

Synthesis-for-Initializability of Asynchronous Sequential Machines*

Montek Singh

Steven M. Nowick

Department of Computer Science
Columbia University
New York, NY 10027

Abstract

We present a new method for the synthesis-for-initializability of asynchronous state machines. Unlike existing approaches, our method incorporates a state assignment step. State assignment is critical, since initializability may be precluded by poor choice of state assignment. Also, we present precise conditions on hazard-free logic, to insure that the circuits are physically initializable.

1 INTRODUCTION

Asynchronous design is enjoying a resurgence, with a number of recent technical and practical advances [9]. In principle, asynchronous systems promise several advantages over synchronous systems: *lower power*, since an asynchronous component computes only when necessary; *improved performance*, since global clock distribution and synchronization can be avoided; and *modularity and ease of design*, since there are no global timing constraints. However, asynchronous design is subtle, since *hazards* and *race conditions* must be avoided for correct operation [1, 2].

Recently, several sound methods have recently been introduced to synthesize asynchronous state machines [3, 4, 5]. These methods are automated and produce low-latency machines which are guaranteed hazard-free at the gate-level. The tools have benefited from a number of hazard-free optimization algorithms, such as two-level [6] and multi-level [1, 7, 8] logic optimization.

Critical to the practical use of asynchronous design is *testability*. Several efforts have addressed the asynchronous testability problem [10, 11, 12, 13, 14]. However, an important related problem, *initialization* of asynchronous circuits, has received little attention.

Initialization is the process of driving a sequential circuit to a known current state, regardless of the power-up state of the circuit. This is an important first phase in the test generation of a sequential circuit [15]. A test generator typically assumes the circuit begins in an unknown state. A sequence of input vectors is derived, called an *initialization sequence*, which drives all state signals to a known *reset state*. If the synthesized implementation cannot be initialized, then a test generator or fault simulator which assumes an unknown initial state will be completely ineffective.

Several techniques have been proposed for initializability of *synchronous* state machines. One solution is to incorporate explicit reset hardware, though the cost is an

additional pin and added wiring. Wehbeh and Saab [16] and Pixley *et al.* [17] focus instead on finding an initialization sequence for a given implementation. An alternative solution is *synthesis-for-initializability*, where initializability is guaranteed by the synthesis procedure itself. In particular, Cheng and Agrawal [18] focus on the role of state assignment on initializability. Recent modifications to their approach have been proposed, which use more relaxed constraints [19].

A synthesis-for-initializability approach for *asynchronous* designs was first proposed by Chakradhar *et al.* [22, 21, 20]. This approach attempts to find a single-vector or multi-vector initialization sequence, assuming a given state assignment. The first method [22, 21] simultaneously searches for an initialization vector (or sequence) and don't-care assignment. If this method fails, the second method [20] modifies the specification by reducing its concurrency, to aid in producing a valid initialization.

The contribution of our paper is a general method for synthesis-for-initializability of asynchronous state machines. Unlike Chakradhar *et al.*, we provide a complete synthesis for initializability method, including a state assignment step. This feature is critical, since initializability may be precluded by poor choice of state assignment. We also indicate that the Chakradhar *et al.* conditions on logic implementations in some cases are insufficient: they may result in designs which are physically and logically uninitializable due to hazards. We present precise conditions to produce a two-level implementation which is both logically and physically initializable.

The paper is organized as follows. Section 2 presents background on synthesis for initializability of synchronous and asynchronous state machines. Section 3 gives an overview of our approach. Section 4 presents our method for single-vector initializability, where we perform state assignment. Section 5 considers the limited case where state assignment is already given. Results are given in Section 6, and Section 7 presents conclusions.

2 BACKGROUND AND PREVIOUS WORK

2.1 Synchronous Initializability

Cheng and Agrawal impose certain constraints on state assignment to translate functional initializability into logic initializability [18]. We first review their approach, then present recent extensions allowing more relaxed constraints on state assignment [19].

Cheng/Agrawal Approach: Consider the functionally initializable machine M in Fig. 1 [18]. At startup, the ma-

* This work was supported by an NSF CAREER Award MIP-9501880 and by an Alfred P. Sloan Research Fellowship.

chine can be in any state, i.e. the initial *state group* consists of all states: (S_1, S_2, S_3, S_4) . $I = 1001$ is a synchronizing sequence of M . The following is the trace of state groups that result as the input sequence 1001 is applied to M :

$$(S_1, S_2, S_3, S_4) \xrightarrow{1} (S_2, S_4) \xrightarrow{0} (S_1, S_3) \xrightarrow{0} (S_1, S_4) \xrightarrow{1} (S_4)$$

Associated with each state group, after state assignment, is its smallest containing cube, called the *group face*. Each group face is represented by a 3-valued vector. Thus, if state encoding $(S_1:01, S_2:00, S_3:10, S_4:11)$ were used, the following group faces would result: $(XX) \xrightarrow{1} (XX) \xrightarrow{0} (XX) \xrightarrow{0} (XX) \xrightarrow{1} (XX)$. This sequence does not converge to a single state, therefore the circuit realized is *logically uninitializable*. A 3-valued simulator can only simulate group faces, not state groups, so there is a loss of information. Therefore, the goal 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.

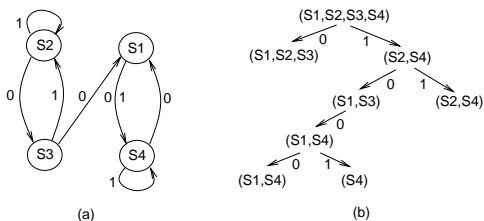


Figure 1: Example FSM and synchronization tree.

Cheng and Agrawal state that the problem can be solved by introducing an additional set of *face-embedding constraints* into the state encoding step. Each of these is a $k \rightarrow 1$ *dichotomy constraint*, $(G_i; s_j)$.¹ In particular, a $k \rightarrow 1$ dichotomy of the form $(G_i; s_j)$ is introduced for every symbolic state s_j not present in the state group G_i . That is, a symbolic state that does not belong to a state group is forbidden from being embedded in it. This applies to all state groups encountered when a synchronizing input is applied to the machine. For the example of Fig. 1, the dichotomies generated are: $\{(S_2S_4; S_1), (S_2S_4; S_3), (S_1S_3; S_2), (S_1S_3; S_4), (S_1S_4; S_2), (S_1S_4; S_3)\}$. A state assignment satisfying these dichotomies is: $(S_1:11, S_2:00, S_3:10, S_4:01)$, resulting in a correct 3-valued simulation: $(XX) \xrightarrow{1} (0X) \xrightarrow{0} (1X) \xrightarrow{0} (X1) \xrightarrow{1} (01)$.

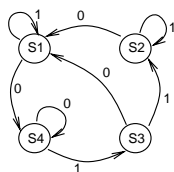


Figure 2: Example to show relaxed state encoding.

