TECHNIQUES FOR LOGIC VALIDATION OF DIGITAL CIRCUITS

by

Hi-Keung Ma

Memorandum No. UCB/ERL M89/140

December 15, 1989

TECHNIQUES FOR LOGIC VALIDATION OF DIGITAL CIRCUITS

by

Hi-Keung Ma

Memorandum No. UCB/ERL M89/140

December 15, 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TECHNIQUES FOR LOGIC VALIDATION OF DIGITAL CIRCUITS

by

Hi-Keung Ma

Memorandum No. UCB/ERL M89/140

December 15, 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Techniques for Logic Validation of Digital Circuits

Hi-Keung Ma

Ph.D.

Department of Electrical Engineering
and Computer Science

ABSTRACT

With the ever-increasing complexity involved in the process of design verification and testing of digital circuits, new approaches and new tools for logic validation are needed. Without high-quality logic validation tools, a design may never be successfully implemented and manufactured. In this thesis, new algorithms for various logic validation tools are presented. New algorithms for combinational logic verification have been developed. Efficient parallel logic verification schemes for large complex circuits that normally require significant amounts of CPU time to verify have also been developed. An efficient deterministic test generation algorithm has been proposed and implemented that works well for mid-sized sequential circuits. For very large sequential circuits, an Incomplete Scan Design approach has been taken that can be used in conjunction with the deterministic test generation algorithm. A fault coverage estimation method for mixed-level circuits has been proposed. Given a test set, this fault coverage estimation method can produce an accurate fault coverage projection with significantly lower computational cost than a deterministic fault simulator. With the increasing density of on-chip components, multiple-fault testing of chips has begun assuming increasing importance. An implicit fault simulation algorithm for multiple-fault detection has been developed and implemented. Finally, synthesis-for-testability methods have been developed that produce fully testable, multi-level or PLA-based, sequential circuits.

Alberto Sangiovanni-Vincentelli
Thesis Committee Chairman

ACKNOWLEDGMENTS

I am indebted to my advisor Alberto Sangiovanni-Vincentelli for his constant support, inspiring guidance and inexhaustible interest in what I did throughout the course of my research at Berkeley. Much of this work would have been impossible but for his enthusiasm and invaluable advice.

I am truly fortunate to have been associated with Professors Robert Brayton and Richard Newton. I have greatly benefited from the numerous valuable discussions with them, whose precious feedback and constructive comments have led to many fruitful ideas in this work. I would like to thank Professor David Donoho for taking his time reading this thesis and providing useful comments.

Srinivas Devadas, my close research associate, long time office-mate and one of my best friends, has been truly supportive, motivating and helpful. The fun-filled moments we had, the silly things we did together and the private jokes we shared will leave a precious, everlasting and pleasurable mark on my memory lane.

Thanks are due to all members of the CAD group for providing the exciting and comfortable working environment. In particular, I would like to thank Albert Wang, Ruey-Sing Wei, Pranav Ashar, Abhijit Ghosh and Alex Saldanha for the many stimulating and useful discussions. I would also like to thank office-mate Gregg Whitcomb for putting up with my untidy office habits, Don Webber, Tom Quarles and Rick Spickelmier for answering many of my UNIX questions, Chuck Kring for lots of great tennis games (he nearly won), and Nick Weiner for providing information about and watching together Chinese movies.

I would like to thank Ruey-Sing Wei, Ajoy Bose and Mike Tong at AT&T Bell Laboratories in Murray Hill, for providing me the opportunity to spend the 1987 summer with their group.

Special thanks are due to Watson Chan and James Dunlap, who I had been fortunate enough to share apartments with for various periods and the early friends I made when I first

arrived in Berkeley. They have been really helpful and tolerant, especially during my adjustment period.

I dedicate this thesis, with love, to my deceased father and my mother whose perpetual love and unfailing support will forever be remembered warmly. I would also like to thank my brother Herman for initiating me into the exciting and rewarding field of Computer-Aided-Design.

Finally, I would like to thank my wife Ching-Yu for her love, encouragement and patience throughout my study at Berkeley.

TABLE OF CONTENTS

CHAPTER 1

Introduction

Logic Validation refers to the process of design verification and testing. There are many reasons why a particular chip coming from a fabrication line may not work. It may be due to a simple production problem where two metal lines are shorted because they are too close to each other on the chip or it may be because of an error introduced during the design process. *Design verification* addresses the latter problem by ensuring that no errors have been introduced from the beginning to the end of the design cycle. *Testing* refers to those tasks needed to assure that a chip functions correctly as it is designed. Results of the testing process may also be used to help identify the causes of a low fabrication yield and this activity is called fault diagnosis.

With the advances in integrated circuit (IC) technology, the number of components which can be placed on a single chip has increased rapidly. This has greatly increased the complexity of the design and testing processes and hence the need for automation. New approaches and new tools for logic validation are desperately needed. Without high-quality logic validation tools, a design may never be successfully implemented and manufactured. In this chapter, a review of the various logic validation tools and their application in the design process is presented.

1.1 Combinational Logic Verification

Synthesis systems [1] that can generate automatically mask-level layout of integrated circuits from high-level descriptions have been receiving considerable research and development effort. In particular, synthesis systems for *Application-Specific-Integrated-Circuit* area is a popular choice for automatic IC generation from high-level specification.

A typical, simplified synthesis system is shown in Figure 1.1. An integrated circuit is described by a high-level, programming-language-like specification. The design is partitioned to reduce the problem to a set of simpler subproblems through combinational block extractions. Each combinational block can be implemented by PLAs or multi-level logic using two-level logic minimizers [2] or multi-level logic optimization systems [3]. *Technology mapping* is applied to target the logic designs towards a specific technology. *Place-and-route* tools are used to produce the final layout.

The synthesis process is in general a very complex one and so verification at each stage of the synthesis process is indispensable. Logic verification tools are used to compare the logic design of integrated circuits at different levels and are needed to make sure that no logic errors have been introduced in the synthesis process. For example, in a synthesis system where a design is translated (synthesized) into a lower level from a higher level description, logic verification is usually performed between functional level (before logic synthesis) and gate level (after logic synthesis), as well as between gate level (before layout generation) and layout level (after layout generation). Because of the great complexity of the optimization tools in use today, it is essential to verify that both the designer and the tools themselves have not introduced any design errors during synthesis. Combinational logic verification is an indispensable step in total design verification for manually or automatically generated designs.

1.2 Test Generation

Testing of ICs is a process used to ensure that a particular chip satisfies its functional specification. For testing a circuit, binary patterns, called test patterns or tests, are applied to

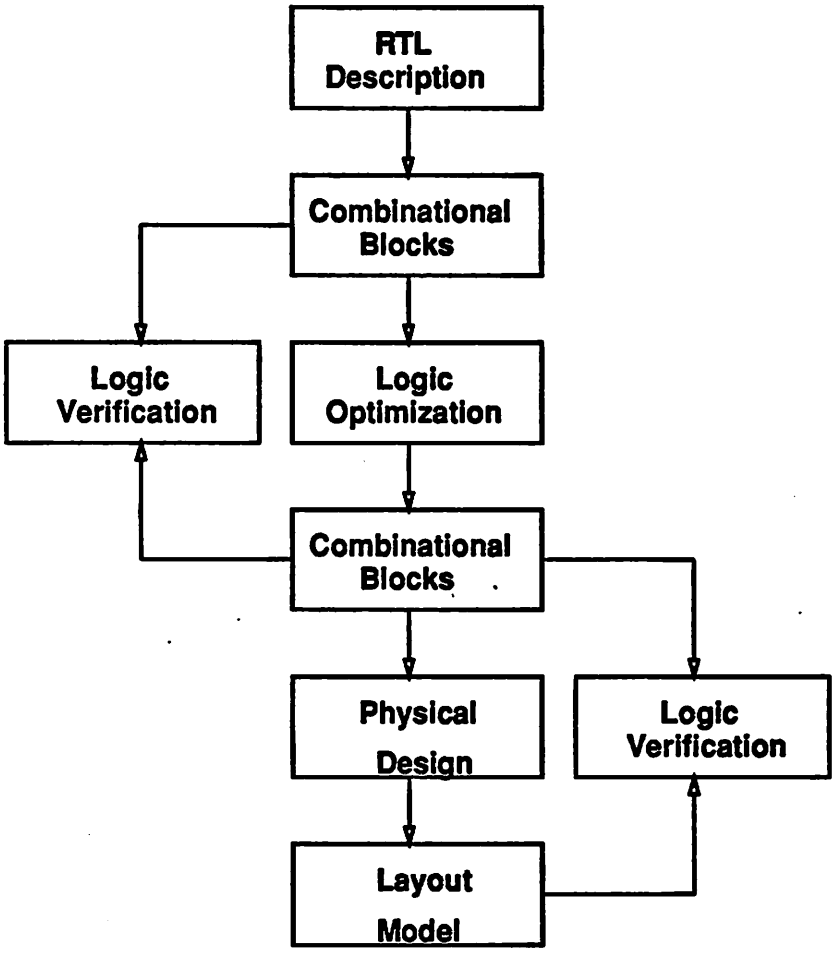


Fig. 1.1 A typical synthesis system

the inputs of the circuit and the response of the circuit is compared with the expected one. Based on the outputs of the circuit under test, bad chips are identified on a go/no-go decision. If the yield of the fabrication process is low, further diagnosis tests may be performed on the malfunctioning chips to identify the causes of the faults. Application of all possible input patterns for combinational circuits, and all possible sequences for sequential circuits, will

guarantee that the chips passing the test are all functionally good. Using exhaustive testing method, the number of test vectors required for a combinational circuit is 2^n and $2^n \times (2^{m+1} - 1)$ for a sequential circuit, where n is the number of primary inputs and m is the number of memory elements in the circuit. However, the exhaustive testing method is infeasible, in terms of testing time per circuit, when the number of inputs is large.

In practice, a set of test patterns that are aimed to detect a high percentage of modeled faults is used. The modeled faults are abstraction of the physical failures. The most widely used fault model has been the *stuck-type* model [4]. Physical failures are assumed to correspond to a line in the gate-level description of the circuit stuck at a 0 or 1 value and an assumption is made that only one fault can occur at a time. It has been empirically shown that a high percentage of the chips passing the set of test patterns for stuck-type faults are correct working chips.

Test generation for combinational circuits has traditionally been considered to be a search problem [5] [6]. A test pattern for a fault is generated by searching through the input space to find an input pattern that excites the fault and propagates its effect to one of the primary outputs. The cost of test generation can be very high and it has been proved that the problem of test generation is NP-complete [7]. It is especially expensive to generate tests for circuits that contain a large number of redundant faults. Redundant faults are faults for which no test can be found after searching, implicitly or explicitly, across the entire input space. The cost for trying to prove no tests exist for redundant faults, i.e. redundancy identification, can be more than 90% of the total test generation time.

In the past few years, great advances have been made for combinational test generation [8] [9] [10]. Test generation and redundancy identification for very large combinational circuits can now be performed efficiently by intelligently utilizing the topological information of the circuits under test. However, test generation for sequential circuits has remained very much an unsolved problem.

Generating tests for sequential circuits is considerably harder than for combinational circuits. Even if the combinational part of a sequential circuit is made fully testable, it may still be impossible to obtain a high fault coverage for the sequential circuit. Some of the inputs and outputs of the combinational part are outputs and inputs respectively of the memory elements, i.e. flip-flops. Test patterns generated considering only the combinational part cannot be readily applied and fault effects cannot be observed directly at the inputs of these memory elements. The controllability and observability of the combinational part can be greatly reduced with the presence of memory elements. Previous approaches to solve the problem of sequential test generation are either extensions to the classical D-Algorithm [11] [12] [13] or based on random techniques [14] [15]. When the number of states of the circuit is large and the tests demand long input sequences, they can be quite ineffective for test generation.

A popular approach to solving the problem of test generation for sequential circuits is to make all the memory elements controllable and observable, i.e. Complete Scan Design [16] [17]. Scan Design approaches have been successfully used to reduce the complexity of the problem of test generation for sequential circuits by transforming the problem into that of combinational test generation which is considerably less difficult. However, there are situations where the cost in terms of area and performance of complete scan design is unacceptable. In addition, Complete Scan Design is a very conservative approach to the sequential testing problem, i.e. latches are made scannable without first examining the sequential testability of the circuit. This can introduce unnecessary area and performance penality. New and novel approaches for sequential test generation are therefore required.

1.3 Fault Simulation

Fault simulation is an essential tool for test generation and grading of test sets. In test generation, whenever a test is found for a fault, fault simulation is used to determine what other faults are also detected by the test so as to avoid unnecessarily generating tests for them. Fault simulation is also used to determine the fault coverage of a given test set, i.e. functional tests, to decide whether more tests are needed. The three typical methods of fault simulation for single stuck-at fault model are parallel, deductive [18] and concurrent [19]. For multiple stuck-at fault model, there is no efficient and feasible fault simulation approach, which has posed a major problem for multiple-fault detection.

The computational cost of fault simulation, even for single stuck-at fault model, is usually very high. It is known that [20] the CPU time and memory requirements for fault simulation for single stuck-at faults are proportional to the square of the number of gates in the circuit. This poses a serious limitation to its use for evaluating test patterns for VLSI circuits. One means to alleviate the problem is to use estimation methods based on statistical analysis [21] [22] rather than deterministic approach. The computational complexity of these estimation methods are greatly smaller and yet an accurate fault coverage projection can be obtained.

Fault simulation for multiple-faults is a very difficult problem due to the enormous number of fault combinations have to be considered. The traditional fault simulation approaches used for single stuck-at fault model is unsuitable and impractical even for small circuits. Alternative approaches based on examination of masking relations [23] among faults involve cumbersome manipulation of Boolean equations and are only feasible for small and restricted type of circuits. In order to perform multiple-fault detection, efficient fault simulation techniques for multiple-fault model are required.

1.4 Synthesis For Testability

As described in Section 1.2, generating tests for sequential circuits is a very hard problem and the cost for trying to prove no tests existing for redundant faults can be more than 90% of the total test generation time. It is therefore very attractive to synthesize fully and/or easily testable sequential circuits. This will guarantee the testability of the synthesized sequential circuit without the use of scannable memory elements and reduce the complexity of the overall test generation process.

It has been well known that optimal logic synthesis can produce fully testable combinational logic designs [24]. The synthesized circuits are guaranteed to be free of redundant faults. On the other hand, the relationship between synthesis and testability for sequential circuits is not understood as well as in the combinational case. And to date no sequential logic synthesis algorithms exist that will guarantee that the resulting sequential circuit is fully testable without resorting to post-design design-for-test techniques such as scan-based methods. Synthesis systems that address the testability problem simultaneously to produce 100% testable sequential designs are very desirable. Synthesis-For-Testability method has emerged both as an exciting research topic and an important tool to alleviate the testing problem.

1.5 Organization of Dissertation

The organization of the dissertation is as follows. A new combinational logic verification approach and its parallel implementation [25] on a multi-processor system is described in Chapter 2. In Chapter 3, a novel approach [26] to test pattern generation for sequential finite state machines that represents a significant departure from previous methods is described. An Incomplete Scan Design methodology for very large sequential circuits is also presented in Chapter 3. A mixed-level fault coverage estimation method [22] based on statistical fault analysis for single stuck-at fault model is described in Chapter 4. In Chapter 5, an implicit fault simulation approach for multiple fault detection is discussed. Synthesis for testability

techniques [27] [28] [29] for PLA-based and multi-level logic implementations of finite state machines are described in Chapter 6.

CHAPTER 2

Logic Verification Algorithms And Their Parallel Implementation

2.1 Introduction

Logic verification tools compare the logic design of integrated circuits at different levels to make sure that in the synthesis process, no logic errors have been introduced. For example, in a synthesis system environment where a design is translated (synthesized) into a lower level from a higher level description, logic verification is usually performed between functional level (before logic synthesis) and gate level (after logic synthesis), as well as between gate level (before layout generation) and layout level (after layout generation). Because of the great complexity of optimization tools in use in synthesis systems today, it is essential to verify that both the designer and the tools themselves have not introduced any design errors during synthesis. Combinational logic verification is an indispensable step in total design verification for manually or automatically generated designs.

