Synthesis for Logical Initializability of Synchronous Finite-State Machines

Montek Singh and Steven M. Nowick

***Abstract*—Logical initializability is the property of a gate-level circuit whereby it can be driven to a unique start state when simulated by a three-valued (0, 1, X) simulator. In practice, commercial logic and fault simulators often require initialization under such a three-valued simulation model. In this paper, the first sound and systematic synthesis method is proposed to ensure the logical initializability of synchronous finite-state machines. The method includes both state assignment and combinational logic synthesis steps. It is shown that a previous approach to synthesis-for-initializability, which uses a constrained state assignment method, may produce uninitializable circuits. Here, a new state assignment method is proposed that is guaranteed correct. Furthermore, it is shown that combinational logic synthesis also has a direct impact on initializability; necessary and sufficient constraints on combinational logic synthesis are proposed to guarantee that the resulting gate-level circuits are logically initializable. The above two synthesis steps have been incorporated into a computer-aided design tool, salsify, targeted to both two-level and multilevel implementations.**

***Index Terms*—Automatic test-pattern generation (ATPG), design for testability, finite-state machines, hazards, initializability, logic simulation, logic synthesis, state assignment, synchronizing sequence, testability, testing, three-valued simulation.**

I. Introduction

INITIALIZABILITY is a property of a circuit that ensures that it can be driven to a unique known state, irrespective of the startup state. Initializability is important in order to physically reset machines if they get out of synchronism. Furthermore, it is required for several fault simulators and nonscan automatic test-pattern generators (ATPGs) to work effectively. Examples of such ATPGs include STG [16] and CONTEST [1].

Two notions of initializability are widely used: *functional initializability* and *logical initializability*. A finite-state machine (FSM) is said to be *functionally initializable* if it is initializable by a series of inputs when *functionally simulated*. 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*. In contrast, a gate-level circuit is said to be *logically initializable* if it is initializable under a series of inputs when simulated by a *three-valued simulator*. The difference between functional and three-valued simulation is that, while the former uses sets of symbolic states to simulate the machine, the latter uses three-valued vectors to keep track of the possible states of the machine. Thus, a precondition for logical initializability of the gate-level circuit is that the underlying finite state machine be functionally initializable.

In practice, the notion of logical initializability is important. Commercial and academic logic and fault simulators and nonscan ATPG tools are often based on a three-valued simulation model [16], [1]. These tools cannot work effectively if the underlying circuit is not three-valued initializable. Note, however, that three-valued simulation is a coarse model, which only safely approximates functional simulation: at any time instant, a three-valued simulator can only approximate the possible states of the machine by representing them with a three-valued vector. In particular, it is well known that a three-valued simulator may compute the logic value of a function to be X (unknown) even when the value can be functionally determined to be a "0" or "1" [3]. Nonetheless, the three-valued model is widely used because of its simplicity and effectiveness.

In this paper, rather than attempt to modify commercial logic and fault simulators by adopting a more accurate simulation model, our aim is to synthesize a circuit itself is logically initializable. Several previous approaches to this problem have been proposed. Some methods only *analyze* a gate-level circuit to search for valid initialization sequences [28], [30]. In contrast, other methods attempt to *synthesize* a logically initializable gate-level circuit from a functionally initializable finite-state machine [7], [8]. In this paper, our strategy is to focus on the synthesis step: to synthesize gate-level implementations of synchronous finite-state machines (with little overhead) that are guaranteed to be logically initializable.

A. Contributions of This Paper

In this paper, the first sound and systematic synthesis method is introduced to ensure the logical initializability of synchronous finite-state machines. It is shown that two synthesis steps—state assignment and combinational logic synthesis—have an impact on logical initializability. The new method therefore provides algorithms for each of these steps.

State Assignment: Miczo first pointed out [18] that state assignment can affect logical initializability. For example, if the sole objective of a state assignment is to minimize the number of state bits, or the amount of logic, an implementation may result that is logically uninitializable: a logic (three-valued) simulator may not be able to initialize the gate-level circuit *even* when the underlying FSM has a valid synchronizing sequence. Later, Cheng and Agrawal proposed a constrained state assignment procedure for logical initializability [7], [8]. We show that

Manuscript received December 31, 1997; revisedJune 16, 1999. This work was supported by the National Science Foundation (under CAREER Award MIP-9501880 and under Award CCR-97-34803, and by an Alfred P. Sloan Research Fellowship. An earlier version of this paper appeared in the *Proceedings of the 10th International Conference on VLSI Design*, January 4–7, 1997.

The authors are with the Department of Computer Science, Columbia University, New York, NY 10027 USA.

Publisher Item Identifier S 1063-8210(00)09515-9.

1063–8210/00$10.00 © 2000 IEEE

the constraints on state assignment imposed by this method are neither necessary nor sufficient.

The contributions of this paper toward state-assignment-for-initializability are twofold: 1) we first identify where the constraints of [7] and [8] can be easily and safely *pruned* and 2) we identify where *additional* new constraints are required (irrespective of whether or not the constraints of [7] and [8] were pruned). The resulting combined set of state assignment constraints is sufficient: the method is guaranteed to produce a state assignment that allows one to synthesize a logically initializable circuit, given a valid functional initialization sequence.

Combinational Logic Synthesis: Several researchers have pointed out the impact of combinational logic synthesis on three-valued simulation and initializability [6], [8]. In one approach to initializability, it is hypothesized that *single-output two-level logic minimization* may ensure initializable circuits [8]. In related research on logic simulation, a *complete sum* two-level implementation is proposed, i.e., including all prime implicants, to guarantee logical simulatability [6]. In this paper, we show that, for logic initializability, the former approach does not always succeed, and the latter approach may be suboptimal.

The contributions of the paper toward combinational logic synthesis are the following.

1) Both necessary and sufficient constraints on combinational logic are proposed to guarantee logical initializability.
2) A two-level logic minimization method is introduced, which incorporates these constraints.
3) Precise constraints for multilevel synthesis are proposed that can ensure initializable logic.

Interestingly, the new constraints are precisely *hazard-freedom constraints* [26] used in the synthesis of asynchronous combinational circuits.

A further contribution is that, unlike previous methods [7], [8], our new method can correctly handle *incompletely specified finite-state machines*. Both the state assignment and combinational logic synthesis steps ensure that consistent logic values can be assigned to "don't-cares" such that three-valued initialization is feasible.

The two modified synthesis steps have been combined into a computer-aided design (CAD) synthesis tool called SALSIFY (state assignment and logic synthesis for initializability of finite state machines). The tool is targeted to both two-level and multilevel circuits. Experimental results indicate that little area overhead is necessitated by the added constraints.

In summary, given a finite-state machine and a valid functional initialization sequence, the new approach provides a complete synthesis path that produces a gate-level circuit guaranteed to be logically initializable.

B. Organization

This paper is organized as follows. Section II summarizes previous work on initializability as well as three-valued simulation. Section III reviews in detail an existing synthesis-for-initializability method that was used as the starting point for our research. Section IV provides a short overview of our entire synthesis method. Section V presents details of the state assignment step of our method, and Section VI presents details of the combinational logic synthesis step. Finally, results on a set of benchmark examples are presented in Section VII, and Section VIII gives conclusions.

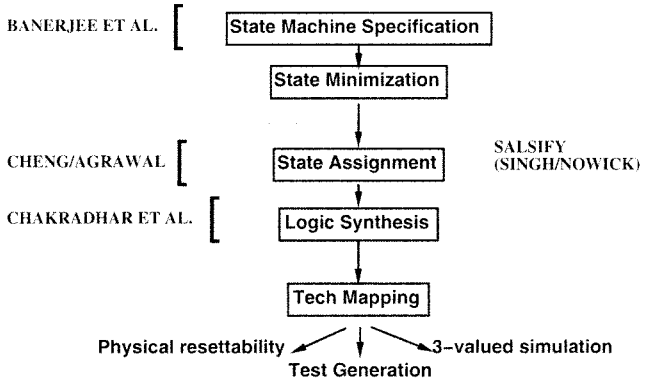

Fig. 1. Synthesis for initializability

II. Previous Work

There has been much work on initializability of finite-state machines, as well as on three-valued, or logic, simulation.

A. Initializability and Synthesis-for-Initializability

Traditionally, several approaches to initializability have been considered. Each of these assumes different models of initializability (such as single or multivector) and of simulation (such as functional or logical). Furthermore, while some methods only *analyze* a state machine to search for valid initialization sequences [21], [28], [30], other methods attempt to *synthesize* a logically initializable gate-level circuit from a functionally initializable finite-state machine [7], [8].

Several analysis methods have been proposed. Rho *et al.* [21] present a search procedure to find *functional initialization sequences*, starting from a finite-state machine description, if any exist. Wehbeh and Saab [28]–[30] propose a method to generate both *functional* and *logical* initialization sequences, starting from a gate-level circuit.

Alternatively, methods have been proposed to synthesize initializable circuits. A typical synthesis path consists of several steps (see Fig. 1). Initializability considerations can be incorporated at various levels. This figure is annotated to highlight some of the recent work on initializability targeting different levels in the synthesis path.