Relaxed State Assignment: When considering encoding of symbolic states, we show that the Cheng/Agrawal

¹An $n \rightarrow k$ dichotomy constraint $(X; Y)$ [1] is the stipulation that the smallest containing cubes of X and Y not intersect. Here, X and Y are sets of states of cardinality n and k respectively. A dichotomy is satisfied if some state bit has value 1 for all states in X and value 0 for all states in Y , or vice versa.

constraints may be relaxed. Consider the machine in Fig. 2. A synchronizing sequence for the machine is 100, resulting in the state group sequence: $(S_1, S_2, S_3, S_4) \xrightarrow{1} (S_1, S_2, S_3) \xrightarrow{0} (S_1, S_4) \xrightarrow{0} (S_4)$. The dichotomy constraints produced by the Cheng/Agrawal method are $\{(S_1S_2S_3; S_4), (S_1S_4; S_2), (S_1S_4; S_3)\}$ and a minimum length state encoding satisfying these constraints requires 3 bits. In fact, the dichotomy $(S_1S_2S_3; S_4)$ is unnecessary. At the second time instant, the next-state value of S_4 is S_4 , which happens to belong to the next state-group, (S_1, S_4) . Therefore, if we let S_4 be embedded in the group-face of (S_1, S_2, S_3) , no harm would accrue. Thus, it is safe to delete the dichotomy $(S_1S_2S_3; S_4)$. The resulting state assignment is $(S_1:00, S_2:01, S_3:11, S_4:10)$, still yielding a correct 3-valued simulation: $(XX) \xrightarrow{1} (XX) \xrightarrow{0} (X0) \xrightarrow{0} (10)$. This solution required only two bits to encode the states, whereas the previous solution requires three. Therefore, we prune the list of dichotomies proposed by Cheng/Agrawal—we delete the dichotomy $(G_i; s_j)$ whenever $NS(s_j, I_i) \in G_{i+1}$, because this is a “safe” embedding. These conditions can be used as the starting point to produce a state encoding that insures logical initializability for synchronous state machines.

2.2 Asynchronous State Machines

We now briefly review background on asynchronous state machines. In its simplest form, an asynchronous state machine is a *Huffman machine* (see Figure 3), with primary inputs, primary outputs, and fed-back state variables [1]. State is stored on the feedback loops, which may have attached delay elements.

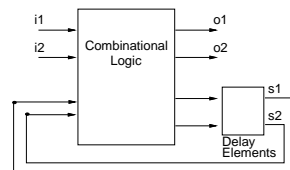


Figure 3: Block diagram of asynchronous state machine

Since there are no clocked latches or flipflops on the feedback lines, the state may change through multi-stepping. That is, an input vector may cause a state change which feeds back, causing another change, and so on.

Unlike synchronous machines, these machines may malfunction due to *critical races* and *logic hazards*. A critical race occurs if the machine may stabilize in an incorrect stable state during a state change [1]. This problem can be avoided by using *critical race-free (CRF)* state assignment. A logic hazard occurs if the combinational logic for outputs or next-state may glitch during an input or state change [1]. This problem can be avoided by constrained combinational logic synthesis (see [6, 2]).

A number of design methods can be used to synthesize these machines, such as *UCLOCK* [5] and *3D* [4]. These two methods insure critical race-free state assignment and hazard-free logic implementation. In this paper, we introduce constraints on state assignment and logic synthesis to be satisfied *in addition* to these CRF and hazard-free requirements, in order to insure initializability.

For the sequel, it will be useful to distinguish some classes of asynchronous finite-state machines. A flow table

	0	1
00	01	01
01	01	xx
11	11	11
10	00	xx

(a)

	0	1
00	01	01
01	01	1x
11	11	11
10	01	x1

(b)

Figure 4: Complete vs. partial don't-cares; single- vs. multi-hop state transitions.

for an FSM may have entries that are don't-cares ("DC's"). These can be either *complete DC's*, as in Fig 4a, or *partial DC's*, as in Fig 4b. By "complete DC's", we mean that each next-state entry in the table is either completely specified or completely unspecified. Most recent asynchronous synthesis methods produce machines with complete DC's only [5, 4]. The case of partial DC's is much less common; it may arise in the case that fed-back and incompletely-specified outputs are used. In our state assignment method of Section 4, we assume all DC's are complete DC's. This allows us to build on our existing synchronous initializability approach, described in Section 2.1. In Section 5, however, we handle both partial and complete DC's in our combinational logic synthesis step.

Further, state transitions can be *single-hop* or *multi-hop*. The flow table of Fig 4a, has a multi-hop transition $10 \xrightarrow{0} 00 \xrightarrow{0} 01$, whereas the corresponding transition in Fig 4b is single-hop ($10 \xrightarrow{0} 01$). Several existing asynchronous synthesis methods produce only single-hop tables, such as UCLOCK [5].

2.3 Asynchronous Initializability

We now review the basic asynchronous synthesis for initializability method of Chakradhar *et al.* Chakradhar *et al.* show in [22] that assignment of don't-cares during the synthesis procedure is intimately related to the initializability of the final implementation. They introduce a novel method that selectively assigns don't-cares to obtain an initializable implementation.

The method starts with a specification of an asynchronous circuit in terms of a Signal Transition Graph (STG). An *Asynchronous State Graph* (ASG) is derived from an STG using a set of *firing rules*. It is also assumed that a state assignment is implicitly specified by the STG itself (several recent STG synthesis methods relax this assumption, and perform explicit state assignment [23]). From the given state assignment and the list of signal transitions, Karnaugh-maps are constructed for each of the next state bits. In general, these Karnaugh-maps will have don't-care entries. The goal of this method is to simultaneously (i) assign values to don't-cares, and (ii) find an initialization vector sequence.

The basic idea of the approach is that of *combinational initialization*: each of the next state bits is initialized in some sequence by a given input vector. Once a state bit is initialized to 0 or 1, it remains stable at that value. Thus, if a certain time a certain state bit is to be initialized to 1 (or 0) the *combinational* next state logic for that state bit should consistently evaluate to 1 (or 0), irrespective of the values of signals that have not yet been initialized. This approach is a good match for asynchronous state machines, since the feedback path has no clocked latches or flipflops: as each next state bit is initialized, it feeds back

as initialized current-state bit.

To initialize the first next-state bit, each next-state Karnaugh map is inspected to see if its output can be determined by some input vector (partially or completely specified). That is, if a portion of the K-map, disregarding don't-cares, has either all 1's or all 0's, then restricting the input to lie in that region will determine the output of the K-map. For this to succeed, all don't-cares in this region must be assigned the same output value as that assigned to the specified minterms in this region. (If all the points in this region are don't-cares, they can be assigned either value.) Each of these "regions" is a cube in the state space.

F		ab			
	fg	00	01	11	10
	00	1	1	0	0
	01	1	1	0	X
	11	X	1	0	1
	10	1	1	0	0

G		ab			
	fg	00	01	11	10
	00	1	0	1	1
	01	X	0	0	1
	11	0	1	0	1
	10	1	0	0	1

Figure 5: Karnaugh Maps for the initializability example.