Several formal verification techniques (e.g. [30] [31] [32] [33] [34] [35]) have been proposed in the past but only a few have been applied due to their complexity and computational requirements. In PROTEUS [36], a number of efficient techniques for combinational logic verification have been developed and implemented. The PROTEUS system includes four basic approaches: verification by justification, verification by cube comparison, verification by exhaustive simulation, and verification by cover generation and simulation. The last approach, called LOVER (LOgic VERification) in PROTEUS, is novel and has given some excellent results compared to existing techniques. Most of logic verification algo-

rithms suffer from the problem of multiplicative blow-up. LOVER was developed with the specific goal of eliminating this problem.

In this chapter, new LOVER-based approaches for verifying the Boolean equivalence of two combinational logic circuits are presented. These approaches compare favorably to other LOVER-based approaches.

Large complex logic circuits require significant amounts of CPU time to verify. Parallel logic verification algorithms are therefore extremely attractive. However, to date, no logic verification technique has been efficiently parallelized.

Parallel logic verification algorithms based on the LOVER approach have been developed for the first time. Novel approaches to parallelizing both general and specific LOVER-based approaches are presented. These parallel implementations can be used over a large number of processors while maintaining high overall efficiency.

This chapter is organized as follows. In the next section, the attractiveness and applications of parallel logic verification and tautology checking schemes is discussed and the LOVER approach to logic verification is reviewed. In Section 2.3, embellishments to the basic LOVER approach is described and new verification algorithms is introduced. The parallelism inherent in the general LOVER approach and a multi-processor implementation exploiting this parallelism is described in Section 2.4. In the same section, a highly parallel version of a specific LOVER-based algorithm, which features a novel parallel enumeration algorithm based on PODEM, is described. Experimental results are presented in Section 2.5 to show that large speed-ups can be achieved when either of these parallelisms are exploited.

2.2 Preliminaries

In this section, the applications of parallel logic verification and tautology checking are first discussed. Then, the difficulties in parallelizing existing logic verification schemes are discussed. Finally, the LOVER approach to verifying the equivalence of two logic circuits is described.

2.2.1 Applications Of Parallel Logic Verification

Combinational logic verification is equivalent to the Boolean tautology problem. Tautology checking has a variety of applications in the logic synthesis area. Two-level logic minimizers can be implemented using tautology checkers alone [2]. Exact logic minimization can expend huge amounts of CPU time. Parallel tautology checking can form the basis for exact or heuristic logic minimization on parallel computers which can result in being able to perform minimizations much more quickly. Recently, multi-level Boolean minimization techniques have been developed which use primarily tautology checking operations [37]. These techniques produce excellent results but can be used only on small circuits due to CPU-time limitations. Parallelizing these techniques can result in being able to handle larger circuits.

Redundancy checking of stuck-at faults in a combinational circuit is also equivalent to the logic verification problem – if the faulty and fault-free circuit are Boolean equivalent, the fault in question is redundant. Combinational test generation algorithms spend significant amounts of time attempting to identify or generate tests for redundant faults. Redundancy checks can be performed more quickly using parallel logic verification schemes.

2.2.2 Difficulties In Efficient Parallel Logic Verification

Parallelizing logic verification algorithms on a coarse grain is quite easy. For example, different outputs of a circuit can be verified in parallel on different processors, using, for instance, Roth's VERIFY algorithm [31]. However, if the number of processors is larger than the number of outputs in the circuit then high efficiencies cannot be obtained. Even if the number of processors equals the number of outputs, verifying one output may take much longer than verifying the others resulting in considerable processor idling times and low efficiences. Ideally, one would like all the processors to do exactly the same amount of work so as to obtain maximum efficiency. This is quite difficult to do and efficient schemes for parallel logic verification have not been proposed thus far.

2.2.3 LOVER

LOVER incorporates a two-set/two-phase approach to logic verification which avoids the multiplicative blow-up problem in traditional logic verification methods and has achieved excellent results in comparison to existing approaches [36].

Let A and B be the two circuits whose equivalence has to be verified. A cube c from C_A^{ON} (the ON-set cover of circuit A) is *enumerated* and *simulated* on B to check if B produces a 1 at its output. If so, the enumeration/simulation process continues with another cube from C_A^{ON}. If, on the contrary, a 0 appears, the verification is completed with the conclusion that A and B are not Boolean equivalent. If an x (unknown) appears, c is split (cube-split) into smaller cubes and re-simulated until a known value appears at the output of B. Cube-splitting and simulation are implicitly exhaustive. The process continues until all cubes from C_A^{ON} have been simulated. A similar process for C_A^{OFF} (the OFF-set cover of circuit A) is then started and processed to the end.

This method is called a two-set/two-phase approach because there are two sets (the ON-set C^{ON} and the OFF-set C^{OFF}) that are to be explicitly verified; and two phases (the enumeration phase and the simulation phase) that are to be performed for each set verification. It is important to note that this framework does not specify which enumeration or simulation algorithm to use. This gives a large degree of freedom in the LOVER approach to verification – many different kinds of simulation and enumeration algorithms can be used. Since simulation is a relatively well understood and developed area, the emphasis is generally placed on developing efficient enumeration algorithms.

Using LOVER, there are only ($n_{ON,A}+n_{OFF,A}$) cube enumeration and simulation passes to be performed where $n_{ON,A}$ and $n_{OFF,A}$ are the number of cubes in C_A^{ON} and C_A^{OFF}, respectively. Though both $n_{ON,A}$ and $n_{OFF,A}$ can be exponentially related to n_i (the number of inputs to the verified circuits) in the worst case, the overall complexity is additive rather than multiplicative. So the problem of multiplicative blow-up is avoided.

In [36], it is indicated that performance of LOVER algorithms vary mostly because different enumeration algorithms are used. Various methods can be used in LOVER for enumeration – justification algorithms as seen in most test pattern generation techniques become enumeration algorithms after suitable modifications to the termination criterion.

Noting that another process, namely cube simulation, follows enumeration using the results from enumeration, the interaction between these two processes should be considered in order to achieve the best overall performance. Although it is generally desirable that the number of cubes enumerated should be as small as possible, the possibility of cube-splitting deserves attention. While verifying two circuits, A and B, an efficient enumeration algorithm may enumerate C_A^{ON} or C_A^{OFF} very efficiently with only a few cubes, but if most of these cubes need to be split several times during simulation on circuit B, the overall verification time suffers.

For ease of reference, in the remainder of this chapter, a LOVER algorithm using a specific justification algorithm X is referred to as LOVER-X. Among all the approaches presented in PROTEUS, it was found that the LOVER-SDIJUST approach was the most efficient. Hence, LOVER-SDIJUST has been used as an example for parallelizing the general LOVER-based approach as described in Section 2.4.1.

2.3 Efficient Enumeration Algorithms

In this section, two enumeration algorithms based on the PODEM justification algorithm are described [38]. LOVER-based approaches are compared to previous approaches to logic verification.

2.3.1 LOVER-PODEM

Enumeration in LOVER can be performed based on the decision tree concept in PODEM [6]. By modifying the termination condition of the implicit enumeration algorithm used in PODEM, both the ON-set and OFF-set can be implicitly, but exhaustively, enumerated. This is illustrated in Figures 2.1 and 2.2.

An example circuit with 5 inputs and a single output is shown in Figure 2.1. The decision tree in LOVER-PODEM while enumerating the ON-set of the output is shown in Figure

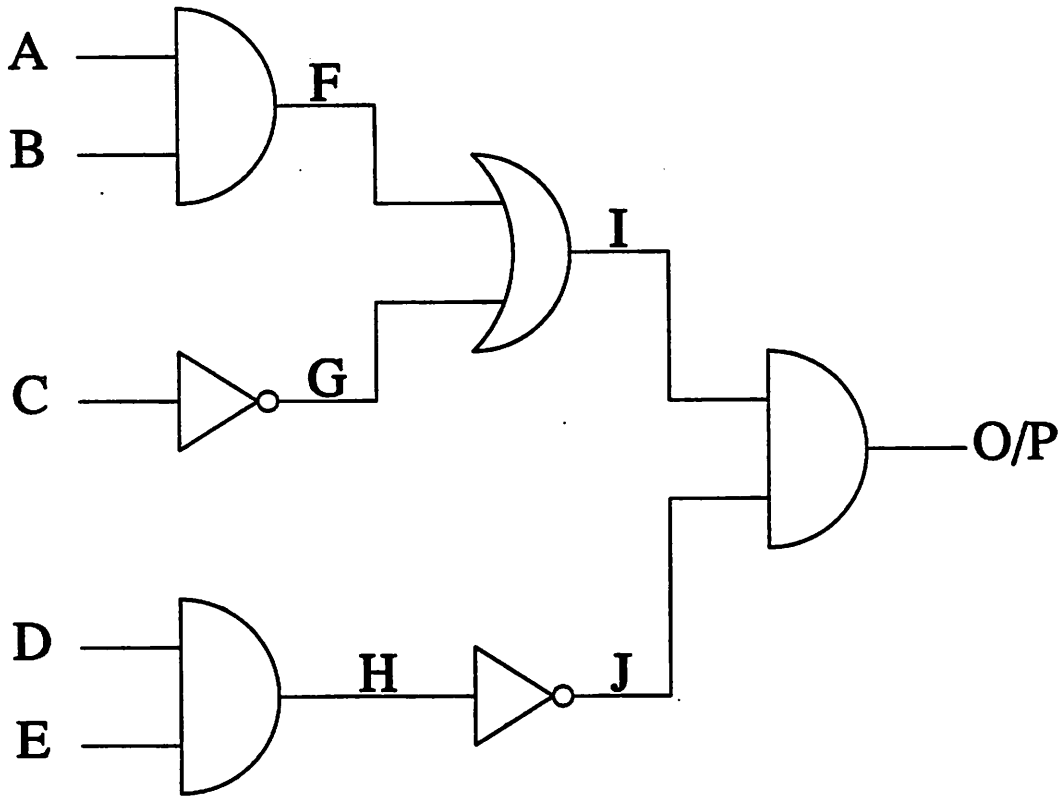


Fig. 2.1 An example circuit

2.2. Note that some OFF-set cubes may be generated. In general, two decision trees are required: one for the ON-set verification and the other for the OFF-set.

Each node in the decision tree represents a primary input (PI) assignment. Initially, all primary inputs are assigned unknown values (corresponding to the node START in the decision tree of Figure 2.2).

Given an initial objective, i.e. to set a primary output line to a 1 or 0, a path is traced from the objective line backwards to a primary input to obtain a PI assignment. A 1 initial objective corresponds to the enumeration of the ON-set and a 0 initial objective corresponds to the OFF-set enumeration. In our example, the objective was to set the primary output line to a 1. The first PI assignment was to set input C to 0 (Figure 2.2).

After each new PI assignment, the circuit is simulated using the current set of PI assignments to see if the value at the objective line has been set up. If not, the backtrace process continues. For example in Figure 2.2, after setting input C (to 0) the value of the primary output is unknown, so the backtrace process continues, selecting and setting input D (to 0).

If the desired value has been achieved, a cube in the corresponding set has been found. In our example, after D has been set, the desired value of the output (= 1) has been set up. The cube --00- has been enumerated in the ON-set of the circuit (Figure 2.2). When verifying against another circuit, this cube would be simulated on the other circuit.

If the opposite value has been set up, the algorithm backtracks to the last PI assignment, tries the alternative value and flags the node to indicate that both assignment choices has been tried. If the alternative has already been tried, the node is removed and the backtrack process continues until an unflagged node with a possible alternative is reached. The backtrack process is also applied when a desired value has been set up at the objective line. After enumerating cube --00-, the algorithm backtracks to the last PI assignment, namely D, and sets it to a different value of 1 (Figure 2.2). This is different from PODEM in which the enumeration process terminates when the desired value is set up at the objective line. When the

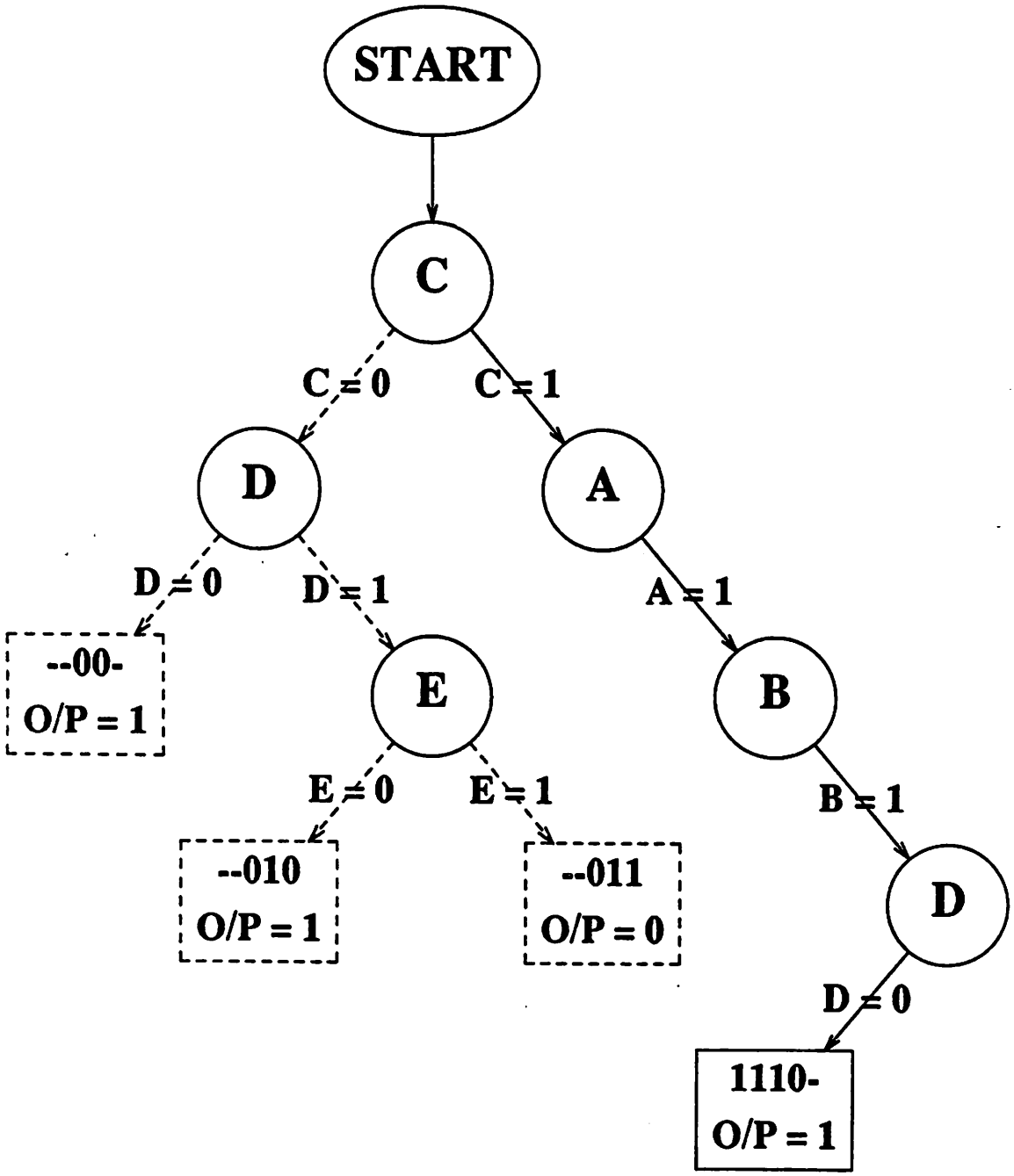


Fig. 2.2 Decision tree of LOVER-PODEM

decision tree is found to be empty in the backtrack process, the total input space for the corresponding set has been implicitly, but exhaustively, enumerated.

