IRREDUNDANT SEQUENTIAL MACHINES VIA OPTIMAL LOGIC SYNTHESIS

by

Srinivas Devadas, Hi-Keung Tony Ma, A. Richard Newton
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M88/52

5 August 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

IRREDUNDANT SEQUENTIAL MACHINES VIA OPTIMAL LOGIC SYNTHESIS

by

Srinivas Devadas, Hi-Keung Tony Ma, A. Richard Newton
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M88/52

5 August 1988

IRREDUNDANT SEQUENTIAL MACHINES VIA OPTIMAL LOGIC SYNTHESIS

by

Srinivas Devadas, Hi-Keung Tony Ma, A. Richard Newton
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M88/52

5 August 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Irredundant Sequential Machines Via Optimal Logic Synthesis

Srinivas Devadas*, Hi-Keung Tony Ma,
A. Richard Newton and Alberto Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

Abstract

It is well known that optimal logic synthesis can ensure fully testable combinational logic designs. In this paper, we show that optimal sequential logic synthesis can produce irredundant, fully testable finite state machines. Test generation algorithms can be used to remove all the redundancies in sequential machines resulting in a fully testable design. However, this method may require exorbitant amounts of CPU time. The optimal synthesis procedure presented in this paper represents a more efficient approach to achieve 100% testability.

Synthesizing a sequential circuit from a State Transition Graph description involves the steps of state minimization, state assignment and logic optimization. Previous approaches to producing fully and easily testable sequential circuits have involved the use of extra logic and constraints on state assignment and logic optimization. In this paper, we show that *100% testability can be ensured without the addition of extra logic and without constraints* on the state assignment and logic optimization. Unlike previous synthesis approaches to ensuring fully testable machines, there is *no area/performance penalty* associated with this approach. This technique can be used in conjunction with previous approaches to ensure that the synthesized machine is easily testable.

Given a State Transition Graph specification, a logic-level automaton that is fully testable for *all single stuck-at* faults in the combinational logic *without access to the memory elements* is synthesized. This procedure represents an alternative to a Scan Design methodology without the usual area and performance penalty associated with the latter method.

*Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge

1 Introduction

Test generation for sequential circuits has long been recognized as a difficult task [3]. A popular approach to solving this problem is to make all the memory elements controllable and observable, i.e. Complete Scan Design [7] [1]. Scan Design approaches transform the sequential testing problem into one of combinational test generation which is considerably less difficult. They also remove all sequential redundancies in a circuit, since direct access is provided to the memory elements. However, there are situations where the cost in terms of area and performance of Complete Scan Design is unaffordable. Also, the testing time associated with Scan Design is high because values have to be *sequentially* scanned into and out of the memory elements one clock cycle at a time.

It is well known that optimal logic synthesis can ensure fully testable combinational logic designs. In this paper, we show that optimal sequential logic synthesis can produce fully testable non-scan finite state machines. Test generation algorithms can be used to remove all the redundancies in sequential machines resulting in fully testable designs. However, in general, this method requires exorbitant amounts of CPU time. The optimal synthesis procedure presented in this paper represents a more efficient approach to achieve 100% testability.

Synthesizing a sequential circuit from a State Transition Graph description involves the steps of state minimization, state assignment and logic optimization. Previous approaches (e.g. [6]) to producing fully and easily testable sequential circuits have entailed the use of extra logic and constraints on state assignment and logic optimization. In this paper, we show that 100% testability can be ensured *without the addition of extra logic and without constraints* on the state assignment and logic optimization. This technique can be used in conjunction with previous approaches to ensure that the synthesized machine is easily testable.

The finite automaton is represented by a State Transition Graph, truth table or by an interconnection of gates and flip-flops. The synthesized/re-synthesized logic-level implementation is guaranteed to be fully testable for *all single stuck-at* faults in the combinational logic *without access to the memory elements.* This procedure represents an alternative to a Scan Design methodology without the usual area and performance penalty associated with the latter method.

Basic definitions and terminologies used are given in Section 2. Various types of redundant faults in sequential circuits are described in Section 3. In Section 4, we outline an optimal synthesis procedure of state minimization, state assignment and logic optimization that produces a highly testable Moore or Mealy finite state machine beginning from a State Transition Graph description.

Any existing sequentially redundant faults in this machine are *implicitly* removed using extended don't care sets in repeated combinational logic minimization. These don't care sets are derived using techniques that check for state equivalence. We give theorems which prove the correctness of these procedures. In Section 5, we discuss the effects of redundancy removal on the state encoding of the machine. In Section 6, we describe how this approach can be used in conjunction with a previously proposed synthesis technique to ensure easily testable machines. In this case, test sequences which detect all single stuck-at faults in the combinational logic can be derived using combinational test generation techniques alone. Preliminary results, which indicate that these procedures are viable for large circuits, are given in Section 7.

2 Preliminaries

A **variable** is a symbol representing a single coordinate of the Boolean space (e.g. a). A **literal** is a variable or its negation (e.g. a or $\overline{a}$). A **cube** is a set C of literals such that $x \in C$ implies $\overline{x} \notin C$ (e.g., $\{a,b,\overline{c}\}$ is a cube, and $\{a,\overline{a}\}$ is not a cube). A cube represents the conjunction of its literals. The trivial cubes, written 0 and 1, represent the Boolean functions 0 and 1 respectively. An **expression** is a set f of cubes. For example, $\{\{a\},\{b,\overline{c}\}\}$ is an expression consisting of the two cubes $\{a\}$ and $\{b,\overline{c}\}$. An expression represents the disjunction of its cubes.

