Synthesis for Logical Initializability of Synchronous Finite State Machines*

Montek Singh
Steven M. Nowick

Department of Computer Science
Columbia University
1214 Amsterdam Ave: Mailcode 0401
New York, NY 10027

{*montek,nowick*}*@cs.columbia.edu*

Technical report: **CUCS-002-98**
January 15, 1998

*This work was supported by an NSF CAREER Award MIP-9501880 and by an Alfred P. Sloan Research Fellowship. This work is an extended version of a conference paper with the same title, that appeared in the proceedings of the 10th International Conference on VLSI Design, Jan 4-7, 1997 [21].

Abstract

A new method is introduced for the synthesis for logical initializability of synchronous state machines. The goal is to synthesize a gate-level implementation that is initializable when simulated by a 3-valued (0,1,X) simulator. The method builds on an existing approach of Cheng and Agrawal, which uses constrained state assignment to translate functional initializability into logical initializability. Here, a different state assignment method is proposed which, unlike the method of Cheng and Agrawal, is guaranteed safe and yet is not as conservative. Furthermore, it is demonstrated that certain new constraints on combinational logic synthesis are both necessary and sufficient to insure that the resulting gate-level circuit is 3-valued simulatable. Interestingly, these constraints are similar to those used for hazard-free synthesis of asynchronous combinational circuits. Using the above constraints, we present a complete synthesis for initializability method, targeted to both two-level and multi-level circuits.

Keywords: logic-simulation, initializability, design, state-transition-graph, finite-state machines, test, tradeoffs, logic synthesis.

1 Introduction

Initializability is a property of a circuit by virtue of which the circuit can be driven to a unique known state, irrespective of the startup state. There are several reasons why initializability is a desirable property. Initializability is needed in order to physically reset machines if they get out of synchronism. Furthermore, a form of initializability called *logical initializability* is *required* for several fault simulators and non-scan automatic test pattern generators (ATPG's) to work effectively. Examples of such ATPG's include STG [2] and CONTEST [1].

The notion of initializability is tightly tied to the model used to simulate the machine. For example, this model could be a functional simulation of an abstract state machine, a 3-valued (0,1,X) logical simulation of a gate-level circuit, or a true-value (0,1) simulation of a gate-level circuit. True-value simulation can be prohibitively expensive since the set of initial states of the machine is often huge. Hence, from the perspective of initializability, since the initial state is unknown, true-value simulation is not very useful, and will not be considered in this paper.

A finite state machine that is initializable by a series of inputs when *functionally simulated*, is said to be *functionally initializable*. Functional simulation keeps track of all the symbolic states the state machine can be in at any time, when subjected to a series of inputs. This series of inputs that initializes the state machine is called its *synchronizing sequence* or *initialization sequence*. A synchronizing sequence may be composed of a single-input vector (*single-vector synchronizing sequence*) or multiple-input vectors (*multi-vector synchronizing sequence*). Similarly, a gate-level circuit that is initializable under a series of inputs when simulated by a 3-valued simulator is said to be *logically initializable*. Therefore, logical initializability refers to initializability under 3-valued simulation. The difference between functional and 3-valued simulation is that while the former uses sets of symbolic states to simulate the machine, the latter works with 3-valued vectors to keep track of the possible states of the machine. Thus, logical initializability of the gate-level circuit requires that the underlying finite state machine be functionally initializable. In this paper, the focus is on synthesis for multi-vector logical initializability.

Traditionally, several different approaches to initializability have been used. Each of these assumes a model of initializability (such as single- or multi-vector) and of simulation (such as functional or 3-valued). Further, while some methods only *analyze* a machine description to search for initialization sequences, other methods go a step further and *synthesize* for initializability. For example, the method of Wehbeh and Saab [3] analyzes the gate-level circuit to determine if it is logically initializable. On the other hand, the method of Cheng and Agrawal [4, 5] attempts to synthesize a logically initializable gate-level circuit from a functionally initializable finite state machine (FSM).

It is well known that state encoding can affect the logical initializability of a finite state machine implementation [4, 5]. If the sole objective of an "optimal" state assignment is to minimize the amount of logic, one may end up with implementations that are logically uninitializable. That is, a logic (3-valued) simulator may not be able to initialize the gate-level circuit *even* when the underlying FSM has a synchronizing sequence. Furthermore, as is shown later, unrestrained combinational logic minimization can adversely impact logical initializability. Therefore, any sound synthesis method for logical initializability must address two issues: *state assignment* and *combinational logic synthesis.*

1.1 Contributions of this paper

In this paper, we present a new synthesis method which provides for both (i) constrained state assignment, and (ii) constrained combinational logic synthesis. The method is the first systematic approach for synthesis-for-logical-initializability which addresses both of these issues.

State Assignment. In previous work [4, 5], a method was proposed for state assignment for initializability. We demonstrate that the constraints on state assignment imposed by that method are neither necessary nor sufficient. The contribution of this paper towards state-assignment-for-initializability is two-fold: (i) we identify where the constraints of [4, 5] can be easily and safely relaxed, and (ii) we identify where additional constraints are needed (irrespective of whether or not the constraints of [4, 5] were relaxed). The new set of constraints used by our state assignment method is *sufficient*; we *guarantee* that our method always produces a state assignment that allows one to synthesize a logically initializable circuit.

Combinational Logic Synthesis. In [5], it was suggested that combinational logic synthesis influences logical initializability, and a synthesis-for-initializability method was proposed. Here, we show that the logic synthesis technique surmised in [5] is not adequate. The contribution of this work towards combinational-logic-synthesis-for-initializability is the following:

(i) we propose both *necessary and sufficient* constraints on combinational logic to guarantee logical initializability, (ii) we present a 2-level logic minimization method that incorporates these constraints, and (iii) we characterize multi-level transformations that can be used to synthesize initializable multi-level logic. Interestingly, these new constraints are similar to hazard-free constraints used in the synthesis of asynchronous combinational circuits.

In summary, given a functionally initializable finite state machine, our synthesis method provides a complete synthesis path that produces a gate-level circuit that is guaranteed to be logically initializable.

1.2 Organization

The paper is organized as follows. Section 2 summarizes previous work on initializability. Section 3 reviews in detail an existing synthesis-for-initializability method that was used as the starting point for our research. Section 4 provides a short overview of our synthesis method. Section 5 presents details of the state assignment step of our method, and Section 6 presents details of the combinational logic synthesis step. Finally, results on a set of benchmark examples are presented in Section 7, and Section 8 gives conclusions.

2 Previous Work

A typical synthesis path consists of several steps (see Fig. 1). Initializability considerations can be incorporated at various levels. The figure is labeled to show some of the recent work on initializability targeting different levels in the synthesis path.

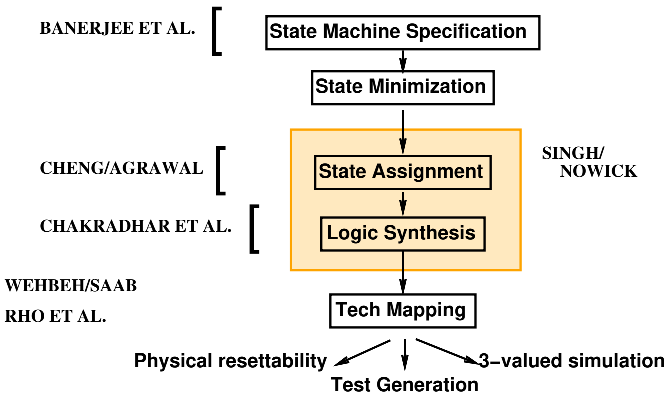

Figure 1: Synthesis for initializability

Banerjee *et. al.* [8] present a technique that targets the highest level in the synthesis path: the top-level functional specification (signal transition graph). The idea is to modify the signal transition graph *specification* to insure functional initializability for an asynchronous circuit specification. However, initializability is achieved only at the cost of some reduction in concurrency.

Cheng and Agrawal [5] target the *state assignment* step in an attempt to produce logically initializable circuits from functionally initializable specifications. This method is applicable to synthesis of synchronous state machines.

The method of Chakradhar *et. al.* [7], targets the *combinational logic synthesis* step for initializability. This method is essentially a search procedure for finding initialization sequences and concomitant don't-care assignments in order to synthesize initializable asynchronous circuits.

Each of the above methods focuses on *synthesis* for initializability. The following methods are *analysis* techniques to find initialization sequences given a circuit description.

Rho *et. al.* [6] analyze a functional description of a state machine, in the form of a state transition graph, and identify *functional* initialization sequences, if any exist. This method uses BDD's [9] and produces minimum-length initialization sequences.

Wehbeh and Saab [3] present a method which determines if a gate-level implementation is initializable. This method is able to generate both *functional* and *logical* initialization sequences from a given gate-level circuit.

In this paper, we present a new procedure for synthesis of initializable synchronous circuits. Our procedure takes as input a functionally initializable finite state machine, and produces a logically initializable gate-level circuit. We provide both (i) a state assignment step, and (ii) a combinational logic synthesis step, both of which are shown to be critical to logical initializability.

From among the previous work just cited, the one that comes closest to our work is that of Cheng and Agrawal [5]. However, their method does not provide for a combinational logic synthesis step specifically constrained for initializability. Moreover, the state assignment step provided by their method is not always correct as far as initializability is concerned. Hence, their method may not always yield initializable circuits.