2.3.2 PLOVER: A Variation Of LOVER-PODEM

In the above enumeration process, the ON-set and OFF-set are enumerated separately with the initial objective being set differently in each case. One can observe that, however, in the enumeration process, backtracking is performed whenever the value of the objective line is set regardless of the value attained. In the enumeration of one of the two sets, cubes in the other set are actually being generated simultaneously but discarded. This represents a wasted effort in the enumeration process. A variation of the LOVER-PODEM method, called PLOVER, is therefore proposed.

In PLOVER, only one decision tree is used with the initial objective set to either a 1 or a 0 for the primary output. Cubes in both the ON-set and OFF-set are collected whenever the value of the output is set. Simulation of cubes from both sets are performed together rather than separately. The verifications of both sets are interleaved to avoid wasted enumeration effort. The choice of the initial objective, either a 1 or 0, is unimportant in terms of the completeness of the ON-set and OFF-set being generated. It only influences the size of the sets, the set corresponding to the initial objective tends to be more compact. Experience has shown that the enumeration process can be made more efficient if advanced knowledge of the relative sizes of the two sets is available by setting the initial objective corresponding to the bigger set.

2.3.3 Comparisons With Other Logic Verification Schemes

The LOVER-based algorithms compare favorably with other techniques to combinational logic verification. In Table 2.1, LOVER-SDIJUST and LOVER-PODEM are compared with Roth's VERIFY algorithm and exhaustive simulation (EXHSIM) on benchmark circuits [39]. The CPU times are on a VAX 11/8650 running ULTRIX. Exhaustive simulation would have taken days for the two larger circuits.

2.4 Schemes for Parallel Logic Verification

The LOVER framework supports various schemes for parallel logic verification. This section contains the description of a static scheduling scheme which can be used regardless of the enumeration algorithms used, and a highly efficient dynamic scheduling scheme using the LOVER-PODEM/PLOVER verification algorithms.

2.4.1 Static Scheduling: Parallelism Inherent In The LOVER Framework

The parallelism inherent in the LOVER framework regardless of what enumeration and simulation algorithms are used can be exploited using a static scheduling scheme [38]. In this scheme, each processor works largely independent of the others with very little inter-processor communication. The enumeration algorithm used has no influence on the speed-up but only on the absolute CPU time expenditure. The synchronization overhead for this scheme is

CKT	LOVER-SDIJUST	LOVER-PODEM	VERIFY	EXHSIM
alu4	7.2s	8.3s	131.0s	1.3s
C432	1.56h	2.15h	> 25h	-
C880	4.3h	4.2h	> 25h	-

Table 2.1 Comparisons of verification algorithms

minimal and the scheme is easily implemented.

A pair of cone circuits can be verified using 1-4 processors as summarized in Table 2.2. Mode 1 is identical to the uni-processor version of the algorithm for each cone. Mode 2 uses two processors one verifying the ON-set and the other the OFF-set. Mode 3 uses three processors, two enumerating the ON and OFF-sets and the third performing simulations. Mode 4 uses four processors, two each for simulation and enumeration.

Simulation is in general more efficient than enumeration. This is because typically several simulations of the circuit are required as part of the enumeration process before the output value becomes known. Using experimental data over a range of logic circuits, it has been determined that among the three processors dedicated to a cone circuit, typically one of the two enumeration processors represents the bottleneck and determines the time it takes to completely verify that cone circuit. Thus, Mode 4 is rarely used since the empirical evidence shows that simulation is usually about twice as fast as enumeration (when using LOVER-SDIJUST) making a fourth processor redundant.

mode no.	no. of processors	comments
1	1	serial algorithm (parallel in cones for entire ckt)
2	2	parallel in sets
3	3	parallel in sets and phases (shared simulation phase)
4	4	parallel in sets and phases

Table 2.2 Modes in static scheduling

2.4.1.1 Mode Selection

Given a circuit, a mode is selected for verifying each output so as to maximize efficiency. Estimates of the complexity in enumerating the cones are used in mode selection. These estimates are relative and are made by examining the structure of the logic network – for example, the number of levels in the network, the number of reconvergent fanouts, and the relative number of gates at each level. Given these estimates and the number of processors to be used, a mode is selected which results in each processor performing as equal an amount of work as possible, relative to the other processors.

Let the number of processors be n_p and the number of outputs n_o. The outputs are sorted in decreasing order of estimated complexity. If $n_p \geq 4 \times n_o$, then Mode 4 is selected for each output regardless of output complexities. Else, if $3 \times n_o \leq n_p < 4 \times n_o$, then Mode 4 is selected for the first $n_p - 3 \times n_o$ outputs and Mode 3 for the rest of the outputs. If $2 \times n_o \leq n_p < 3 \times n_o$ then Mode 3 is selected for the first $n_p - 2 \times n_o$ outputs and Mode 2 for the rest.

If the number of processors is less than twice the number of outputs, i.e. $n_p \leq 2 \times n_o$, then for efficiency reasons more than one cone may be allocated to a single processor. This is because it may be advantageous to use Mode 3, i.e. three processors, for a complex cone circuit. When the number of processors available is small compared to the number of outputs, efficient parallel verification can be achieved by assigning sets of outputs to the different processors, such that the sum totals of enumeration and simulation tasks to be performed by each processor are approximately the same. If the sets of tasks are exactly identical in complexity and CPU time requirements, an ideal 100% overall efficiency can obtained.

To find a good grouping of cones in multi-output circuits, the estimates of the complexity in enumerating the cones are used. Initially, Mode 1 is assumed for all outputs and a grouping of outputs is found such that each group has minimally varying complexity. If a group has a single cone circuit which represents the bottleneck, then the mode for that

output/group is increased and the remaining outputs are regrouped into fewer groups, again with minimally varying complexities. If this increases estimated efficiency then the new grouping/mode is adopted. Now, the bottleneck may be a different group or the same one. Modes are increased for the bottleneck groups till either Mode 3 is reached or efficiency drops.

2.4.1.2 Experimental Results Using The Balance Parallel Computer

A parallel LOVER-SDIJUST algorithm has been developed and implemented on the Sequent Balance 8000 multi-processor [40] using the static scheduling scheme described.

In Table 2.3, the results obtained on two circuits from [39] using 1-8 processors on the Sequent multi-processor are given. Two different implementations of each example circuit, which were equivalent, were verified against each other. Because the two circuits being compared were different in each example, cube-splitting was required during simulation. The percentage of cube-splitting was approximately 10% in both examples. The absolute verification time is proportional to the number of cubes that have to be split during simulation. Cube-splitting also affects the speed-ups that can be obtained via parallelization, but to a much lesser extent.

CKT	#inputs	#outputs	speed-up/mode						
			2	3	4	5	6	7	8
C880	60	26	1.90/1	2.60/1	3.30/2	3.82/2	4.70/3	4.70/3	4.70/3
C432	36	7	1.95/1	2.80/1	3.50/2	4.30/2	5.40/2	5.40/2	6.10/3

Table 2.3 Results using static scheduling

Both these circuits are complex and the uni-processor verification time on the Sequent is about 38 and 17 hours respectively using the LOVER-SDIJUST algorithm. Respectable speed-ups have been obtained over 8 processor configurations for both examples. The highest mode used for any cone circuit on different processor configurations is also indicated.

The first example saturates after 6 processors because verifying one output in the circuit is significantly more time consuming than any of the others, and a maximum of three processors can be used on a single cone given a static scheduling scheme. This output thus becomes the bottleneck in the parallel verification process. Since Mode 3 provides the highest degree of parallelism, i.e. 3 processors for a single output, it is used when the number of processors is equal to or exceeds 6. Three processors work on the bottleneck output and the remaining on the rest of the outputs. Better results are obtained in the second example, although it has fewer outputs, because no single output overwhelms the others in complexity. Three processors are required for the bottleneck output in this case, only when the number of processors equals 8.

In the following section, a dynamic scheduling scheme which enables an arbitrary number of processors to be used to verify a single cone circuit is described. Using this scheme, high processor utilization (and overall efficiency) can be obtained regardless of the number of processors available and the complexity of individual cone circuits.

2.4.2 Dynamic Scheduling

A dynamic scheduling scheme using the new PODEM decision tree based enumeration method PLOVER is devised [38]. Based on this scheme, high processor utilization on any kind of circuit is achieved. In the following section, how the enumeration method can be efficiently parallelized using dynamic scheduling will be described. Initially, for ease of explanation, a single-output circuit will be assumed while describing the parallel algorithm. Later, means of extending the algorithm to handle multiple-output circuits will be described.

In the LOVER framework as described in Section 2.2, the two main tasks performed in the verification process are enumeration and simulation. Cubes are continuously enumerated on one circuit and simulated on the other to check any functional discrepancies between the two.

The chief goal of dynamic scheduling is to continually distribute equal amounts of work among processors to avoid wasteful idling and achieve high processor utilization. Good processor utilization during enumeration can be achieved by repeatedly breaking up the enumeration task(s) into smaller ones and assigning them to different processors – an enumeration algorithm that is tailored for such a parallel application has been devised.

The parallel enumeration algorithm is based on the PLOVER algorithm described in Section 2.3. The input space is divided up into disjoint sub-spaces and each processor enumerates implicitly all possible input patterns in an assigned sub-space in parallel. Sub-spaces are further broken up, again disjointly, if some processors finish their assigned enumeration in the input sub-space before the others. Thus, even if the initially assigned sub-spaces are very different in enumerative complexity, processors which complete their tasks early don't remain idle but help other processors in completing their enumeration task.

Cube simulation on the cone circuit can be performed by any processor whenever the accumulated number of cubes generated by a processor is equal to the number of cubes that can be simulated in parallel by a parallel simulation algorithm. By proceeding in such fashion, a equal amount of verification work is assigned to each available processor and full utilization of processor time is achieved by continuously keeping all processors at work in parallel.

2.4.2.1 A Parallel Enumeration Algorithm

The enumeration algorithm used in PLOVER described in Section 2.3 is well suited for a parallel application. Whenever a new PI assignment is made, two disjoint input spaces are implicitly developed by the decision tree. These two input spaces correspond to the 0 and 1 values of the newly assigned input and the old values of all the previously assigned inputs. Some input values may still be unknown. Since these two input spaces are disjoint, they can be enumerated by two different processors in parallel with the guarantee that the resulting two sets of enumerated cubes will also be disjoint. Thus, no redundant enumeration work is done using this technique – each processor enumerates on a different branch of the decision tree.

Disjoint input spaces are continually generated by all the processors doing the enumeration every time a new PI assignment is made. After a processor performs a PI assignment, it picks one of the disjoint spaces and continues enumeration on that space. As soon as a processor completes enumerating its present input space it then picks up another branch which corresponds to previously generated input spaces by other processors which have not yet been enumerated. This process continues till the entire input space has been enumerated. The selection of a new input space by a processor on the completion of its initially assigned task (this input space would have been generated by some other processor) entails an initialization overhead. It is therefore desirable to select the largest unenumerated input space available which corresponds to the space with the minimum number of assigned primary inputs.

In the example decision tree of Figure 2.2, the PI assignments, C=0 and C=1, correspond to two disjoint input spaces. Processors can enumerate on these two different branches of the tree in parallel.

2.4.2.2 Implementation

The decision tree is an ordered list of nodes and is implemented as a stack. Each processor owns a separate stack which corresponds to the input space currently being enumerated by it. Whenever a new PI assignment is made, a new unflagged node is pushed onto the top of the stack. And whenever a backtrack step is made, the node on the top of the stack is examined. If the node is unflagged, the alternative value is assigned to the corresponding input and the node is flagged to indicate both choices have been tried. If the node is found to be flagged, it is popped from the stack. Enumeration of a particular input space is completed when the stack becomes empty. The stack is therefore treated as a FILO queue by the owning processor. When a stack is being manipulated by a processor, no other processor is allowed to access it.

The selection of a new input space by a processor is done by popping nodes from the bottom of the stack of another processor and pushing them onto the processor's own stack. This popping and pushing process continues until the first unflagged node is reached. This unflagged node is flagged and the corresponding input is assigned the alternative value creating a new disjoint input space on which the processor enumerates. The popping of nodes begins from the bottom of the stack rather from the top so as to obtain the largest unenumerated space to minimize initialization overhead. The implementation of the parallel enumeration algorithm is illustrated in the pseudo-code below.

2.4.2.3 Incorporating Dynamic Scheduling Into A Global Verification Scheme

Circuits with an arbitrary number of outputs can be efficiently verified using the dynamic scheduling scheme described thus far by using all the processors to verify each output, and verifying the outputs sequentially. Another option is to perform enumeration directly on the multiple-output circuit rather than using conning – each output has to be set to 1 or 0 for every cube enumerated.

```
parallel_enumerate()
{
  while (enumeration_not_finished) {
    if (output_is_not_set) {
      find_new_pi_assignment();
      push a new unflagged node on top of its stack, S1;
      simulate the current set of pi assignments;
    }
    else {
      if (output is a 1) a cube from ON-set is generated;
      else a cube from OFF-set is generated;
      while (S1 is not empty AND node on top of S1 is flagged) {
        pop a node from the top of S1;
      }
      if (an unflagged node is found) {
        flag node and assign alternative value to the pi;
        simulate the current set of pi assignments;
      }
      else {
select:   select a non empty stack S2 of another processor;
        while (node at bottom of S2 is flagged AND S2 is not empty) {
          pop the node and push on top of S1;
          assign the pi value corresponding to that node;
        }
        if (an unflagged node is found) {
          pop the node and push on top of S1;
          flag node and assign alternative value to the pi;
          simulate the current set of pi assignments;
        }
        else go to select;
      }
    }
  }
}
```

Greater efficiency is gained by incorporating the dynamic scheduling strategy into a global verification scheme. Initially, each processor tries to pick and verify an output, enumerating and simulating over the entire input space. If a processor runs out of unverified outputs it then helps the processors which have not completed their outputs, via dynamic scheduling. Thus, the overhead of selecting new input spaces to enumerate on is minimized. To further minimize the initialization overhead incurred in the selection of a new input space by a processor on the completion of its initial assigned task during dynamic scheduling, a feature

called a preferred stack mechanism is implemented. This feature restricts a processor to enumerate on unfinished input space of one cone circuit before switching to another one by assigning different priorities to input spaces of different cone circuits during input space selection. This prevents a processor from unnecessary switching between unfinished input spaces of different cone circuits and increases overall efficiency.

2.4.2.4 Results

Results for five examples using dynamic scheduling are given in Table 2.4. In the table, h, m and s stand for hours, minutes and seconds respectively. The first two examples are benchmark circuits from [39]. The number of outputs for the five examples are 3, 26, 2, 1 and 8 respectively. Two different, but equivalent, implementations of each example circuit (the second implementation was obtained by performing a partial collapse of the original benchmark implementation) were verified for equivalence. Regardless of the number of outputs, the number of processors used and the great variations in logic complexities among different cones circuits in the benchmarks, in every case except example 5, the speed-ups are very close to the ideal values. The reason that the speed-up deviates from the ideal value when the number of processors is large in example 5 is because the verification time is so small, circuit read-in time becomes significant; it is about 11s and represents over 40% of the total run-time in the 8-processor case and over 50% in the 12-processor case. If the circuit read-in time is deducted from the total run-time, the speed-up in the verification phase is again close to the ideal value.

The amount of cube-splitting affects absolute performance but not the speed-ups obtained due to parallelization. This is because of the load balancing characteristic of the dynamic scheduling scheme. The percentage of cubes that had to be split during simulation varied between 5 and 15% for the examples of Table 2.4.

A profile of the time each processor spent enumerating and simulating on the various outputs in example C880 is shown in Figure 2.3 for a 8 processor configuration. In the figure,

CKT	Number of processors									
	1		2		4		8		12	
	ABS. time	speed up	ABS. time	speed up	ABS. time	speed up	ABS. time	speed up	ABS. time	speed up
C432*	10.9h	1	5.49h	1.99	2.78h	3.92	1.38h	7.92	0.97h	11.22
C880	33.9h	1	17.0h	1.99	8.54h	3.98	4.28h	7.92	2.97h	11.41
ex1	69.7m	1	35.0m	1.99	17.7m	3.94	9.10m	7.68	6.25m	11.14
ex2	95.4m	1	48.4m	1.99	24.2m	3.96	12.2m	7.84	8.50m	11.22
alu4	104s	1	57.5s	1.81	35.3s	2.95	23.2s	4.49	18.3s	5.67

C432* : only the first three outputs

Table 2.4 Results using dynamic scheduling

the time profiles for each output have been normalized and the absolute verification time for each output is indicated to the right of the plot. As can be seen, the outputs which take a long time to verify have been shared out amongst many processors illustrating the excellent load balancing characteristics of the dynamic scheduling scheme, which is key to obtaining high overall efficiences.