A cube may also be written as a bit vector on a set of variables with each bit position representing a distinct variable. The values taken by each bit can be 1, 0 or 2 (don't care), signifying the true form, negated form and non-existence respectively of the variable corresponding to that position. A **minterm** is a cube with only 0 and 1 entries.

A finite state machine is represented by its **State Transition Graph** (STG), $G(V,E,W(E))$ where V is the set of vertices corresponding to the set of states S, where $||S|| = N_s$ is the cardinality of the set of states of the FSM, an edge joins v_i to v_j if there is a primary input that causes the FSM to evolve from state v_i to state v_j, and $W(E)$ is a set of labels attached to each edge, each label carrying the information of the value of the input that caused that transition and the values of the primary outputs corresponding to that transition. In general, the $W(E)$ labels are Boolean expressions. The number of inputs and outputs are denoted N_i and N_o respectively. The input combination and present state corresponding to an edge or set of edges is $(i,\ s)$, where i and s are cubes. The fanin of a state, q is a set of edges and is denoted $fanin(q)$. The fanout of a state q is denoted $fanout(q)$. The output and the fanout state of an edge $(i,\ s) \in E$ are $o((i,\ s))$ and

$n((i,\ s)) \in V$ respectively.

Given N_i inputs to a machine, 2^{N_i} edges with minterm input labels fan out from each state. A STG where the next state and output labels for every possible transition from every state are defined corresponds to a **completely specified machine**. An **incompletely specified machine** is one where at least one transition edge from some state is not specified.

A starting or initial state is assumed to exist for a machine, also called the **reset state**. Given a logic-level finite state machine with N_b latches, 2^{N_b} possible states exist in the machine. A state which can be reached from the reset state via some input vector sequence is called a **valid state** in the STG. The input vector sequence is called the **justification sequence** for that state. A state for which no justification sequence exists is called an **invalid state**. Given a fault F, the State Transition Graph of the machine with the fault is denoted G^F.

A State Transition Graph G_1 is said to be **isomorphic** to another State Transition Graph G_2 if and only if they are identical except for a renaming of states.

The fault model assumed is **single stuck-at**. A finite state machine is assumed to be implemented by combinational logic and feedback registers. Tests are generated for stuck-at faults in the combinational logic part.

A primitive gate in a network is **prime** if none of its inputs can be removed without causing the resulting circuit to be functionally different. A gate is **irredundant** if its removal causes the resulting circuit to be functionally different. A gate-level circuit is said to be **prime** if all the gates are prime and **irredundant** if all the gates are irredundant. It can be shown that a gate-level circuit is prime and irredundant if and only if it is 100% testable for all single stuck-at faults.

We differentiate between two kinds of redundancies in a sequential circuit. If the effect of the fault cannot be observed at the primary outputs or the next state lines, beginning from any state, with any input vector, the fault is deemed **combinationally redundant**. A **sequentially redundant** fault is a fault that cannot be detected by any input sequence and is not combinationally redundant.

To detect a fault in a sequential machine, the machine has to be placed in a state which can then excite and propagate the effect of the fault to the primary outputs. The first step of reaching the state in question is called **state justification**. The second step is called **fault excitation-and-propagation**.

An edge in a State Transition Graph of a machine is said to be **corrupted** by a fault if either the fanout state or output label of this edge is changed because of the existence of the fault. A

path in a State Transition Graph is said to be corrupted if at least one edge in the path has been corrupted.

3 Origin of Redundant Faults in Sequential Circuits

There are two classes of redundant faults in a sequential circuit, namely, combinationally and sequentially redundant faults. Combinationally redundant faults (CRFs) are due to the presence of lines/wires in the logic circuit that do not contribute to the primary output or the next state functions. Replacement of these lines by constants will not change the functionality of the combinational logic in the sequential circuit. CRFs cannot be detected even if all the memory elements of the sequential circuit are made scannable. Sequentially redundant faults (SRFs), on the other hand, are related to the temporal characteristics of the sequential circuit. Although SRFs alter the next state functions and hence the State Transition Graph (STG), or the State Transition Table (STT), representing the sequential circuit, they cannot be detected without making some of the latches scannable.

A stuck-at fault is sequentially redundant if

1. It causes only interchange and/or creation of equivalent states in the STG of the finite state machine (*type* 1 SRF) or

2. It is not combinationally redundant but does not corrupt any fanout edge of a valid state that is reachable from the reset state (*type* 2 SRF) or

3. It transforms the original machine isomorphically, i.e. the faulty machine is equivalent to the good machine but with a different encoding (*type* 3 SRF). (There exists an isomorphism between the original and the faulty machine.)

While the CRF, *type* 1 and *type* 2 SRF redundant faults are common occurrences during design process due to improper utilization of don't cares, the *type* 3 SRF is relatively unlikely to exist. We will use an example to illustrate the existence of sequentially redundant faults.