3 Background

The Cheng–Agrawal Method

This section reviews the Cheng and Agrawal state assignment method [4, 5]. Given a finite state machine and a synchronizing sequence, the basic approach of the method is to constrain the state encoding step to insure logical initializability.

An example first shows how state encoding can affect logical (3-valued) initializability.

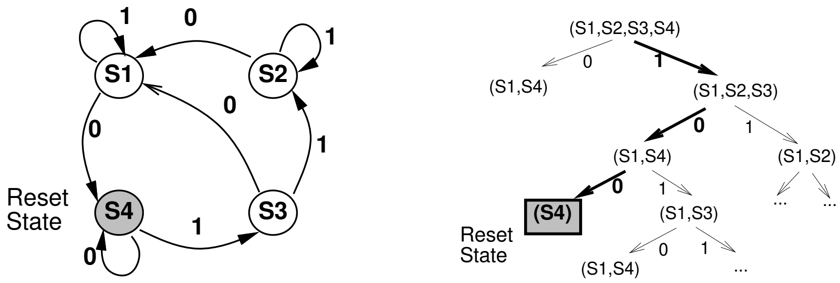

Figure 2: Example FSM and synchronization tree.

Example 3.1. Consider the functionally initializable machine M in Fig. 2. At startup, the machine can be in any state: S_1 or S_2 or S_3 or S_4. The term *state group* refers to a set of states. Thus, the initial state group of the machine is written as $(S_1S_2S_3S_4)$. When the series of inputs $1 \longrightarrow 0 \longrightarrow 0$, simply written as 100, is applied to the machine, the machine is driven to a unique known state, S_4, irrespective of the initial state. Therefore, $I = 100$ is called a *synchronizing sequence* of M. The following is the trace of state groups, or *state group sequence*, that results as the input sequence 100 is applied to M:

$$(S_1S_2S_3S_4) \xrightarrow{1} (S_1S_2S_3) \xrightarrow{0} (S_1S_4) \xrightarrow{0} (S_4) \tag{1}$$

Therefore, the machine is functionally initializable.

Associated with each state group, after state assignment, is its smallest containing cube, or *group face*. Each group face is represented by a 3-valued vector. Thus, if state encoding $(S_1 : 00,\ S_2 : 01,\ S_3 : 11,\ S_4 : 10)$ were used, the group face corresponding to (S_1S_2) would be $0X$, the group face corresponding to (S_4) would be 10, and the one corresponding to (S_1S_3) would be XX. The *group face sequence* is the trace of group faces that results when the series of inputs is applied. Thus, for the machine in the above example, the group face sequence is:

$$XX \xrightarrow{1} XX \xrightarrow{0} 1X \xrightarrow{0} 10.$$

Since the 3-valued simulation converges, the resulting implementation is logically initializable.

□

Example 3.2. Assume the state encoding ($S_1 : 00$, $S_2 : 01$, $S_3 : 10$, $S_4 : 11$) is used for the machine of Example 3.1. instead. In this case, the group face sequence is:

$$XX \xrightarrow{1} XX \xrightarrow{0} XX \xrightarrow{0} XX.$$

This sequence does not converge to a single state, therefore the circuit realized here is *logically uninitializable.* □

Example 3.2 demonstrated the impact of state assignment on logical initializability — unconstrained state encoding can render circuits uninitializable by a 3-valued simulator, even though the state group sequence functionally converges to a unique state. This problem is due to the fact that a 3-valued simulator can only simulate group faces, not state groups; there is a loss of information during 3-valued simulation.

The goal of the Cheng and Agrawal method is to produce a state assignment that allows the sequence of group faces to "track" the sequence of state groups, and therefore insure logical initializability. To this end, the method introduces an additional set of *face-embedding constraints* into the state assignment step. Constraints are in the form of *dichotomies* [13, 14].

A dichotomy constraint, or simply dichotomy, is written as $(X; Y)$, where X and Y are disjoint sets of states. The constraint $(X; Y)$ is the stipulation that the smallest containing cubes of X and Y, after state encoding, do not intersect. This dichotomy constraint is satisfied by a state encoding if some state bit has the value 1 for all states in X and the value 0 for all states in Y, or vice versa. If the cardinality of the set X is n and the cardinality of Y is k, then the constraint $(X; Y)$ is called a *type $n \rightarrow k$ dichotomy.*

The constraints of Cheng and Agrawal are type $n \rightarrow 1$ dichotomies, also called *face embedding constraints.* If a face embedding constraint has a left side consisting of only a singleton state, then the constraint is called a *trivial* face embedding constraint. An $n \rightarrow 1$ dichotomy of the form $(G_i; s_j)$ is introduced for every symbolic state s_j not present in the state group

G_i in the state group sequence. *That is, a symbolic state that does not belong to a state group is forbidden from being embedded in its group face,* after state encoding. This requirement applies to all state groups encountered when a synchronizing sequence is applied to the machine.

Given a functional initialization sequence

$$G_1 \xrightarrow{I_1} G_2 \xrightarrow{I_2} \cdots \xrightarrow{I_n} G_{n+1},$$

the Cheng-Agrawal face-embedding constraints *(FEC's)* can be formalized as follows:

$$\text{Cheng-Agrawal } \mathbf{FEC} = \{\ (G_i; s_j) \mid s_j \notin G_i\}\,.$$

Example 3.3. Given the finite state machine of Fig. 2, and the synchronizing sequence 100 (Eqn 1), the non-trivial Cheng-Agrawal face embedding constraints are: $(S_1S_2S_3; S_4)$, $(S_1S_4; S_2)$ and $(S_1S_4; S_3)$. A state assignment satisfying these dichotomies is: (S_1 : 000, S_2 : 010, S_3 : 001, S_4 : 100). Observe that the first state bit satisfies dichotomy $(S_1S_2S_3; S_4)$: the bit is 0 for S_1, S_2 and S_3, but is 1 for S_4. Similarly, the other two state bits satisfy the remaining dichotomies. Thus, this state encoding insures that the state code for S_4, 100, is not embedded in the group face of $(S_1S_2S_3)$, $0XX$, during the first time step in 3-valued simulation: $XXX \xrightarrow{1} 0XX$. Fig. 3 shows graphically the state group sequence and the corresponding group face sequence after 3-valued simulation. 3-valued simulation converges to the correct value, 100: $\quad XXX \xrightarrow{1} 0XX \xrightarrow{0} X00 \xrightarrow{0} 100$.

□

4 New Synthesis-for-Initializability Method: Overview

The technique presented in this paper is a synthesis method for logical (3-valued) initializability. Given a finite state machine and a synchronizing input sequence, our method consists of the following two steps:

Step #1: constrained state assignment

(a) generate relaxed face embedding constraints *(RFEC's)*

(b) generate don't-care intersection constraints *(DCIC's)*

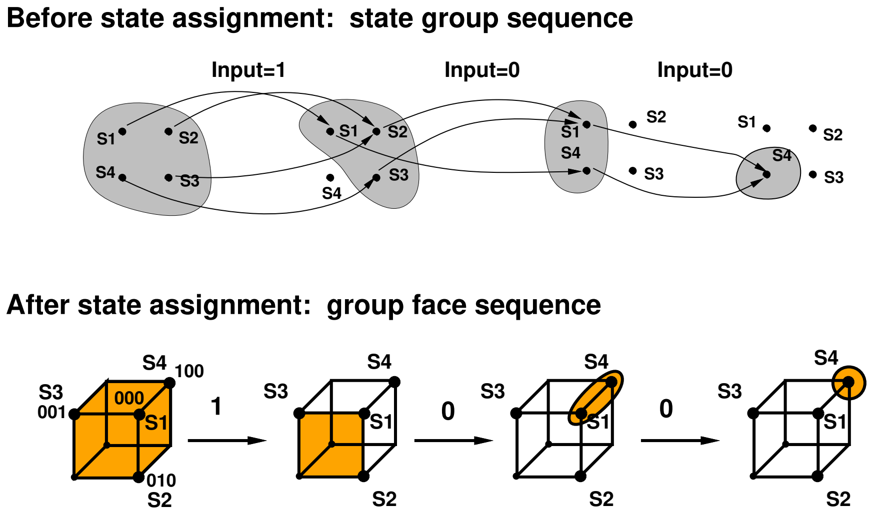

Figure 3: Groups faces "track" state groups

Step #2: constrained combinational logic synthesis

In Step #1 (a), we introduce relaxed face-embedding constraints. In Step #1 (b), we enumerate additional constraints that allow us to guarantee that the state assignment produced will enable logical initializability. These additional constraints, called *don't-care intersection constraints*, are shown to be critical to initializability.

Step #2 is also critical to achieving logical initializability. In this step, we formulate precise conditions on 2-level and multi-level logic to guarantee initializability, and present a logic synthesis method that incorporates these constraints.

The next two sections present our method in detail.

5 Step #1: Constrained State Assignment

This section presents our new constrained state assignment step. Section 5.1 discusses face-embedding constraints. First, it is shown how existing face-embedding constraints are overly restrictive. Then, it is shown how those constraints can be relaxed, yielding our *relaxed face-embedding constraints, or RFEC's.*

In Section 5.2, it is shown that face-embedding constraints, whether the original Cheng-Agrawal variety, or our relaxed version, by themselves are insufficient. Additional sufficiency constraints, called *don't-care intersection constraints (or DCIC's)*, are therefore formulated to guarantee initializability.