2.5 Conclusions

In this chapter, new algorithms based on the LOVER approach for combinational logic verification is presented. For the first time, parallel logic verification schemes has been developed and high overall efficiencies over a large number of processors is achieved.

Parallelism inherent in the LOVER approach can be exploited using a static scheduling scheme. The advantage with this approach is that it is independent of the enumeration and simulation algorithms used. High speed-ups have been obtained on benchmark circuits.

A dynamic scheduling scheme using a PODEM-based enumeration algorithm has been developed. Excellent results have been obtained on different circuits with arbitrary numbers of processors. High efficiences of over 95% have been obtained using this scheme.

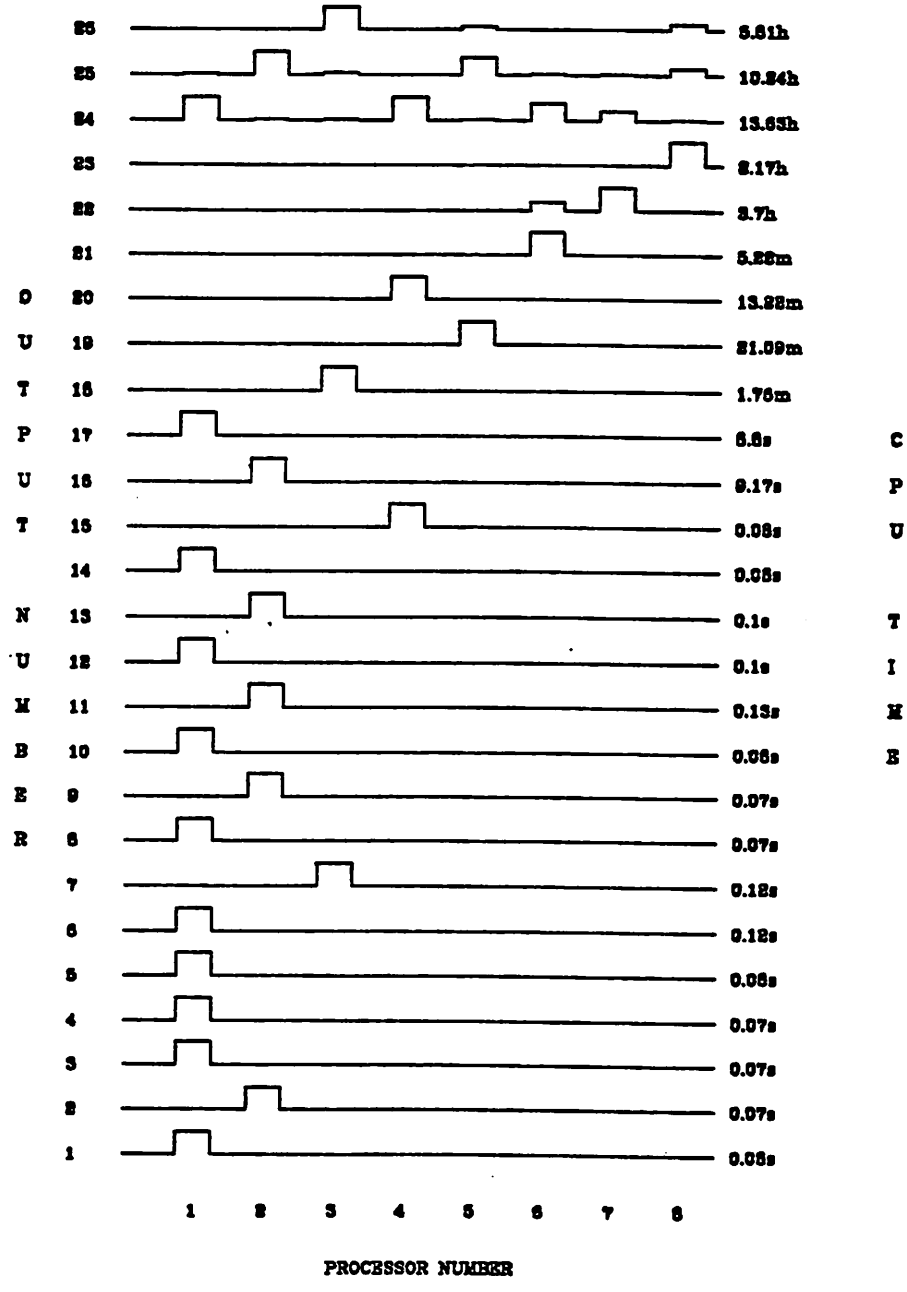


Fig. 2.3 Processor output time profiles for C880

CHAPTER 3

Sequential Test Generation

3.1 Introduction

Test generation for sequential circuits has long been recognized as a difficult task [41] [42] [4] [43]. Unstructured random sequential designs are very difficult to test. One common approach to improve the testability of a sequential circuit is to add test points to the circuitry so that tests can be applied more readily and fault effects can be observed better. But this method is not systematic and relies on designer ingenuity.

A popular approach to solving the problem of test generation for sequential circuits is to make all the memory elements controllable and observable, i.e. Complete Scan Design [16] [17]. Scan Design approaches have been successfully used to reduce the complexity of the problem of test generation for sequential circuits by transforming the problem into that of combinational test generation which is considerably less difficult. The design rules of Scan Design also constrain the sequential circuits to be synchronous so that the normal operation of the sequential circuit is free of critical races. However, there are situations where the cost in terms of area and/or performance and/or testing time of Complete Scan Design is unaffordable. In addition, even though the general sequential testing problem is very difficult, there are cases where test generation can be effective. Simply making all the memory elements scannable in a sequential circuit without first investigating how difficult is the problem of generating tests for it could unduly incur unnecessary area cost.

The difficulty in generating a test usually lies with: 1) setting the states of the memory elements into a certain combination so that the fault under test is excited; 2) propagating the fault effect to the primary outputs. An input sequence is usually required in both cases (if

such a sequence exists). In general, the longer the length of the shortest input sequence needed to perform Steps 1) and 2), the more difficult it is to find an input sequence to test the circuit. Both approaches mentioned above attempt to shorten the length of the input sequence. In the Scan Design approach, the length of the input sequence is reduced to one when all memory elements are made scannable.

Several approaches [11] [14] [12] [44] [15] [13] have been taken in the past to solve the problem of test generation for sequential circuits. They are either extensions to the classical D-Algorithm or based on random techniques [14] [15]. When the number of states of the circuit is large and the tests demand long input sequences, they can be quite ineffective for test generation. This is because no *a priori* knowledge of the length of the test sequence is available. In the extended D-Algorithm methods, a large amount of effort may be wasted in trying to find short sequence tests for faults that require long ones. Random testing techniques are based on continuous simulations and grading of test vectors according to simulation results. They too can be very time consuming for difficult faults that have only a few long test sequences. In this chapter, a new approach to test pattern generation for sequential finite state machines that represents a significant departure from previous methods is described.

An efficient deterministic sequential test generation algorithm based on extensions to the PODEM [6] justification algorithm has been developed. The problem of generating tests for faults that require a lengthy input sequence is handled efficiently by the intelligent use of information contained in a partial State Transition Graph (STG) and the integration of new algorithms based on the concept of **state space enumeration.**

It is assumed that the sequential circuit under test is synchronous and free of races under simple design rules. It is also assumed that there is a reset state for the synchronous sequential machine and memory elements such as D flip-flops are identified and represented as logical primitives to facilitate loop cutting in transforming the synchronous sequential circuit into an iterative array. A part of the State Transition Graph (STG) of the finite state machine

using purely structural information is extracted first, i.e. the gate-level description of a sequential circuit. The construction of the partial STG is based on an efficient algorithm that finds paths from the reset state to different valid states (states reachable from the reset state) in the STG. For circuits with relatively few states, a partial STG including all valid states is built. For circuits with a large number of states, only a subset of valid states is included in the partial STG. The partial STG is then used in conjunction with efficient fault excitation-and-propagation and state justification algorithms to generate tests for line stuck-at faults. Tests have been successfully generated for finite state machines with a large number of states using reasonable amounts of CPU time and obtained close to maximum possible fault coverages.

For very large sequential circuits, an Incomplete Scan Design methodology has been developed. First, using the sequential testing algorithm, test sequences are generated for a large number of possible faults in the given sequential circuit. A **minimal subset of memory elements** is then found, which if made observable and controllable will result in easy detection of all remaining irredundant but difficult-to-detect faults. The deterministic test generation algorithm is again used to generate tests for these faults in the modified circuit (the circuit with the identified memory elements made scannable). Detection of all irredundant faults as in the Complete Scan Design case, but at significantly less area and performance cost, is guaranteed. The length of the test sequences for the faults can be bounded by a prescribed value – in general, a trade-off exists between the number of memory elements required to made scannable and the maximum allowed length of the test sequence.

Preliminaries and basic definitions are given in the next section. Previous approaches to solving the sequential test generation problem is described in section 3.3. The deterministic test generation algorithm is described in Section 3.4 [26]. This algorithm has been implemented in the program, STALLION. Results obtained on several circuits using STALLION are presented in Section 3.5. In Section 3.6, the Incomplete Scan Design methodology [45] to

sequential test generation is introduced. Algorithms to identify the critical memory elements are presented. Results using STALLION in conjunction with the Incomplete Scan Design approach are given in Section 3.7.

3.2 Preliminaries

3.2.1 Introduction

A general sequential circuit is shown in Figure 3.1. It is realized by combinational logic and feedback registers. The general sequential test generation problem involves finding primary input sequences which can excite the faults in the circuit and propagate their effects to the primary outputs. No access to the inputs and outputs of the memory elements (the next state and present state lines, respectively) is given.

The initial state of the machine (e.g. when powered on) may be unspecified or specified. The fault model used in test generation also varies. Certain assumptions are made regarding the sequential circuit to be tested.

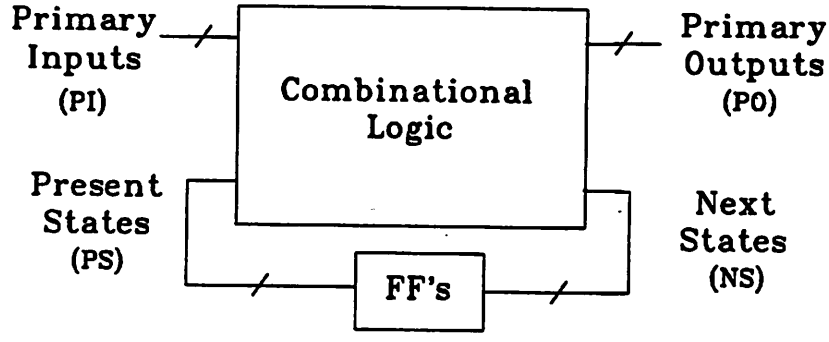


Figure 3.1: A Sequential Circuit

(1) The machine is assumed to have a **reset state,** R. All input test sequences begin from this reset state.

(2) The fault model is assumed to be **single stuck-at.**

(3) The memory elements are assumed to be represented as distinct logic primitives. Tests are generated for all single stuck-at fault in the combinational logic part of the sequential machine including faults on the present state lines, primary inputs, next state lines and primary outputs. Faults within the flip-flops are not considered.

3.2.2 Difficulties in Sequential Test Generation

In combinational test generation, a single test vector suffices to excite a fault in the circuit and propagate its effect to the primary outputs. A sequential machine, however, has to be first placed in a state which can excite the fault and only then can the effect of the fault be propagated to the primary outputs. Placing the machine in the required state and propagating the effect of the fault to the primary outputs may each require a sequence of input vectors.

In sequential test generation, the search space for a test sequence for a fault is usually very large. For a synchronous sequential circuit with m inputs and n memory element, the search space for a fault is $2^m \times 4^n$. The maximum possible length of a test sequence is $r + s - 1$ where r is the number of states of the good machine and s is the number of states in the bad machine. In general, r and s is not known in advance. The worst case complexity of the problem is 4^n times the complexity of generating a single test vector for the combinational part of the circuit. Some faults in the circuit may be **redundant,** i.e. they cannot be detected by *any* test sequence. Large amounts of effort may be wasted in trying to generate tests for redundant faults, which are, in general, quite difficult to identify.

3.2.3 Basic Definitions

The conventional iterative array model [4] used in sequential test generation is shown in Figure 3.2. The combinational logic block of the original sequential machine (Figure 3.1), with a fault, F, to be detected in it, has been duplicated in each time-frame. Beginning with the present state lines in time-frame 1, PS^1, set to the reset state values, it is required to produce an input sequence, $PI^1, PI^2, .., PI^n$, for some n, which when applied to time-frames 1, 2, .. , n propagates the effect of the fault, F to the primary output lines of the n-th time-frame, PO^n. This input sequence is called a **test sequence** for the fault.

A state is considered as a bit vector of length equal to the number of memory elements (latches or flip-flops) in the sequential circuit. In general, a state is a **cube**, i.e. the values of the different bit positions (state lines) may be 0, 1 or x (don't care). A state with only 0's or 1's as bit values is called a **minterm** state.

A state is said to **cover** another state if the value of each bit position in the first state is either a x or is equal to the value of the corresponding bit position in the second state.

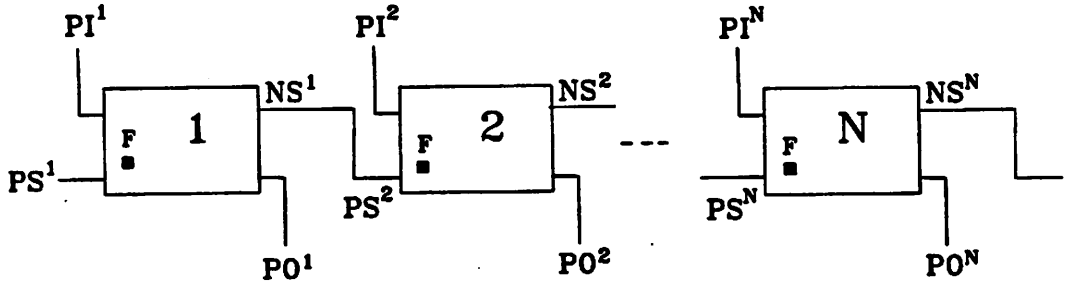


Figure 3.2: Iterative Array Model for Sequential Circuit

The process of finding an input sequence which places the machine, initially in its reset state, R, into a given state, S, is called **state justification.** The input sequence in question is called a **justification path.** State justification may be **forward** state justification or **backward** state justification, depending on whether the search is conducted from R forwards or from S backwards.

The search space in sequential test generation is deemed to be the product of two spaces, namely the **input space** and the **state space.** The **dimension** of each space is equal to the number of Boolean variables in the space. For example, the dimension of the input space is equal to the number of primary inputs. These spaces correspond to the universal input cube and the universal state cube respectively (the universal cube is a cube with all x entries of length equal to the dimension of the space).

A space can be **enumerated** by exhaustively searching a set of cubes which add up to the universal cube corresponding to that space. **Minterm enumeration** implies that each cube searched is a minterm. Minterm enumeration on a n-dimensional space, implies 2^n combinations have been searched. **Implicit enumeration** (or **implicit cube enumeration** or **cube enumeration**) involves exhaustively searching a n-dimensional space via cubes such that the number of cubes searched is significantly less than 2^n.