The State Transition Table of a finite state machine is shown in Figure 1. The machine has 5 states and the states 010 and 110 are equivalent. The logic implementation of the combinational part of the machine is shown in Figure 2. The fault $w1$ stuck-at-0 (s-a-0) changes the original STT to the one shown in Figure 3. The third field (next state) of the second row is changed from 110 to 010 by the fault. Since 010 and 110 are equivalent states in the original STT, the fault $w1$ s-a-0

0 100 010 1
1 100 110 0
0 010 110 1
1 010 000 0
0 110 010 1
1 110 000 0
0 000 001 0
1 000 110 1
0 001 000 0
1 001 100 1

Figure 1: Original Finite State Machine

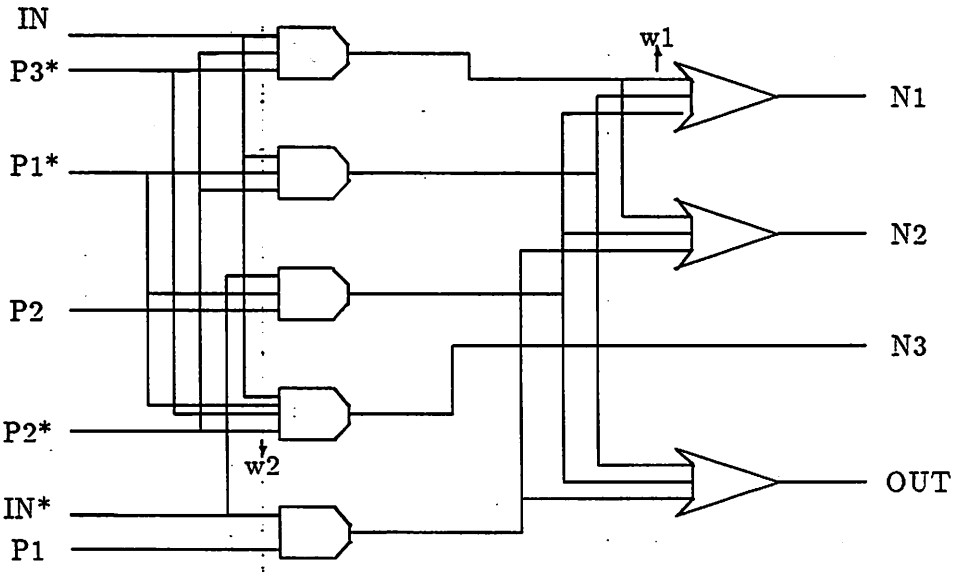

Figure 2: Combinational Logic of FSM

0 100 010 1

1 100 010 0

0 010 110 1

1 010 000 0

0 110 010 1

1 110 000 0

0 000 001 0

1 000 110 1

0 001 000 0

1 001 100 1

Figure 3: Faulty FSM with w1 s-a-0

0 100 010 1

1 100 110 0

0 010 111 1

1 010 000 0

0 110 010 1

1 110 000 0

0 000 001 0

1 000 110 1

0 001 000 0

1 001 100 1

0 111 010 1

1 111 000 0

Figure 4: Faulty FSM with w2 s-a-1

0 100 010 1
1 100 110 0
0 010 110 1
1 010 001 0
0 110 010 1
1 110 001 0
0 001 000 0
1 001 110 1
0 000 001 0
1 000 100 1

Figure 5: Faulty FSM with a *type* 3 SRF

only causes an interchange of two equivalent states of the machine and is therefore sequentially redundant. The fault $w2$ s-a-1 changes the machine to the one shown in Figure 4. The fault creates an extra state 111 and changes the third field of the third row of the original STT from 110 to 111. However, it can be proved that 111 is equivalent to 010 and 110 and therefore the fault $w2$ is also sequentially redundant.

If the detection of a fault in the combinational logic requires the machine to be brought to an invalid state (e.g. 101), then the fault is a SRF of *type* 2. A *type* 3 SRF may change the original machine to the one shown in Figure 5. Note that the faulty machine represents an equivalent machine with a different encoding. The encodings for the states 000 and 001 in the original machine have been swapped. An isomorphism exists between the original and the faulty machine.

Theorem 3.1 *: A redundant fault in a finite state machine is either a CRF or type 1 or type 2 or type 3 SRF.*

Proof (by contradiction): Assume a fault, F, is a redundant fault but not a CRF or *type* 1 SRF or *type* 2 SRF or *type* 3 SRF. Since F is not a CRF or a *type* 2 SRF, there must be an input sequence, beginning from the reset state, that will bring the machine to a state that can excite the fault and propagate its effect at least to some of the next state lines. Since F is not a *type* 1 or *type* 3 SRF, the fault effect on the next state lines will not cause an interchange or creation of