Banerjee *et al.* [2] present a technique for *asynchronous* synthesis that targets the highest level in the synthesis path: the top-level specification (*signal transition graph*). The idea is to modify specification itself to ensure functional initializability. Initializability is achieved only at the cost of some reduction in concurrency. This approach targets a different synthesis level (behavioral), design style (asynchronous), and type of initializability (functional) from our proposed approach.

Miczo pointed out [18] that *state assignment* can affect initializability. In particular, it is shown that while some state encodings of a given state machine produce logically initializable

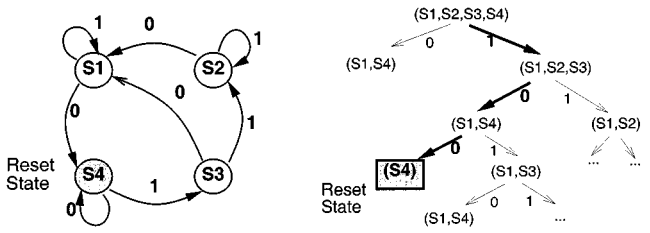

Fig. 2. Example FSM and synchronization tree.

circuits, other encodings do not. Cheng and Agrawal [8] provide a constrained state assignment algorithm to produce logically initializable circuits from functionally initializable synchronous specifications. However, as will be shown, their proposed synthesis constraints are neither necessary nor sufficient. In this paper, we target an alternative set of state assignment constraints which are guaranteed sufficient.

The method of Chakradhar *et al.* [5] for *asynchronous* synthesis targets the *combinational logic synthesis* step for initializability. Their method is essentially a search procedure for finding initialization sequences and concomitant don't-care assignments in order to synthesize initializable asynchronous circuits. However, the method targets a different design style (asynchronous) and only considers the combinational logic functionality (it does not include logic covering requirements). In this paper, we provide constraints on logic synthesis for the corresponding synchronous problem and also include logic covering requirements.

Alternative approaches select an appropriate subset of flip-flops to be partially reset [15], [20].

B. Three-Valued Simulation and Hazard Freedom

Three-valued simulation has been used for decades both for logic simulation as well as hazard analysis. The earliest reference to three-valued simulation is perhaps the use of ternary algebra by Yoeli and Rinon in 1964 to study hazards in combinational circuits [32]. Subsequently, three-valued algebras were used by Eichelberger [10] to detect hazards in logic circuits, and by Jephson *et al.* [11] to simulate the operation of digital circuits in the presence of unknown values.

There have been a few attempts at synthesis of combinational logic to ensure simulatability or initializability [6], [4], [8]. The technique of [6] produces a two-level circuit that can be successfully simulated by using a *complete sum* implementation, i.e., including all prime implicants. Another two-level synthesis approach [4] notes the need for *consensus* products to enhance simulatability but does not guarantee simulatable circuits. For the more limited problem of insuring logical initializability, it has been suggested that *single-output two-level minimization* might ensure a correct implementation [8].

However, none of these methods has noted the tight connection between three-valued simulatability and hazard freedom. In particular, while the constraints proposed by the first two of these approaches relate to constraints for asynchronous hazard-free design [10], they do not explicitly note that hazard-free synthesis may provide the *precise conditions* for three-valued simulatability. In fact, we are unaware of any prior work that has formally proved the exact correspondence between simulatable logic and static hazard-free logic.

In this paper, the correspondence between three-valued simulatability and static hazard freedom is formally proved (Section VI), thus providing precise requirements enabling us to synthesize simulatable circuits. Furthermore, while these earlier methods only focus on two-level requirements for simulatability [6], [4], [8], our new formulation allows one to synthesize fully simulatable multilevel circuits as well.

III. Background—The Cheng–Agrawal Method

This section reviews the Cheng and Agrawal state assignment method [7], [8], which is the starting point of our method. Given a finite-state machine and a synchronizing sequence, the basic approach is to use a constrained state assignment step to ensure logical initializability. After highlighting the impact of state assignment on logical initializability, the details of the Cheng–Agrawal state assignment method are reviewed.

A. Impact of State Assignment on Logical Initializability

It is well known that state assignment may have an effect on logical (three-valued) initializability [18], [7], [8]. An example illustrates the impact and also introduces some useful terminology.

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 state S_4 irrespective of the initial state. Therefore, $I = 100$ is called a *synchronizing sequence* of M. The trace of state groups, or *state group sequence*, that results when the input sequence 100 is applied to M is

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

Therefore, the machine is *functionally initializable*.

When logical initializability is required, however, the synchronizing sequence must also converge under three-valued simulation. Once states have been encoded, a three-valued simulator uses a sequence of *group faces* to represent the simulation trace. Each group face is a three-valued vector, representing the "smallest containing cube" for the corresponding binary states. A group face is computed as follows: the ith bit in the group face is 1 (0) if the ith bit in the encoding of all the states of the group is 1 (0); otherwise, it is X. Thus, for example, if the state encoding (S_1: 00, S_2: 01, S_3: 11, S_4: 10) were used, the group face corresponding to (S_1S_4) would be the smallest cube containing 00 and 10: $X0$. Similarly, the group face of (S_4) is ten, and the group face of $(S_1S_2S_3)$ is XX.

The *group face sequence* is the resulting 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

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

Since the three-valued simulation converges, the resulting implementation is *logically initializable*. □

Next, consider a different state assignment for the same machine.

Example 3.2: Assume that state encoding (S_1: 00, S_2: 01, S_3: 10, S_4: 11) is used for the specification of Fig. 2 instead. In this case, the group face sequence is

$$XX \xrightarrow{1} XX \xrightarrow{01} XX \xrightarrow{01} XX.$$

This sequence does not converge to a single state; therefore, the implementation is *logically uninitializable*. □

Example 3.2 demonstrates the impact of state assignment on logical initializability—arbitrary state encoding can render circuits uninitializable by a three-valued simulator, even though the state group sequence *functionally* converges to a unique state. This problem arises since a three-valued simulator can only simulate group faces, not state groups; there is a loss of information during three-valued simulation. Thus, even though both of the state encodings of Examples 3.1 and 3.2 are satisfactory for functional or physical initializability, the state encoding of Example 3.2 is inadequate for three-valued initializability.

B. State Assignment for Logical Initializability

The goal of the Cheng–Agrawal method is to produce a state assignment, such as that of Example 3.1, that provides logical initializability. That is, the state assignment should allow the sequence of group faces to "track" the sequence of state groups, and therefore ensure logical initializability. To this end, the method introduces constraints into the state assignment step. These constraints are in the form of *dichotomies* [17], [25].

A dichotomy constraint, or simply dichotomy, is written as $(X; Y)$, where X and Y are disjoint sets of states. A 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 state set X is n and the cardinality of Y is k, the constraint $(X; Y)$ can be called a *type* $n \rightarrow k$ *dichotomy*.

The constraints used by Cheng and Agrawal are type $n \rightarrow 1$ dichotomies, also called *face-embedding constraints* [17]. An $n \rightarrow 1$ dichotomy of the form $(G_i; s_j)$ is introduced for every singleton 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. More formally, 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 (FECs) are written as

$$\text{Cheng–Agrawal } \mathbf{FEC} = \{(G_i; s_j) | s_j \notin G_i\}. \quad (2)$$

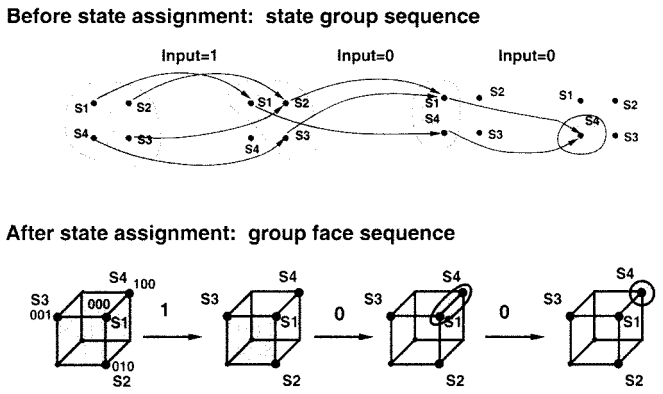

Fig. 3. Groups faces "track" state groups.

Example 3.3: Given the finite-state machine of Fig. 2 and the synchronizing sequence 100 [(1)], the Cheng–Agrawal face-embedding constraints are $(S_1S_2S_3; S_4)$, $(S_1S_4; S_2)$, and $(S_1S_4; S_3)$. Note that *trivial* face-embedding constraints can always be omitted: those whose left side consists of a singleton state. A state assignment satisfying these dichotomies is (S_1: 000, S_2: 010, S_3: 001, S_4: 100). In this example, the first state bit satisfies dichotomy $(S_1S_2S_3; S_4)$: the bit is 0 for states S_1, S_2 and S_3, but is 1 for state S_4. Similarly, the other two state bits satisfy the remaining dichotomies. Thus, the state encoding ensures that the state code for S_4, 100, is not embedded in the group face $0XX$ of state group $(S_1S_2S_3)$ during the first step in the three-valued simulation: $XXX \xrightarrow{1} 0XX$.

