table input column, I_1, critical race-free constraints are needed to avoid interference between an unstable transition and either stable states (case 1), or other unstable transition (case 2).

Case 1: I_1 contains an unstable transition $t_1 : S_0 \to S_1$, and a distinct stable state S_2. To avoid a critical race, a *2-to-1* dichotomy constraint $d_1 = \{S_0, S_1; S_2\}$ must be satisfied [41]. At the same time, in our framework, the unstable transition t_1 defines a symbolic required cube for next-state S_1, which must be covered. Therefore, some symbolic implicant, p, must cover the transition and implement the destination next-state function, S_1. Hence, $stategroup(p) \supseteq \{S_0, S_1\}$. Now, S_2 is stable in I_1, and so the next-state function S_1 is 0 there. Therefore, our encoding constraints include $\{stategroup(p); S_2\}$, which subsumes dichotomy d_1.

Case 2: I_1 contains unstable transitions $t_1 : S_0 \to S_1$ and $t_2 : S_2 \to S_3$, where $S_1 \neq S_3$. In this case, a *2-to-2* dichotomy constraint $d_2 = \{S_0, S_1; S_2, S_3\}$ must be satisfied [41]. Again, each unstable transition defines a symbolic required cube. Therefore, some implicant p covers transition t_1 and implements next-state S_1, so $stategroup(p) \supseteq \{S_0, S_1\}$. Meanwhile, next-state function S_1 is 0 throughout transition t_2. Hence, our encoding constraints include $\{stategroup(p); S_2, S_3\}$, which subsumes dichotomy d_2. □

Theorem 7.4 *The cover C is hazard-free for every specified input and state transition.*

Proof Sketch. A hazard-free implementation requires that: (i) each required cube is contained in some implicant of B; and (ii) no implicant of B illegally intersects any privileged cube [30]. Condition (i) was shown to hold in Theorem 7.1. For condition (ii), there is a one-to-one mapping between symbolic and binary privileged cubes. Further, symbolic hazard-free minimization ensures that no illegal intersections occur in the symbolic cover. This property is maintained in the binary cover, due to the encoding constraints of Section 6. □

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).

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.*

Proof Sketch (by contradiction). Assume some hazard-free binary cover $\widetilde{\mathcal{OC}}$ exists which has smaller cardinality than $\mathcal{OC}$. Each implicant in $\widetilde{\mathcal{OC}}$ is dhf-prime, and corresponds to a symbolic dhf-prime in the symbolic cover (since primality is preserved by instantiation). The number and types of output transitions are also preserved by instantiation, and there is thus a one-to-one mapping between required cubes for M_S and M_E. Therefore, there must exist a corresponding symbolic hazard-free cover of size $|\widetilde{\mathcal{OC}}|$. However, this is impossible, since $\mathcal{OC}$ corresponded to an exactly minimal symbolic cover of cardinality $> |\widetilde{\mathcal{OC}}|$. □

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 assignments*.

8 Program Implementation

We have implemented our algorithms for symbolic hazard-free logic minimization, constraint generation, and constraint solution.

Unlike a previous hazard-free minimizer [30], which handled only single-output binary-valued functions, our new minimizer provides exact minimization of *multi-valued* functions. It therefore performs exact minimization of *multi-output* binary-valued functions as a special case.

The new minimizer makes use of *mincov* [34] for efficient solution of the minimum covering problem (the former minimizer did not). An added benefit is that mincov offers a *heuristic mode*, which allows us to handle large hazard-free minimization problems. Our constraint solution mechanism supports both exact (via *dichot*) and heuristic (via *nova*'s simulated annealing mode) solutions. Unlike standard *nova*, however, our modified simulated annealing algorithm accepts fully general dichotomies and handles both *compulsory* and *non-compulsory* constraints.

The program is implemented in portable C++, running on NeXT, Sun Sparc, and IBM RS/6000. It is currently integrated into the unclocked asynchronous sequential synthesis system UCLOCK [28], and can easily be integrated into other synthesis systems, e.g. 3D [46], MEAT [7], or LCLOCK [29][3]. Where available, we make use of existing highly optimized tools for certain steps, e.g. prime implicant generation (*espresso-mv* [34]), exact and heuristic unate covering (*mincov* [34]), dichotomy solution (*dichot* [36]) and partial encoding constraint satisfaction via simulated annealing (*nova* [45]).