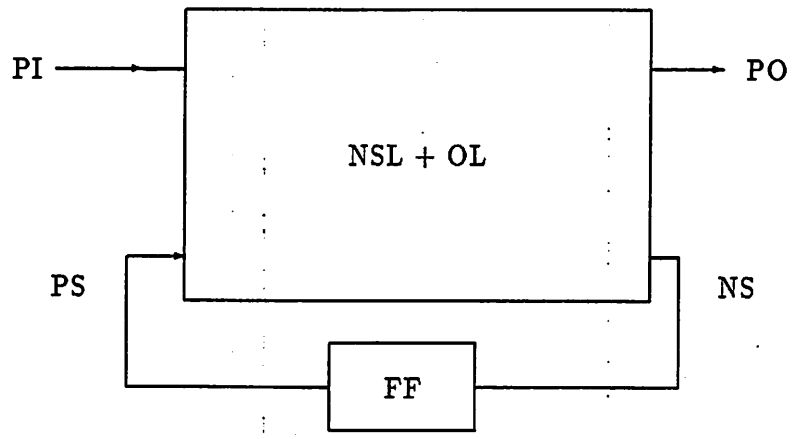

Figure 6: General Sequential Machine Model

equivalent states or an isomorphic mapping of states. This means the good state and the faulty state can be differentiated by a propagation sequence, i.e. the fault effect is propagated to the primary outputs, which means that the fault is testable. □

To synthesize a fully testable finite state machine, we must ensure that none of those redundant faults described above exist in the synthesized machine. Steps in our synthesis procedure are designed to achieve this goal.

4 Irredundant Fully Testable Sequential Machines

A general model for a Mealy finite state machine is shown in Figure 6. It is realized by a combinational logic block, which implements the output and next state logic functions, and feedback registers. The Moore machine can be viewed as a special case of a Mealy machine, where the outputs depend only on the present state of the machine.

We first describe the optimal synthesis procedure in Section 4.1. In Section 4.2, we prove that the resulting machine has no CRFs, *type* 2 or *type* 3 SRFs. Experimental results indicate that the machine has very few redundancies. In Section 4.3, we present a modified synthesis procedure using extended don't care sets in repeated combinational logic minimization which ensures that *type* 1 SRFs do not exist in the synthesized machine. The machine can be thus made fully testable. In Section 4.4, we briefly discuss how finite automata represented at the truth table or at the logic-level can be made fully testable.

4.1 The Synthesis Procedure

The procedure consists of the steps of state minimization, state assignment and combinational logic optimization. These steps are described in the sequel.

1. **State Minimization:** Given an original, possibly incompletely specified, State Transition Graph specification G^O we obtain a state minimum representation, G^M, using algorithms similar to those proposed in [13]. G^M has N_s valid states and satisfies the property that no two states are equivalent. State minimization for completely specified State Transition Graphs can be accomplished in $O(Nlog(N))$ time where N is the number of states in the machine, but is NP-complete for incompletely specified machines.

2. **State Assignment:** We encode the states in G^M, namely Q. The number of encoding bits N_b can be arbitrarily large ($N_b \geq log_2(||Q||)$). State assignment algorithms like those in [8] and [5] can be used, which find a state assignment that heuristically minimizes the area of the combinational network after optimization.

3. **Combinational Logic Optimization:** Given the encoded machine, which is now a combinational logic specification, we synthesize a prime and irredundant combinational logic network which implements both the next state logic and output logic functions. The transitions from the unused state codes, are used as don't cares during the minimization. The number of inputs to the network will be $N_i + N_b$ and the number of outputs will be $N_o + N_b$. Prime and irredundant two-level networks can be produced using two-level logic minimizers like ESPRESSO [2]. Prime and irredundant multi-level networks can be synthesized using techniques like those in [9].

We will have N_b latches in the synthesized sequential machine (denoted S^M) and 2^{N_b} valid and invalid states in the completely specified State Transition Graph (denoted G).

4.2 Correctness of Procedure

We can prove that the sequential machine synthesized by the procedure of the previous section is irredundant for all CRFs, *type* 2 and *type* 3 SRFs.

The following theorem follows from the definition of state minimality. It is given in [11].

Theorem 4.1 : *Given a state minimized (reduced) machine M with N_s states, no machine with fewer states can realize the same terminal behavior. Also, any machine with the same number of*

states that realizes the same behavior has to be M or isomorphic to M.

We can show that stuck-at faults cannot produce a faulty State Transition Graph that is isomorphic to the true State Transition Graph if the combinational logic implementing the next state and output logic functions is prime and irredundant. Isomorphic faulty and true State Transition Graphs imply that the fault has no other effect than changing the state encoding of the machine.

Theorem 4.2 : *If the combinational circuit implementing the next state and output logic functions is prime and irredundant, then any fault F in the circuit cannot produce a G^F that is isomorphic to G.*

Proof: For G^F to be isomorphic to G, no two edges $e_1,\ e_2\ \in G$ should be merged in G^F. Consider a prime and irredundant circuit, C_1, implementing G. The circuit is levelized from the primary outputs to the primary inputs. Gates generating primary outputs are assigned level 0 and a gate that drives gates with levels $l_1,\ l_2\ ..,\ l_n$ has a level equal to $MIN(l_i) + 1$. The gates at level j are $g_{j1},\ g_{j2},\ ..g_{jN_j}$. The outputs of these gates constitute a set of N_j variables $v(j)$. Without loss of generality, consider a fault F at the output of g_{j1}. We have vectors $(v_1,\ v_2)$, corresponding to values of the $v(j)$, that differ in bit 1 such that v_1 detects F and v_2 does not. Obviously, v_2 produces the same output as the faulty output of v_1. Since the circuit is irredundant, primary input combinations $(i_1,\ i_2)$ have to exist that produce a $(v_1,\ v_2)$ satisfying this property.