Example. Let f and g be the next state functions for a two-bit state encoding, and let a and b be the input bits, as shown in Fig 5. All bits are assumed initially unknown. In the region where $a = 0$, the function f evaluates to 1 or X *irrespective* of the current values of b , f and g . Therefore, $0XXX$ ($a = 0, b = X, f = X, g = X$) is a partial assignment that initializes f to 1 provided all the don't-cares of f in $0XXX$ are changed to 1's. At the end of one such step, some input bits and some state bits will have been initialized. Thus, the current "region of interest" will have been narrowed down to a sub-cube of the search space because some input bits and some state bits have been determined. For the example above, the new region of interest that needs to be further explored is $0X1X$ ($a = 0, b = X, f = 1, g = X$). Within this smaller region, the steps just mentioned are repeated. That is, each of the K-maps is then inspected to see if its output can be determined solely by the input and state bits that have already been initialized, possibly assigning values to input bits and don't-cares. Fig 6 shows the *initialization sequence tree* for the above example. The bits shown are in the order $abcf$ g. The circular nodes marked "F" are failure nodes, whereas the success nodes hold the corresponding solutions. \square

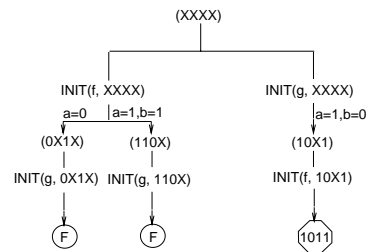


Figure 6: Initialization Sequence Tree

The algorithm of Chakradhar *et al.* implements this approach by implicitly enumerating the entire search space consisting of all possible assignments to input bits and don't-cares. A tree is implicitly built where each edge corresponds to either an assignment to certain input bits, or

to some don't-cares. If, at any step, more than one option exists, the algorithm not only tries all of them recursively, but it tries them in all possible orderings until failure is reached on all branches.

In more detail, the method uses a function called **INIT**. **INIT** is repeatedly called for each uninitialized state bit. The function examines the K-maps for the next state logic for each state bit, and determines initialization vectors based on the current partial assignment of state and input bits and don't-cares. The derivation of initialization vectors may require the assignment of currently unassigned inputs and don't-cares. If, at any step, more than one choice is available, each of them is explored by recursively calling **INIT**. The procedure either produces a list of initialization vectors (along with associated don't-care assignments), or reports that no single-vector initialization sequence exists. A drawback of this approach is that in the worst case, it tries all possible permutations of choices that lie along a path in the tree. We show later that such repetition can be avoided.

If at a certain step the procedure determines that a single-vector initialization sequence does not exist, then search is restarted from that point onwards using a different input assignment.

If the method fails to synthesize an initializable circuit, Chakradhar *et al.* indicate that the original STG specification may be modified to achieve initializability. Banerjee *et al.* present such STG transformations for initializability in [20]. However, a drawback of this approach is a reduction in concurrency.

3 OVERVIEW OF OUR APPROACH

We now present two new approaches for synthesis-for- initializability.

Section 4 introduces a complete synthesis-for-initializability method, including a state assignment step. Our method is restricted to single-vector initialization, for state machines with “complete don't cares”. Many existing synthesis methods fall into this category [5, 4, 3]. We first show how state assignment has a direct impact on the initializability of an asynchronous state machine. We then extend the synchronous state assignment approach of Section 2.1 to the asynchronous case. Finally, we present hazard-free logic requirements which must be satisfied to insure both logical and physical initializability.

Section 5 presents our second approach, which assumes a *given* state assignment. In this case, we can only modify combinational logic synthesis to achieve, or preserve, initializability. As with the approach of Chakradhar *et al.*, we allow both “partial” and “complete don't cares”, and consider both single-vector and multi-vector initialization. Our method includes efficient techniques to prune the initialization search space. In addition, we address issues of hazard-free logic requirements to insure initializability during multi-vector initializability, which have not been considered in previous work.

4 STATE ASSIGNMENT FOR SINGLE-VECTOR INITIALIZABILITY

This section presents a complete path to synthesize circuits that are initializable by a single-vector. The procedure transforms an asynchronous FSM specification into a

circuit which is physically initializable at the logic level.

An asynchronous circuit is initializable by a single-vector if application of the vector drives the circuit from an unknown start state to a unique known state. The transition from the unknown state to the known one may involve multiple hops while the input is held constant. This model of initializability is the same as the one used in [22] and will be used throughout this section. In the context of synchronous initializability of Section 2.1, this corresponds to application of a single vector repeated over (possibly) several time-steps.

The flow table of Fig 7(a) illustrates the impact of state assignment on asynchronous initializability.

		0	1
S_1		S_2	S_2
S_2		S_3	S_2
S_3		S_3	S_4
S_4		-	S_1

(a)

		0	1
01		00	00
00		11	00
11		11	10
10		xx	01

(b)

		0	1
00		01	01
01		11	01
11		11	10
10		xx	00

(c)

Figure 7: Impact of state encoding on initializability

Example. Assume that the state encoding of Figure 7(b) is used. If the input is held constant at 0, a 3-valued simulator will derive the following sequence of 3-valued vectors for the state bits: $XX \xrightarrow{0} XX \xrightarrow{0} XX \dots$. Therefore, the circuit is not logically initializable. If, on the other hand, the state encoding of Figure 7(c) is used, the 3-valued simulation trace that results is: $XX \xrightarrow{0} X1 \xrightarrow{0} 11$, so the circuit is logically initializable. \square

In summary, the state assignment step should incorporate initializability considerations to translate functional initializability (at the FSM level) into logic initializability (at the simulation level). Logic synthesis is then modified to insure initializability at the gate-level.

4.1 The Single-Hop case

For flow tables with “single-hopping” only (see Section 2.2), initializability is surprisingly simple. We first highlight this important special case, somewhat informally, and then in Section 4.2 present the general approach for “multi-hopping” which subsumes it.

Consider the state diagram in Figure 8 and corresponding flow table in Figure 8(a). This machine obeys the single-hopping constraint: each unstable state leads directly to a stable state. The following theorem characterizes the notion of a *functional initialization vector* for asynchronous single-hop machines.

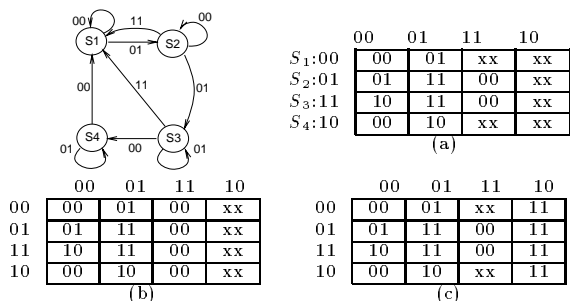


Figure 8: FSM and Flow Table for the Single-Hop example.