Fig. 3 shows graphically the state group sequence and the corresponding group face sequence after three-valued simulation. three-valued simulation converges to the correct value, 100

$$XXX \xrightarrow{1} 0XX \xrightarrow{0} X00 \xrightarrow{0} 100.$$

□

IV. New Synthesis-for-Initializability Method—Overview

Our proposed synthesis method for logical initializability builds partly on the state assignment method of Cheng–Agrawal, but with significant extensions. It also includes a complete combinational logic synthesis step.

Given an incompletely specified finite-state machine and a synchronizing input sequence, a new constrained state assignment step is proposed, in Step 1. This step in turn has two parts: First, it generates a set of *relaxed face-embedding constraints* (RFECs), which is a pruned version of the set of original Cheng–Agrawal face-embedding constraints. Second, it is shown that neither our RFECs nor the earlier FECs alone are sufficient to ensure logical initializability. Therefore, additional constraints, called *don't-care intersection constraints* (DCICs), are imposed that guarantee logical initializability.

Next, in Step 2, once a valid state assignment has been formed, it is shown that the actual combinational logic synthesis step is critical to obtaining a logically initializable circuit. New constraints on logic synthesis are proposed, which are both necessary and sufficient for logical initializability.

Interestingly, these constraints are shown to be precisely a form of static *hazard-freedom constraints* [26] used in the synthesis of asynchronous combinational circuits. Techniques for synthesizing both two-level and multilevel initializable circuits are then presented.

V. Step 1: Constrained State Assignment

This section presents the new constrained state assignment step. Section V-A introduces ther new relaxed face-embedding constraints, and Section V-B introduces the don't-care intersection constraints.

A. Step 1(a)—Relaxed Face-Embedding Constraints

It is now demonstrated that the Cheng–Agrawal face-embedding constraints can be overly restrictive. In some cases, they may be safely pruned.

The intuition is as follows. The idea of logical initializability is to apply an input sequence to drive a machine to a single state. The result is a narrowing sequence of state groups (or group faces). The Cheng–Agrawal method imposes constraints on state assignment to insure that no state lying *outside* the current simulated state group will be embedded within it, after state assignment. Such an embedding may "derail" the simulation. In contrast, we show that, in some cases, such an "outlier" state *may* safely be embedded within such a state group. Such an embedding is permitted as long as this state's destination (next state) *reconverges*, i.e., lies within the next state group in the simulation sequence. Thus, certain embeddings are admissible and will not derail the three-valued simulation.

1) Cheng–Agrawal Constraints—A Reexamination:

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

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

Clearly, at least three state bits are required to satisfy all three constraints. However, a careful examination of the state transition diagram of Fig. 2 indicates that, in fact, the dichotomy $(S_1S_2S_3;\, S_4)$ is unnecessary. Consider the transition $(S_1S_2S_3) \longrightarrow^{0} (S_1S_4)$. Note that state S_4 also has a transition on input 0 to S_4, which belongs to the next state group, (S_1S_4). Therefore, it is safe to allow S_4 to be embedded in the group-face of $(S_1S_2S_3)$: even though S_4 is not part of the current 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

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

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

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

In sum, by pruning the set of face-embedding constraints, a shorter length state encoding can be used (two state bits instead of three) that still ensures logical initializability.[1] □

2) Safe Embeddings: The notion of safe embeddings can now be formally defined. Let M be a finite-state machine having a functional initialization sequence, $G_1 \xrightarrow{I_1} G_2 \xrightarrow{I_2} \cdots \xrightarrow{I_n} G_{n+1}$. Here, G_i is the ith state group in the initialization sequence and I_i is the input applied to 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 $\mathrm{NS}(s_j, I_i)$ is 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

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

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 three-valued simulation for G_{i+1} will still result in the same value as it would if s_j were not embedded in the group face of G_i.

3) Relaxed Face-Embedding Constraints: Using the above notion, the set of Cheng–Agrawal face-embedding constraints can safely be pruned. The original Cheng–Agrawal face-embedding constraints were [(2)]

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

Our new RFECs are

$$\mathbf{RFEC} = \{(G_i;\, s_j) | \mathrm{NS}(s_j, I_i) \notin G_{i+1}\}. \tag{3}$$

The RFEC constraints are clearly a subset of the original Cheng–Agrawal constraints: $s_j \in G_i$ implies $\mathrm{NS}(s_j, I_i) \in G_{i+1}$. Moreover, for any state s_j such that $s_j \notin G_i$ but $\mathrm{NS}(s_j, I_i) \in G_{i+1}$, the Cheng–Agrawal constraints include the dichotomy $(G_i;\, s_j)$, whereas the RFEC constraints do not.

B. Step 1(b)—Don't-Care Intersection Constraints

It is now shown that face-embedding constraints alone, whether the original Cheng–Agrawal FECs or the new RFECs, are *insufficient* to ensure a state assignment that allows logical initializability. New constraints, called *don't-care intersection constraints*, are therefore introduced to ensure initializability.

The intuition is as follows. The earlier face-embedding constraints ensure that all symbolic states are encoded correctly. However, for a three-valued simulation, *don't-cares* must *also* be assigned correct next-state values. A don't-care,

[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.

after state assignment, may arise in two ways: 1) it may be present in the original FSM specification (i.e., unassigned next-states in an incompletely specified machine) and 2) it may appear after state assignment, in binary state codes that have no corresponding symbolic state. If don't-care entries are assigned arbitrary values, three-valued simulation may be "derailed," and logical initialization may fail. Interestingly, it is shown that, after arbitrary state assignment, there may exist no feasible assignment of values to don't-cares which ensures logical initializability. Therefore, state assignment itself must be constrained to guarantee that don't-care assignment for initializability is feasible.

In this section, it is first shown how don't-care (DC) assignment has an impact on logical initializability. Next, conditions on DC assignment are formulated to ensure that DC assignment does not adversely affect logical initializability. It is then shown that, under arbitrary state assignment, these conditions may be unsatisfiable, and three-valued simulation may fail. Finally, new sufficient constraints on state assignment are introduced (DCICs) to ensure that DC assignment for logic initializability is always feasible.

1) Impact of DC Assignment on Logical Initializability: The following example illustrates the impact of DC assignment on logical initializability.

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

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

There are no required face-embedding constraints, whether the original Cheng–Agrawal FECs or our RFECs, since this state group sequence is empty (the dichotomy constraints are trivial). Fig. 4 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. Such a state code is an *unassigned state code* or a *nonsymbolic state*. There are no specified next-state transitions for nonsymbolic states; they are all *don't-care next-state transitions*.[2]

Care must be taken in assigning the next-state values to this unassigned state code, 11. Such values will eventually be assigned during a later stage in the synthesis path (e.g., combinational logic synthesis). Suppose that this latter synthesis step assigns this DC entry for state 11 on input 1 with a next state value 11. In this case, the three-valued simulation trace is

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

Thus, 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 are assigned arbitrary values (during logic synthesis), a noninitializable circuit may result.

[2]Alternatively, in an incompletely specified FSM, a don't-care next-state transition can also arise in a state code corresponding to a *symbolic state* if it has an undefined next-state entry.

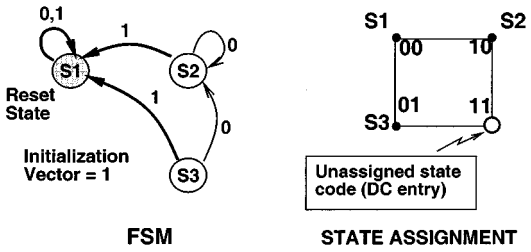

Fig. 4. The issue of assignment to don't-care entries.

To avoid this problem, suppose the next state of 11 on input 1 is now be assigned the value 00 (corresponding to S_1). The following three-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. □

2) Don't-Care Assignment for Logical Initializability: Based on the previous discussion, 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 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 nonsymbolic) have a don't-care next-state entry on input I_i, i.e., $\mathrm{NS}(s, I_i) =$ don't-care. Suppose that state assignment has been completed and that the state code of state 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 ensure initializability. Such a DC assignment must be performed for every such s and i.

More formally, assuming that $\mathrm{StateCode}(s)$ 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 as a *tracking requirement*:

$$\begin{aligned} \forall s, i \qquad & (\mathrm{StateCode}(s) \in \mathrm{GroupFace}(G_i) \Rightarrow \\ & \mathrm{StateCode}(\mathrm{NS}(s, I_i)) \in \mathrm{GroupFace}(G_{i+1})). \end{aligned} \tag{4}$$

This requirement ensures that during three-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 the expected singleton state. Thus, given a synchronizing sequence, the tracking requirement of (4) is a sufficient condition for logical initializability.

3) Infeasible Satisfaction of the Tracking Requirement: In the preceding example, a DC assignment could be applied to ensure initializability. However, this is not always the case.