Next, consider a fault, F, at the input of g_{j1}. This fault has to be propagated to the primary outputs via the output of g_{j1}. Since the circuit is prime, we have an input vector, i_1, that detects the fault and produces vector v_1 corresponding to the variables $v(j)$. Also, there has to exist a v_2, produced by i_2, which differs from v_1 in bit 1 for some v_1 that detects F. Obviously, i_2 produces the same output as the faulty output of i_1.

Thus, for any fault in the circuit, we have a input vector pair $(i_1,\ i_2)$ such that i_1 detects the fault, i_2 does not and the output of i_2 is equal to the faulty output of i_1 (and different from the true output of i_1). $(i_1,\ i_2)$ correspond to two edges $(e_1,\ e_2)\ \in G$. These edges have merged in G^F. This means G^F and G are not isomorphic. □

Theorem 4.3 : *If G^M contains 2^{N_b} states where N_b is the number of latches in S^M, S^M is fully testable.*

Proof: No fault in the machine can result in an increase in the number of states, since the true machine has the maximum possible number of states, namely 2^{N_b}. Since G^M is reduced, we

know that no machine with fewer than 2^{N_b} states can realize the behavior of G^M. All faults are combinationally irredundant, since the combinational logic is prime and irredundant. For a combinationally irredundant fault F to be sequentially redundant, the faulty machine G^F has to be isomorphic to the true machine G. By Theorem 4.2, this is not possible. Therefore, S^M is fully testable. □

The above theorem is quite a strong result. Given a State Transition Graph G^M, if extra states can be added to G^M such that the resulting graph $G^{M'}$ is reduced and has 2^n states, then the synthesized machine $S^{M'}$ is guaranteed to be fully testable. Of course, adding the extra states and edges to G^M constitutes an area overhead. If G^M has less than 2^{N_b} states, the unused state codes can be used as don't care states to minimize the combinational specification.

Lemma 4.4 : *An invalid state in the State Transition Graph is never required to detect a fault in* S^M.

Proof: All unused state codes are used as don't cares during logic minimization. Invalid states can only correspond to some unused state code. Since the combinational network is prime and irredundant, there always exists a valid state that detects any fault that the invalid state detects. □

We now use the preceding results to prove the partial irredundancy theorem for machines whose G^M has $N_s < 2^{N_b}$ states.

Theorem 4.5 : *The sequential machine* S^M *produced by the synthesis procedure may contain only type* 1 *SRFs.*

Proof: Follows directly from Theorem 3.1, Theorem 4.2 and Lemma 4.4. □

We can also show that a subset of the faults in the machine are always testable.

Theorem 4.6 : *All primary input (PI) and primary output (PO) line stuck-at faults are testable in* S^M. *Further, if* S^M *has been obtained using a minimum bit encoding on* G^M, *all present state (PS) and next state (NS) line stuck at faults are also testable.*

Proof: A primary input i exists in G^M, if and only if there exists a pair of edges e_1 and e_2 that differ in bit i alone which go to different next states in G^M or assert different outputs. A primary input stuck-at fault, F, cannot increase the number of states, i.e. $||G^F|| \leq ||G||$. G^F is not isomorphic to G since e_1 and e_2 have merged in G^F. Therefore, F is testable.

A primary output o exists in G^M, if and only if there exists a pair of edges e_1 and e_2 which assert both values of the output, 0/1. When the machine makes the transition corresponding to the edge which asserts the value of the output different from the stuck value, the fault will be detected.

A stuck-at fault on a present state line results in a faulty machine with a maximum of 2^{n-1} states, where n is the number of latches in the machine. Since S^M has been obtained via a minimum bit encoding, the number of states in the reduced machine G^M is $N_s > 2^{n-1}$. This means that G^F cannot realize the behavior of S^M (G^M).

The argument that the next state line faults are testable is similar to the argument that the present state line faults are testable. □

4.3 Eliminating Redundancies Via Extended Don't Care Sets

In this section, we show how the testability of the synthesized machine S^M can be increased by removing possible *type* 1 SRFs through succeeding logic minimization steps, *without explicitly identifying these redundancies.* Redundancies are identified and removed *implicitly* via the use of *extended don't care sets.*

A simple *type* 1 SRF was illustrated in Section 3. We have a situation where an invalid state q has identical fanout to some valid state v_1. An edge from v_2 to v_1 is corrupted to go to q. F only corrupts one edge in the State Transition Graph and propagates only one time-frame. In the general case, a *type* 1 SRF can propagate multiple time-frames, when the invalid state q is equivalent to the true valid state v_1, but does not have identical fanout.

These redundancies are likely to occur, especially if a large number of unused state codes exist. One can ensure that these redundancies do not occur by specifying an extended don't care set. The following steps are taken in an extended synthesis procedure:

1. State assignment and logic optimization are performed as before. Logic optimization uses the invalid states as don't cares.

2. Given the prime and irredundant logic network, the State Transition Graph, G, corresponding to the network is extracted. All invalid states $iv \in G$ that are equivalent to valid states $v \in G$ are found. It should be noted that G is a completely specified combinational logic function, which also represents an encoded State Transition Graph.