In a sequential circuit, a fault may be **redundant,** i.e. untestable. There are two kinds of redundancies in a sequential circuit. The first kind is deemed **combinationally redundant** – the effect of the fault cannot be excited or propagated to the *primary outputs or the next state lines*, beginning from any state, with any input vector. A **sequentially redundant** fault is a fault which can be excited by some input vector but its effect cannot be propagated to the *primary outputs.*

3.3 Previous Work In Sequential Testing

3.3.1 The Extended D-algorithm for Synchronous Circuits

The test generation procedure for a self-initializing test sequence based on the iterative array model using the extended D-algorithm approach is as follows:

(1) Determine the maximum number of time frames, p, allowed for test generation.

(2) Choose an initial value for p and construct the iterative array model with y_I of unknown value. (If a reset state is given, y_I will be assigned the value of the reset state.)

(3) Choose the time frame q from which the D-drive must be organized. Apply the D-algorithm to find a test for the multiple fault f^p so that a D or $\bar{D}$ appears at one of the outputs $z_1, z_2, \ldots z_p$. If a test is found, exit; otherwise, continue.

(4) If possible, increment p by 1 and return to Step 3); otherwise, exit with no test.

Since the length of test sequence cannot be determined *a priori*, a large amount of effort may be wasted in trying to generating tests with inappropriate choice of p.

3.3.2 Weighted Random Test-pattern Generator

In a random test-pattern generator, sequence of random patterns are applied to the circuit. In general, all primary inputs (PI's) of the circuit have the same weights, i.e. each PI is exercised approximately the same number of times averaging over a long period of time. In the weighted random test-pattern generator [14], different weights are assigned to the PI's in proportion to their relative importance, i.e. some PI's are exercised more often than others. Single input change between two consecutive patterns is assumed.

One way to determine the weight assigned to each PI is to measure the amount of gate switching activity produced inside the circuit as the result of exercising that PI. A set of random patterns is simulated on the circuit. The number of gates changing for the first time from a logic 1 to 0, and vice versa, due to the switching of any of the PI's, is counted. The switching activity count is then accumulated over the complete set of random patterns. By

comparing the activity created by all PI's, different weights can be determined for all PI's. However, this method suffers from the fact that the importance of the order of patterns applied to detect a fault is ignored. Furthermore, test sequences consisting of more than one change between consecutive patterns cannot be generated.

A dynamic adaptive technique has been used to partially alleviate the problem of ignoring the order of test patterns mentioned above in weighted random test-pattern generation. This technique introduces the rates of changes of activity into the function of determining the weight for each PI. Results show that this technique achieves a significant improvement in fault coverage over the static weighted random test-pattern generators. A reduction technique is used to reduce the total number of random patterns generated as random approaches usually create a large number of test patterns. Random pattern techniques offer no guarantees of test coverage/redundancy identification unlike deterministic test pattern generators.

3.4 A Deterministic Sequential Test Generation Algorithm

3.4.1 Introduction

Given a reset state for the sequential machine, two different strategies can be employed to detect faults in the machine.

(1) Forward propagation from the reset state.

(2) A two-stage process of forward propagation and state justification.

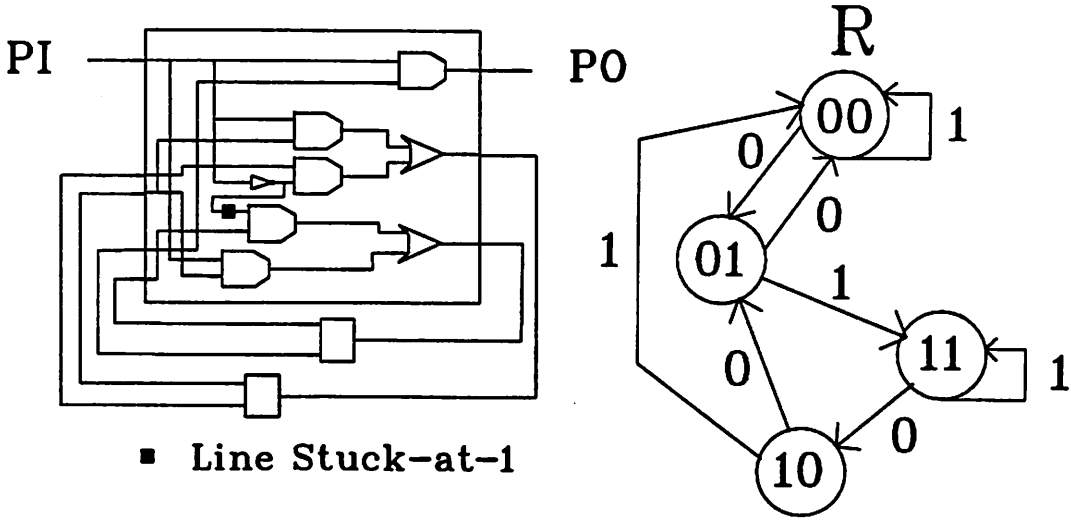
In the iterative array model, (1) corresponds to fixing the present state lines in the first time-frame, PS^1, to the reset state values, and attempting to find n and $PI^1, .., PI^n$ so as to propagate the fault(s) to PO^n.

In (2), the test generation process is decomposed into more tractable subproblems. First, an input sequence, $T1 = (PI^i, PI^{i+1}, .., PI^n)$ and an initial state $S0$ that excite and propagate the effect of the fault to the PO^n is found. This step is called the **fault-excitation-and-propagation** and $T1$ is called the **fault propagation** or simply the **propagation** sequence.

Next, state justification on $S0$ is performed, i.e. a path from R to $S0$ is found, consisting of another sequence of input vectors, $T0 = (PI^1, PI^2, .., PI^{i-1})$. $T0$ is called the **set-up sequence** or the **justification sequence**. The test sequence is the concatenation of the set-up and propagation sequences, $T0$ and $T1$.

It should be noted that both fault propagation and state justification, in general, need a sequence of input vectors. An irredundant fault may be such that its effect can be propagated to the *next state lines alone* in the i-th time frame. Time frames $i+1$ through n are required to propagate the effect of the fault to the primary outputs. Similarly, given an initial state, $S0$, even a minimum-length justification path for $S0$ may require more than one input vector.

In Figures 3.3 & 3.4, these two approaches are illustrated. A sequential circuit with 2 latches is shown in Figure 3(a). A stuck-at fault on a line in the combinational logic is to be detected. The State Transition Graph of the circuit, with the reset state marked R is shown in Figure 3.3(b). In Figure 3.4(a), the test sequence, TS, produced using **Approach 1** is shown

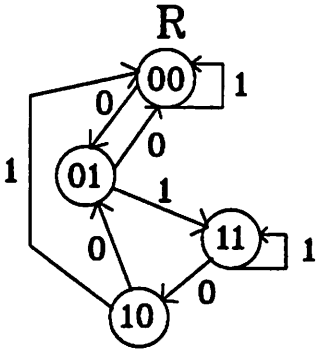


(a) Sequential Machine **(b) State Transition Graph**

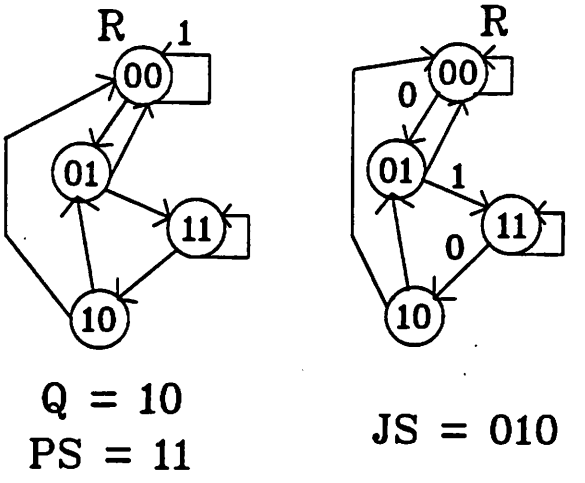
Figure 3.3

(highlighted in the State Transition Graph). In Figure 3.4(b) the propagation sequence, PS, and initial exciting state, Q, using **Approach 2** are shown. The state justification sequence, JS, is shown in Figure 3.4(c).

In this section, a deterministic sequential test generation algorithm, incorporating new techniques for fault propagation and state justification based on extensions to the PODEM



(a) Test Sequence using Approach 1



(b) Propagation Sequence **(c) Justification Sequence**

Figure 3.4

justification algorithm, is presented. In the new algorithm, two different steps of forward fault propagation and state justification, are performed. Information contained in a partial State Transition Graph of the sequential circuit is exploited to facilitate detection of faults requiring long test sequences.

This section is organized as follows. In Section 3.4.2, the overall strategy used in generating test sequences for single stuck-at faults is described. Extraction of the fully or partially connected State Transition Graph from the logic level sequential circuit is described in Section 3.4.3. The fault excitation-and-propagation and state justification algorithms are described in Section 3.4.4 and 3.4.5 respectively. The detection of a special class of redundant faults is described in Section 3.4.6.

3.4.2 The Overall Strategy

A test generation procedure which assumes the existence of a complete State Transition Graph (STG) description of the *fault-free* sequential circuit is first outlined.

Assuming that the complete STG of the fault-free sequential circuit is available, test generation for a fault under test can be done by first finding an input sequence $T1$ and an initial state $S0$ that excite and propagate the effect of the fault to the primary outputs within 4^n time-frames, where n is the number of latches in the sequential circuit. Then, every path from the reset state, R, to any state $S1$ that covers $S0$, a potential setup sequence, in the complete STG is fault simulated. Since the STG corresponds to the fault-free machine, it is not guaranteed that the path from R to $S1$ exists under faulty conditions. If a path $T0$ (setup sequence) to a state $S1$ that covers $S0$ can be found under fault conditions, a test sequence $T2$ is generated by concatenating the path $T0$ with $T1$. Even though a setup sequence $T0$ may not be found, the fault may still be detected by one of the potential setup sequences through fault simulation. If this is the case, that particular potential setup sequence itself can serve as a test sequence $T2$. If no test sequence can be found, a new fault propagation sequence $T1$ and a new initial state $S0$ which is *disjoint* from all previously generated ones is searched and

the procedure is repeated.

The algorithm is complete, i.e. if a fault is testable, a test will be found given sufficient time. The main drawbacks of this method are: (1) the memory storage for the complete STG may be unreasonably large and the generation of the complete STG may demand astronomical CPU time; (2) fault simulation of all potential setup sequences is extremely time consuming. A remedy to (1) is to generate the potential setup sequences on-the-fly using a state justification algorithm that searches for paths from the reset state to the $S0$'s under fault-free conditions. Another alternative to (1) is to perform state justification under faulty conditions so as to ensure that the justification path found is a setup sequence. In either case, no information of the STG is required/used.

A test generation algorithm following the ideas presented above is as follows.

Algorithm Structure 1

(1) Find an (new) fault propagation sequence $T1$ and an (new) initial state $S0$ that will excite and propagate the effect of the fault under test to the primary outputs within 4^n time frames using the fault-excitation-and-propagation algorithm (described in Section 3.4.4). If no solution exists, exit without a test.

(2) Find a (new) path $T0$ (potential setup sequence) from the reset state to the initial state $S0$ using a state justification algorithm in the fault-free machine. If no solution exists, go to (1).

(3) Fault simulate the potential setup sequence $T0$. If it detects the fault, generate the test sequence $T2$ directly from $T0$ and go to (5). Else if it is a valid setup sequence, go to (4). Else if $T0$ neither detects the fault nor is a setup sequence go to (2).

(4) Concatenate the setup sequence $T0$ that represents the justification path from the reset state to the initial state $S0$ with $T1$ to form $T2$ which is the test sequence for the fault under test.

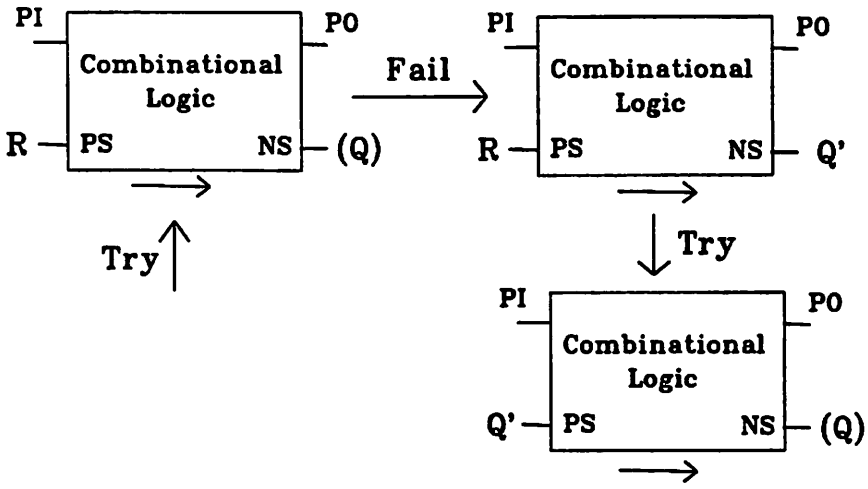
(5) Exit with a test sequence.

Even though this algorithm is potentially effective, state justification in general is difficult when the setup sequence is long. In addition, some states may need to be justified more than once. Enhancements to this basic algorithm are discussed in the next section.

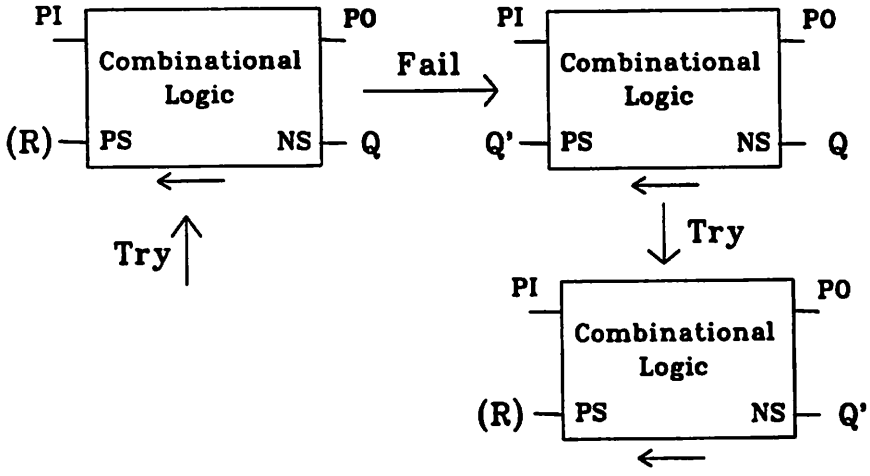
Enhancing the Basic Strategy: As mentioned in Section 3.2.2, state justification can be performed in two different ways, via forward searching or backward searching. These two different search techniques are illustrated in Figure 3.5(a) and Figure 3.5(b). In forward state justification, if a single input vector which places the machine from R to the state to be justified, Q, cannot be found, a state Q', reachable from R via a single input vector is found. Then, a path from Q' to Q is searched for. In backward state justification, if a single input vector which places the machine from R to the state to be justified, Q, cannot be found, a state Q', which reaches Q via a single input vector is found. Then, a path from R to Q' is searched for.

Both forward and backward state justification, in faulty or fault-free machines, can be extremely time consuming as the search space is very large. It is true that, given the justification algorithms used, some states are easier to justify backwards rather than forwards and vice versa.

Thus, an important enhancement to the test generation strategy, which *combines the advantages of forward and backward state justification*, is to generate *a partial STG containing as many valid states* (and paths from the reset state to them) as possible through forward searching/enumeration (as described in Section 3.2) and to use a backward justification algorithm on-the-fly during test generation if the initial state, $S0$, required for a fault does not exist in the partial STG. Note that the partial STG may contain all the valid states in the complete STG but contains much fewer edges. States and edges may be *added* to the partial STG via backward justification during test generation.



(a) Forward State Justification



(b) Backward State Justification

Figure 3.5

The second drawback of Algorithm Structure 1, that the fault simulation of all potential setup sequences is very time consuming, does not actually pose a problem. From the observations in the experiments carried out, if $T0$ is an invalid setup sequence, it is very likely to be a test sequence *by itself*. There is no need to concatenate, $T0$ with $T1$ to produce

a test sequence, $T2$. Therefore, there is rarely the need for fault simulation of more than one potential setup sequence for a fault.