It is now shown that, after an arbitrary state assignment, satisfying the tracking requirement may not always be feasible. In particular, the example below shows that, using a state assignment which satisfies the Cheng–Agrawal constraints (or our

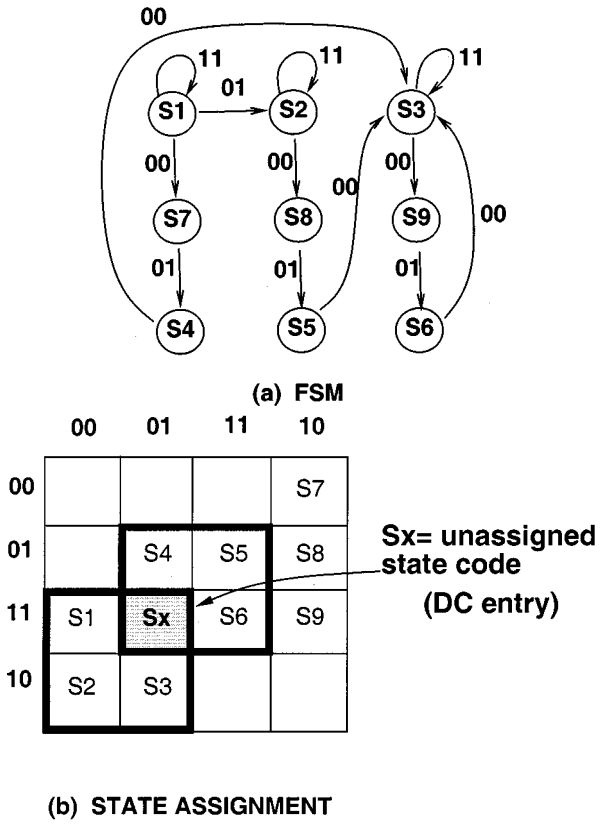

Fig. 5. Example illustrating unsatisfiable tracking requirement.

relaxed FECs) may result in an implementation that is logically uninitializable *for every possible assignment of don't-cares*. This problem occurs because the tracking requirement may impose conflicting DC assignments for certain total states.

Example 5.3: Consider the example of Fig. 5(a).[3] The machine is functionally initializable. Applying the following synchronizing sequence results in a sequence of state groups:

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

Fig. 5(b) shows a 4-bit state assignment that satisfies all the resulting Cheng–Agrawal face-embedding constraints, as well as our new relaxed face-embedding constraints. Bit vector 0111 is an unassigned state code, or nonsymbolic state, which is labeled S_x. The detailed analysis below shows that logical initializability is impossible with the given state assignment. In particular, S_x *cannot be assigned any next-state value* for input 00 while still preserving logical initializability.

For the moment, don't cares will be ignored. If a *true-valued* simulation is performed on the specified states, and the results are collapsed at every time step into a three-valued vector, the following simulation trace occurs:

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

[3]This state machine is incompletely specified, but note that our analysis also applies to completely specified machines.

Ideally, the same result should be obtained for three-valued simulation when the effect of next-state transitions out of state S_x is included in the simulation.

The goal in the DC assignment to state S_x is to satisfy the tracking requirements outlined above. We begin by noting that state S_x is embedded in the group faces of both $(S_1S_2S_3)$ and $(S_4S_5S_6)$. Both of these state groups have specified transitions in the initialization sequence on the *same input* 00. The latter embedding mandates that the next-state value of S_x be set to S_3 in order to meet the tracking requirement [(4)]. The former embedding requires that the next-state value be set to any state in the column containing $(S_7S_8S_9)$. Since the group faces (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 nonsymbolic state S_x on input 00, which simultaneously satisfies both tracking requirements. Therefore, the result is always a logically uninitializable circuit.

Examining the example 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 three-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, $(S_1S_2S_3)$, receives input 00. The group face associated with $(S_1S_2S_3)$ 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 $10XX$. As a result, all four state bits are reset to X.

Alternatively, if S_x were assigned a next-state transition that was embedded within the third group face $10XX$, as desired, then initialization would proceed normally at this step but would be thrown off course on the final input vector. For example, assuming now that NS$(S_x, 00) = 1010$, the following three-valued simulation results:

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

Thus, since there are conflicting don't-care assignment requirements, the implementation is logically uninitializable. □

4) Satisfying the Tracking Requirement–Don't-Care Intersection Constraints: In this section, the state assignment step itself is modified to guarantee that the tracking requirement always can be satisfied, and don't cares can be correctly assigned.

Once again, consider a machine with the initialization sequence

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

Let a binary state s belong to two distinct group faces corresponding to state groups G_i and G_j

$$\text{StateCode}(s) \in \text{GroupFace}(G_i)$$
$$\text{StateCode}(s) \in \text{GroupFace}(G_j).$$

Then the tracking requirement [(4)] results in the following pair of constraints:

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

This pair of constraints may be unsatisfiable if I_i and I_j are identical inputs (revisited in the initialization sequence). That is, if $I_i = I_j$, the above two constraints reduce to one:

$$\text{StateCode}(\text{NS}(s, I_i)) \in \text{GroupFace}(G_{i+1}) \cap \text{GroupFace}(G_{j+1}). \quad (5)$$

In this case, the tracking requirement is unsatisfiable *precisely* when the two next group faces, GroupFace(G_{i+1}) and GroupFace(G_{j+1}), are *disjoint* (i.e., have an empty intersection): there will then be no consistent assignment to the DC entry for NS(s, I_i).

To ensure that the tracking requirement can be met, new constraints are added to the state assignment. Informally, the constraints force the two group faces in the above simulation sequence [GroupFace(G_i) and GroupFace(G_j)] to be disjoint when there is a possibility that they may impose conflicting DC assignments on any common unassigned states.

More formally, GroupFace(G_i) and GroupFace(G_j) in the simulation sequence are forced to be disjoint whenever two conditions hold: 1) the inputs applied to G_i and G_j in the synchronizing sequence are *identical* (i.e., $I_i = I_j$) and 2) the corresponding next *state groups* G_{i+1} and G_{j+1} are disjoint. The motivation is that any unassigned state S_x that lies in both of the "source" group faces [GroupFace(G_i) and GroupFace(G_j)] will have two requirements on its next-state assignment under input I_i: its next state must lie in both GroupFace(G_{i+1}) and GroupFace(G_{j+1}). However, if the next-state groups G_{i+1} and G_{j+1} are disjoint, there is a possibility that their corresponding group faces [GroupFace(G_{i+1}) and GroupFace(G_{j+1})] will also be disjoint after state assignment. In that case, the requirements on the DC assignment of S_x cannot be simultaneously satisfied.

Thus, our conservative solution is to separate the "source" group faces: that is, if the symbolic *state groups* G_{i+1} and G_{j+1} are disjoint, *force* the binary *group faces* of G_i and G_j to be nonintersecting.[4] In this case, no conflicting DC assignment can ever occur, since the "source" group faces of G_i and G_j no longer intersect.

After imposing these constraints on state assignment, either the next state groups G_{i+1} and G_{j+1} intersect [and, hence, GroupFace(G_{i+1}) and GroupFace(G_{j+1}) will intersect] or the current group faces GroupFace(G_i) and GroupFace(G_j) will be made disjoint. In each case, the tracking requirement of (5) is now satisfiable, with no conflicting next-state DC assignments. This new constraint between GroupFace(G_i) and GroupFace(G_j) can be written as an $n \rightarrow k$ type dichotomy between two state groups, G_i and G_j: (G_i; G_j).

[4]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.

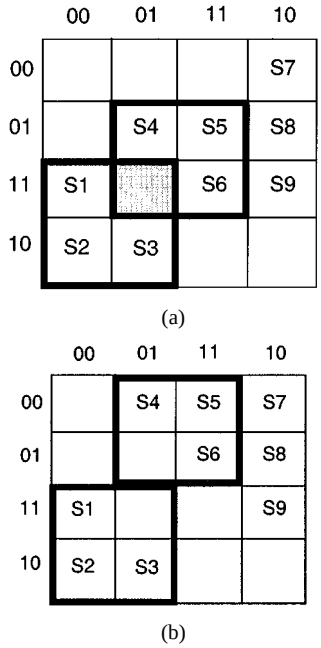

Fig. 6. (a) Bad state encoding and (b) good encoding

Example 5.3 (Continued): In the above example, we therefore add a dichotomy constraint, ($S_1S_2S_3$; $S_4S_5S_6$), since the inputs in the synchronization sequence are identical ($I_2 = I_4 = 00$) for these two state groups, and the next state groups [$G_3 = (S_7S_8S_9)$ and $G_5 = (S_3)$] are disjoint. After satisfying this 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. Effectively, there is no longer an S_x that is shared by two different group faces, since they have been forced apart. Consequently, the new synthesized machine is logically initializable. □

The dichotomy constraints introduced above are called "don't-care intersection constraints." They are formalized as follows, where G_i and G_j are any two state groups appearing in the initialization sequence and I_i and I_j are the corresponding input vectors:

$$\mathbf{DCIC} = \{(G_i; G_j) | (I_i = I_j) \wedge (G_{i+1} \cap G_{j+1} = \phi)\}. \quad (6)$$

C. Solving the Initializability Constraints

Together, the RFECs of Section V-A and the DCICs of Section V-B are sufficient to produce a state assignment that allows the synthesis of a logically initializable machine. Both RFECs and DCICs are dichotomy constraints. It is well-known that any set of dichotomy constraints can always be solved. For example, a *one-hot* [26], which uses one state bit for every symbolic state, can be used to satisfy a set of dichotomies. However, such a code is potentially expensive in terms of the number of state bits. Therefore, more efficient algorithms have been developed to solve dichotomies using fewer state bits [27], [31], [22], [9].