5.1 Step #1(a): Face embedding constraints

The following example illustrates how the Cheng-Agrawal constraints may be overly restrictive.

Example 5.1. Once again, consider the machine in Fig. 2. 100 is a synchronizing sequence for the machine, resulting in state group sequence:

$$(S_1S_2S_3S_4) \xrightarrow{1} (S_1S_2S_3) \xrightarrow{0} (S_1S_4) \xrightarrow{0} (S_4)$$

From Example 3.3, the dichotomy constraints produced by the Cheng-Agrawal method were:

$$\{\ (S_1S_2S_3; S_4),\ (S_1S_4; S_2),\ (S_1S_4; S_3)\ \}$$

Clearly, at least 3 state bits are required to satisfy all three constraints. However, a careful look at the state transition diagram of Fig. 2, shows us that, in fact, the dichotomy $(S_1S_2S_3; S_4)$ is unnecessary. Consider transition $(S_1S_2S_3) \xrightarrow{0} (S_1S_4)$. Note that state S_4 also has a transition on input 0 to S_4, which happens to belong to the next state-group, (S_1S_4). Therefore, *it is safe to let S_4 be embedded in the group-face of* $(S_1S_2S_3)$: even though S_4 is not part of the correct state group, $(S_1S_2S_3)$, S_4 also has a transition on the given input which drives it to the correct next state group (S_1S_4). We call this scenario a *safe embedding* of S_4 in $(S_1S_2S_3)$, and therefore can delete the dichotomy $(S_1S_2S_3; S_4)$. Thus, there is now a smaller set of dichotomy constraints to solve:

$$\{(S_1S_4; S_2), (S_1S_4; S_3)\}$$

Both of these constraints are satisfied by the following 2-bit state assignment: (S_1 : 00, S_2 : 01, S_3 : 11, S_4 : 10), still yielding a correct 3-valued simulation:

$$XX \xrightarrow{1} XX \xrightarrow{0} X0 \xrightarrow{0} 10.$$

In sum, by relaxing the set of face-embedding constraints, we have produced a shorter length state encoding (2 state bits instead of 3) that still insures logical initializability.[1] □

[1] In this example, we use 100 as the synchronizing sequence even though 00 is a shorter synchronizing sequence. However, the same problem can arise even starting with a minimum-length sequence.

Safe Embeddings

We now formally characterize safe embeddings. Given a finite state machine, M, and a functional initialization sequence, $G_1 \xrightarrow{I_1} G_2 \xrightarrow{I_2} \cdots \xrightarrow{I_n} G_{n+1}$. That is, G_i is the ith state group in the initialization sequence, and I_i is the input seen by G_i. Let NS(*current-state, input*) be the next-state function. An embedding of state s_j in the group face of state group G_i is *safe* whenever the transition out of s_j on the current input, $\mathrm{NS}(s_j, I_i)$, goes to a state in the specified next-state group, G_{i+1}; that is, whenever $\mathrm{NS}(s_j, I_i) \in G_{i+1}$. In this case,

$$G_i \xrightarrow{I_i} G_{i+1} \qquad \Longrightarrow \qquad (G_i \cup \{s_j\}) \xrightarrow{I_i} (G_{i+1} \cup \mathrm{NS}(s_j, I_i))$$

and, therefore, as desired, $(G_i \cup \{s_j\}) \xrightarrow{I_i} G_{i+1}$ if $\mathrm{NS}(s_j, I_i) \in G_{i+1}$.

The embedding is safe because, even if s_j is embedded within the group face of G_i, the 3-valued simulation for G_{i+1} will still converge to the same value as it would if s_j were not embedded in the group face of G_i.

Relaxed Face Embedding Constraints

Using the above notion, the list of Cheng-Agrawal FEC constraints can be pruned. The original Cheng-Agrawal face-embedding constraints (FEC's) were:

$$\text{Cheng-Agrawal } \mathbf{FEC} = \{\ (G_i; s_j) \mid s_j \notin G_i\}$$

Our new relaxed face-embedding constraints (RFEC's) are:

$$\mathbf{RFEC} = \{\ (G_i; s_j)\ \mid \mathrm{NS}(s_j, I_i) \notin G_{i+1}\} \tag{2}$$

It is easy to see that the RFEC constraints are a subset of the original Cheng-Agrawal constraints: $s_j \in G_i$ implies $\mathrm{NS}(s_j, I_i) \in G_{i+1}$.

5.2 Step #1(b): Don't-care Intersection Constraints

This section demonstrates that face-embedding constraints alone, whether the original Cheng-Agrawal variety or our relaxed version, *do not guarantee* a state assignment that allows logical initializability. Additional constraints, called *don't-care intersection constraints,* are therefore introduced to insure initializability. The need for don't-care intersection constraints, or

DCIC's, arises from the fact that the assignment of binary values to don't-care, or unspecified, next states is critical to the logical initializability of the state machine.

In what follows, it is first shown how don't-care *(DC)* assignment impacts logical initializability. Then, conditions on DC assignment are formulated to insure that DC assignment does not adversely affect logical initializability. It is shown how satisfiability of these conditions on DC assignment is critically dependent on state assignment. Finally, state assignment constraints (DCIC's) are introduced, which insure that DC assignment conditions can be satisfied.

Impact of DC Assignment on Logical Initializability

The following example illustrates how a careful DC assignment is critical to achieving initializability.

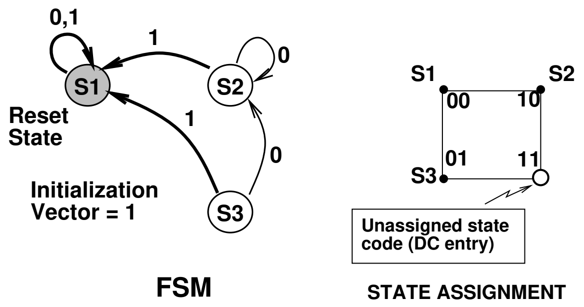

Figure 4: The issue of assignment to don't-care entries.

Example 5.2. Consider the state machine of Fig. 4. Applying the input vector 1 functionally initializes the machine to the unique reset state S_1. Thus, the machine has a single-vector initialization sequence: $I = 1$. The corresponding state group sequence is:

$$(S_1S_2S_3) \xrightarrow{1} (S_1)$$

The set of face-embedding constraints, whether the original Cheng-Agrawal FEC's, or our RFEC's, for this state group sequence is empty (the dichotomy constraints are trivial). Fig. 4 also shows a state encoding that trivially satisfies all the face-embedding constraints (since there are none). Two bits are used to encode the three states, (S_1 : 00, S_2 : 10, S_3 : 01). The fourth state code, 11, has no associated symbolic state. We call such a state code an *unassigned state code*, or a *non-symbolic state*. There are no specified next-state transitions for non-symbolic states; they are all *don't-care (DC) next-state transitions.* In addition, there may also be *symbolic states* having unspecified next state entries which are also don't-care next-state transitions.

In this example, the non-symbolic state 11 has a DC next state transition on input 1. This DC entry will eventually get assigned some value during some later stage in the synthesis path (*e.g.* during combinational logic synthesis). Suppose that this latter synthesis step fills in this DC entry for state 11 on input 1 with the next state value 11. In this case, the following group face sequence results from 3-valued simulation:

$$XX \xrightarrow{1} XX$$

.

The result is a logically uninitializable circuit. Simulation fails because the assigned next state transition from state 11 to 11, on input 1, lies *outside* of the group face of the destination state group, (S_1), thus throwing initialization off course. Therefore, if DC transitions can be assigned arbitrary values (during logic synthesis), a non-initializable circuit may result. □

Example 5.2 (contd.). To remedy this problem, let us now assign the DC next-state transition of 11 on input 1, the value 00 (corresponding to S_1). The following 3-valued simulation results:

$$XX \xrightarrow{1} 00$$

The circuit is now initializable. In this case, initializability is achieved by assigning to the DC next-state entry a value lying *within* the next group-face, 00. □

While in this example, a DC assignment can be applied to insure initializability, we show shortly that, given an arbitrary state assignment, this is not always the case.

Don't-Care Assignment for Initializability

In Example 5.2, we saw how sometimes initializability can be achieved by proper DC assignment. The key to proper DC assignment is to assign to every DC next-state entry in the current group face, a value that *lies within the next group-face.* More formally, let the machine M have the following initialization sequence:

$$G_1 \xrightarrow{I_1} G_2 \xrightarrow{I_2} \cdots G_i \xrightarrow{I_i} G_{i+1} \cdots \xrightarrow{I_n} G_{n+1}.$$

Let a state s (symbolic or non-symbolic) have a don't-care next-state entry on input I_i, *i.e.*, $\mathrm{NS}(s, I_i)$ = don't-care. Suppose the state code of s is embedded in the group face of state group G_i. Then, assigning the next-state $\mathrm{NS}(s, I_i)$ of s to lie *within the group face of* G_{i+1} will insure initializability. Such an assignment must be done for every such s and i.

Assuming StateCode(s) represents the binary state code of s, and GroupFace(G) represents the binary group face of the state group G, this condition can be written more formally as the following *tracking requirement:*

$$\forall s, i \qquad \mathrm{StateCode}(s) \in \mathrm{GroupFace}(G_i) \;\;\Rightarrow\;\; \mathrm{StateCode}\left(\mathrm{NS}(s, I_i)\right) \in \mathrm{GroupFace}(G_{i+1}) \quad (3)$$

This tracking requirement insures that during 3-valued simulation, GroupFace(G_i) is always followed by GroupFace(G_{i+1}), thus insuring initializability. By virtue of the definition of a synchronizing sequence, the final group face is guaranteed to be a singleton state.