Theorem 4.1. Let F be an asynchronous flow table which has “complete don’t cares” and “single-hopping” only. Then an input vector I is a valid single initialization vector *if and only if* F has at most 1 distinct next-state entry on I (that is, if and only if all specified transitions on I have the same destination).

Proof. Clearly, if there are no specified transitions, then all the next-state entries on I are DC’s, and can be assigned the same arbitrary value, as in Figure 8(c). If there are one or more specified transitions, all of them to the same destination state s_0 , then the remaining DC entries can also be assigned the value s_0 . Then, the machine is initializable to s_0 in one hop, as shown in Figure 8(b). However, if, on input I , there are transitions to more than one destination state, then, obviously, the machine cannot be functionally initialized in one hop by I . \square

In summary, if a flow table F is single-hop only, the following simple procedure can be used to find all single-vectors for initialization: select any input vector I where F has at most 1 distinct next-state entry.

Once a single functional initialization vector is selected, don’t-care assignment is performed. The don’t care assignments shown in Figures 8(b) and (c) use a simple “direct” technique for assignment. More optimal techniques for don’t-care assignment are introduced in the next subsection as well. Finally, in Section 4.2.4, we show that a *hazard-free logic covering requirement* must be met, to insure both logical and physical initializability. Given an appropriate don’t-care assignment, and hazard-free logic cover, application of the vector will physically initialize the machine to a single stable state, *regardless* of the start-up state of the machine. These results are a special case of the general procedure described in detail in Section 4.2.

Note that state assignment will have *no impact* in this case. Though we are free to choose a state assignment, Section 4.2.2 indicates that initializability is independent of the choice of state assignment.

4.2 The Multi-Hop case

For FSM’s that have specified multi-hop transitions, state assignment *has* an impact. In this section, we present a new method producing a state assignment for initializability machines with multi-hop transitions. Our state assignment method is essentially a special-case of the procedure presented in Section 2.1 for synchronous circuits. However, additional issues specific to asynchronous circuits must be dealt with—these are discussed in Sections 4.2.1 through 4.2.4.

4.2.1 Synchronizing sequence

As in Section 2.1, the first step in our synthesis procedure is to find a synchronizing sequence. However, asynchronous machines have different operation than synchronous machines, since there are no clocked latches or flipflops. Therefore, the nature of a *valid synchronizing sequence* for asynchronous machines is different.

There are two key differences. First, in the asynchronous case, the input vector must drive the machine to a single *stable* state; if the state were unstable, the machine would never be in a known state. Therefore, we require a synchronization sequence, consisting of a single-vector, which leads to single stable state. Also, note that “multi-

hopping” under a single input vector corresponds to a synchronization sequence where I may repeated over several time-steps. This is a restriction on choice of synchronization sequence.

A second difference between asynchronous and synchronous state machines is that the former are usually *incompletely specified*, *i.e.*, contain don’t-care entries. However, synchronization sequences usually apply only to completely-specified machines [18]. We must extend the definition of synchronizing sequence to also include those that result in an empty state group. As an example, the FSM of Figure 8 has no specified transitions on input 10. We consider this input vector to be a valid synchronizing sequence, leading to the *empty* state group: $(S_1, S_2, S_3, S_4) \xrightarrow{10} ()$.

Incorporating the above two modifications into existing techniques for finding synchronizing sequences is easy. For example, the method of building a synchronization tree can be constrained to follow only those paths which have edges labeled by a *constant* vector, which terminate in a stable state, or in an empty state group. If no such initialization vector is found, the FSM is not initializable by a single-vector.

This notion of single-vector synchronizing sequence is summarized by the following general theorem, which presents the necessary and sufficient conditions on a *functional asynchronous initialization vector*.

Theorem 4.2. Let F be an asynchronous flow table which has “complete don’t cares” (allowing both single- and multi-hopping). Then an input vector I is a valid single initialization vector, *if and only if* F has at most 1 stable state in column I and there are no cycles.

Proof. Clearly, if there is a cycle on input I , or if there are two or more stable states on input I , then I cannot be an initialization vector. On the other hand, if there is exactly 1 stable state, s_0 , and there are no cycles on input I , then it is clear that all don’t-cares can be assigned such that all unstable states eventually converge to s_0 (one such assignment is to change all don’t-cares to s_0). If there are no stable states and no cycles, then there must exist some state s_0 which has a don’t-care next-state entry. This don’t-care can then be made the stable state, and any remaining don’t-cares assigned accordingly. \square

In the sequel, we will refer to s_0 as the *destination state* of vector I .

A valid initialization vector is a *functional initialization sequence* for an asynchronous machine. The goal of synthesis-for-initializability is to translate functional initializability of the asynchronous specification into physical (logical) initializability in the implementation. In the remaining subsections, synthesis steps are introduced which insure that initializability is preserved.

4.2.2 State Assignment

Once a single repeated vector synchronizing sequence is found, the entire state assignment procedure of Section 2.1 is carried out—the sequence of state groups is enumerated, face-embedding constraints are generated, and a state encoding that satisfies these constraints is produced. In general, *critical-race-free* constraints must be generated by the synthesis method [1]. These require added constraints,

which are already addressed by existing synthesis methods [5, 4].

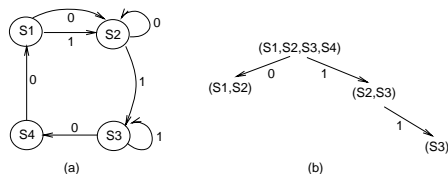


Figure 9: Example of an asynchronous FSM and its synchronization tree.

Example. Consider the FSM of Fig 9a. The synchronization tree and the synchronizing sequences are shown in Fig 9b. Suppose the input sequence (1, 1) is chosen. The following is the resulting sequence of state groups:

$$(S_1 S_2 S_3 S_4) \xrightarrow{1} (S_2 S_3) \xrightarrow{1} (S_3) \quad \square$$

Note that the state groups exhibit a *telescoping* property—every state group is a subset of its predecessor. This telescoping property among state groups always holds if the input sequence consists of a single repeated vector. Following is the face-embedding constraint generated: $(S_2 S_3; S_1)$. A state encoding which satisfies this constraint is: $(S_1:00, S_2:01, S_3:11, S_4:10)$.

As a special case, for *one-hop* machines, a synchronizing sequence always has length 1. For example, for the state diagram in Figure 8, the synchronizing sequence is $(S_1, S_2, S_3, S_4) \xrightarrow{11} (S_1)$. In this case, *no* face-embedding constraints are generated by our relaxed assignment method. As a result, for one-hop machines, there are no encoding constraints for initializability: any state assignment can be used.

4.2.3 The issue of Don't-cares

As discussed earlier, assignment to DC's may need to be constrained; a poor DC assignment can ruin initializability.

The starting point of don't-care assignment is the state assignment of the previous subsection. State assignment insures that the sequence of state groups in the initialization sequence is "tracked" by the corresponding 3-valued vectors, or group faces, after state assignment. Furthermore, as indicated, these state groups have a "telescoping" property: each state group is embedded in the preceding state group in sequence.