Finally, an efficient test generation algorithm combining the advantages of forward enumeration and backward justification by using the partial STG is as follows.

Algorithm Structure 2

(1) Find an (new) fault propagation sequence $T1$ and an (new) initial state $S0$ that will excite and propagate the effect of the fault under test to the primary outputs within a prescribed number of time frames using the fault-excitation-and-propagation algorithm (described in Section 3.4.4). If no solution exists, exit without a test.

(2) Search for a path (potential setup sequence) $T0$ from the reset state to $S0$ in the partial STG. If it is found, go to (5).

(3) If the partial STG includes all valid states, go to (1).

(4) Find a path $T0$ from the reset state to the initial state $S0$ using the backward state justification algorithm (described in Section 3.4.5). If no solution exists, go to (1).

(5) Fault simulate the potential setup sequence $T0$. If it detects the fault, generate the test sequence $T2$ directly from $T0$ and go to (7). Else if it is a valid setup sequence, continue. Else go to (1).

(6) Concatenate the setup sequence $T0$ that represents the path from the reset state to the initial state $S0$ with $T1$ to form $T2$ which is the test sequence for the fault under test.

(7) Exit with a test sequence.

The initial state $S0$ can be a cube containing don't care bits or a minterm with every state bit specified. In the case of a cube, a path from the reset state to a minterm covered by $S0$ can serve the purpose of a setup sequence.

3.4.3 State Transition Graph Extraction

The input to the logic-level extraction program is the combinational logic specification of the finite state machine as well as information about latch inputs and outputs, i.e. present and next state lines. The output is a partial State Transition Graph (STG) of the finite state machine. A node in the STG represents a distinct minterm state and an edge between two nodes represents an input combination (cube) that drives the finite state machine from one specific state to another.

The STG extraction algorithm first cube-enumerates all fanout edges from the given reset state. Whenever a new edge is found, it is added to the current STG if the next state it fans into does not exist in the STG. Each next state is then picked as a new starting state. The procedure is repeated until no more distinct valid states can be found. Since each state reachable from the reset state is picked as a new starting state and the fanout of each starting state is enumerated, all the edges in the complete STG will be implicitly, but exhaustively enumerated. The partial STG constructed is a tree, i.e. there is only a single path from the reset state to any other state. This is to restrict the storage space for the partial STG so that synchronous sequential machines with a very large number of states can be handled.

The algorithm used to cube-enumerate the fanout edges from a state is an extension to the implicit enumeration algorithm of PODEM [6]. Initially, all the primary inputs and next states of the logic-level finite state machine are set to unknown values. The logic-level circuit is simulated with the present state lines fixed at their specified values. An unknown next state line is then picked and a path is backtraced from it to an unknown primary input with the objective of setting the value of the chosen next state line to a known value. A 1 or 0 is assigned to that primary input. The circuit is then simulated again. The setting of primary inputs and simulation of the circuit is continued until all next state lines are set to known values – a fanout edge is enumerated. Whenever an edge is found, but rejected because it leads to a state already in the partial STG, backtracking is performed on the input cube to

where a primary input was first set to a known value. The opposite value is assigned to it. The simulation and primary input setting is then repeated. When no more backtracking can be done, all the edges from a state are implicitly, but exhaustively enumerated.

The extraction process can proceed in either a depth-first or a breadth-first fashion. In the breadth-first fashion, the path from the reset state to any state in the partial STG is the shortest one. The test sequences generated are shorter but the total number of test sequences is greater than using a depth-first algorithm. There are hard limits, $L1$ and $L2$, for the total number of states to be included in the final STG and the number of states at each level from the given initial state. $L1$ is used to restrict the memory usage and $L2$ restricts the maximum length of the test sequence. The pseudo-code below illustrates the partial STG extraction process in a depth-first fashion. **Extract()** is initially called with the reset state of the sequential circuit and the level equal to 0.

```
Extract(State, level)
{
  PresentState = State;
  PrimaryInput = unknown;
  simulate the circuit;

  while (not all edges have been enumerated) {

    if ((TotalNumStates ≥ L1) |
      (NumStates[level] ≥ L2)) break;

    if (not all NextState lines are set) {
      find_new_pi_assignment();
      simulate with current set of pi assignments;
    }
    else {
      if (NextState is not in the partial STG) {
        add NextState to partial STG;
        TotalNumStates = TotalNumStates + 1;
        NumStates[level] = NumStates + 1;

        Extract(NextState, level + 1);
      }
      else {
        backtrack to the last set primary input
```

```
        and assign an alternative value to it;
        simulate with current set of pi assignments;
      }
    }
  }
}
```

In Figure 3.6, the STG extraction process for the sequential machine of Figure 3.3 is illustrated. Beginning from the reset state, the extraction algorithm proceeds in depth-first fashion, building up the partial STG.

An alternative to the backtracing/backtracking approach to STG enumeration described above is forward simulation on the input space given a starting present state. The forward simulation process begins with all the input lines set to unknown values. Inputs are set randomly to 0 or 1 in a pre-specified order till all the next state lines are all set to known values. Backtracking on primary input values is done after setting all next state lines. However, this

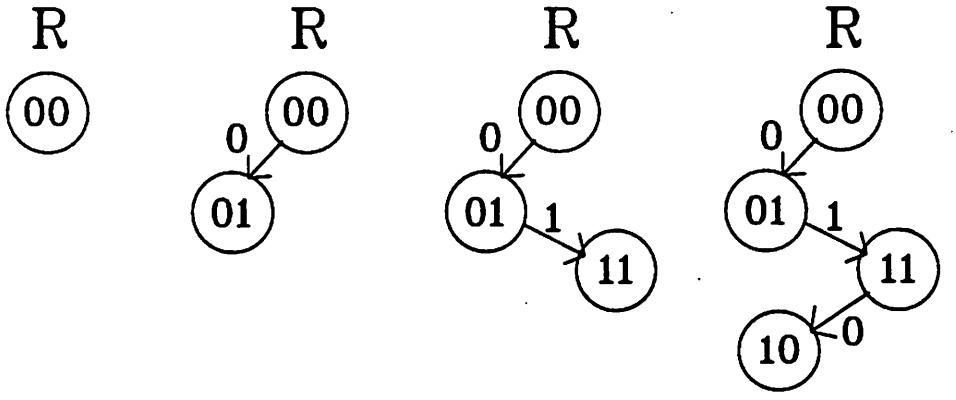


Figure 3.6: STG extraction

approach is less efficient than the approach described earlier because a primary input value may be unnecessarily set in order to set the next state lines. This can lead to a great amount of redundant simulations. On the contrary, in the backtracing/backtracking approach, the backtracing process makes sure that the next primary input to be set and the simulation following the value-setting always contribute to the setting of the next state lines.

3.4.4 The Fault Excitation-and-Propagation Algorithm

The Fault Excitation-and-Propagation algorithm (FEP) is based on the decision tree concept of the test pattern generation algorithm PODEM. It uses 9-valued simulation as opposed to the conventional 5-valued simulation used in PODEM to handle the multiple-fault effect in sequential test generation (a fault is repeated in each time-frame). FEP uses the conventional iterative array model, shown in Figure 3.2, for generating an input propagation sequence $T1$ and an initial state $S0$ to excite and propagate the effect of the fault under test to the primary outputs within a prescribed number of time-frames. The initial state $S0$ produced by FEP is a fault-free state (line values 0, 1 or x). The iterative array is considered wholly as a combinational circuit with primary inputs of different time-frames time-indexed (PI^1, PI^2, etc) and the present state lines of the first time-frame treated as pseudo inputs (PS^1, PS^2, etc). The initial state $S0$ is specified by the pseudo input values. FEP first tries to propagate the fault effect to the primary outputs of the first time-frame. If it fails, it will use the primary outputs of the second time-frame for fault propagation and so on until the prescribed number of time-frames is reached.

If a fault is combinationally redundant, FEP will not be able to propagate the effect of the fault to the primary outputs or the next state lines of the first time-frame. FEP continues to the second and succeeding time-frames only if the effect of the fault has been propagated to the next state lines of the previous time-frame.

FEP uses *two decision trees*, one for the primary inputs of different time-frames and the other for the initial state $S0$, as opposed to only one in PODEM. The two decision trees are

built through backtracing and backtracking processes similar to those used in PODEM. The present state lines of the first time-frame are treated similarly to the primary inputs during the fault excitation-and-propagation process. Values of the present state lines and primary inputs of different time-frames are continuously set one at a time through the backtracing process and the iterative array is simulated whenever a primary input or a pseudo input is set to a known value. The value-setting-and-simulation process continues until the effect of the fault under test is excited and propagated to the primary outputs of at least one of the time-frames or when the backtracking limit is reached. Backtracking takes place whenever it can be established that under the current set of primary input and pseudo input assignments, the effect of the fault under test cannot be excited and/or observed at the primary outputs of the specified time-frame with further input assignments. Backtracking during the search for $T1$ and $S0$ is done on both decision trees.

FEP employs *the concept of disjoint state space enumeration* to make sure that all the tests it generates for a specific fault will have disjoint initial states $S0$; this is necessary because of the loop in the test generation process described in Section 3.4.2. Whenever the search for a new test is begun, the primary input decision tree ($D1$) for the previous test is scratched completely but the present state decision tree ($D2$) of the initial state $S0$ is retained. Immediately, backtracking is done on $D2$. Then, the value-setting-and-simulation process is carried out as described above. The reason that tests generated for a specific fault by FEP should all have disjoint $S0$'s is related to how FEP is used in the test generation process as described in Section 3.4.2. For a specific fault, a new test is requested only if a path from the reset state to the $S0$ in the previous test cannot be found, neither in the extracted STG nor through the state justification algorithm described in Section 3.4.5. Therefore, all tests generated for a specific fault should have disjoint $S0$'s.

A single decision tree could have been used instead of two separated ones as described above. Instead of completely resetting all primary input values to unknown, i.e. scratching the

entire primary input decision tree, when a new search is started, one can simply backtrack on the single decision tree to where a pseudo input (present state line) is first set to a known value and assign it the opposite value. This means that some primary inputs may be preset to 0 or 1 values when beginning the search for a new initial state $S0$ (which is disjoint from the previous one, since one state bit value has been changed to an opposite value). Due to the inherent characteristics of the enumeration approach of PODEM, it is more efficient to begin a search with as small a number of preset primary inputs as possible. Therefore the double decision tree method is used.

3.4.5 The State Justification Algorithm

Given a goal state $S0$, the state justification algorithm (SJ) attempts to find a path (setup sequence) from the reset state to it. $S0$ can be a cube containing don't care state bits or a minterm with every state bit specified. In the case of a cube, SJ needs only to find a path to any minterm state that is covered by $S0$.

SJ performs backward justification from $S0$ to R, given a prescribed limit on the number of backtracks, to bound CPU time usage. First, SJ sets the next state lines to $S0$ and enumerates all the fanin edges to $S0$. SJ then checks to see whether any of the states the edges fanout from cover the reset state or a state in the partial STG. If such a state exists, a path is found. Otherwise, SJ picks each fanin state as a new goal state and carries out fanin edge enumeration again. The procedure is repeated until a path is found or no path can be found. SJ actually proceeds in a depth-first fashion and there is a limit on the maximum length of the justification sequence.

The fanin-edge enumeration algorithm is an extension to the PODEM enumeration algorithm. Here, multiple lines (the next state lines) values have to be justified simultaneously rather than a single output line as in PODEM. The concept of state space enumeration is also employed in SJ. There are two decision trees to be maintained as in Section 3.4.4, i.e. one ($D1$) for the primary inputs and the other ($D2$) for the present state lines. All the present

state lines and primary inputs are set to unknown values initially. Through backtracing and backtracking processes, the primary inputs and present state lines are continuously set to some known values, 1 or 0, until all the next state lines are found to be set to their specified values through simulation. Whenever the search for a new fanin edge is begun, $D1$ is completely scratched but $D2$ is retained. Immediately, backtracking is done on $D2$. Then, the enumeration procedure is repeated again with a new fanout state. All edges (fanning out of disjoint states) fanning into a state have been implicitly enumerated when no more backtracking is possible. The pseudo code below illustrates the state justification algorithm proceeding in depth-first fashion. Breadth-first search is an alternative.

```
Justify_State(State)
{

  PresentStateLines (ps) = unknown;
  PrimaryInputs (pi) = unknown;
  simulate the circuit;

  while (not all fanin states to State are enumerated) {
    while (not all the NextState lines are justified) {

      find_new_pi/ps_assignment();
      simulate circuit with current set of pi/ps assignments;

      if (there are conflicts on NextState line values) {

        backtrack to the last set pi in D1 or ps in D2
                  and assign an alternative value to it;
        simulate with current set of pi and ps assignments;
      }
    }
    if (a fanin state is found) {
      if (fanin state covers reset state or any state in partial STG) {
        a path is found;
        return;
      }
      else Justify_State(fanin state);
    }

    if (a path is not found) {

          /* scratch D1 */
      scratch all pi assignments;
```

```
        backtrack to the last set ps in D2 and
        assign an alternative value to it;
        simulate with current set of ps assignments;
      }
    }
  }
```

3.4.6 Detection Of Redundant Faults

The difficulty in test generation for sequential circuits does not just lie with finding tests for the difficult-to-detect but testable faults. The determination of redundant faults is equally formidable if not more difficult. Obtaining a low fault coverage does not necessary mean that the test generator is inadequate if it can be shown that the fault coverage is close to the maximum achievable value. However, to determine whether faults, that no test has been generated for, are redundant or testable may demand an astronomical amount of CPU time.

As defined in Section 3.2, two classes of redundant faults exist in a sequential circuit. Combinationally redundant faults are detected using the fault-excitation-and-propagation algorithm, FEP. In general, they are easier to find than sequentially redundant faults.

For the purpose of judging how close the fault coverage obtained by the new sequential test generator is to the maximum possible value, all sequentially redundant faults based on Theorem 3.1 given below are found and other undetected faults are treated as possibly testable faults. This gives a lower bound on the number of redundant faults in a given circuit.

Definition 3.1: An edge in the State Transition Graph is said to be *corrupted* by a stuck-at fault if the effect of the fault can be excited and propagated to the primary outputs and/or next state lines by the input vector corresponding to the edge with the present state lines values set to the fanin state of the edge.

Theorem 3.1: In order for a stuck-at fault to be detected, the fault should at least corrupt one fanout edge from a valid state that is reachable from the reset state in the state transition

graph.

Proof: In order to detect a fault a test sequence starting from the reset state and ending with a corrupted edge in the STG is needed. If a fault does not corrupt any fanout edge *from a valid* state in the STG, no test sequence can detect the fault since no corrupted edge can be reached from the reset state. □

Determining this special class of redundant faults requires the extraction of a partial STG containing all valid states reachable from the reset state. The procedure to find these redundant faults is based on the FEP algorithm described in Section 3.4.4. A single time-frame is used and *all next state lines are treated as primary outputs.* (During test generation, the fault effect has to be propagated to the primary outputs alone, and therefore FEP may require multiple time-frames). It should be noted that FEP attempts to find a fault-free state and input vector which detects the given fault. All tests are generated, for a potential redundant fault, with disjoint initial states. If none of the initial states covers any of the states in the partial STG, the fault under test is redundant.

3.5 Sequential Test Generation Results

The test generation algorithms described in the previous section have been implemented in the program STALLION [26]. STALLION consists of about 10,000 lines of *C* code and runs in a VAX-UNIX environment.

Results and time profiles using STALLION for eight finite state machines which are described in Table 3.1 are given in Table 3.2 and 3.3 respectively. In the tables *m* and *s* stand for minutes and seconds respectively. For each example in Table 3.1, the number of inputs (#inp), number of outputs (#out), number of gates (#gate), number of latches (#lat), and the number of equivalent faults (#eqv. faults) are indicated. In Table 3.2, the number of test sequences (#test seq.), total number of test vectors in all the test sequences (#vect), maximum test sequence length (max. seq. len.), fault coverage, percentage of provably redundant faults

(using Theorem 1), total fault coverage including detected and provably redundant faults (tfc), and CPU time on a VAX 11/8800 are indicated for each example. CPU times for extracting the partial State Transition Graph, test sequence generation, fault simulation, miscellaneous setup and for the entire test generation process are given in Table 3.3. In Table 3.2 and Table 3.3, *m* stands for minutes and *s* for seconds.