Satisfying the Tracking Requirement

In essence, given an initialization input sequence, the tracking requirement of Eqn. 3 is a sufficient condition for logical initializability. However, given an arbitrary state assignment, satisfying the tracking requirement may not always be feasible. Here is an example where using the Cheng-Agrawal constraints (or our relaxed FEC's) results in a circuit that is uninitializable *for every assignment of don't-cares.* This happens because the tracking requirement imposes conflicting next-state assignments for some s and i.

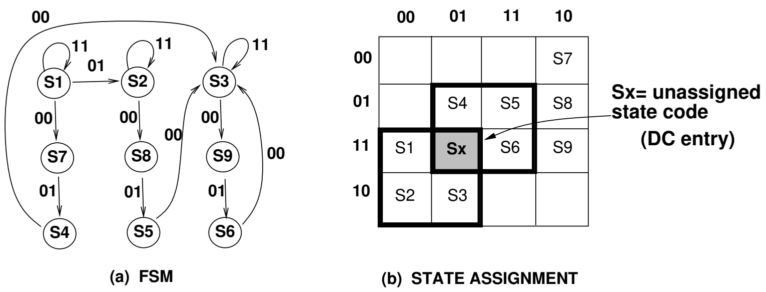

Figure 5: Example illustrating unsatisfiable tracking requirement.

Example 5.3. Consider the example of Fig. 5(a).[2] Applying a synchronizing sequence gives the following state groups:

$$(S_1S_2S_3S_4S_5S_6S_7S_8S_9) \xrightarrow{11} (S_1S_2S_3) \xrightarrow{00} (S_7S_8S_9) \xrightarrow{01} (S_4S_5S_6) \xrightarrow{00} (S_3).$$

A state assignment that satisfies all the Cheng-Agrawal FEC constraints, or our RFEC constraints, is shown in Fig. 5(b). Bit-vector 0111 is an unassigned state code, or non-symbolic state, which we call S_x. S_x has as yet unassigned next state transitions.

A careful analysis follows that shows that S_x *cannot be assigned any NS value* for input 00 while preserving initializability. For a moment, let us ignore any transitions out of S_x. If we now do a *true-valued* simulation, and collapse the result at every time step into a 3-valued vector, we get the following simulation trace (ignoring the effect of transitions out of S_x and other unassigned state codes):

$$XXXX \xrightarrow{11} 0X1X \xrightarrow{00} 10XX \xrightarrow{01} X1X1 \xrightarrow{00} 0110.$$

Ideally, we would like to obtain the same result for 3-valued simulation when the effect of transitions out of state S_x is included in the simulation. We begin by noting that the state S_x is embedded in the group-faces of $(S_1S_2S_3)$ as well as $(S_4S_5S_6)$. Both the groups have specified transitions in the initialization sequence on the *same input:* 00. The latter embedding mandates the NS value of S_x be set to S_3 in order to meet the tracking requirement (Eqn 3). However, the former embedding requires the NS value to be set to a state in the column containing $(S_7S_8S_9)$. Since the state groups (S_3) and $(S_7S_8S_9)$ are disjoint, *these two conditions are not simultaneously satisfiable.* That is, no next state DC assignment exists, for non-symbolic state S_x on input 00, which simultaneously satisfies both tracking requirements. Therefore, the result is always a logically uninitializable circuit.

In more detail, the embedding of S_x within $(S_4S_5S_6)$ mandates the next-state value of S_x on input 00 to be set to S_3. With this DC assignment, the 3-valued simulation trace is:

$$XXXX \xrightarrow{11} 0X1X \xrightarrow{00} XXXX \xrightarrow{01} XXXX \xrightarrow{00} XXXX.$$

The machine is not logically initializable. Observe that initializability gets derailed when the second state group, $(S_1S_2S_3)$, sees the input 00. The group face associated with $(S_1S_2S_3)$

[2]This state machine is incompletely-specified, but our analysis also applies to completely-specified machines.

is $0X1X$. The next state group, $(S_7S_8S_9)$, has an associated group face, $10XX$. However, the unassigned state code, 0111 (labeled S_x), which is embedded within the group face of $(S_1S_2S_3)$ has been assigned to next-state 0110 (S_3) on input 00, which lies outside of next group face, $10XX$. This uninitializes all the four state bits. If, instead, S_x were assigned a next state transition that was embedded within the third group face, $10XX$, as desired, then initialization proceeds normally at this step, but is now thrown off course at the last time step. For example, assuming now that $\mathrm{NS}(S_x, 00) = 1010$, the following 3-valued simulation results:

$$XXXX \xrightarrow{11} 0X1X \xrightarrow{00} 10XX \xrightarrow{01} X1X1 \xrightarrow{00} XX10.$$

Thus, conflicting don't-care assignment requirements can render a design uninitializable.
□

Don't-Care Intersection Constraints to Insure Satisfiability of the Tracking Requirement

Examples 5.2 and 5.3 demonstrate that don't-care assignment is critical to logical initializability, and therefore it cannot be left entirely to the later steps in the synthesis path. Example 5.2 showed that sometimes there is an assignment of don't-cares to meet the tracking requirement and, thus, make the circuit initializable. However, Example 5.3 showed that sometimes, *given an encoded machine, there may be no way to assign don't-cares to meet the tracking requirement.*

In this section, it is shown how the state assignment step itself can be modified so that the encoded machine *always* insures that a don't-care assignment is feasible, and thus allows the tracking requirement to be satisfied.

Once again, consider a machine with the initialization sequence:

$$G_1 \xrightarrow{I_1} G_2 \xrightarrow{I_2} \cdots G_i \xrightarrow{I_i} G_{i+1} \cdots G_j \xrightarrow{I_j} G_{j+1} \cdots G_n \xrightarrow{I_n} G_{n+1}.$$

Let a state s belong to two group faces corresponding to the state groups G_i and G_j:

$$\mathrm{StateCode}(s) \in \mathrm{GroupFace}(G_i)$$

$$\mathrm{StateCode}(s) \in \mathrm{GroupFace}(G_j)$$

Then the tracking requirement (Equation 3) is the following simultaneous pair of constraints:

$$\text{StateCode}\,(\text{NS}(s, I_i)) \in \text{GroupFace}\,(G_{i+1})$$

$$\text{StateCode}\,(\text{NS}(s, I_j)) \in \text{GroupFace}\,(G_{j+1})$$

The pair of constraint equations may be unsatisfiable if I_i and I_j are the same inputs values. That is, if $I_i = I_j$, the above conditions reduce to the following:

$$\text{StateCode}\,(\text{NS}(s, I_i)) \quad \in \quad \text{GroupFace}\,(G_{i+1}) \cap \text{GroupFace}\,(G_{j+1}) \tag{4}$$

In this case, the tracking requirement is unsatisfiable precisely when GroupFace (G_{i+1}) and GroupFace (G_{j+1}) are *disjoint:* there is no consistent assignment to the DC entry for $\text{NS}(s, I_i)$.

To insure that the tracking requirement can be met, we constrain the state assignment. In more detail, we circumvent this problem by forcing GroupFace(G_i) and GroupFace(G_j) to be disjoint whenever two conditions hold: (i) the inputs seen by G_i and G_j are identical (*i.e.*, $I_i = I_j$), *and* (ii) the next state groups G_{i+1} and G_{j+1} are disjoint. The idea is to insure that if the *state groups* G_{i+1} and G_{j+1} are disjoint, then the *group faces* of G_i and G_j are forced to be non-intersecting.[3] In this case, no conflicting DC assignment can ever occur, since the group faces of G_i and G_j will no longer intersect. As a result, either state groups G_{i+1} and G_{j+1} intersect (hence, GroupFace(G_{i+1}) and GroupFace(G_{j+1}) will intersect), or GroupFace(G_i) and GroupFace(G_j) will be made disjoint. In each case, Eqn. 4 is now satisfiable, without conflicting next-state DC assignments. This non-intersection constraint between GroupFace(G_i) and GroupFace(G_j) is written as an $n{\rightarrow}k$ type dichotomy between G_j and G_k: $(G_j; G_k)$.

Example 5.3 (contd.) In the above example, we therefore add dichotomy $(S_1S_2S_3; S_4S_5S_6)$ because $I_2 = I_4 = 00$, and state groups $G_3 = (S_7S_8S_9)$ and $G_5 = (S_3)$ are disjoint. After solving with the new dichotomy, the result is the new state encoding of Fig. 6(b). This encoding does not suffer from the problem of conflicting tracking requirements since there is no counterpart of S_x here: GroupFace($\{S_1S_2S_3\}$) and GroupFace($\{S_4S_5S_6\}$) are now forced to be disjoint. Consequently, the new synthesized machine is logically initializable.

□

[3] Note that it will never happen that G_{i+1} and G_{j+1} are disjoint, but G_i and G_j have a symbolic state in common, since in this case they must have a symbolic next-state in common.