VI. Step 2: Combinational Logic Synthesis

Once constrained state assignment is complete, combinational logic synthesis is performed. However, combinational logic synthesis can adversely affect three-valued simulatability, and hence logical initializability. That is, even after an FSM has been state encoded in accordance with the method of Section V, arbitrary combinational logic minimization can produce a logically uninitializable implementation.

A previous approach to synthesis for logical initializability [8] suggested that *single-output two-level minimization* (as opposed to multioutput minimization) might ensure logical simulatability. However, in this section, it is demonstrated that this approach is neither necessary nor sufficient. In addition, it has the drawback that it is limited to two-level implementations, unlike the proposed approach, which can produce multilevel implementations.

In Section VI-A, it is first shown how combinational logic synthesis has an impact on three-valued simulation. Then, in Section VI-B, the key result of this section is presented: a theorem that precisely relates *three-valued simulatability* of a circuit with *hazard freedom* of asynchronous circuits. In particular, it is proved that a combinational circuit is three-valued simulatable if and only if it *static hazard free* [19], [10] for certain multiple-input changes. Based on this result, in Section VI-C, both two-level and multilevel synthesis methods for logical initializability are presented, which leverage off of existing hazard-free logic synthesis methods.

A. How Logic Synthesis Affects Three-Valued Simulatability

The following example motivates how logic synthesis can affect logical initializability under three-valued simulation.

Example 6.1: Let Y be the Boolean function of three variables a, b, and c, shown in the Karnaugh map of Fig. 7(a). Let Y be implemented as an AND–OR circuit using two product terms: $Y = ab + \overline{a}c$. Suppose the gate-level implementation of Fig. 7(a) is simulated by a three-valued simulator. Assume that the primary inputs are set to $abc = X11$. For $abc = X11$, Y is functionally equal to one, as seen from the Karnaugh map. However, a three-valued simulator may evaluate Y as follows:

$$Y = ab + \overline{a}c = X \cdot 1 + X \cdot 1 = X + X = X.$$

Therefore, the above implementation of Y is *logically uninitializable* since it is not correctly simulatable to the value 1.

The Karnaugh map of Fig. 7(b) shows an alternate implementation of Y that is *logically initializable*: $Y = ab + bc$ (where the shaded region represents the added product term bc). In this case, three-valued simulation yields the correct result

$$Y = ab + bc = X \cdot 1 + 1 \cdot 1 = X + 1 = 1.$$

Initialization succeeds in this case since bc evaluates to one *irrespective* of the value of a. Therefore, this implementation of Y is correctly simulatable for the input combination $abc = X11$. □

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 three-valued simulator.

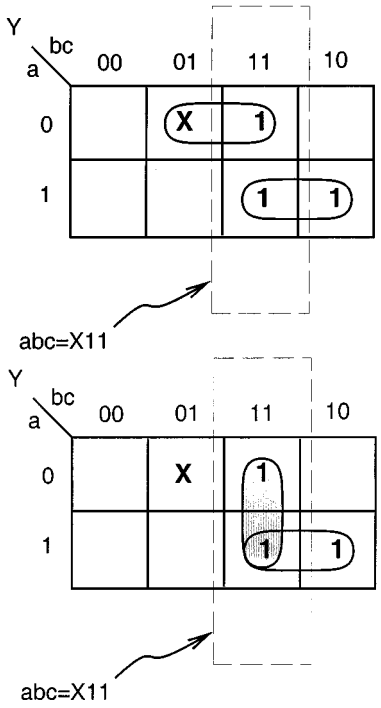

Fig. 7. (a) Uninitializable and (b) initializable implementations of Y

B. Simulatability and Hazard Freedom

This section states and proves the fundamental correspondence between two different properties of an arbitrary combinational circuit: three-valued simulatability on the one hand and static hazard freedom on the other.

Hazard freedom refers to the absence of glitches at the outputs of a combinational circuit when the inputs to the circuit undergo a transition [26]. Hazards are usually not a problem in clocked, or synchronous, systems since all the circuit outputs are assumed to have stabilized before the arrival of the clock tick. However, in unclocked, or asynchronous, systems, freedom from hazards is usually critical to correct operation.

In this section, the notion of three-valued simulatability, and hence logical initializability, of a given synchronous circuit is related to hazard freedom. The goal of this exercise is to establish a direct correspondence between these two seemingly different properties, and then leverage off of existing techniques for hazard-free synthesis in order to synthesize logically initializable circuits.

1) Example and Overview: The correspondence between three-valued simulatability and hazard freedom is now illustrated by an example. Consider the two-level implementation of Fig. 8 (from Example 6.1). In a *three-valued simulation* framework, the highlighted column corresponding to $abc = X11$ represents *indeterminacy* in the values of the inputs—the value of a is unknown. In order for Y to be simulatable to one for this input combination, it is required that the cube bc be contained in some product of the cover. In the terminology of [19], bc is called a *required cube*; the stipulation that the two-level

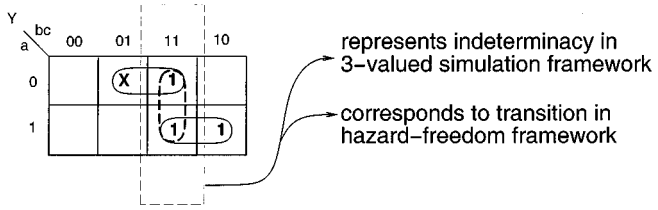

Fig. 8. Simulatability and hazard-freedom.

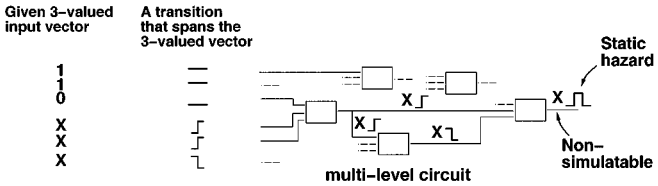

Fig. 9. A general multilevel circuit

implementation of Y must include at least one product term that covers bc is a *covering requirement*.

In a *hazard-freedom* framework, the function Y is regarded as being the output of an asynchronous combinational circuit. Here, the input column $abc = X11$ is now viewed as representing the *input transition* $011 \rightarrow 111$ (or, equivalently, $111 \rightarrow 011$), which *spans* $X11$. It is well known that to ensure a glitch-free output Y—i.e., to ensure that Y remains at one throughout the input transition, or *static hazard free*—it is necessary that some product term in the implementation of Y cover the cube bc [19], [26], [10]. Otherwise, the product term $\overline{a}c$ may turn off *before* the product term ab turns on, causing the "1" output to momentarily switch to "0" and then back to "1." The presence of a product term covering the cube bc ensures that the output does not glitch; bc holds the "1" value throughout the transition. Once again, in the terminology of [19], bc is a required cube; the stipulation that a hazard-free two-level implementation of Y must include at least one term that covers bc is a *static hazard-free covering requirement*.

This correspondence between three-valued simulatability and static hazard freedom can now be formally stated.

> Given a three-valued input vector I, the covering requirement for three-valued simulatability of an implementation of Y is identical to the static hazard-free covering requirement to ensure a hazard-free implementation of Y for any input transition that spans I.Thus, for the given example of Fig. 8, the following states the relation between three-valued simulation and the transient (asynchronous) behavior:

If the implementation of Y is *not correctly simulatable* to one over the input combination $X11$, then each input transition spanning $X11$ (i.e., $011 \rightarrow 111$ and $111 \rightarrow 011$) has a *static logic hazard* for the same implementation.

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

The remainder of this section is devoted to defining several notions related to simulatability (Section VI-B2) and hazard freedom (Section VI-B3), and then formally proving the above correspondence between the two (Section VI-B4).

2) Three-Valued Simulation of a Network: The following definitions formalize the notion of three-valued simulation.

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

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

□

For example, the binary vector $\beta = 110010$ is covered by three-valued vector $\alpha = 1X001X$. The next definition formalizes the notion of three-valued simulation of a single-output combinational gate.

Definition 2 (Three-Valued Simulation of a Gate): Given a gate G corresponding to a Boolean function f of n variables, f: $\{0, 1\}^n \rightarrow \{0, 1\}$, and given a three-valued input vector α, the gate output is simulated by

$$\begin{cases} 0, & \text{iff } f(\beta) = 0 \,\forall\, \beta \text{ covered by } \alpha \\ 1, & \text{iff } f(\beta) = 1 \,\forall\, \beta \text{ covered by } \alpha \\ X, & \text{iff } f(\beta_0) = 0,\ f(\beta_1) = 1 \\ & \quad \text{for some binary vectors} \\ & \quad \beta_0, \beta_1 \text{ covered by } \alpha. \end{cases}$$

□

The definition states that a gate is simulated to X under a three-valued vector α if the vector covers at least one minterm where G has value 1 and at least one minterm where G has value 0; otherwise, the gate is simulated to the appropriate binary value, 1 or 0. Given a three-valued input α, Definition 2 can be generalized from gate simulation to a gate network simulation, by a topological traversal from the inputs toward the output, applying Definition 2 once to each gate.

3) Hazard Simulation of a Network: The following presents basics of hazards in combinational logic. First, Kung's nine-valued hazard simulation algebra [12] is reviewed, and then the notion of hazard simulation is formalized.