The goal of don't-care assignment is to insure that this tracking is still preserved, once don't-cares are assigned. For the single-vector case of this section, there is in fact a trivial DC assignment that always works: assign to every DC next state entry the value of the *destination state* for this input vector. Henceforth, DC assignments of this type will be called *direct assignments*.

Though this approach is always sufficient, it may be suboptimal, as shown by the following example.

	0	1
00	01	01
01	01	11
11	10	11
10	00	XX

(a)

	0	1
00	01	01
01	01	11
11	10	11
10	00	11

(b)

	0	1
00	01	01
01	01	11
11	10	11
10	00	X1

(c)

Figure 10: Example of DC assignments: direct vs. incremental

Example. Fig 10a shows a flow table for the example of Fig 9 after state assignment. Fig 10b shows a direct DC assignment, using input vector 1: the DC in state 10 is assigned to the destination state 11. Fig 10c shows another DC assignment that constrains a fewer number of DC's, but still insures initializability, as shown by the following simulation trace: $XX \xrightarrow{1} X1 \xrightarrow{1} 11$ \square

The *incremental DC assignment* of Fig 10c is clearly better. It can be implemented by a 2-level AND-OR logic that requires only one product term to implement the first state bit, whereas the assignment of Fig 10b requires two. Before presenting a formal procedure for assigning DC's, we propose the following definitions:

The *current state cube* is a 3-valued vector that represents all the states that the machine could be in at a given time instant. The initial current state cube consists of all X 's since the machine can power up in any state.

The i th bit of the current state cube is said to be *1-determinizable* at a given time for a given input if it is currently X and the i th bit of the next state entry of every state in the current state cube is either a 1 or an X . A similar definition holds for *0-determinizability*. State bit i is said to be *determinizable* if it is 1-determinizable or 0-determinizable. Below is an algorithm that performs DC assignment incrementally, by examining the next state functions for each state bit in a certain order:

```

algorithm assign_DC's(flow_table, current_state_cube, inp_vector)
1 while there exists a bit i that is determinizable on inp_vector
2   if bit i is 1-determinizable then
3     set_bits(i, current_state_cube, inp_vector, flow_table, 1)
4   else //bit i is 0-determinizable//
5     set_bits(i, current_state_cube, inp_vector, flow_table, 0)
6   output flow_table
end of algorithm

```

```

algorithm set_bits(i, current_state_cube, inp, flow_table, val)
  modify flow_table by setting bit i of the NS entries on
  input inp of all states in the current_state_cube, to val
  set bit i of current_state_cube to val
end of algorithm

```

Application of the above algorithm to the machine of Fig 9 produced the better DC assignment of Fig 10c.

It is easily seen that the algorithm always produces a valid DC assignment, irrespective of the choice made in step 1. A simple proof follows from the fact that determinizing bit i can never adversely affect determinizability of bit j ; it can only enhance determinizability of bit j . Based on this, we make an important observation that the order in which bits are chosen by our algorithm is unimportant as far as finding a solution is concerned. If a solution exists, a solution will always be found by our algorithm.

However, the choice made in step 1 (choosing an i , if more than one exists) may affect the optimality of the solution, as measured by the number of DC's assigned. A simple heuristic is to make the algorithm greedy—always choose that bit which requires assignment to the fewest DC's at that time instant. The next example illustrates this point.

Example. In Fig 11a, observe that both bits 1 and 2 are immediately determinizable. Fig 11b shows the resulting DC assignment if the bits are determined in the order 123. Fig 11c shows that a better DC assignment is produced if bits are determined in the order 213. \square

	0	1
000	111	-
001	110	-
011	111	-
010	111	-
110	xxx	-
111	111	-
101	110	-
100	xxx	-

	0	1
000	111	-
001	110	-
011	111	-
010	111	-
110	111	-
111	111	-
101	110	-
100	11x	-

	0	1
000	111	-
001	110	-
011	111	-
010	111	-
110	111	-
111	111	-
101	110	-
100	x1x	-

(a)
(b)
(c)

Figure 11: Example of different incremental DC assignments.

4.2.4 Physical Initializability

Even if a good state encoding and DC assignment are used, a physical implementation may still be uninitializable. This problem is due to hazards resulting from improper logic synthesis.

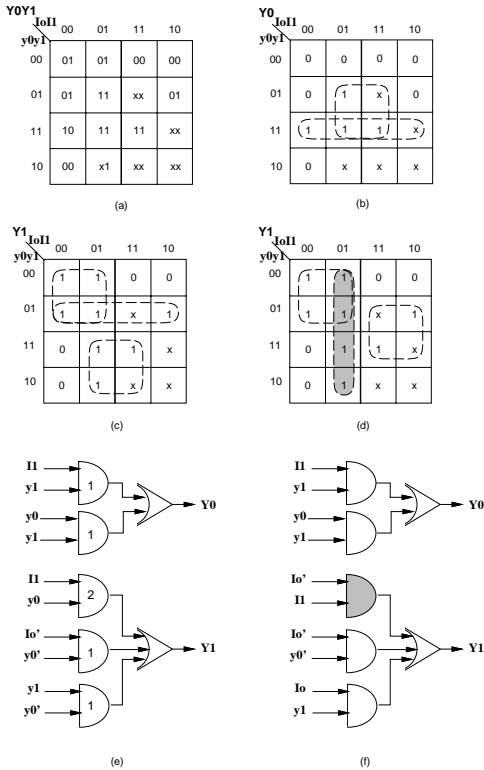


Figure 12: Example illustrating issue of hazards

Example. Fig 12 shows an example where initializability is ruined because of a poor choice of logic implementation. Fig 12(a) shows a two-bit state encoding for an FSM, with state variables y_0 and y_1 . Input 01 is a valid initialization vector, where initialization occurs in 2 hops. Suppose the covering of Fig 12b–c is used to synthesize a 2-level AND–OR implementation of the next state variables Y_0 and Y_1 . We show that the implementation of Fig 12(e) may be physically uninitializable.

Assume the machine starts up in state $y_0y_1 = 01$ and then the vector $I_0I_1 = 01$ is applied to the inputs. The next state function evaluates to 11. However, the static $1 \rightarrow 1$ transition of Y_1 is not hazard-free as there is no product term that remains 1 during the transition $01 \rightarrow 11$ (see

Fig 12c) [1, 6]. Assume the gates have delays as shown in (e). The next state function will glitch and evaluate to 10 momentarily. This glitch causes the machine to enter the state 10 instead. The next state value for 10 is 01. Assuming this transition to be a smooth one, the circuit ends up in state 01 once again. This machine, thus, cycles indefinitely and is physically uninitializable. \square

Let us now try to synthesize a hazard-free implementation for the same machine. Observe that during the first time step, y_1 is determinized to 1. The condition for hazard-freedom is that there be a product term that covers y_1 for all next state entries in the input column [1, 6]. Using the terminology of [6], cube $I_0' I_1$ is called a *required cube*. This cube *must* be contained in some product of the cover for Y_1 to avoid a static-1 hazard, as the current state is unknown and may be changing.