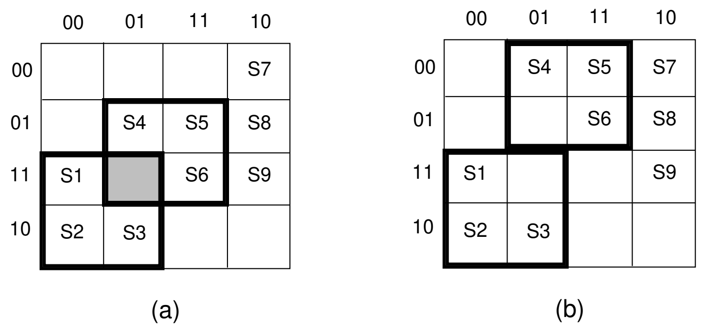

Figure 6: (a) bad state encoding, (b) good encoding

The constraints introduced above are called "don't–care intersection constraints" *(DCIC)* and are formalized as follows:

$$\mathbf{DCIC} = \{(G_i; G_j) \mid (I_i = I_j) \wedge (G_{i+1} \cap G_{j+1} = \phi)\} \tag{5}$$

5.3 Solving Initializability Constraints

Together, the relaxed face-embedding constraints (RFEC's) and the don't-care intersection constraints (DCIC's) are sufficient to produce a state assignment that enables the synthesis of a logically initializable machine.

The RFEC as well as DCIC constraints are dichotomy constraints. Any set of dichotomy constraints can always be solved. For example, a *one-hot code* [12] which uses one state bit for every symbolic state satisfies all dichotomies that can be defined over the set of states. However, such a code is potentially expensive in terms of the amount of logic used to synthesize the circuit. Therefore, several well known methods have been developed for efficiently solving a set of dichotomy constraints (*e.g.,* Scherzo [10], Nova [11]).

6 Step #2: Combinational Logic Synthesis

Once constrained state assignment is complete, combinational logic synthesis can be performed. However, combinational logic synthesis can still adversely affect logical initializability. That is, even after an FSM has been state encoded in accordance with the method of Section 5, unrestrained combinational logic minimization can render it logically uninitializable.

In [5], Cheng and Agrawal point out that combinational logic synthesis influences logical initializability. They surmise that initializability can be preserved by applying *single-output logic minimization* for each output, as opposed to performing *multi-output logic minimization.* However, we discovered that this restriction is neither necessary nor sufficient.

In what follows, it is first shown how combinational logic synthesis can impact 3-valued simulation. Then, the key result of this section is presented: a theorem that relates 3-valued simulatability of a circuit with *hazard-freedom* of asynchronous circuits. In particular, it is shown that our new constraints on logic synthesis for initializability correspond precisely to hazard-free synthesis requirements (*cf.* Nowick [16], Eichelberger [15]). Finally, a multi-level synthesis method for initializability is presented that leverages off of existing hazard-free synthesis methods.

6.1 How logic synthesis affects 3-valued simulatability

The following example illustrates how logic synthesis can affect logical initializability under 3-valued simulation.

Example 6.1. Let Y be the Boolean function of three variables a, b and c shown in the K-map of Fig. 7(a). Let Y be implemented in 2-level AND–OR logic using two product terms: $Y = ab + \bar{a}c$.

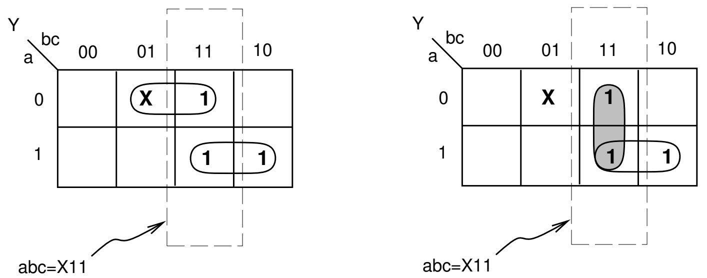

Figure 7: (a) Uninitializable, and (b) initializable implementations of Y

Suppose this gate-level implementation is simulated by a 3-valued simulator. Assume that the primary inputs are set to $abc = X11$. For $abc = X11$, Y is functionally equal to 1, as seen from the K-map. However, a 3-valued simulator may evaluate Y as follows:

$$Y \;=\; ab + \bar{a}c \;=\; a + \bar{a} \;=\; X + X \;=\; X$$

Therefore, the above implementation of Y is *logically uninitializable* since it is not correctly simulatable.

The K-map of Fig. 7(b) shows an alternate implementation of Y which is *logically initializable:* $Y = ab + bc$ (where the shaded region represents the added product term bc). 3-valued simulation in this case yields the correct value:

$$Y \;=\; ab + bc \;=\; a + 1 \;=\; X + 1 \;=\; 1$$

The reason initialization succeeds in this case is that the product bc evaluates to 1 *irrespective* of the value of a. Therefore, this implementation of Y is correctly simulatable for the input combination $abc = X11$. Hence, this gate-level implementation is logically initializable to $Y = 1$ under 3-valued simulation with the inputs held at $abc = X11$. □

Thus, combinational logic synthesis is critical to insuring logical initializability. Any synthesis method that does not incorporate initializability considerations cannot guarantee that the resulting gate-level circuit will be initializable by a 3-valued simulator.

6.2 Simulatability and Hazard-freedom

This section states and proves a correspondence between two different properties of a general multi-level circuit: 3-valued simulatability on the one hand, and hazard-freedom on the other.

Example 6.1 showed that it is sometimes necessary to include a certain product term in the 2-level implementation for initializability. In that example, bc was such a product term in the implementation for Y. In the asynchronous terminology of [16], bc is called a *required cube;* the stipulation that the 2-level implementation of Y must include at least one term that *covers* bc is a *hazard-free covering requirement.*

We now point out the correspondence between 3-valued simulatability and hazard-freedom for the synthesized circuit of Fig. 8.

In the simulatability framework, the highlighted column corresponding to $abc = X11$ represents *indeterminacy* in the values of the inputs—the value of a was unknown. In order for Y to be simulatable to 1 for this input combination, it is required that the cube bc be covered by some term of the cover.

Now regard the function Y as being the output of an asynchronous combinational circuit. Also, view the input column $abc = X11$ as representing the *input transition* $011 \rightarrow 111$ (or, equivalently, $111 \rightarrow 011$) which *spans* $X11$. Then, it is well known ([16, 12, 15]) that to insure

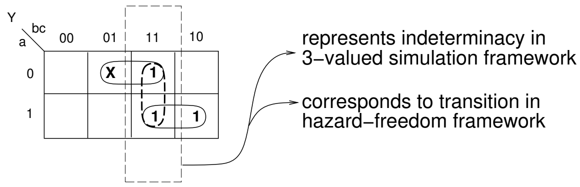

Figure 8: Simulatability and Hazard-freedom

a glitch-free output Y—*i.e.,* to insure that Y remains 1 throughout the input transition, free of *static hazards*—it is essential that some product term in the implementation of Y cover the cube bc.

> *Thus, given a 3-valued input vector I, the covering requirement for 3-valued simulatability of an implementation of Y is identical to the hazard-free covering requirement to insure a hazard-free implementation of Y for an input transition that spans I.*

Thus, for the given example of Fig. 8, the following states the relation between 3-valued simulation and the transient (asynchronous) behavior:

> If the implementation of Y is *not correctly simulatable* to 1 over the input combination $X11$, then all input transitions spanning $X11$ (*i.e.* $011 \rightarrow 111$, *and* $111 \rightarrow 011$), are *hazardous* for the same implementation.

This result can be generalized to an arbitrary multi-level circuit of Fig. 9 as follows. Replace the 3-valued input vector by a corresponding input transition that spans the 3-valued input. Then, if the the output of the circuit has a static hazard, the circuit is non-simulatable for that 3-valued input, and vice-versa.

The rest of this section is devoted to defining several notions related to simulatability (Section 6.2.1) and hazard-freedom (Section 6.2.2), and formally proving the correspondence between the two (Section 6.2.3).

6.2.1 3-valued Simulation of a Network

Definitions regarding 3-valued simulation follow.

Definition 6.1. Given a 3-valued vector $\alpha \in \{0, 1, X\}^n$, a binary vector $\beta \in \{0, 1\}^n$ is *covered* by α iff

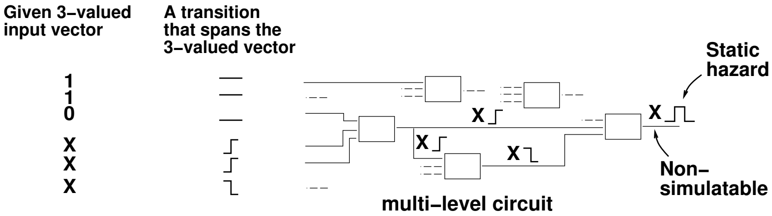

Figure 9: A general multi-level circuit

$$
\begin{aligned}
\alpha_i = 0 \quad &\Rightarrow \quad \beta_i = 0 \\
\alpha_i = 1 \quad &\Rightarrow \quad \beta_i = 1
\end{aligned}
$$

□

For example, binary vector $\beta = 110010$ is covered by 3-valued vector $\alpha = 1X001X$. The following defines how 3-valued simulation works at the gate level for a single output gate.