3. Given a valid state v_1 and invalid states iv_1, iv_2, .. iv_K that are equivalent to v_1, then the fanin of v_1 is re-specified as $n(fanin(v_1)) = DC(v_1, iv_1, iv_2, .. iv_K)$. $DC()$ implies

that any of the enclosed state entries can be used. In practice, if v_1 and some or all of the iv_k, $1 \leq k \leq K$ can be merged into a single cube, c, then every occurrence of v_1 in the next state field of G is replaced by c. G with this extended don't care set is made prime and irredundant via logic minimization to produce G'.

4. G' may have some invalid states, which could be different from the invalid states in G. These invalid state codes are used as don't cares and G' is made prime and irredundant under this new don't care set to produce G''.

5. If $G' = G''$, exit. Else $G \longleftarrow G''$, go to Step 2.

Theorem 4.7 : *The procedure above converges, and the resulting machine after convergence will not have any simple type 1 SRFs, type 2 SRFs or type 3 SRFs.*

Proof: The procedure converges when succeeding logic minimizations have produced the same result. Each logic minimization operates on the result of the previous logic minimization with the additional don't care set that is provided. We are guaranteed that the overall cost function (e.g. the number of lines in the network) has a finite decrease if the logic function is altered. Therefore, the sequence of logic minimizations must eventually converge, and on the last call, return an unchanged network, η. No *type* 3 SRFs will exist in the prime and irredundant network η by Theorem 4.2. Since the invalid states have been used as don't cares to produce η and the network is unchanged since then (even though additional minimizations may have been performed), no *type* 2 SRFs can exist.

Finally, using the don't care sets corresponding to the equivalent states, ensures that for each fault F there will exist at least one corrupted edge that goes to a state, q^F, that is *not* equivalent to the true next state, q, in the true machine G, regardless of whether the q^F is invalid or valid. η is unchanged since the use of the invalid states as don't cares, so an edge fanning out of a valid state has to exist with this property. $q^F \in G^F$ has to become equivalent to $q \in G$ for F to be redundant, but that would mean that F is not a simple *type* 1 SRF. Therefore, F is testable or not a simple *type* 1 SRF. □

More complicated *type* 1 SRFs may exist, though experimental evidence indicates that this is extremely rare. These redundancies correspond to the case, where $q^F \in G$ is not equivalent to $q \in G$ but $q^F \in G^F$ becomes equivalent to $q \in G$, making F redundant. A larger set of don't cares can ensure that these *type* 1 SRFs do not occur in the machine. The synthesis procedure

described above is unchanged except for introducing an additional don't care set in **Step 3** where G' is produced, as described below.

Step 3b: Given an invalid state q_2 that is not equivalent to a valid state q_1, the set of input combinations $i_{ne}(q_1, q_2)$ are found which make this pair not equivalent. If q_2 were equivalent to q_1 then $i_{ne} = \phi$. The don't care specification is $n(fanin(q_1)) = DC(q_1, q_2)$, with a constraint on a subset of fanout edges of q_2 if q_2 is picked rather than q_1. The constraint is that

$$o(i_{ne}(q_1, q_2), q_2) = o(i_{ne}(q_1, q_2), q_1) \bigwedge n(i_{ne}(q_1, q_2), q_2) = n(i_{ne}(q_1, q_2), q_1)$$

This set of don't cares and associated constraints are found for each valid and invalid state pair that are not equivalent. Optimal use of the don't cares while satisfying these constraints ensures full testability.

Theorem 4.8 : *Using the additional don't care set in the synthesis procedure will result in a fully testable machine.*

Proof: We know by Theorem 4.7, that no simple *type* 1 SRFs, *type* 2 SRFs or *type* 3 SRFs will exist in the machine. Using the additional don't cares will ensure that there will always be an edge from a valid state that is corrupted to q^F instead of q such that $q^F \in G \neq q \in G$ and $q^F \in G^F \neq q \in G$. Therefore, G^F and G can be differentiated by distinguishing q^F and q and F is testable. □

The enhanced procedure will remove all *type* 1 SRFs in the machine which has been synthesized as described in the previous section. In practice, only the simple don't cares of **Step 3** suffice to ensure full testability, allowing a locally optimal solution with no redundancies to be reached; the more complicated don't cares of **Step 3b** are *not* required. That is fortunate, since current logic optimization programs are quite restricted in the specification and optimal usage of don't cares.

The procedure is quite CPU-intensive since repeated combinational logic minimizations have to be performed. Experimental results (Section 7) indicate that the machine prior to using the extended don't care sets is highly testable, and in some cases, fully testable. Removing the few redundancies can be accomplished using reasonable amounts of CPU time. The fact that a network has to repeatedly made prime and irredundant in order to ensure full testability for a sequential circuit, indicates that synthesizing irredundant sequential circuits is more difficult than synthesizing irredundant combinational circuits.

4.4 Synthesis from Logic-Level Descriptions

In this section, we describe how complete or partial re-synthesis of logic-level circuits can be performed so as to ensure irredundant sequential machines. Given a combinational specification of a circuit in the form of a truth table, i.e. a previously encoded finite state machine, the following steps are performed in re-synthesis. The combinational specification has $N_i + N_b$ inputs and $N_o + N_b$ outputs, where N_b is the number of encoding bits used (latches) in the state assignment process.