As can be seen the new test generation technique obtains close to the maximum possible fault coverage in all the examples. The extraction of the partial STG consumes a relatively small amount of CPU time with respect to the total TPG time in all cases. Fault simulation constitutes a large percentage of total TPG time in most cases except in sse, as can be seen in Table 3.3. The fault simulator uses a parallel-fault event-driven technique and a more sophisticated one using concurrent techniques will significantly speed up the test generation process. The reason that test generation time is the dominant constituent in the total CPU time in sse is because a great amount of time is consumed in trying to find tests for the large number of redundant faults.

The first five examples are finite state machines obtained from various industrial sources. The example sbc is the snooping bus controller [46] in the SPUR chip set. It was

CKT	#inp	#out	#gate	#lat	#eqv. faults
cse	7	7	192	4	680
sse	7	7	130	6	486
planet	7	19	606	6	2028
sand	9	6	555	6	1932
scf	27	54	959	8	3338
mult4	9	9	170	15	506
sbc	40	56	1011	28	3008
stage	131	64	2700	64	9155

Table 3.1: Statistics for 8 example circuits

CKT	#test seq.	#vec	max. seq. len.	fault cov. (%)	red.* fault (%)	tfc§ (%)	CPU† time
cse	96	472	8	99.71	0.29	100.0	53.2s
sse	46	284	10	84.57	15.23	99.8	69.9s
planet	80	1191	26	97.39	2.56	99.95	12.6m
sand	165	1077	24	94.36	5.18	99.54	22.4m
scf	136	2238	21	94.37	3.86	98.23	83.0m
mult4	17	120	24	96.25	2.96	99.21	40.6s
sbc	168	1063	24	95.68	2.66	98.34	62.1m
stage	139	425	26	93.97	6.03	100.0	154m

* percentage of provably redundant faults
§ total fault coverage including detected and provably redundant faults
† All times are obtained on a VAX 11/8800

Table 3.2: Results for circuits

CKT	STG Extraction	Test Generation	Fault Simulation	Miscell.	Total
cse	0.9s	8.3s	43.8s	0.2s	53.2s
sse	0.4s	52.2s	17.1s	0.2s	69.9s
planet	3.2s	1.2m	11.4m	0.7s	12.6m
sand	4.6s	10.7m	11.6m	0.6s	22.4m
scf	13.9s	11.5m	71.2m	1.2s	83.0m
mult4	27.5s	9.7s	6.5s	0.2s	43.9s
sbc	12.4m	28.3m	21.4m	1.3s	62.1m
stage	10.1m	50.8m	94.1m	1.0m	154m

Table 3.3: Time profiles for example circuits

synthesized using the multiple level logic optimization system MIS [3].

So far, it has not been able to perform direct comparisons with other deterministic sequential test generation systems (Any sequential test generation program which is publicly available cannot be obtained). However, in [47], performance figures were quoted for the D-

Algorithm-based test generation system in [44], STG, for some publicly available benchmarks. Figures from [47] are reproduced here along with STALLION's results for the same examples in Table 3.4. The CPU times are in seconds on a VAX 11/8650. In all aspects and for all the examples STALLION's results are vastly superior to STG.

STG does not have an interactively running fault simulator. Thus the fault list is not updated and the test generator produces tests for each fault separately. Also, for each test, a new initialization is attempted. These two factors increase the run time for STG.

The example **planet** has faults that require long test sequences (Table 3.2). Because of this, the differences between the performances of STG and STALLION are more marked in **planet** than in the other two examples.

3.6 An Incomplete Scan Design Approach

3.6.1 Introduction

In the previous sections, a sequential testing algorithm effective for mid-sized sequential finite state machines is presented. In this section, a new Incomplete Scan Design approach to test generation for large sequential circuits is presented. First, using the efficient sequential testing algorithm, STALLION, test sequences are generated for a large number of possible faults in

EXAMPLE	STALLION			STG		
	tfc§	#vect.	CPU time	tfc§	#vect.	CPU time
sse	99.80	284	69.9	99.50	676	1134
mult4	99.21	120	40.6	93.15	148	1490
planet	99.95	1191	754.0	61.00	132	19388

§ total fault coverage including detected and provably redundant faults

Table 3.4: Comparison with STG [44]

the given circuit. A **minimal subset of memory elements** is then found, which if made observable and controllable will result in all remaining irredundant but previously difficult-to-detect faults being easily detected in the modified circuit by STALLION. Detection of all irredundant faults as in the Complete Scan Design case can be guaranteed, but at significantly less area and performance cost. The length of the test sequences for the faults can be bounded by a prescribed value – in general, a trade-off exists between the number of memory elements required to made scannable and the maximum allowed length of the test sequence.

Related work in this area has involved using functional vector sets to initially detect faults in the sequential circuit [48]. After functional vector fault simulation, a set of memory elements is made scannable so as to ensure that test sequences each consisting of a *single* test vector can detect the remaining undetected faults. The new approach described is not restricted to using single vector test sequences to detect faults. A minimal set of critical memory elements is found which when made scannable allows the sequential test generation algorithm to detect the faults via *multiple* vector test sequences.

This section will focus on the algorithm used in the identification of the minimal subset of memory elements. The overall strategy used in the identification of the critical memory elements is described in Section 3.6.2. Heuristic selection algorithms used in the last stage of the identification process are described in Section 3.6.3.

3.6.2 The Global Strategy

The overall structure of the Incomplete Scan Design algorithm is shown below. The algorithm incorporates the sequential test generation algorithm, STALLION, described in Section 3.4. It is given the sequential circuit S and a set of faults to be detected, F. It produces a set of test sequences, T, each beginning from the reset state of the machine, R, and identifies a set of memory elements, M, to be made scannable. The T are such that they detect the faults F in S^M, the sequential circuit, S, with the memory elements, M, made observable and controllable.

Incomplete Scan Design Algorithm

(1) Given the sequential circuit S, for each fault $f \in F$, attempt to find a fault-excitation-and-propagation sequence, P_f, which will propagate the effect of f to the primary outputs if possible else to the next state lines. The length of P_f is limited to *MAX_PROP_LEN*. If P_f does not propagate f to the primary outputs, a set of next state lines, NS_f, to which the effect of f can be propagated to, is found. The set of faults which can be propagated only to next state lines is called F^{NS}.

(2) All distinct state vectors in $P_f{}^0$, the first vector in the sequence P^f are found. Call this set of distinct state vectors, K.

(3) Generate *MAX_STATE* states, Q_i, at different levels, i, from R, in the State Transition Graph (STG) of S. i varies from 1 to *MAX_LEVEL*.

(4) For each $k \in K$, find the memory lines to made scannable such that each state $q \in Q_i$ covers k. For each k generate *MAX_CHOICE* best (with the least number of lines) choices for line sets, which if made scannable will result in the covering of k by $q \in Q_i$.

(5) Given K and *MAX_CHOICE* choices of line sets for each $k \in K$, and a line set NS_f for each $f \in F^{NS}$, select a line set for each k and a single line from each NS_f line set so the number of distinct lines to be made scannable, M, is minimized.

In **Step 1**, propagation sequences are found for faults in the sequential circuit, S, using the sequential test generation algorithm, STALLION. While running STALLION, the present state lines of S are set last so the state vectors K, found in **Step 2**, have as many don't care bits in them as possible. STALLION produces a starting state, $S0$, and an input vector sequence, I, which propagate the effect of the fault to either the primary outputs or the next state lines of S. If the effect of the fault is propagated to the primary outputs, then it only remains to justify $S0$, else the complete set of next state lines to which the effect of the fault

can be propagated, NS_f, has to be found as well.

In **Step 3** a large set of states in the State Transition Graph of S, and paths from the reset state leading to each state are found. These states are extracted using the STG extraction algorithm described in Section 3.4.3. The number of states and the lengths of the justification paths are bounded by *MAX_STATES* and *MAX_LEVEL* respectively.

The **distance** between two arbitrary bit vectors of length N, $A(i)$ and $B(i)$, where each bit can take the values of 0, 1 and x (don't care) is defined as the number of bits where $A(i)$ is 1 and $B(i)$ is 0 or *vice versa* over $i = 1, .. N$. Given a state $q \in Q_i$ and a state $k \in K$, the distance between q and k then gives the number of state line values that q and k differ in. It is easy to see that if the state lines which are different between q and k are made controllable, any justification sequence for q has to work for k. Don't care bits in k do not contribute to distance since they can take the values of 0 or 1. So to minimize distance between the states in K and Q_i, the number of don't care bits in K is maximized by setting the state lines last in **Step 1.**

In **Step 4**, for each pair of q and k, the set of lines which are to be made scannable for the justification sequence of q to be usable for k is found. Then, for each k, *MAX_CHOICE* such line sets with the least number of lines and secondarily the smallest justification sequence length are selected.

Given these *MAX_CHOICE* sets of lines for each k, a heuristic algorithm (**Step 5**) is used to select one particular line set, (corresponding to one particular $q \in Q_i$) for each k, so as to minimize the total number of distinct lines in all the line sets selected. Simultaneously, for each fault, f, which could not be propagated to the primary outputs, a line from NS_f (the set of next state lines to which f can be propagated) is selected. Two selection algorithms which have been employed are described in the next section.

After the heuristic selection process, a minimal number of state lines to be made scannable and justification sequences for K are identified. A set of test sequences, T, is formed by

concatenating the justification sequences, J, with the propagation sequences, P, generated using STALLION. T will detect all undetected faults in S^M, namely, F.

3.6.3 Heuristic Algorithms For Selection

The subproblem to be solved is as follows. There are N elements (each corresponding to a fault), each with a set of line groups. The number of line groups may vary. It is assumed that for any given element, none of the line groups is a superset or subset of any other line group of the same element. The goal is to identify a line group for each element such that the number of distinct lines in the selected line groups is minimum. Some of the elements may have a set of line groups each containing a single line. For example, during test generation, faults that can only be propagated to next state lines will result in an element with the property mentioned above.

Two heuristic algorithms which produce minimal solutions are described below. **Algorithm 1** is a very fast greedy algorithm run with different starting points. **Algorithm 2** first finds all lines which are definitely required, if any. For example, if for any element, a line exists in all its choices, that line is definitely required. Then, N_e best elements and line groups in these elements are picked at a time. The complexity of this algorithm is O(N^{N_e}). The larger the N_e is, the better the solution can potentially be ($N_e = N$ is exhaustive search) but more CPU time is required. It have been found that $N_e = 3$ gives near-optimal results within acceptable run times.

Algorithm 1

```
for( i = 1;  i ≤ N;  i++ ) {
   Element  =  e [i];
   for ( j = 1;  j ≤ Element.NumChoices;  j++ ) {
      Solution  =  greedy( Element.Choice[j], Element ) ;
   }
}
select best Solution ;


greedy( lineGroup, element ) {

  Lines  =  Lines ∪ lineGroup ;
  for ( i = 1;  i ≤ N;  i++ ) {
     if ( e [i]  ≠  element ) {
        for ( j = 1;  j ≤ e [i].NumChoices;  j++ ) {
           card  =  |Lines ∪ e [i].Choice[j] |
        }
        pick k so card is minimum ;
        Lines  =  Lines ∪ e [i].Choices[k] ;
     }
  }
  return( Lines ) ;
}
```

Algorithm 2

```
find all required lines, Lines ;

while ( choices to be made ) {

   pick N_e elements, e_1, .. e_Ne and choices
   for the elements, c_1, .. c_Ne minimizing
   |Lines ∪ e_i[c_i]|   i = 1, N_e

   mark choices made for e_i
}
```

3.7 Results Using Incomplete Scan Design

In this section, results obtained on several circuits using STALLION in conjunction with an Incomplete Scan Design methodology are presented. Results obtained on seven circuits are summarized in Table 3.5. In the table, the number of inputs (#inp), outputs (#out), gates (#gate), and latches (#lat) in each circuit is indicated. The percentage of combinationally redundant faults (which cannot be detected even using Complete Scan Design), initial fault coverage achieved by STALLION, the number of latches made scannable, the final fault coverage in the modified circuit and the CPU times used for selection on a VAX 11/8650 are also given. The CPU times for sequential test generation varied between a minute for the smaller examples to an hour for the largest example sbc on a VAX 11/8650 (Table 3.2). It should be noted that the CPU time used for sequential test generation can be bounded by limiting the number of backtracks allowed. However, if this is done, quite possibly fewer faults will be detected and more scan latches may be required.

EXAMPLE	#inp	#out	#gate	#lat	red. fault %	initial fault cov.	scan latches	final fault cov.	CPU time (secs)
sse	7	7	130	6	0.0	84.57	3	100.0	4.9
sand	9	6	555	6	0.21	94.31	2	99.79	5.3
scf	27	54	959	8	0.51	94.67	2	99.49	1.8
donfile	2	1	232	12	0.0	63.60	5 9	96.34 100.0	11.1 14.3
sckt4	3	6	160	21	1.09	26.55	6 9	92.93 98.91	2.1 2.4
sbc	35	51	1011	33	2.83	81.25	5 10	95.68 97.17	9.1 17.1
lex1	27	52	395	97	0.0	75.86	39	97.94	6.3

Table 3.5: Incomplete Scan Design Results

As can be seen, making a small subset of latches scannable increases the fault coverage obtained to the maximum possible value or very close to the maximum possible value for all the circuits. For instance in example **sckt4**, making 9 out of 21 latches scannable raised the fault coverage from 26.25% to the maximum possible value of 98.91% (1.09% of the faults are combinationally redundant). Depending on the fault coverage required, differing numbers of scan latches suffice. Trade-offs can be made for each example as indicated in Table 3.5.

The examples **donfile** and **sckt4** originally have a large number of sequentially redundant faults, which is why the fault coverage obtained by STALLION for these two examples is low. Making some memory elements scannable allows detection of these faults increasing the fault coverage to the maximum possible value.

The results demonstrate the advantages in using a combined approach of sequential test generation and Incomplete Scan Design. A large percentage of faults are detected using the efficient sequential test generation algorithm and the remaining irredundant faults are detected by the same algorithm after making a minimal subset of flip-flops observable and controllable.

3.8 Conclusions

A novel approach to test generation for synchronous sequential finite state machines has been presented in this paper. An efficient deterministic test generation algorithm for sequential circuits have been developed. The efficacy of the new method stems from the integration of several new algorithms that are based on the concept of state space enumeration. The new approach involves extracting a partial State Transition Graph and using it in conjunction with fault excitation-and-propagation and state justification algorithms in generating tests. The problem of generating tests for faults that require a lengthy input sequence is shown to be handled efficiently by the new method through the intelligent use of the path information contained in the STG and the coordinated interaction of the various algorithms. Tests have been successfully generated for finite state machines with a large number of states using reasonable amounts of CPU time and close to maximum possible fault coverages are obtained. A new

method of detecting a special class of redundant faults in determining how close the fault coverage obtained to the maximum possible value has also been described.

For very large sequential circuits, an Incomplete Scan Design approach to test generation have been developed. In this approach, the test generation algorithm is first used to generate tests for a large subset of faults in the circuit. A minimal set of critical memory elements is then found, which if made observable and controllable will result in easy detection of all remaining irredundant but difficult-to-detect faults. The deterministic test generation algorithm is again used to generate tests for these faults in the modified circuit (the circuit with the identified memory elements made scannable). Detection of all irredundant faults as in the Complete Scan Design case can be guaranteeds, but at significantly less area and performance cost.

CHAPTER 4

Mixed-Level Fault Coverage Estimation

4.1 Introduction

Fault simulation is an essential process for VLSI design and serves as an important part of the test-generation process. It is used to determine the fault coverage of a given test (test pattern or test sequence), that is, to find all the faults detected by the test. There are three typical methods of fault simulation: parallel, deductive [18], and concurrent [19] fault simulation and the last method has been shown to be most effective. The computational cost of fault simulation is usually very high. It is known that [20] the CPU time and memory requirements for fault simulation are proportional to the square of the number of gates in the circuit. This poses a serious limitation to its use for evaluating test patterns for VLSI circuits.