Definition 6.2 (3-valued simulation of a gate). Given a gate G corresponding to a Boolean function f of n variables, $f : \{0, 1\}^n \to \{0, 1\}$, and given a 3-valued input vector α,

$$
\begin{array}{l}\text{the gate output} \\ \text{is simulated by}\end{array}
\left\{
\begin{array}{cl}
0 & \text{iff } f(\beta) = 0 \quad \forall\, \beta \text{ covered by } \alpha \\
1 & \text{iff } f(\beta) = 1 \quad \forall\, \beta \text{ covered by } \alpha \\
X & \text{iff } f(\beta_1) = 0, f(\beta_2) = 1 \\
 & \qquad \text{for some binary vectors } \beta_1, \beta_2 \text{ covered by } \alpha
\end{array}
\right.
$$

□

Given a 3-valued input α, Definition 6.2 can be generalized from gate simulation to circuit simulation, by a topological traversal from the inputs towards the output, applying Definition 6.2 once to each gate.

6.2.2 Hazard Simulation of a Network

We now discuss basics of hazards in combinational logic. We first review Kung's 9-valued algebra [18] which will be needed to prove our later results.

Kung's algebra is a transition algebra that classifies a transition on a wire into one of 9 values: $\{\mathbf{0}, \mathbf{1}, \uparrow, \downarrow, S0, S1, D+, D-, *\}$. The first two values, **0** and **1,** represent *hazard-free* static $0 \to 0$ and static $1 \to 1$ outputs respectively. Values $\uparrow, \downarrow, S0, S1, D+$ and $D-$ are *transient values* and represent transitions and hazards. $\uparrow$ and $\downarrow$ denote hazard-free $0 \to 1$

and $1 \to 0$ transitions, respectively. $S0$ and $S1$ denote *hazardous* static $0 \to 0$ and $1 \to 1$ transitions, respectively. $D+$ and $D-$ represent *hazardous* dynamic $0 \to 1$ and $1 \to 0$ transitions, respectively. Finally, $*$, which represents a don't-care transition, will not be needed for the remainder of this section.

An *input transition*, or a *multiple-input change*, on a set of input wires $x_1 \ldots x_n$ can be described as a vector $\delta = \delta_1 \ldots \delta_n$ of corresponding values in Kung's algebra, where $\delta_i \in \{\mathbf{0}, \mathbf{1}, \uparrow, \downarrow, S0, S1, D+, D-\}$.

Since we are trying to relate 3-valued simulation to hazard-freedom, it is important to give basic definitions for hazard-freedom. We present the classical notion of an *atomic gate* in the context of hazard-freedom:

Definition 6.3 (atomic gate). An *atomic gate* is a combinational logic gate that can be modeled as an instantaneous Boolean operator followed by an arbitrary finite delay. □

The next proposition indicates that, for the purpose of hazard simulation, any input combination (minterm) that might be reachable during an input transition is assumed to be reachable.

Proposition 6.0 (reachable inputs). Let $x = x_1 \ldots x_n$ be a set of wires and let $\delta = \delta_1 \ldots \delta_n$ be a corresponding input transition. Then, the set of input combinations, Λ, reachable on transition δ is the set of all minterms $m = m_1 \cdots m_n \in \{0,1\}^n$ such that,

$$\begin{aligned} m_i = 0 \quad &\Rightarrow \delta_i \ \in \ \{\mathbf{0}\} \cup \{\uparrow, \downarrow, S0, S1, D+, D-\} \\ m_i = 1 \quad &\Rightarrow \delta_i \ \in \ \{\mathbf{1}\} \cup \{\uparrow, \downarrow, S0, S1, D+, D-\} \end{aligned}$$

Proof. In a hazard model which assumes arbitrary gate and wire delays, worst case behavior is assumed [12, 18]. Hence, if a minterm is reachable by some sequence of transitions on the set of wires, x, it is assumed reachable. □

Proposition 6.1 (hazard simulation of an atomic gate). Let G be an atomic gate for a Boolean function f. Let there be an input transition δ at the inputs. Let us denote the set of all the inputs reachable on this transition by Λ. For the purpose of hazard simulation, any input that is reachable for some combination of gate and wire delays is assumed to be reached. Then,

(a) If $f(\beta) = 0 \quad \forall\ \beta \in \Lambda$, the gate output stays at 0 throughout the transition, and is, therefore, hazard-free: **0.**

(b) If $f(\beta) = 1 \quad \forall\, \beta \;\in\; \Lambda$, the gate output stays at 1 throughout the transition, and is, therefore, hazard-free: **1.**

(c) If $f(\beta_1) \;=\; 0, f(\beta_2) \;=\; 1$ for some $\beta_1, \beta_2 \;\in\; \Lambda$, the gate output either exhibits a monotonic transition or is hazardous for this input change

Proof. Part (a) follows directly from the definition of an atomic gate (Definition 6.3)—if at all times the inputs seen by the gate are those for which f evaluates to 0, then the gate output must constantly stay at 0. Part (b) is proved similarly.

Part (c): while the inputs are changing, the gate sees an input for which $f = 0$ and another input for which $f = 1$. By definition of an atomic gate, the instantaneous operator evaluates to two different values during the transition. Therefore, by virtue of Proposition 6.0 (reachable inputs), the output will produce a transient value, *i.e.*, one of $\{\uparrow, \downarrow, S0, S1, D+, D-\}$.
□

For a given input transition, δ, hazard simulation of a *circuit network* corresponding to Boolean function f is performed by a topological traversal from the inputs towards the output, applying Proposition 6.1 once to each gate.

6.2.3 Transformation: 3-valued vector $\longrightarrow$ input transition

Based on the above, there is a *natural transformation* of a 3-valued *input vector* α to a corresponding *input transition* δ in Kung's 9-valued algebra. For the following, assume α is an arbitrary 3-valued vector where the ith bit is α_i. A corresponding input transition δ is constructed as follows: replace each 0 in α by a $0 \to 0$ transition (**0**), each 1 by a $1 \to 1$ transition (**1**), and each X by *any one* of the transient values $\{\uparrow, \downarrow, S0, S1, D+, D-\}$. More formally, denote this transformation by the operator τ,

$$\delta \quad \in \quad \tau(\alpha) \quad = \quad \left\{ v \;\middle|\; \begin{array}{l} v_i = \mathbf{0} \text{ if } \alpha_i = 0 \\ v_i = \mathbf{1} \text{ if } \alpha_i = 1 \\ v_i \in \{\uparrow, \downarrow, S0, S1, D+, D-\} \text{ if } \alpha_i = X \end{array} \right\}$$

for all bits i of vector α, where τ describes a set of corresponding input transitions. For example, if $\alpha = X10X$ is a 3-valued input, then $\delta =\uparrow 10 \downarrow$ is one such input transition, corresponding to vector α.

We now have all the tools needed to state and prove the key theorem relating 3-valued simulation and hazard-freedom of an arbitrary multi-level network. The proof will essentially consist of a topological traversal of the circuit, applying the above propositions once to every gate.

We introduce the notation $Sim^f_{3-val}(\alpha)$ to represent the result of 3-valued simulation of a circuit with output f for the 3-valued input vector α. Similarly, we use $Sim^f_{hazard}(\delta)$ to denote the result of hazard simulation of f for the 9-valued input vector δ. Given these definitions, the following key theorem lets us deduce the result of 3-valued simulation of a circuit from the result of hazard simulation, and vice-versa.

Theorem 6.1. Let f be a Boolean function implemented by a gate level network of atomic gates G, let α be any 3-valued input vector, and let $\delta \in \tau(\alpha)$ be any corresponding input transition. Then, the 3-valued and hazard simulation results for the implementation G of the function f correspond, as follows:

$$\begin{array}{lcl} Sim^f_{3-val}(\alpha) = 0 & \Longleftrightarrow & Sim^f_{hazard}(\delta) = \mathbf{0} \\ Sim^f_{3-val}(\alpha) = 1 & \Longleftrightarrow & Sim^f_{hazard}(\delta) = \mathbf{1} \\ Sim^f_{3-val}(\alpha) = X & \Longleftrightarrow & Sim^f_{hazard}(\delta) \in \{\uparrow, \downarrow, S0, S1, D+, D-\} \end{array}$$

Or, in short, $Sim^f_{hazard}(\delta) \ \in \ \tau\left(Sim^f_{3-val}(\alpha)\right)$.

Proof. We prove that the above correspondence holds for *any* gate output ℓ in the network G. The proof is by induction on the "depth" of the sub-circuit in the transitive fan-in of ℓ, where "depth" of this sub-circuit is defined as the number of gates on the longest path to ℓ from any of the primary inputs.

Induction Base: Let $depth(\ell) = 0$. Then, ℓ must be one of the primary input wires, and the result holds by virtue of the definition of the τ operator.

Induction Hypothesis: Assume the results holds for all wires ℓ of depth less than k, $k \geq 1$.

Induction Step: Let wire ℓ be at a depth of k. Then, ℓ is the output of a gate with inputs $i_1 \dots i_n$. Let g represent the Boolean function corresponding to the gate. Under 3-valued simulation, these inputs are represented by a 3-valued vector α_g of length n. Each of these inputs lies at a depth less than k. We now show that the same correspondence holds for output ℓ of gate g. There are 3 cases:

(1) The function g has value 0 for all inputs covered by α_g, *i.e.* $g(\beta) = 0$ for each binary vector β covered by α_g. Then, in 3-valued simulation, g is simulated to 0, by Definition 6.2. Thus, $Sim^g_{3-val}(\alpha_g) = 0$. In hazard simulation, let the input transition seen by the gate be denoted by δ_g. By the induction hypothesis, $\delta_g \ \in \ \tau(\alpha_g)$. Then, by

definition of τ each transient in δ_g corresponds to an X in α_g, each 0 in δ_g corresponds to a 0 in α_g, and each 1 in δ_g corresponds to a 1 in α_g. Therefore, by Proposition 6.0, the reachable inputs during input transition δ_g are all covered by α_g. Therefore, by Proposition 6.1(a), the gate output is hazard-free under input transition δ_g, with value **0.** That is, $Sim^g_{hazard}(\delta_g) = \mathbf{0}$.