1. The combinational specification is made disjoint in the present state field (the last N_b inputs). A cube entry in the field is identical to another cube entry or does not intersect it. A two-level cover can be made disjoint using the SHARP operation in [2].

2. The specification is now treated as a State Transition Table, with each distinct entry in the present state and next state field representing a distinct state. If some states cannot be reached from the reset state (invalid states), they are deleted from the description. The State Table is now state minimized. Some states (represented by cubes or minterms) may be removed because of being equivalent to other states.

3. The encoded State Transition Table represents a combinational logic specification that can be made prime and irredundant. A fully testable machine can be synthesized via the procedures of Section 4.2 and 4.3.

The re-synthesis procedure can be extended to begin from a logic-level description. In this case, the State Transition Graph of the machine is extracted using the efficient cube-enumeration techniques presented in [4]. Given this (encoded) State Transition Graph, Steps 1-3 described above are carried out as before.

5 Effect of Redundancy Removal via Logic Minimization on State Encoding

If a combinationally redundant line is removed from a logic network, network functionality remains unchanged. Similarly, when a sequentially redundant but combinationally irredundant line is removed from a sequential machine, the terminal behavior of the machine remains unchanged. However, the State Transition Graph of the machine, and the state encoding are affected by redundancy removal via repeated logic minimization.

Two things may happen during redundancy removal:

1. A state may be added to the State Transition Graph, which is equivalent to some other valid state. An edge is redirected from some valid state to this originally invalid state.

2. A valid state may be replaced by an originally invalid state. In effect, the encoding of a symbolic state is changed.

Removing a redundant line decreases network complexity. Therefore, *both these effects are due to a sub-optimal state assignment.*

The occurrence of the first effect is due to the fact that state assignment is performed on a state minimized Graph. It is well known [10] that state splitting may be required for an optimal state assignment. Unfortunately, the state assignment problem is difficult enough as it is without adding the extra degree of freedom of being able to split states. The faulty, but equivalent, State Graph corresponds to a "better" state assignment with (at least) one state split into two (or more) components.

The occurrence of the second effect is due to a state assignment that is not locally optimal for the reduced State Graph, even without the addition of extra states. The replacement of a state code by an unused state code results in a "better" machine. Since all state assignment techniques (e.g. [5] [8]) are heuristic and attempt to predict a complicated logic optimization process that follows, it is quite possible that redundancies producing this effect of replacing a state code by another will exist. We have found in our experience, however, that the first effect is much more frequent.

When a reduced machine has 2^n states, and the number of encoding bits used is n, states cannot be split and no unused state codes exist. Thus, any state assignment is locally optimal and the machine will be fully testable by Theorem 4.3. If an optimal state assignment can be found exploiting the freedom of state splitting, then the resulting logic implementation will be fully testable. Repeated logic minimization, as described in Section 4.3, has the effect of changing a sub-optimal state encoding to a locally optimal encoding that corresponds to a fully testable machine.

6 Easily Testable Sequential Machines

The proposed optimal synthesis procedure ensures full testability. Optimal synthesis, typically results in a machine that is relatively easy to test. However, sequential test generation can be a

time-consuming process even for fully testable machines.

Previously, an approach to ensure easy testability of sequential machines via constrained state assignment and logic optimization was proposed [6]. While the constraints on synthesis result in some area overhead, test sequences to detect all single stuck-at faults in the combinational logic of the machine can be obtained via combinational test generation techniques alone. This approach can bound the length of the propagation sequence of any fault to be less than or equal to a prescribed value (e.g. ≤ 1).

The approach presented in this paper can be used in conjunction with the approach in [6] to produce fully and easily testable sequential machines. The state assignment and logic optimization constraints for easy testability can be accommodated without sacrificing full testability.

Thus, irredundant sequential machines with minimal area overheads can be synthesized and the test sequences for all faults in the machine can be obtained without resorting to sequential test generation.

7 Results

In this section, we present some preliminary results obtained using the synthesis procedures described in Section 4. Intensive optimization is necessary to obtain fully testable designs. If this optimization can be carried out, then the synthesized machine will occupy minimal area. There is no area/performance overhead associated with this procedure. However, the CPU time requirements have to be evaluated.

We chose some examples in the MCNC 1987 Logic Synthesis Workshop as test cases, whose statistics are given in Table 1. Beginning from a State Transition Graph description, G, the following steps were performed in the synthesis procedure.

1. **State Assignment:** Binary codes were assigned to the states in G using the program KISS [8]. The encoding length in some cases was greater than the minimum required. The codes were all minterms, and some minterms were not used. The combinational logic specification, a truth table, after encoding is denoted T.

2. **Logic Optimization:** T, with all the unused state codes specified as don't cares, was optimized using ESPRESSO, and algebraically factored to produce a multi-level logic network C. C was prime and irredundant.