9 Experimental Results

A preliminary set of experiments was run on industrial examples ([7, 21, 32, 47]) using our optimal encoding and logic minimization algorithms. Results appear in Figure 9. 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. 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.

References

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

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

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

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

[5] 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.

[6] P.K. Datta, S.K. Bandyopadhyay, and A. K. Choudhury. A graph theoretic approach for state assignment of asynchronous sequential machines. *International Journal of Electronics*, 65(6):1067–1075, 1988.

[7] A. Davis, B. Coates, and K. Stevens. Automatic synthesis of fast compact self-timed control circuits. In *1993 IFIP Working Conference on Asynchronous Design Methodologies (Manchester, England)*, 1993.

[8] S. Devadas, H.-K. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State assignment of finite state machines targeting multi-level logic implementations. *IEEE Transactions on CAD*, CAD-7(12):1290–1300, December 1988.

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

[10] P.D. Fisher and S.-F. Wu. Race-free state assignments for synthesizing large-scale asynchronous sequential logic circuits. *IEEE Transactions on Computers*, 42(9):1025–1034, September 1993.

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

[12] 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]though this system would benefit only from the hazard-free, not the critical race-free, nature of the solution

DESIGN	I/S/O	opt-fixed		optimal		base-crf	
		bits	cubes	bits	cubes	bits	cubes
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	-[a]	4	55
stetson-p2	8/13/12	4	31	10	37	4	36

Figure 5: Experimental Results

[a]Exact logic minimization failed due to insufficient virtual memory in prime generation

[13] K. Keutzer, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis for testability techniques for asynchronous circuits. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*, November 1991.

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

[15] P.N. Lam, H.F. Li, and S.C. Leung. Optimization of state encoding in distributed circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(5):581–588, May 1994.

[16] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 302–308. Association for Computing Machinery, June 1991.

[17] B. Lin and S. Devadas. Synthesis of hazard-free multi-level logic under multiple-input changes from binary decision diagrams. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design*, November 1994.

[18] B. Lin and A. R. Newton. Synthesis of multiple level logic from symbolic high-level description languages. In *IFIP Conference on VLSI*, pages 187–196, August 1989.

[19] B. Lin, C. Ykman-Couvreur, and P. Vanbekbergen. A general state graph transformation framework for asynchronous synthesis. In *Proceedings of the 1994 European Design Automation Conference*, September 1994.

[20] C.N. Liu. A state variable assignment method for asynchronous sequential switching circuits. *JACM*, 10:209–216, April 1963.

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

[22] A.J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Institute on Concurrent Programming, pages 1–64. Addison-Wesley, Reading, MA, 1990.

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

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

[25] 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.

[26] 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.

[27] S. M. Nowick and D. L. Dill. Synthesis of asynchronous state machines using a local clock. In *ICCD-1991*.

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

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

[30] S.M. Nowick and D.L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. In *ICCAD-1992*.

[31] S.M. Nowick, N.K. Jha, and F. Cheng. Synthesis of asynchronous circuits for stuck-at and robust path delay fault testability. In *Proceedings of VLSI Design 95*, January 1995.

[32] S.M. Nowick, K.Y. Yun, and D.L. Dill. Practical asynchronous controller design. In *ICCD-1992*.

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

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

[35] Richard Rudell. Logic synthesis for VLSI design. Technical Report UCB/ERL M89/49, Berkeley, 1989.

[36] 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.

[37] G. Saucier. State assignment of asynchronous sequential machines using graph techniques. *IEEE Transactions on Computers*, C-21(3):282–288, March 1972.

[38] Gabriele Saucier. Next-state equations of asychronous sequential machines. *IEEE Transactions on Computers*, EC-21(4):397–399, April 1972.

[39] 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.

[40] C.-J. Tan. State assignments for asynchronous sequential machines. *IEEE Transactions on Computers*, C-20(4):382–391, April 1971.

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

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

[43] Kees van Berkel. Handshake circuits: an intermediary between communicating processes and VLSI. Technical report, Eindhoven Institute of Technology, 1992. Ph.D. Thesis.

[44] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment of finite state machines for optimal two-level logic implementations. In *Design Automation Conference*, pages 327–332, June 1989.

[45] 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.

[46] Kenneth Yun, David Dill, and Steven M. Nowick. Synthesis of 3D asynchronous state machines. In *ICCD-1992*.

[47] K.Y. Yun and D.L. Dill. Automatic synthesis of 3D asynchronous finite-state machines. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society Press, November 1992.