(2) The function g has value 1 for all inputs covered by α_g. By a similar line of reasoning as above, $Sim^g_{3-val}(\alpha_g) = 1$ and $Sim^g_{hazard}(\delta_g) = \mathbf{1}$.

(3) $g(\beta_1) = 0, g(\beta_2) = 1$ for some β_1, β_2 covered by α_g. Then, by Definition 6.2, $Sim^g_{3-val}(\alpha_g) = X$. By Proposition 6.0, all those inputs that are covered by α_g are reachable. Therefore, both β_1 and β_2 are reachable during the input transition δ_g. By Proposition 6.1, $\beta_1, \beta_2 \in \Lambda$. Combining this result with Proposition 6.1(c), we conclude that $Sim^g_{hazard}(\delta)$ is a transient value, *i.e.*, one of $\{\uparrow, \downarrow, S0, S1, D+, D-\}$.

Since these three are the only scenarios possible, the proof is complete. □

What we have shown above is that if indeterminacy at the input (*i.e.,* a value of X) is translated into an input transition (*i.e.,* $\uparrow$ or $\downarrow$) then indeterminacies on wires elsewhere in the circuit manifest themselves as transitions or hazards on those wires.

We are now ready to give a corollary to this theorem that gives a precise equivalence between non-simulatability and existence of a static hazard. But first, we give precise definitions of "simulatability" and "non-simulatability," terms which we have thus far used informally.

Definition 6.4 (simulatability/non-simulatability). Let a Boolean function f be implemented by a gate level network of atomic gates G. Let α be any 3-valued input vector for which we wish to simulate the circuit output. Then, implementation G is said to be *simulatable* for input α if and only if all of the following hold:

1. $Sim^f_{3-val}(\alpha) = 1$ if $f(\beta) = 1$ for all binary inputs β covered by α.
2. $Sim^f_{3-val}(\alpha) = 0$ if $f(\beta) = 0$ for all binary inputs β covered by α.
3. $Sim^f_{3-val}(\alpha) = X$ if $f(\beta_0) = 0$ and $f(\beta_1) = 1$ for some binary inputs β_0, β_1 covered by α.

G is said to be *non-simulatable* for input α if it is not simulatable for α. □

It can be easily proved that there are only two situations in which G can be non-simulatable:

1. $Sim^f_{3-val}(\alpha) = X$ and $f(\beta) = 1$ for all β covered by α, or
2. $Sim^f_{3-val}(\alpha) = X$ and $f(\beta) = 0$ for all β covered by α.

That is, for a non-simulatable implementation, simulation yields the value X even though f is either functionally equal to 0 over all inputs covered by α, or functionally equal to 1 over all such inputs.

The following key corollary now shows that non-simulatability implies existence of a static logic hazard transition, and vice-versa.

Corollary 6.1 (non-simulatability $\iff$ static logic hazard transition). Let f be a Boolean function implemented by a gate level network of atomic gates G, and let α be any 3-valued input vector. If G is *non-simulatable* for 3-valued vector α, then G has a *static logic hazard* for each input transition $\delta \in \tau(\alpha)$. Conversely, if G has a static logic hazard for some input transition $\delta \in \tau(\alpha)$, then G is non-simulatable for the 3-valued vector α.

Proof. By Definition 6.4, if G is non-simulatable for α then $Sim^f_{3-val}(\delta) = X$. Then, Theorem 6.1 implies that $Sim^f_{hazard}(\delta) \in \{\uparrow, \downarrow, S0, S1, D+, D-\}$. That is, $Sim^f_{hazard}(\delta)$ is a *transient*. However, Definition 6.4 also implies that f is either functionally equal to 0 over all inputs covered by α, or functionally equal to 1 over all such inputs. Therefore, $Sim^f_{hazard}(\delta)$ has to be a static logic hazard.

To prove the converse, assume G has a static logic hazard for some transition $\delta \in \tau(\alpha)$. Then, by Theorem 6.1, $Sim^f_{3-val}(\delta) = X$. However, by definition of a static logic hazard, f is either functionally equal to 0 over all inputs covered by α, or functionally equal to 1 over all such inputs. Therefore, by Definition 6.4, G is non-simulatable for 3-valued input α. □

6.2.4 Summary

The key result of the previous subsection was that given a 3-valued input, for simulatability, the circuit should be *static logic hazard-free* for certain input transitions. Conversely, any circuit realization that is free of static logic hazards for those input transitions, is also logically simulatable.

6.3 Combinational Logic Synthesis for Initializability

Corollary 6.1 provides a technique for combinational logic synthesis for initializability: (i) identify the input transitions that span the 3-valued input vectors encountered in the group face sequence, and (ii) synthesize a circuit that is free of static hazards for those input transitions. We consider both 2-level and multi-level logic synthesis.

2-level: For the special case of a 2-level AND-OR implementation, the conditions for hazard freedom have been presented in [15, 16, 12]. To eliminate static logic hazards ([12, 15]), constraints imposed on logic synthesis are of the form of required cubes. A required cube is a cube that must be covered by some product term of the cover. Techniques for minimization of hazard-free logic based on required cubes are well known [16, 17]. Moreover, the input transitions are *function hazard free,* since the function value is all 0 (or all 1) throughout the transition. Therefore, the constraints for static logic hazard-freedom can always be solved [16].[4]

Multi-level: The duality between simulatability and hazard-freedom enables us to do multi-level logic synthesis for simulatability as follows: (a) do 2-level hazard-free logic synthesis on the appropriate input transitions that span the 3-valued vectors, and (b) use multi-level transformations that do not introduce any static hazards (see [18]). Corollary 6.1 provides the basis for the correctness of this procedure. Alternatively, direct multi-level hazard-free synthesis methods based on BDD's can be used [19].

7 Results

Tables 1 and 2 present the results of our synthesis-for-initializability method of Sections 5 and 6 on several synchronous state machine examples from the MCNC89 benchmark suite [20]. We compare our new method with both the Cheng-Agrawal method, and a base method.

Comparison of synthesis methods. For Cheng-Agrawal, we consider two variants. Since the original Cheng-Agrawal method does not provide any special combinational logic minimization step, the first variant we used, called *CA*, consisted of Cheng-Agrawal state assignment for initializability followed by regular 2-level minimization. However, since Section 6 of this paper demonstrated that combinational logic synthesis is critical to initializ-

[4] A simple proof that a solution always exists follows from the fact that a trivial cover that is the sum of all the prime implicants will always satisfy all the static logic hazard-free covering requirements. Obviously, this solution may be very expensive, but, in practice, when an exact hazard-free minimizer is used, the overhead in satisfying hazard constraints is often negligible [16].

ability, we used a second variant that consisted of Cheng-Agrawal state assignment followed by *our* combinational logic synthesis method of Section 6. We call this method *CA+HF*. Finally, the *BASE* method we used consisted of optimal state encoding followed by "regular" 2-level combinational logic minimization; it does not consider initializability.

Results of our method were compared with *BASE*, *CA* and *CA+HF* using the following two criteria: (i) *effectiveness*, and (ii) *optimality*. With regard to effectiveness (explained later) our method outperformed the existing methods. With regard to optimality (as measured by metrics explained later), our method incurred low logic overhead in achieving initializability.

Benchmark examples. Results for 14 state machines from the MCNC89 benchmark suite are presented. Each machine was functionally initializable, and the same initialization sequence was used for each synthesis method.

For each machine, the optimal state encoding constraints of [13] were first generated. Then, the initializability constraints for each of the synthesis methods (except for the *BASE* method which uses none) were generated. All the dichotomy constraints were then solved to obtain a final state assignment.

Next, 2-level multi-output logic minimization was performed to synthesize a gate-level circuit. The circuit was then simulated with a 3-valued simulator to verify whether or not it was *actually* logically initializable for the synchronizing sequence used for its synthesis.

7.1 Effectiveness of the synthesis methods

In Table 1 we focus on the most important property of the synthesized circuits: *logical initializability*. We list whether or not the final gate-level implementation was *actually* initializable by the synchronizing sequence used for synthesis, when simulated by a 3-valued simulator. As expected, the trends show that logical initializability is generally enhanced as we move across the table from left to right.