EX	#inp	#out	#states	#edges
ex1	2	2	6	24
ex2	2	1	13	57
s1	8	6	20	110
planet	7	19	48	118
dfile	2	1	24	96
scf	27	54	128	168

Table 1: Statistics of Benchmark Examples

EX	#lat:	#gate	fault cov.	%abo. fault	l.o. time	TPG time	%red. fault	r.i. time
ex1	2	23	97.92	2.08	0.5s	2.0s	2.08	1.1s
ex2	5	35	98.15	1.85	2.2s	41.8s	1.85	6.1s
s1	5	105	99.79	0.21	5.5s	303s	0.21	4.0s
planet	6	193	100.0	0.0	10.5s	141.8s	0.0	0.0s
dfile	6	77	97.80	2.20	6.2s	331.8s	2.20	41.8s
scf	8	402	100.0	0.0	121.4s	82.2m	0.0	0.0s

Table 2: Synthesis Procedure Results

Tests were generated for the resulting sequential machine M whose combinational logic is implemented by C. Test generation was accomplished using the program STALLION [12]. The number of encoding bits used in state assignment (#lat), the number of gates in C (#gate), the fault coverage obtained (fault cov.) by STALLION and the percentage of aborted/possibly redundant faults (%abo. fault) are given in Table 2. The CPU times logic optimization (l.o. time) and test generation (TPG time) are also given. All the aborted faults were checked for redundancy using algorithms in STALLION. The number of redundant faults (%red. fault) and the CPU time expended during redundancy identification (r.i. time) is given in Table 2. The CPU times for state assignment and the initial state minimization were negligible are not given. In the tables, s stands for CPU seconds on a VAX 11/8650 and m for CPU minutes. For all the cases, the machine produced is highly testable, $\geq$ 99% fault coverage. The larger examples, **scf** and **planet** which have significantly more outputs than latches are fully testable.

The examples of Table 2 with $<$ 100% fault coverage were re-synthesized using the extended don't care set as described in Section 4.3. The CPU time to check for equivalence between invalid and valid states (s.e. time), number of logic minimizations (#logic mini.), CPU time spent in

EX	s.e. time	#logic mini.	l. o. time	fault cov.	TPG time
ex1	0.5s	1	0.5s	100.0	2.1s
ex2	6.5s	7	22.4s	100.0	40.6s
s1	1.0s	1	6.1s	100.0	298.2s
dfile	10.2s	3	25.5s	100.0	747.7s

Table 3: Results using Extended Don't Care Sets in Synthesis

logic minimization (l.o. time), the final fault coverage (fault cov.) using STALLION and the test generation time (TPG time) are indicated in Table 3. The CPU time required for the state equivalence checks and the extra logic minimization steps are less than sequential test generation and redundancy identification (Table 2). Using the simple don't cares (**Step 3** in Section 4.3) resulted in fully testable designs in all cases. We have yet to find an example where this is not the case.

8 Conclusions

We have described a synthesis procedure that produces an optimized logic implementation of a sequential machine from a State Transition Graph description of the machine. This procedure typically results in a highly testable machine, with very few redundancies. These redundancies can be implicitly eliminated using state equivalence checking and combinational logic minimization to produce a fully testable machine. No direct access to the memory elements is required.

The optimal synthesis procedure described involves the steps of state minimization, state assignment and logic optimization. It is applicable to Moore or Mealy finite state machines. This procedure has no associated area/performance overhead unlike Scan Design methodologies. It can be used in conjunction with previous synthesis approaches to ensure easily testable sequential machines. In this case, test sequences which detect all single stuck-at faults in the sequential machine can be obtained via combinational test generation and depth-first search on the State Transition Graph.

Ongoing work includes the generalization of these methods to arbitrary interconnections of finite state machines.

9 Acknowledgements

This work was supported in part by the Semiconductor Research Corporation, the Defense Advanced Research Projects Agency under contract N00039-86-R-0365 and a grant from AT&T Bell Laboratories. Their support is gratefully acknowledged.

References

[1] V. D. Agarwal, S. K. Jain, and D. M. Singer. Automation in design for testability. In *Proc. of Custom Integrated Circuit Conference*, May 1984.

[2] R. K. Brayton, G. D. Hachtel, Curt McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[3] M. A. Breur and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, 1976.

[4] S. Devadas, H-K. T. Ma, and A. R. Newton. On the verification of sequential machines at differing levels of abstraction. In *IEEE Transactions on CAD*, June 1988.

[5] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Mustang: state assignment of finite state machines for optimal multi-level logic implementations. In *Int'l Conference on Computer-Aided Design (ICCAD)*, November 1987.

[6] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Synthesis and optimization procedures for fully and easily testable sequential machines. In *Proc. of International Test Conference*, September 1988.

[7] E. B. Eichelberger and T. W. Williams. A logic design structure for lsi testability. In *Proc. 14th Design Automation Conference*, June 1977.

[8] G. De Micheli et. al. Optimal state assignment of finite state machines. In *IEEE Transactions on CAD*, July 1985.

[9] K. Bartlett et. al. Multi-level logic minimization using implicit don't cares. In *IEEE Transactions on CAD*, June 1988.

[10] J. Hartmanis and R. E. Stearns. Some dangers in the state reduction of sequential machines. In *Information and Control*, September 1962.

[11] F. J. Hill and G. R. Peterson. Introduction to switching theory and logical design. John Wiley and Sons, 1981.

[12] H-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli. Test generation for sequential finite state machines. In *Proc. of Int'l Conference on Computer-Aided Design (ICCAD)*, November 1987.

[13] M. C. Paul and S. H. Unger. Minimizing the number of states in incompletely specified sequential circuits. In *IRE Transactions on Electronic Computers*, September 1959.