Once y_1 is determinized, the current state cube becomes $X1$. During the second time step, y_0 gets determinized to 1. To ensure there are no glitches, the required cube $I_0' I_1 y_1$ must be contained in some product of the cover for Y_0 .

Fig 12d shows a cover which makes the next state function for y_1 hazard-free. The corresponding circuit, shown in Fig 12f, will be initialized correctly to state 11, *irrespective of any gate delays*.

Precise conditions on 2-level AND–OR logic for insuring that physical initializability is not hampered by hazards can be stated as follows:

If state bit y_i is determinized to 1 at a certain time step, then it is required that a product of the cover of y_i span all the next state entries in the current state cube on the current input.

These can be visualized as “vertical” covering requirements within a column, as shown in Fig 12d. The algorithm of Section 4.2.3 for DC assignment can be augmented to generate these covering requirements at the same time a DC assignment is produced. These can then be passed to the logic minimizer for hazard-free logic minimization [6].²

5 LOGIC SYNTHESIS FOR INITIALIZABILITY, GIVEN A STATE ASSIGNMENT

The aim of this section is to synthesize an initializable circuit from an FSM whose state assignment is specified. Obviously, this setting affords less freedom than that of the previous section, where state assignment was not given.

Our algorithm accepts as input an incompletely specified flow table and attempts to output a sequence of input vectors and a DC assignment which make the circuit logically initializable. Thus, we solve the same problem as the one addressed by Chakradhar *et al.* [22].

Our method generates precise covering requirements for guaranteeing physical initializability. Moreover, we avoid needless exploration of regions of the search space where existence of a solution can be trivially ruled out. In particular, we perform simple tests to prune the search space.

We first present a procedure for single-vector initialization in Sections 5.1–5.3, and then consider multi-vector initialization in Section 5.4.

²Interestingly, these requirements are identical to the ones for 3-valued simulatability presented in [19] for synchronous circuits. Thus, the above requirements also insure that the circuit is initializable by a 3-valued simulator.

5.1 Synchronizing Sequence

Assume that the FSM specification is available as an incompletely specified flow table. Some of the next state entries in the table may be unspecified or incompletely specified. Thus, we allow not only entries that are all don't-cares (such as $XXXX$), but also entries that are partial don't-cares (such as $XX01$).³

To find a synchronizing sequence, and a concomitant DC assignment, we follow a *generate-and-test approach*. We choose a candidate input column, and then verify if a good DC assignment is possible for that column. If this justification step fails, we repeat the procedure with a different input column until all columns are exhausted (in which case the circuit is not initializable by a single-vector).

By Theorem 4.2, an input column is a single-vector initialization sequence *only if* no more than one state is stable in that column *and* there are no cycles. Otherwise, the machine could start up in *any* one of the stable states, making it impossible to specify a unique end state. We use this property to prune our search space as follows: we pick an input column only if it contains at most one stable state. In the event that no such column can be picked, the circuit is uninitializable by a single-vector. (However, one can then use the multi-vector initialization method of Sec 5.4.) Thus, we have a fast means of rejecting circuits that are uninitializable by single vector.

Unlike the method of Chakradhar *et al.*, our method separates the search in the input space from the search in the space of don't cares. While their method explores both spaces concurrently, our technique first chooses an input vector and then searches in the space of all DC assignments. This is advantageous because search in the input space can be greatly pruned by rejecting those inputs that have more than one stable states in their columns.

5.2 DC assignment

Once a candidate input vector is selected, we perform a justification step. This step involves assigning some of the DC bits a 1 or 0 value so that the machine converges to a unique stable state. Our procedure is an extension of the DC assignment algorithm of Section 4.2.3.

```

algorithm justify(flow_table, current_state_cube, inp_vector)
  while there exists a bit that is determinizable on inp_vector
    if there exists a bit i which is 1-determinizable
      but not 0-determinizable
        set_bits(i, current_state_cube, inp_vector, flow_table, 1)
    else if there exists a bit i which is 0-determinizable
      but not 1-determinizable
        set_bits(i, current_state_cube, inp_vector, flow_table, 0)
    else if there exists a bit i that is both 1-determinizable
      and 0-determinizable
        set_bits(i, current_state_cube, inp_vector, flow_table, 1)
        if justify(flow_table, current_state_cube) succeeds then
          return success
        set_bits(i, current_state_cube, inp_vector, flow_table, 0)
  if all bits in current_state_cube have been determinized,
    output flow_table and return success
  else return failure
end of algorithm

```

Note that the procedure may encounter a bit that is both 1-determinizable and 0-determinizable simultaneously. This happens if the i th bits of the next state entries in the current-state-cube are all X 's. In this case, we need to

³The latter situation may arise, for example, in specifications that use fed-back partially-specified outputs.

explore both choices. The recursion in the above algorithm accomplishes this. Further note that the above algorithm is biased in favor of bits that are not both 1- and 0-determinizable simultaneously. That is, as long as other choices exist, we do not select a bit that will cause the recursive call to be taken. This delays the recursive call and results in an improved performance because of a smaller search tree.

	0	1		0	1		0	1
000	11x	-	000	11x	-	000	11x	-
001	x1x	-	001	x1x	-	001	x1x	-
011	01x	-	011	011	-	011	010	-
010	01x	-	010	011	-	010	010	-
110	xxx	-	110	01x	-	110	01x	-
111	01x	-	111	01x	-	111	01x	-
101	11x	-	101	11x	-	101	11x	-
100	xxx	-	100	x1x	-	100	x1x	-
	(a)		(b)			(c)		

Figure 13: Example showing DC assignment for initializability.

When applied to the flow-table of Fig 13a, the algorithm results in the DC assignment of Fig 13b, assuming that the bits are chosen in order: 2, 1, 3 and that bit 3 is determinized to 1. Fig 13c shows the result if bit 3 is determinized to 0 instead.

5.3 Physical Initializability

As demonstrated in Section 4.2.4, hazards are indeed an important issue in single-vector initialization. Physical initializability can be destroyed by the presence of hazards. Our algorithm enumerates precise covering requirements which insure that hazards do not destroy initializability of the gate-level circuit.

It was shown in Section 4.2.4 that the precise condition for physical initializability under a single-vector input sequence is the following: if state bit y_i is determinized to 1 at a certain time step, then it is required that a product of the cover of y_i span all the next state entries in the current state cube on the current input. As before, the algorithm for DC assignment can be augmented to generate these *required cube* covering requirements at the same time a DC assignment is produced. These requirements can then be passed on to a hazard-free 2-level logic minimizer [6] during logic synthesis.

5.4 Multi-vector initializability

If the algorithm of Sections 5.1–5.3 is unable to initialize a circuit by a single-vector, a multiple-input vector sequence can be used. We show below that hazards must be considered both (i) in *selecting* a multi-vector initialization sequence, and (ii) during the subsequent logic synthesis step: if hazards are ignored, the circuits may not initialize correctly.