The *BASE* method fared poorest in initializability, whereas, as expected, *OUR METHOD* produced the best results, since our method always guarantees initializability. All circuits produced by *OUR METHOD* were logically initializable. In comparison, *CA+HF* cannot guarantee initializability for two benchmarks, *dk27* and *dk512*. The reason lies in the fact that DCIC constraints are necessary to guarantee initializability, and *CA+HF* does not use them. Therefore, as in the case of *dk27* and *dk512*, while one particular implementation

Name	**BASE**	**CA**	**CA+HF**	**OUR METHOD**
dk14	×	×	**√**	**√**
dk15	√	√	**√**	**√**
dk17	×	×	**√**	**√**
dk27	×	×	√	**√**
dk512	×	×	√	**√**
ex3	×	×	**√**	**√**
ex5	×	×	**√**	**√**
lion9	×	×	**√**	**√**
bbtas	×	√	**√**	**√**
bbara	√	×	**√**	**√**
beecount	×	×	**√**	**√**
train11	×	×	**√**	**√**
s8	×	×	**√**	**√**
shiftreg	√	√	**√**	**√**

Legend: × means synthesized circuit was *uninitializable.*

√ means synthesized circuit was initializable, but *not all implementations* of this circuit that can result from this method *will be initializable.*

√ means that *all implementations* of this circuit that can result from this method *are guaranteed initializable.*

Table 1: Comparison of the correctness of the four synthesis methods

produced by *CA+HF* may be initializable, another one may not be. Finally, a comparison of the *CA* and *CA+HF* columns demonstrates the critical importance of our new combinational logic synthesis step for initializability: using the same state assignment, 11 circuits in *CA* were uninitializable, while all synthesized circuits were initializable using our constrained logic synthesis method in *CA+HF*.

7.2 Optimality of the synthesis methods

			BASE			CA			CA+HF			OUR METHOD			
Circuit Name	No. of states	Len. of Sync. Seq.	No. of $n \to 1$ encoding cons.	State code length	No. of gates	No. of $n \to 1$ encoding cons.	State code length	No. of gates	No. of $n \to 1$ encoding cons.	State code length	No. of gates	No. of $n \to 1$ encoding cons.	No. of $n \to k$ encoding cons.	State code length	No. of gates
dk14	7	2	32	5	25	35	5	25	35	5	26	35	0	5	24
dk15	4	1	9	4	17	9	4	17	9	4	17	9	0	4	17
dk17	8	3	34	4	17	26	4	17	26	4	19	28	0	4	21
dk27	7	4	19	3	8	24	4	9	24	4	9	22	1	4	9
dk512	15	4	101	6	19	113	6	19	113	6	20	113	1	6	19
ex3	10	2	31	7	18	35	7	17	35	7	18	36	0	7	19
ex5	9	2	35	7	15	40	7	16	40	7	17	39	0	7	16
lion9	9	3	22	7	8	36	8	8	36	8	10	31	0	7	10
bbtas	6	3	7	3	13	9	4	11	9	4	11	8	0	4	11
bbara	10	2	30	5	28	33	5	27	33	5	28	33	0	5	28
beecount	7	1	16	5	11	16	5	11	16	5	12	16	0	5	12
train11	11	2	42	10	10	50	11	11	50	11	12	49	0	11	12
s8	5	4	0	3	10	6	3	10	6	3	11	6	0	3	11
shiftreg	8	3	28	3	4	24	3	6	24	3	6	24	0	3	6

Table 2: Comparison of Synthesis Methods

Table 2 compares the optimality of the four synthesis methods as measured by several parameters—(i) number of state encoding constraints, (ii) state code length, and (iii) number of gates.

Number of state encoding constraints. The column "No. of $n \to 1$ encoding cons." lists the number of optimal encoding constraints plus the number of face embedding constraints (FEC's, or RFEC's) used for state assignment for initializability. For all of these columns, only the number of *irredundant* constraints is listed; a constraint that is subsumed by other constraints is not counted. Note that the number of dichotomy constraints is a very rough indicator of the restrictiveness of those constraints; one more restrictive dichotomy may subsume several smaller dichotomies (*e.g.,* $\{(abc;d)\}$ is more restrictive than $\{(ab;d),(ac;d)\}$).

Additionally, for *OUR METHOD*, the number of DCIC constraints is shown in the column "No. of $n \to k$ cons." Our method needed DCIC constraints for only two circuits, and, moreover only one DCIC for each. We recall that DCIC constraints are critical for guaranteeing initializability. Thus, whereas existing methods may not always achieve initializability, our method uses DCIC's to guarantee initializability often at very little cost.

State code length. Code length, or the number of state bits used to encode the machine, provides one parameter to compare optimality of the methods. As expected, the *BASE* method always produced the shortest code length because it uses the least constraining set of constraints. In all examples except one, *OUR METHOD* produced state encodings that were the same length as codes produced by *CA* or *CA+HF*. In example *lion9*, *OUR METHOD* produced a shorter encoding, using 7 state bits instead of 8. This indicates that, while our face-embedding constraints, RFEC's, are less restrictive than the Cheng-Agrawal face-embedding constraints, they had little impact on resulting code lengths. It is possible that RFEC's will have a greater impact on larger examples. However, more importantly, one should note that *OUR METHOD* guarantees initializability.

Gate count. The column "No. of gates" lists the number of gates used in the final circuit implementation. From the table, it is clear that *OUR METHOD* incurs low logic overhead over the *BASE* method in order to insure initializability (215 gates total used by *OUR METHOD* for the 14 examples vs. 203 gates total used by *BASE*). A comparison with *CA* and *CA+HF* also shows that the gate counts of circuits produced *OUR METHOD* compare favorably with those of *CA* and *CA+HF*.

8 Conclusions

This paper has presented a new synthesis-for-logical-initializability method. The method provides both a state assignment step, and a combinational logic synthesis step.

For state assignment, we introduced two sets of constraints. First, relaxed face embedding constraints were presented. These constraints are safely relaxed versions of existing face-embedding constraints [5]. Second, don't-care intersection constraints were introduced and were shown to be critical for initializability.

For combinational logic synthesis, it was first shown that unconstrained logic minimization can render a circuit logically uninitializable under 3-valued simulation. Next, necessary and sufficient conditions on combinational logic for initializability were enumerated. Finally, synthesis methods to generate two-level and multi-level logic for initializability were presented.

Combined together, given a functionally initializable specification, our synthesis method guarantees logical initializability for the resulting circuit under 3-valued simulation. Benchmark results show low logic overhead.

9 Acknowledgments

We thank Robert Fuhrer and Michael Theobald of Columbia University for help with synthesizing the circuits. We also thank Prof. Niraj Jha of Princeton University for discussions on initializability and testability.

References

[1] V.D. Agrawal, K.T. Cheng, and P. Agrawal, "A directed search method for test generation using a concurrent fault simulator," *IEEE Trans. CAD,* vol. CAD-8, pp. 131-138, Feb. 1989.

[2] S. Mallela and S. Wu, "A sequential circuit test generation system," in *Proc. of ITC,* Philadelphia, PA. 1985, pp. 57–61.

[3] J.A. Wehbeh and D.G. Saab, "On the initialization of sequential circuits," in *Proc. of ITC,* pp. 233–239, 1994.

[4] K. Cheng and V. Agrawal, "State assignment for initializable synthesis," in *Proc. ICCAD,* pp. 212–215, 1989.

[5] K. Cheng and V. Agrawal, "Initializability consideration in sequential machine synthesis," *IEEE Trans. Comput.,* vol 41, pp. 374–379, Mar. 1992.

[6] J.-K. Rho, F. Somenzi, and C. Pixley, "Minimum length synchronizing sequences of finite state machines," *Proc. DAC,* pp. 463–468, 1993.

[7] S.T. Chakradhar, S. Banerjee, R.K. Roy, and D.K. Pradhan, "Synthesis of initializable asynchronous circuits," in *Proc. 7th Int. Conf. on VLSI Design,* pp. 383–388, Jan. 1994.

[8] S. Banerjee, R.K. Roy, S.T. Chakradhar, and D.K. Pradhan, "Initialization Issues in the Synthesis of Asynchronous Circuits," in *Proc. ICCD-1994.*

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

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

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

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

[13] G.D. Micheli, *Synthesis and Optimization of Digital Circuits,* McGraw-Hill (1994).

[14] J.H. Tracey, "Internal state assignments for asynchronous sequential machines," *IEEE-TEC,* vol. EC-15, no, 4, pp. 551–560, Aug. 1966.

[15] E.B. Eichelberger, "Hazard detection in combinational and sequential switching circuits," *IBM J. Res. Develop.,* vol 9, no. 2, pp. 90–99, 1965.

[16] S.M. Nowick and D.L. Dill, "Exact Two-level Minimization of Hazard-free Logic with Multiple-Input Changes," *IEEE Trans. CAD,* vol. CAD-14, pp. 986–997, Aug. 1995.

[17] M. Theobald and S.M. Nowick, "An implicit method for hazard-free two-level logic minimization," *in Proc. Intl. Symposium on Advanced Research in Asynchronous Circuits and Systems,* Mar. 1998.

[18] D.S. Kung, "Hazard-Non-Increasing Gate-Level Optimization Algorithms," *in Proc. ICCAD,* pp 631–, 1992.

[19] B. Lin and S. Devadas, "Synthesis of Hazard-Free Multilevel Logic Under Multi-Input Changes from Binary Decision Diagrams," *IEEE Trans. CAD,* vol. 14, pp 974–985, Aug. 1995.

[20] R. Lisanke, "Finite-State Machine Benchmark Set v1.0," http://www.cbl.ncsu.edu/pub/Benchmark_dirs/LGSynth89/fsmexamples/, 1989 MCNC International Workshop on Logic Synthesis.

[21] M. Singh and S.M. Nowick, "Synthesis for logical initializability of synchronous finite state machines," *in Proc. of Intl. Conf. on VLSI Design,* pp 76–80, Jan. 1997.