Kung introduced a hazard simulation algebra that classifies a transition on a wire into one of nine values $\{\mathbf{0}, \mathbf{1}, \uparrow, \downarrow, S0, S1, D+, D-, *\}$. The first two values, $\mathbf{0}$ and $\mathbf{1}$, represent *hazard-free* static $0 \rightarrow 0$ and static $1 \rightarrow 1$ transitions, respectively. Values $\uparrow$, $\downarrow$, $S0$, $S1$, $D+$ and $D-$ are *transient values* and represent transitions and hazards. $\uparrow$ and $\downarrow$ denote hazard-free dynamic $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions, respectively. $S0$ and $S1$ denote *hazardous* static $0 \rightarrow 0$ and $1 \rightarrow 1$ transitions, respectively. $D+$ and $D-$ represent *hazardous* dynamic $0 \rightarrow 1$ and $1 \rightarrow 0$ 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 $\delta_i \in \{\mathbf{0}, \mathbf{1}, \uparrow, \downarrow, S0, S1, D+, D-\}$.

In order to relate three-valued simulation to hazard freedom, it is important to give basic definitions for hazard freedom. The

first is the classical notion of an *atomic gate* in the context of hazard freedom.

Definition 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 1 (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 $m = m_1 \ldots m_n \in \{0, 1\}^n$ such that

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

Proof: In a hazard model that assumes arbitrary gate and wire delays, worst case behavior is assumed [26], [12]. Hence, if a minterm may be reachable under the given input transition by some sequence of permitted events on the set of wires x, it is assumed reachable. ■

Proposition 2 (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 the set of all the inputs reachable on this transition be noted 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. Therefore, we have the following.

a) If $f(\beta) = 0 \quad \forall \beta \in \Lambda$, the gate output stays at zero throughout the transition and is therefore hazard free: **0**.
b) If $f(\beta) = 1 \quad \forall \beta \in \Lambda$, the gate output stays at one 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 3)—if at all times the inputs seen by the gate are those for which f evaluates to zero, then under an atomic gate model, the gate output must stay constant at zero. Part b) is proved similarly. Part c) follows as well, since during the input transition, 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 1 (reachable inputs), the output will produce a transient value, i.e., one of $\{\uparrow, \downarrow, S0, S1, D+, D-\}$. ■

The above definitions can easily be generalized to hazard simulation of a gate network: given an input transition δ, hazard simulation of a gate network is performed by a topological traversal from the inputs toward the output, applying Proposition 6.1 once to each gate.

4) Transformation—Three-Valued Vector $\longrightarrow$ Input Transition: Based on the above definitions, we now formulate a *natural transformation* between an *input vector* α (used in three-valued simulation) and a corresponding *input transition* δ (used in hazard simulation, under Kung's nine-valued algebra).

In the following, assume that α is an arbitrary three-valued vector whose ith bit is α_i. A corresponding input transition δ is constructed by replacing each zero entry in α by a $0 \rightarrow 0$ transition (**0**), each one entry by a $1 \rightarrow 1$ transition (**1**), and each X entry by *any one* of the transient values $\{\uparrow, \downarrow, S0, S1, D+, D-\}$. More formally, the transformation is denoted by the operator τ

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

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

5) The Correspondence Theorem: We now have all the tools needed to state and prove the key theorem relating three-valued simulation and hazard freedom of an arbitrary multilevel network. The proof will essentially consist of a topological traversal of the circuit, applying the above propositions once to each gate.

We introduce the notation $\text{SIM}^f_{3-\text{valued}}(\alpha)$ is introduced to represent the result of three-valued simulation of a circuit with output f for the three-valued input vector α. Similarly, $\text{SIM}^f_{\text{hazard}}(\delta)$ denotes the result of hazard simulation of f for the nine-valued input vector δ. Given these definitions, the following key theorem lets us deduce the result of three-valued simulation of a circuit from the result of hazard simulation, and vice versa.

Theorem 1 (The Correspondence Theorem): Let f be a Boolean function implemented by a network G of atomic gates, let α be any three-valued input vector, and let $\delta \in \tau(\alpha)$ be any corresponding input transition. Then, the three-valued and hazard simulation results for the implementation G of the function f correspond, as follows:

$$\text{Sim}^f_{3-\text{valued}}(\alpha) = 0 \iff \text{Sim}^f_{\text{hazard}}(\delta) = \mathbf{0} \tag{7}$$
$$\text{Sim}^f_{3-\text{valued}}(\alpha) = 1 \iff \text{Sim}^f_{\text{hazard}}(\delta) = \mathbf{1} \tag{8}$$
$$\text{Sim}^f_{3-\text{valued}}(\alpha) = X \iff \text{Sim}^f_{\text{hazard}}(\delta) \in \{\uparrow, \downarrow, S0, S1, D+, D-\}. \tag{9}$$

That is

$$\text{Sim}^f_{\text{hazard}}(\delta) \in \tau\left(\text{Sim}^f_{3-\text{valued}}(\alpha)\right).$$

Proof: The above correspondence is shown to hold for *every* gate output ℓ in the network G. The proof is by induction on the "depth" of the subcircuit in the transitive fan-in of ℓ, where "depth" of this subcircuit 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 that 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 $i_1 \ldots i_n$. Let g represent the Boolean

function corresponding to the gate. Under three-valued simulation, these gate inputs are represented by a three-valued vector α_g of size 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 three cases.

Case 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 three-valued simulation, g is simulated to zero, by Definition 2. Thus, $\text{SIM}^g_{three-valued}(\alpha_g) = 0$. 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 zero in δ_g corresponds to a zero in α_g, and each one in δ_g corresponds to a one in α_g. Therefore, by Proposition 1, the reachable inputs during input transition δ_g are all covered by α_g. Therefore, by Proposition 2(a), the gate output is hazard free under input transition δ_g, with value $\mathbf{0}$. That is, $\text{SIM}^g_{\text{hazard}}(\delta_g) = \mathbf{0}$.

Case 2: Function g has value 1 for all inputs covered by α_g. By similar reasoning as above, $\text{SIM}^g_{3-\text{valued}}(\alpha_g) = 1$ and $\text{SIM}^g_{\text{hazard}}(\delta_g) = \mathbf{1}$.

Case 3: In this case, the gate evaluates to both zero and one; i.e., $g(\beta_1) = 0$ and $g(\beta_2) = 1$ for some β_1, β_2 covered by α_g. Then, by Definition 2, $\text{SIM}^g_{3-\text{valued}}(\alpha_g) = X$. By Proposition 1, all those inputs that are covered by α_g are reachable by a corresponding input transition. Therefore, both β_1 and β_2 are reachable during the input transition δ_g. By Proposition 2, $\beta_1, \beta_2 \in \Lambda$. Combining this result with Proposition 2(c), $\text{SIM}^g_{\text{hazard}}(\delta)$ must be a transient value in hazard simulation, i.e.,, one of $\{\uparrow, \downarrow, S0, S1, D+, D-\}$. ■

The above theorem states that, given an input vector α, a three-valued simulation of a gate network will result in value 0 (1) *if and only if* the corresponding hazard simulation on input transition δ is hazard free at $\mathbf{0(1)}$. The three-valued simulation will result in X *if and only if* the corresponding hazard simulation corresponds to an output change (either hazard free or hazardous).

Tthe key corollary to this theorem that gives a precise equivalence between nonsimulatability and existence of a static hazard. But first, precise definitions of "simulatability" and "nonsimulatability,"are needed, terms that we have thus far used informally..

Definition 4 (Simulatability/Nonsimulatability): Boolean function f be implemented by a network G of atomic gates. Let α be any three-valued input vector to be used to simulate the circuit output. Then, network G is said to be *simulatable* for input α if and only if one of the following holds:

1) $\text{SIM}^f_{3-\text{valued}}(\alpha) = 1$;
2) $\text{SIM}^f_{3-\text{valued}}(\alpha) = 0$;
3) $\text{SIM}^f_{3-\text{valued}}(\alpha) = X$, and $f(\beta_0) = 0$ and $f(\beta_1) = 1$ for some binary inputs β_0, β_1 covered by α.

G is said to be *nonsimulatable* for input α if it is not simulatable for α. For example, if $\text{Sim}^f_{3-\text{valued}}(\alpha) = X$, whereas $f(\beta) = 0$ *for all* binary inputs β covered by α, then G is nonsimulatable; clearly, the simulation does not accurately represent the value of the Boolean function. □

It can be easily proved that there are only two cases in which a network G can be nonsimulatable:

1) $\text{SIM}^f_{3-\text{valued}}(\alpha) = X$ and $f(\beta) = 1$ for all β covered by α;
2) $\text{SIM}^f_{3-\text{valued}}(\alpha) = X$ and $f(\beta) = 0$ for all β covered by α.

In each case, given a nonsimulatable implementation, three-valued simulation yields the value X even though f is either functionally equal to one over all inputs covered by α or functionally equal to one over all such inputs.

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

Corollary 1 (Nonsimulatability $\Longleftrightarrow$ Static Logic Hazard Transition): Let f be a Boolean function implemented by a network G of atomic gates, and let α be any three-valued input vector. If G is *nonsimulatable* for three-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 any input transition $\delta \in \tau(\alpha)$, then G is nonsimulatable for the three-valued vector α.