The starting point for the multi-vector method is the point from where the single-vector method could not proceed. Thus, some of the state bits have been initialized by the first vector, possibly none. We then choose another input vector such that each state bit which is initialized by the first vector remains initialized by the second vector. This restriction, which appears in [22], is used to insure convergence of the initialization algorithm. The justification step of Section 5.2 is then applied for this vector to

initialize as many state bits as possible; additional input vectors are then selected until initialization is complete.

Hazard considerations limit the choice of the next input vector. In particular, we require that the next state logic for the initialized bits must be glitch-free *during* the input change; otherwise, initialized state bits could become uninitialized due to glitching behavior.

In greater detail, let the circuit be in a current state cube \mathcal{S} when the input changes from I_1 to I_2 . Let \mathcal{I} represent the *input transition cube* $[I_1, I_2]$. To insure that bits once initialized in I_1 remain initialized during the input change, we constrain I_2 to be an input such that, for all columns in \mathcal{I} , the next state of every state in \mathcal{S} lies within \mathcal{S} . The circuit then makes a transition from *total state* $\mathcal{S} \cdot I_1$ to $\mathcal{S} \cdot I_2$. Put another way, we pick an I_2 such that the next state functions for the known state bits are constant, and therefore *free of function hazards*, for each transition within the cube $\mathcal{S} \cdot [I_1, I_2]$.

In addition, we require that the gate-level implementation of the next state logic, for each of the known state bits, be *free of logic hazards*.

These two conditions are, therefore, equivalent to insuring a function-hazard-free and logic-hazard-free transition for a cube spanning the total-state change $\mathcal{S} \cdot \mathcal{I}$. For 2-level AND-OR logic, the logic-hazard-free requirement can be stated as follows [6, 1]:

For each of the state bits in \mathcal{S} that are 1, some product term must cover the total transition cube $\mathcal{S} \cdot \mathcal{I}$. For each of the state bits in \mathcal{S} that are 0, no covering requirement is needed (an AND-OR circuit will be hazard-free).

In contrast, existing methods for multi-vector asynchronous initialization do not place any further restrictions on what types of input changes are allowed, and therefore may allow function hazards [21]. In particular, they invoke the function hazard free requirement only at the end points of the transition, rather than over the entire transition cube $[I_1 I_2]$, so incorrect input sequences might be used. In addition, these methods do not include a requirement to avoid logic hazards during the input vector change.

BCY	bcy							
	000	001	011	010	110	111	101	100
ad								
00	000	---	---	---	110	---	100	000
01	---	---	011	---	111	011	101	---
11	---	101	001	110	110	---	101	---
10	010	101	---	110	110	---	101	---

Figure 14: Function hazard in multiple input change transition

Example. Consider the Karnaugh map of Figure 14 ($qr42$ from [21]), where a and d are primary inputs, and b , c and y are state variables (also used as primary outputs).

At the functional level, 00 followed by 11 would appear to be a valid initialization sequence. Once input 00 is applied, the current state of the circuit is initialized to $bcy = XX0$. Once 11 is applied, in state $bcy = XX0$, next-state bit Y remains functionally initialized to 0, and in fact, the remaining bits become initialized as well: $BCY = 110$ (with appropriate don't-care assignment).

However, based on the above discussion, this 2-vector

sequence is in fact *invalid*. In particular, given the partial state vector $bcy = XX0$, there is a *static-1 function hazard* during the input change from 00 to 11, in the initialized next-state bit Y . This hazard is apparent as a $1 \rightarrow 0 \rightarrow 1$ change as the total state changes from $ad = 00, bcy = 110$, through $ad = 01, bcy = 110$, to $ad = 11, bcy = 110$. A circuit synthesized for initializability using (00,11) as the input sequence may fail to initialize correctly, since the initialized next-state bit, Y , may glitch in the implementation during the input change, changing the current state back to XXX . Our approach will detect the function hazard which occurs during the sequence (00,11), and avoid using this sequence.

An alternative valid initialization sequence is (00,10) (this is the sequence actually used in [21]). In this case, the initialized next-state bit, Y , is 0 wherever it is specified in the total state cube $ad = X0, bcy = XX0$. To make the transition function-hazard-free, remaining don't-care entries for Y in this cube are then set to 0. As a result, the total state cube is now *fully-specified* and *function-hazard-free*. There is no logic hazard-free covering requirement, since Y makes a static-0 transition. \square

6 RESULTS

Table 1 compares results of our synthesis-for-initializability method, *ASM-INIT*, with an asynchronous synthesis method, *UCLOCK* [5], which does not consider initializability. Results are for single-vector initialization of single-hop machines only (*i.e.*, method of Section 4).

Ten circuits were synthesized, using two approaches. The first five circuits were synthesized using arbitrary critical-race-free state assignment followed by single-output hazard-free minimization [6]. The last five circuits were synthesized using an optimal fixed-length state encoding followed by multi-output hazard-free minimization [25]. Column *I/S/O* indicates the number of inputs, state bits and outputs, respectively, for each example.

Column *Base* shows results using the existing asynchronous synthesis tool, *UCLOCK* [5]. Sub-columns *prod* and *lit* indicate number of products and literals in the circuits, respectively. After synthesis, the number of input vectors which can actually initialize each circuit is given in sub-column *#init vec*.

Column *ASM-INIT* shows results of our synthesis-for-initializability method. Here, the number of input vectors *before* synthesis which satisfy the conditions of Theorems 4.1 and 4.2 and, therefore, can be used for initializable synthesis is indicated in sub-column *#init vec*. Circuits were synthesized by randomly selecting one such vector for each circuit. In sub-column *direct DC*, don't-cares were assigned *directly*; in subcolumn *incremental DC*, don't-cares were assigned *incrementally*. Sub-columns *prod* and *lit* indicate number of products and literals in the circuits, respectively, and *DC* indicates the total number of don't-cares assigned to insure initializability.

The table indicates the impact of *ASM-INIT* on initializability of several of the examples. In *pscsi-isend-opt*, the synthesized *Base* circuit was *not* initializable. In contrast, *ASM-INIT* identifies 10 input vectors which could be used for initializable synthesis. For the remaining examples, the base circuits were already initializable. However, in almost every case, many more input vectors could be

Circuit name	I/S/O	Base			#init vec	ASM-INIT					
		prod	lit	#init vec		direct DC			incremental DC		
						DC	prod	lit	DC	prod	lit
chu-ad-opt-e	3/1/3	2	3	6	6	0	2	3	0	2	3
dme-e	3/2/3	5	8	4	5	0	5	8	0	5	8
dme-fast-e	3/3/3	8	14	2	4	0	8	14	0	8	14
sbuf-read-ctl	3/2/3	4	7	5	7	0	4	7	0	4	7
pe-send-ifc	5/3/3	18	49	2	21	21	18	51	10	18	49
sbuf-send-ctl-opt	3/2/3	11	24	3	5	0	11	24	0	11	24
sbuf-read-ctl-opt	3/2/3	7	15	5	7	0	7	15	0	7	15
p SCSI-ircv-opt	4/2/3	10	26	8	14	0	10	26	0	10	26
p SCSI-trcv-opt	4/3/3	13	26	10	14	12	14	29	4	13	26
p SCSI-isend-opt	4/3/3	18	55	0	10	18	21	61	8	-*	-*

*result unavailable due to interfacing issues with the *hfmin* tool [25]

Table 1: Results

used as initialization vectors by *ASM-INIT*. For *pe-send-ifc*, the results were dramatic: the base circuit had only 2 initialization vectors, while our method identified 21 initialization vectors in the flow table which could be used for initializable synthesis.