Proof: Suppose that G is nonsimulatable for input vector α. Then, by Definition 4, $\text{SIM}^f_{3-\text{valued}}(\delta)$ must equal X. As a result, Theorem 1 implies that $\text{SIM}^f_{\text{hazard}}(\delta) \in \{\uparrow, \downarrow, S0, S1, D+, D-\}$. That is, $\text{SIM}^f_{\text{hazard}}(\delta)$ is a *transient*. However, Definition 4 also implies that f is either functionally equal to zero over all inputs covered by α, or functionally equal to one over all such inputs. Hence, under input transition δ, the function itself is stable at one (or zero) throughout the entire transition, and therefore is a static 1 (or static 0) transition. A transient at the output therefore indicates a logic hazard. Therefore, $\text{SIM}^f_{\text{hazard}}(\delta)$ must be a static logic hazard, $\mathbf{S1}$ (or $\mathbf{S0}$).

To prove the converse, assume that G has a static logic hazard for some transition $\delta \in \tau(\alpha)$. Then, by Theorem 1, $\text{SIM}^f_{3-\text{valued}}(\delta) = X$. However, by definition of a static logic hazard, f is either functionally equal to zero over all inputs covered by α or functionally equal to one over all such inputs. Therefore, by Definition 4, G is nonsimulatable for three-valued input α. ■

6) Summary: In summary, the key result of this subsection is that, given an input vector, to ensure three-valued simulatability, the circuit should be *static logic hazard-free* for a given set of input transitions. (These input transitions will correspond to the input vectors taken from the group face sequence.) Conversely, any circuit realization that is free of static logic hazards for those input transitions is also three-valued simulatable.

C. Combinational Logic Synthesis for Initializability

The correspondence between nonsimulatability and the existence of static logic hazards (Corollary 1) provides a complete method for the synthesis of initializable logic, after state assignment is complete. The input vectors, which are applied to the circuit, are those that arise in the group face simulation sequence (when applying the functional initialization sequence). *Three-valued simulatability* is guaranteed by synthesizing a combinational circuit that is *static hazard free* for each of the corresponding input transitions.

In particular, the method is as follows: 1) identify the input transitions that span the three-valued input vectors encountered in the group face sequence and 2) synthesize a circuit that is free of static hazards for those input transitions. Such static hazard-free logic can always be synthesized, for any set of (function hazard-free) input transitions, for both two-level and multilevel logic [10], [19].

Two-Level: For the special case of a two-level AND-OR implementation, the conditions for hazard freedom have been presented in [10], [19], and [26]. To eliminate static logic hazards, constraints imposed on logic synthesis make use 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 [19], [24]. 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 [19].[5]

Multilevel: Two different approaches can be used to synthesize multilevel logic that is static hazard free for the given input transitions. First, hazard-free circuits can be synthesized in two steps: 1) perform two-level hazard-free logic synthesis (as above), then 2) apply *hazard-nonincreasing multilevel transformations,* which do not introduce any static hazards [12]. A wide range of algebraic transformations have been identified as hazard nonincreasing, including factorization and De Morgan's law. Alternatively, direct methods can be used for the synthesis of hazard-free multilevel circuits based on binary decision diagrams (BDDs) [13].

VII. Results

The two constrained synthesis steps—state assignment and logic minimization—have been combined into a new CAD synthesis tool called SALSIFY. The tool is targeted to incompletely specified synchronous FSMs. It can be used to synthesize both two-level and multilevel circuits.

Results are compared with three alternative methods on examples from the MCNC89 benchmark suite [14]. These methods include: 1) a *BASE* method, which uses the KISS optimal state assignment, but otherwise no additional constraints on state assignment or logic minimization; and 2) the Cheng–Agrawal (CA) method, which includes their constraints on state assignment, coupled with the KISS optimality constraints, but no constraints on logic minimization. Finally, to highlight the impact of our new *hazard-free logic minimization step* (Section VI), we run a hybrid method 3) CA + HF, which combines the existing Cheng–Agrawal state assignment method with the new hazard-free logic synthesis step. In each case, the focus is on two-level logic implementations. Results of the methods are compared with respect to two criteria: 1) *logical initializability* and 2) *overhead*.

Methodology: Results for 14 state machines from the MCNC89 benchmark suite are presented. Each machine is functionally initializable, and the same initialization sequence is used for each synthesis method. For each machine, the optimal state encoding constraints of [17] are first generated. Then, the appropriate initializability constraints for each of the different synthesis methods are generated (except for the *BASE* method, which uses none). The dichotomy constraints are then solved to obtain a final-state assignment. Next, two-level multioutput logic minimization is performed (either nonhazard-free using SIS's espresso-exact or hazard free [19], [24]) to synthesize a gate-level circuit. The circuit is then simulated with a three-valued simulator to verify whether it was logically initializable for the synchronizing sequence used for its synthesis.

A. Logical Initializability

Table I focuses on the most important property of the synthesized circuits: *logical initializability*. The table indicates if the gate-level implementation was actually initializable by the synchronizing sequence used for synthesis, when simulated by a three-valued simulator. As expected, the trend shows that logical initializability improves in moving across the table from left to right.

The *BASE* method fares poorest in logical initializability, whereas SALSIFY guarantees logically initializable circuits in every case. Note that in some cases (see, BASE, CA, CA + HF), the synthesized circuit happens to be logically initializable, while the method itself does not include sufficient constraints to guarantee initializability. In these cases, either the state assignment happens to satisfy our additional new constraints (DCICs), even though these constraints were not included in the method; or the logic synthesis step happens to ensure hazard-free logic, even though the method did not require it.

For example, note that for two benchmarks $dk27$ and $dk512$, the CA + HF method does not guarantee initializability, but the actual implementations happen to be initializable. However, an alternative implementation, which uses a different solution to the state assignment constraints of CA + HF that does not happen to satisfy our new required DCIC constraints, could be logically uninitializable.

A comparison of SALSIFY and CA highlights the effectiveness of our new method over the earlier Cheng–Agrawal approach: in the latter, only three out of 14 circuits are logically initializable, while all circuits produced by SALSIFY are logically initializable.

Finally, a comparison of the CA and CA + HF columns highlights the critical importance of our new combinational logic synthesis step for initializability: using the same state assignment, while 11 circuits in CA are uninitializable, all synthesized circuits happened to be logically initializable when our constrained logic synthesis method was included (CA + HF).

We should point out that one conclusion is to use the CA + HF method: generate a circuit, and test if it is logically initializable. While initializability is not guaranteed, it may in fact hold in many cases. Alternatively, one can use our RFECs + HF, with a pruned set of face-embedding constraints. However, only the full method, SALSIFY, which includes DCIC's, is guaranteed to produce a logically initializable circuit. Since DCIC's are so rarely needed, this is the most straightforward approach.

[5]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. Such a solution is expensive, but unnecessary; in practice, when an exact hazard-free minimizer is used, the overhead in satisfying hazard constraints is often negligible [19].

TABLE I
COMPARISON OF THE CORRECTNESS OF THE FOUR SYNTHESIS METHODS

Name	BASE	CA	CA+HF	SALSIFY
dk14	×	×	√	√
dk15	√(×)	√(×)	√	√
dk17	×	×	√	√
dk27	×	×	√ (×)	√
dk512	×	×	√ (×)	√
ex3	×	×	√	√
ex5	×	×	√	√
lion9	×	×	√	√
bbtas	×	√(×)	√	√
bbara	√(×)	×	√	√
beecount	×	×	√	√
train11	×	×	√	√
s8	×	×	√	√
shiftreg	√(×)	√(×)	√	√

Legend:
× The synthesized circuit was **uninitializable.**
√ (×) The synthesized circuit was **initializable,** but *not all implementations* of this benchmark example that can result from this method will be initializable. (The method does not guarantee initializability of circuit.)
√ *All implementations* of this benchmark example using this method are guaranteed **initializable.**

TABLE II
COMPARISON OF SYNTHESIS METHODS

Circuit Name	No. of states	Len. of Sync. Seq.	BASE			CA			CA+HF			SALSIFY			
			No. of $n \rightarrow 1$ encoding cons.	State code length	No. of gates	No. of $n \rightarrow 1$ encoding cons.	State code length	No. of gates	No. of $n \rightarrow 1$ encoding cons.	State code length	No. of gates	No. of $n \rightarrow 1$ encoding cons.	No. of $n \rightarrow k$ 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

B. Overhead

Table II evaluates the overhead of the four synthesis methods as measured by three parameters: 1) number of state encoding constraints, 2) state code length, and 3) number of gates.

Number of State Encoding Constraints: The column "no. of $n \rightarrow 1$ encoding cons." lists the total number of face-embedding constraints used in state assignment. These include the KISS optimality constraints, as well as any additional face-embedding constraints for initializability (either FECs or RFECs). Additionally, for SALSIFY, the number of DCICs is shown in the column "no. of $n \rightarrow k$ cons."

For all such 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 only a very rough indicator of the restrictiveness of those constraints. For example, a single dichotomy may subsume several smaller dichotomies [e.g., $\{(abc; d)\}$ is more restrictive than $\{(ab; d), (ac; d)\}$]. The table shows only moderate variance in the number of face-embedding constraints. Also, while the *BASE* method tends to have fewer constraints than the initializability methods (e.g., train11), this is not always the case (e.g., dk17).

Interestingly, SALSIFY required use of our the new DCICs for only two circuits (dk27, dk512), and, moreover, only one DCIC for each. These DCIC constraints were shown to be critical for guaranteeing initializability. Thus, whereas existing methods may not always achieve initializability, our method uses DCICs to guarantee initializability often at little cost.

State Code Length: Code length, or the number of state bits used to encode the machine, is another 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 only four examples, though, was one extra state bit used in CA and CA + HF, over the base method. Interestingly, SALSIFY produced state encodings that were the same length as codes produced by CA or CA + HF, except for one example, *lion9*, where SALSIFY produced a shorter encoding, using seven state bits instead of eight.

In sum, while the new face-embedding constraints, RFECs, are a pruned version of the Cheng–Agrawal FECs, this relaxation had little impact on resulting code lengths. It is possible that RFEC's will have a greater impact on larger examples. However, more important, the state code length across all of the initializability methods differed little from the BASE method.

Gate Count: The column "No. of gates" lists the number of gates used in the final two-level AND-OR circuit implementation. For the purpose of this comparison, it is assumed that each product term is implemented using one AND gate, and that all the products are summed together using one OR gate. From the table, it is clear that SALSIFY incurs low logic overhead over the BASE method in order to ensure initializability (215 gates total used by SALSIFY for the 14 examples versus 203 gates total used by BASE). A comparison with CA and CA + HF also shows that the gate counts of circuits produced SALSIFY compare favorably with those of CA and CA + HF.

VIII. Conclusions

This paper has presented the first sound and systematic method for the synthesis for logical initializability of synchronous FSM's. The method provides both a state assignment step and a combinational logic synthesis step.

For state assignment, two sets of dichotomy constraints were introduced. First, relaxed face-embedding constraints were presented. These constraints are safely pruned versions of existing face-embedding constraints [8]. Second, don't-care intersection constraints were introduced and were shown to be critical for initializability.

For combinational logic synthesis, it was first demonstrated that unconstrained logic minimization can render a circuit logically uninitializable under three-valued simulation. Necessary and sufficient conditions on combinational logic synthesis for initializability were presented. These conditions are identical to ensuring static logic hazard freedom for input transitions that correspond to three-valued vectors that arise when applying the initialization sequence. Finally, synthesis methods to generate two-level and multilevel logic for initializability were presented.

Combined together, given a functionally initializable specification, our synthesis method guarantees logical initializability for the resulting circuit under three-valued simulation. In addition, unlike existing methods, it can correctly handle incompletely specified finite-state machines and can produce multilevel circuits as well. Benchmark results show low logic overhead.

The basic results of this paper, especially the correspondence between three-valued simulatability and static hazard freedom, may also be applicable not only for logical initializability but also for the synthesis of fully simulatable multilevel circuits (e.g., [6]).

Acknowledgment

The authors would like to thank R. Fuhrer and M. Theobald of Columbia University for help with synthesizing the circuits. They would also like to thank N. Jha of Princeton University, J. Brzozowski of the University of Waterloo, and I. Pomeranz of The University of Iowa for discussions on initializability and testability and the reviewers for insightful comments.

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. Computer-Aided Design*, vol. 8, pp. 131–138, Feb. 1989.

[2] S. Banerjee, R. K. Roy, S. T. Chakradhar, and D. K. Pradhan, "Signal transition graph transformations for initializability," in *Proc. Eur. Conf. Design Automation (EDAC)*, 1994.

[3] M. A. Breuer, "A note on three-valued logic simulation," *IEEE Trans. Comput.*, vol. C-21, pp. 399–402, Apr. 1972.

[4] J. L. Carter, B. K. Rosen, G. L. Smith, and V. Pitchumani, "Restricted symbolic evaluation is fast and useful," in *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, 1989, pp. 38–41.

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

[6] S. J. Chandra and J. H. Patel, "Accurate logic simulation in the presence of unknowns," in *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, 1989, pp. 34–37.

[7] K. Cheng and V. Agrawal, "State assignment for initializable synthesis," in *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, 1989, pp. 212–215.

[8] ——, "Initializability consideration in sequential machine synthesis," *IEEE Trans. Comput.*, vol. 41, pp. 374–379, Mar. 1992.

[9] O. Coudert, "Two-level logic minimization: An overview," *Integration*, vol. 17, pp. 97–140, 1994.

[10] E. B. Eichelberger, "Hazard detection in combinational and sequential switching circuits," *IBM J. Res. Develop.*, vol. 9, pp. 90–99, Mar. 1965.

[11] J. S. Jephson, R. P. McQauarrie, and R. E. Vogelsberg, "A three-value computer design verification system," *IBM Syst. J.*, vol. 8, no. 3, pp. 178–188, 1969.

[12] D. S. Kung, "Hazard-nonincreasing gate-level optimization algorithms," in *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, 1992.

[13] B. Lin and S. Devadas, "Synthesis of hazard-free multilevel logic under multi-input changes from binary decision diagrams," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 974–985, Aug. 1995.

[14] R. Lisanke, "Finite-state machine benchmark set v1.0," in *Proc. Int. Workshop Logic Synthesis*, 1989.

[15] Y. Lu and I. Pomeranz, "Synchronization of large sequential circuits by partial reset," in *Proc. IEEE VLSI Test Symp.*, Apr. 1996, pp. 93–98.

[16] S. Mallela and S. Wu, "A sequential circuit test generation system," in *Proc. Int. Test Conf.*, 1985, pp. 57–61.

[17] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[18] A. Miczo, "The sequential ATPG: A theoretical limit," *Proc. Int. Test Conf.*, pp. 143–147, 1983.

[19] S. M. Nowick and D. L. Dill, "Exact two-level minimization of hazard-free logic with multiple-input changes," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 986–997, Aug. 1995.

[20] I. Pomeranz and S. M. Reddy, "On the role of hardware reset in synchronous sequential circuit test generation," *IEEE Trans. Comput.*, pp. 1100–1105, Sept. 1994.

[21] J.-K. Rho, F. Somenzi, and C. Pixley, "Minimum length synchronizing sequences of finite state machines," in *Proc. ACM/IEEE Design Automation Conf.*, 1993, pp. 463–468.

[22] A. Saldanha, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, "Satisfaction of input and output encoding constraints," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 589–602, May 1994.

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

[24] M. Theobald and S. M. Nowick, "An implicit method for hazard-free two-level minimization," in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 58–69.
[25] J. H. Tracey, "Internal state assignments for asynchronous sequential machines," *IEEE Trans. Electron. Comput.*, vol. EC-15, pp. 551–560, Aug. 1966.
[26] S. H. Unger, *Asynchronous Sequential Switching Circuits*. New York: Wiley-Interscience, 1969.
[27] T. Villa and A. Sangiovanni-Vincentelli, "NOVA: State assignment of finite state machines for optimal two-level logic implementation," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 905–924, Sept. 1990.
[28] J. A. Wehbeh and D. G. Saab, "On the initialization of sequential circuits," in *Proc. Int. Test Conf.*, 1994, pp. 233–239.
[29] ——, "Initialization of sequential circuits and its application to atpg," in *Proc. IEEE VLSI Test Symp.*, 1996, pp. 246–251.
[30] J. A. Wehbeh and D. G. Saab, "Initialization of sequential circuits and its application to ATPG," *J. Electron. Testing: Theory Applicat.*, vol. 13, no. 3, pp. 259–271, Dec. 1998.
[31] S. Yang and M. Cieselski, "Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization," *IEEE Trans. Computer-Aided Design*, vol. 10, p. 412, Jan. 1991.
[32] M. Yoeli and S. Rinon, "Applications of ternary algebra to the study of static hazards," *J. ACM*, vol. 11, no. 84, 1964.

Montek Singh received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Delhi, in 1993 and the M.S. degree in computer science from Columbia University, New York, in 1996, where he is pursuing the Ph.D. degree in computer science.

His research interests include asynchronous design and computer-aided design of VLSI systems, with a special focus on high-speed and low-power asynchronous pipeline designs.

Steven M. Nowick received the B.A. degree from Yale University, New Haven, CT, and the Ph.D. degree in computer science from Stanford University, Stanford, CA, in 1993.

He is an Associate Professor of computer science at Columbia University, New York. His research interests include asynchronous circuits, computer-aided digital design, low-power and high-performance digital systems, logic synthesis, and formal verification.

Prof. Nowick recently received two large-scale NSF ITR awards for research on aysnchronous systems. He also received an NSF Faculty Early Career Award (1995), an Alfred P. Sloan Research Fellowship (1995), and an NSF Research Initiation Award (1993). He received a Best Paper Award at the 1991 IEEE International Conference on Computer Design. He was a Best Paper Finalist at the 1993 Hawaii International Conference on System Sciences and at the 1998 IEEE Async Symposium. He was a Guest Editor of the February 1999 issue of the PROCEEDINGS OF THE IEEE on asynchronous circuits and systems. He also was Cofounder and Program Committee Cochair of the 1st IEEE "Async" Symposium (1994) and was Program Committee Cochair of the 5th IEEE "Async" Symposium (1999). He is a member of several international program committees, including ICCAD, DATE, ICCD, IWLS, Async, and ARVLSI.