The table also shows that circuits synthesized using *ASM-INIT* had little logic overhead. Except for *p SCSI-trcv-opt* and *p SCSI-isend-opt* using direct DC assignment, the number of product terms produced by *ASM-INIT* and the *Base* method is identical. Moreover, for the more optimal *incremental DC* assignment, all circuits except *p SCSI-isend-opt* could be synthesized using no more products and no more literals than the *Base* circuits.

Finally, the table indicates the potential benefit of incremental versus direct DC assignment. In *pe-send-ifc*, for example, our direct assignment method assigned 21 don't-cares, while our incremental method assigned only 10 don't-cares. We expect that for larger examples, the benefit of the incremental approach may increase, resulting in fewer products and literals than the direct approach.

7 CONCLUSIONS AND FUTURE WORK

We have introduced two new methods for the synthesis-for-initializability of asynchronous state machines. Unlike existing approaches, the first method includes a state assignment step. This feature is critical, since initializability may be precluded by poor choice of state assignment. Use of state assignment may also allow the synthesis of initializable machines without the need to use concurrency-reducing transformations on the specification. The second method assumes that state assignment is given, but handles a larger class of machines, and considers both single-vector and multi-vector initialization. In both methods, we included conditions on hazard-free logic, to insure that the circuits are initializable.

For future work, we intend to extend our approach of Section 4 to handle state assignment for *multi-vector initialization*. In addition, we will apply our method to larger examples.

Acknowledgment: We thank Robert Fuhrer of Columbia University for help in synthesizing the circuits.

REFERENCES

- [1] S.H. Unger. *Asynchronous Sequential Switching Circuits*. New York: Wiley-Interscience, 1969.
- [2] Steven M. Nowick, *Automatic synthesis of burst-mode asynchronous controllers*. Technical report: CSL-TR-95-686, Stanford University. Ph.D. Thesis, Mar. 1993.
- [3] S.M. Nowick and D.L. Dill. "Automatic synthesis of locally-clocked asynchronous state machines," in *ICCAD-1991*, pp. 318-321.
- [4] K.Y. Yun, D.L. Dill, and S.M. Nowick, "Synthesis of 3D asynchronous state machines," in *ICCD-1992*, pp. 346-350.
- [5] S.M. Nowick and B. Coates, "UCLOCK: Automated design of high-performance unlocked state machines," in *ICCD-1994*, pp. 434-441.
- [6] S.M. Nowick and D.L. Dill, "Exact two-level minimization of hazard-free logic with multiple-input changes," *IEEE Trans. CAD*, 14(8), pp. 986-997, Aug. 1995.
- [7] D.S. Kung, "Hazard-non-increasing gate-level optimization algorithms," in *ICCAD-1992*, pp. 631-634.
- [8] B. Lin and S. Devadas, "Synthesis of hazard-free multilevel logic under multiple-input changes from binary decision diagrams," *IEEE Trans. CAD*, 14(8), pp. 974-985, Aug. 1995.
- [9] G. Birtwistle and A. Davis (eds.). *Asynchronous Digital Circuit Design*, Springer-Verlag (Workshops in Computing series), London 1995.
- [10] S.M. Nowick, N.K. Jha and F.-C. Cheng, "Synthesis of Asynchronous Circuits for Stuck-at and Robust Path Delay Fault Testability," in *8th Int. Conf. on VLSI Design*, Jan. 1995.
- [11] P. A. Bearel and T. H.-Y. Meng, "Semi-modularity and self-diagnostic asynchronous control circuits," in *Adv. Res. in VLSI*, Mar. 1991, pp. 103-117.
- [12] A. J. Martin and P. J. Hazewindus, "Testing delay-insensitive circuits," in *Adv. Res. in VLSI*, Mar. 1991, pp. 118-132.
- [13] K. Keutzer, L. Lavagno, and A. L. Sangiovanni-Vincentelli, "Synthesis for testability techniques for asynchronous circuits," *IEEE Trans. CAD*, 14(12), pp. 1569-1577, Dec. 1995.
- [14] M. Roncken, "Partial scan test for asynchronous circuits illustrated on a DCC error corrector", in *Int. Symp. Adv. Res. in Async. Circuits & Systems (Async94)*, pp. 247-256, Nov. 1994.
- [15] M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.
- [16] J.A. Wehbeh and D.G. Saab, "On the initialization of sequential circuits," in *Int. Test Conf.* pp. 233-239, Oct. 1994.
- [17] C. Pixley and G. Beihl, "Calculating resettability and reset sequences," in *ICCAD-91*, pp. 376-379.
- [18] K. Cheng and V. Agrawal, "Initializability consideration in sequential machine synthesis," *IEEE Trans. Comput.*, vol 41, pp. 374-379, Mar. 1992.
- [19] M. Singh and S.M. Nowick, "Synthesis for Logical Initializability of Synchronous Finite State Machines," submitted to a conference (presented at *1996 Int. Test. Synth. Workshop*).
- [20] S. Banerjee, R.K. Roy, S.T. Chakradhar, and D.K. Pradhan, "Initialization Issues in the Synthesis of Asynchronous Circuits," in *ICCD-1994*, pp. 447-452.
- [21] S. Banerjee. *New Techniques for Synthesis and Testing of Asynchronous Circuits*. PhD Thesis, UMass Amherst, May 1995.
- [22] S.T. Chakradhar, S. Banerjee, R.K. Roy, and D.K. Pradhan, "Synthesis of initializable asynchronous circuits," in *IEEE Trans. VLSI Systems*, 4(2), pp. 254-262, June 1996.
- [23] L. Lavagno, C.W. Moon, R.K. Brayton, and A.L. Sangiovanni-Vincentelli, "An Efficient Heuristic Procedure for Solving the State Assignment Problem for Event-Based Specifications," in *IEEE Trans. CAD*, 14(1), pp. 45-60, Jan. 95.
- [24] L. Lavagno, M. Kishinevsky, and A. Liroy, "Testing redundant asynchronous circuits by variable phase splitting," in *European Design Automation Conf.*, Sept. 1994.
- [25] R.M. Fuhrer, B. Lin, and S.M. Nowick, "Symbolic Hazard-Free Minimization and Encoding of Asynchronous Finite State Machines," in *ICCAD-95*, pp. 604-611.