Recently, Jain and Agrawal [21] proposed a statistical fault analysis technique, called STAFAN, which is able to produce an accurate test coverage projection given a set of test vectors. Only fault-free simulation results are required for the fault coverage calculation and the computational complexity is greatly reduced. Presently, this method can only be used for circuits modeled with basic Boolean gates. In many cases, however, it is desirable and advantageous to be able to apply the technique of STAFAN to mixed-level circuits consisting of MOS transistors, logic gates and combinational functional blocks. First, this will provide the designers the capability to zoom-in at a particular portion of the design. Secondly, it will greatly enhance the usefulness of the method in the early phase of the design when some parts of the circuit are only functionally known. Thirdly, test coverage for transistor stuck-open and stuck-closed faults in CMOS circuit can also be estimated for a given set of test vectors.

In this chapter, algorithms for estimation of fault coverages for mixed-level combinational circuits are presented [22]. The concept of STAFAN is reviewed in Section 4.2. The observability formulae developed for combinational Multiple-Input Multiple-Output (MIMO) functional blocks is described in Section 4.3. A STAFAN-like technique applied to CMOS transistor circuits for estimating transistor fault detection probabilities is described in Section 4.4. A mixed-level FAult Coverage Estimation tool (FACE) is described in Section 4.5. Results obtained for some circuits to evaluate the efficiency of the estimation tool are presented in Section 4.6.

4.2 Preliminaries

STAFAN computes a detection probability estimate for each line fault in a logic circuit based on the concepts of controllability and observability. These two measures are defined as probabilities of controlling and observing the line. Fault-free simulation results are used to compute estimates of controllability and observability. The detection probability estimate of a fault is given by the product of the appropriate observability and controllability. Fault coverage is estimated from the computed detection probabilities.

Each line l in a logic circuit is associated with four quantities, one-controllability $C1(l)$, zero-controllability $C0(l)$, one-observability $B1(l)$, and zero-observability $B0(l)$. $C1(l)$ and $C0(l)$ are estimated as

$$C1(l) = \frac{one-count}{N} ;$$

$$C0(l) = \frac{zero-count}{N} ;$$

where

N = number of vectors simulated ;

$one-count$ = number of times l is set to 1 ;

$zero-count$ = number of times l is set to 0 ;

$B1(l)(B0(l))$ is the probability of observing line l at a primary output when the value of l is 1(0). Observability is computed by the following procedures : i) set all primary output lines $B1$ and $B0$ to 1 ; ii) propagate observabilities through logic cells based on observability formulae. The probability of detecting line l stuck-at-1(s-a-1) is estimated as

$$D1(l) = B0(l) \times C0(l) ;$$

And the probability of detecting line l stuck-at-0(s-a-0) is estimated as

$$D0(l) = B1(l) \times C1(l) ;$$

Special procedures are used to unbias the detection probability estimates. Certain assumptions are made in computing observabilities for lines with fan-out branches to reduce the amount of computation time. The estimated fault coverage is the sum of all the estimated detection probabilities divided by the total number of faults. Results for practical circuits show that in most cases STAFAN fault coverage estimates are very close to exact fault coverage, even though the estimates for any single line may be inaccurate [49].

4.3 Observability Propagation Formulae

The computation of controllabilities for input and output lines of a functional block is performed as described in Sections 4.2. The formulae that allow backward propagation of observabilities from the outputs of a MIMO functional block to its inputs are described here [22].

For the sake of simplicity, first consider a single-output logic block as shown in Figure 4.1. Line l is one of the inputs and line m is the output line. The value of line l can be observed through the output either as a 0 or a 1 whenever the line is sensitized. This means the observabilities of line l are dependent on the one-observability and the zero-observability of output m and the sensitization of line l. For the one-observability $B1(l)$ of line l, this can be translated mathematically into the following formula:

$$B1(l) = B1(m) \times P(m=1/l=1,S1(l)) \times P(S1(l)/l=1)$$
$$+ B0(m) \times P(m=0/l=1,S1(l)) \times P(S1(l)/l=1) ;$$

where

$P(m=1/l=1,S1(l))$ = conditional probability that output m equal to 1 when line l is equal to 1 and being sensitized ;

$P(m=0/l=1,S1(l))$ = conditional probability that output m equal to 0 when line l is equal to 1 and being sensitized ;

$P(S1(l)/l=1)$ = conditional probability that line l is being sensitized when it is equal to 1 ;

the formula can be simplified to

$$B1(l) = [B1(m) \times P(m=1,l=1,S1(l)) + B0(m) \times P(m=0,l=1,S1(1))] / C1(l)$$

where

$C1(l)$ = one-controllability of line l ;

The computation of the P's terms in the observability formula is demonstrated by explaining how the value of $P(m=1,l=1,S1(l))$ is obtained. A counter $SM(1,1)$ is kept for the input line for the computation of $P(m=1,l=1,S1(l))$. The counter is incremented whenever the output m is equal to 1 and l is equal to 1 and being sensitized. Sensitization condition is

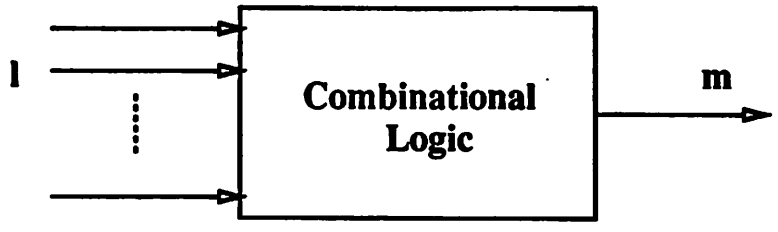


Fig. 4.1 Multiple-Input-Single-Output Block

checked by changing line l to its opposite value and evaluating the functional block again. If the output is changed to 0, line l is being sensitized. The value of $P(m=1,l=1,S1(l))$ is the value of counter $SM(1,1)$ divided by the number of vectors N being simulated ,i.e.

$$P(m=1,l=1,S1(l)) = \frac{SM(1,1)}{N} ;$$

The value of $P(m=0,l=1,S1(l))$ can be similarly obtained by keeping another counter $SM(0,1)$ and its value is given by

$$P(m=0,l=1,S1(l)) = \frac{SM(0,1)}{N} ;$$

Zero-observability of line l can be similarly derived as

$$B0(l) = [B1(m) \times P(m=1,l=0,S0(l) + B0(m) \times P(m=0,l=0,S0(1))] / C0(l)$$

;

where

$C0(l)$ = zero-controllability of line l ;

For a MIMO block with n output lines, the input line l can be observed through any output. If $B1(l,m)$ is the probability of observing $l=1$ through output m, the total one-observability of line l is

$$B1(l) = \text{Prob}\{l=1 \text{ is observed through one of the outputs}\}$$

$$= S1 - S2 + S3 - S4 + ... + (-1)^{n-1}SN$$

where

$$S1 = \sum_{j=1}^{n} B1(l,j) ;$$

$S2 = \sum \text{Prob}\{l=1$ is observed through output i and $j\}$;

$S3 = \sum \text{Prob}\{l=1$ is observed through output i, j and $k\}$;

.

.

.

$SN = \text{Prob}\{l=1$ is observed through all outputs$\}$;

The exact computation of the observability formula would require a detailed analysis of the topological structure of the circuit. The computational cost could be very high. The computation would be greatly simplified if the assumption that all output lines have observation paths independent of others is made. The formula is reduced to

$$B1(l) = \bigcup_{j=1}^{n} B1(l,j);$$

This is analogous to the computation of observability for a line with several fanout branches for which the assumption that all fanout branches have independent observation paths is proved experimentally feasible [50].

Similarly, the zero-observability can be derived as

$$B0(l) = \bigcup_{j=1}^{n} B0(l,j);$$

The above formulae can be greatly simplified for primitive logic gates based on their logic properties. For example, the $P(m=1,l=0,S0(l))$ term and the $P(m=0,l=1,S1(l))$ term are both equal to zero for an AND gate. The simplified formulae for logic gates are equivalent to those derived in [21].

4.4 Computation Of Transistor Fault Detection Probability

The fault model used for CMOS transistor gates consists of transistor stuck-open and stuck-short faults. Most prevalent physical failures are representable by this fault model [51]. It is also assumed that the path from ground dominates the path from Vdd in a CMOS gate, i.e., a Vdd to GND path caused by a stuck-short fault in a CMOS gate will force the gate output to a logic zero. Hence, PMOS transistor stuck-short faults are treated as undetectable. In order to apply the principle of STAFAN at transistor level, each CMOS transistor gate is replaced by an equivalent logic model. A simple example is used to illustrate how transistor fault detection probabilities can be computed based on the logic model.

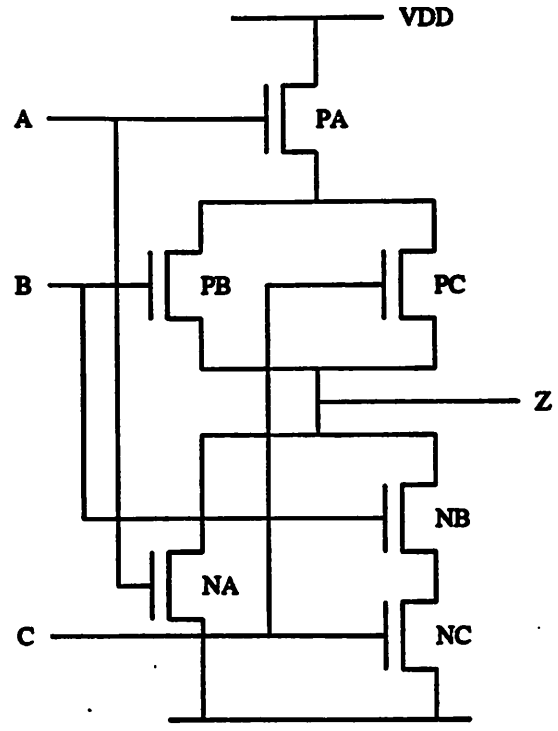


Fig. 4.2 A 3-Input CMOS Gate

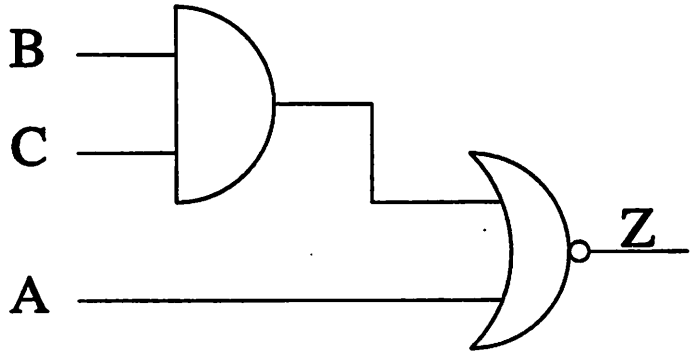


Fig. 4.3 Logic Model of the CMOS Gate

A simple CMOS gate is shown in Figure 4.2. The equivalent logic model is shown in

Figure 4.3. This model can be used in simulation of the mixed-level circuit and detection probabilities of each line stuck-at fault in the logic model will be estimated using formulae described in Sections 4.2 and 4.3. This information is used to compute all transistor fault detection probabilities as described below.

4.4.1 NMOS Transistor Stuck-short Faults

The NMOS transistor stuck-short faults correspond to gate input line s-a-1 faults in the logic model [52]. For example, the stuck-short fault of transistor *NA* in Figure 4.2 corresponds to the input line *A* s-a-1 fault in Figure 4.3. The detection probability for transistor *NA* stuck-short is equal to the probability of detecting the input line *A* s-a-1, i.e.,

$$DS(NA) = D1(A);$$

Similarly, all NMOS transistor stuck-short detection probabilities can be obtained.

4.4.2 NMOS And PMOS Transistor Stuck-open Faults

The detection of transistor stuck-open faults requires two-pattern tests [53]. In order to detect a NMOS transistor stuck-open in a CMOS gate by a test vector, one of the necessary conditions is that the previous gate output is set to one. Similarly, to detect a PMOS transistor stuck-open fault, the previous gate output should be 0. The dependence of detecting transistor stuck-open fault on the memory state of the gate output will be reflected in the stuck-open fault detection probability formula given below.

Two special counters *SN* and *SP* are kept during the simulation for computing the probabilities of detecting transistor stuck-open faults in a CMOS gate. *SN* is incremented on a vector only if the CMOS gate previous output is equal to 1 and the present output is equal to 0. *SP* is incremented if the previous output is equal to 0 and present output is equal to 1. Again using the same example CMOS gate, it will be shown here how the detection probabilities of transistor stuck-open faults can be obtained. The stuck-open fault of transistor *NA* corresponds to the input line *A* s-a-0 fault. The probability of detecting *NA* stuck-open is

given by

$$DO(NA) = D0(A) \times \frac{SN}{N \times C0(Z)} ;$$

where

$D0(A)$ = probability of detecting line A s-a-0 ;

SN = value of counter SN ;

N = number of test vectors ;

$C0(Z)$ = zero-controllability of the gate output ;

$\frac{SN}{N \times C0(Z)}$ = Prob{previous output = 1/present output = 0} ;

The stuck-open fault of transistor PA corresponds to the input line A s-a-1 fault. The probability of detecting NA stuck-open is given by

$$DO(PA) = D1(A) \times \frac{SP}{N \times C1(Z)} ;$$

where

$D1(A)$ = probability of detecting line A s-a-1 ;

SP = value of counter SP ;

N = number of test vectors ;

$C1(Z)$ = one-controllability of the gate output ;

$\frac{SP}{N \times C1(Z)}$ = Prob{previous output = 0/present output = 1} ;

$\frac{SN}{N \times C0(Z)}$ and $\frac{SP}{N \times C1(Z)}$ account for the dependence of detecting stuck-open fault on CMOS gate memory state.

The above formulae allow using a single logic model to estimate both transistor stuck-open and stuck-short faults. Observabilities could also be effectively propagated through CMOS transistor gates with the logic models.

4.4.3 Unbiasing Transistor Fault Detection Probability

As shown in [2], fault detection probabilities obtained from the simulation of N vectors are biased estimates containing random errors. Unbiasing requires modifying the detection probability as

$$D = 1 - \frac{(1-D)^N}{W(D)} ;$$

where

D = detection probability estimate ;

$$W(D) = 1 + (N-1) \times \frac{\beta^2}{6} \times \frac{D}{(1-D)} ;$$

The value of β is determined experimentally. By matching estimated fault coverage with fault simulation results for an actual circuit, the value of $\frac{\beta^2}{6} = 5$ is found to experimentally produce good estimates for other gate-level circuits [2]. However, it was found that this β value is unsuitable for CMOS transistor circuits where unconventional stuck-open faults are present. Estimated fault coverage tends to be over-optimistic. A more appropriate value, after experimentation with several circuits, is found to be $\frac{\beta^2}{6} = 3$.

4.4 Implementation

The mixed-level fault coverage estimator is implemented in the program FACE [22]. It accepts input files consisting of primitive logic gates, functional blocks and CMOS transistor gates. The main parts of the program are a function extractor, a gate-level modeler, a logic simulator and a fault coverage estimator. Various components of the tool are described below.

4.5.1 Function Extractor and Gate-level Modeler

These two programs map all the CMOS transistor gates into equivalent logic models. The function extractor first builds up a connectivity graph representing the transistor circuit. It then identifies all the gate nodes and generates the logic expressions [54] [55]. A tree-like data structure similar to those mentioned in [56] is used to store the logic expressions. The gate-level modeler uses the logic expression information to generate equivalent logic models for all the CMOS gate. These logic models are attached to the existing logic subcircuit to build up an equivalent Boolean network. This equivalent network is used by the logic simulator and the fault coverage estimator to calculate detection probabilities.

4.5.2 Logic Simulation

The logic simulator is a three-valued simulator programmed to simulate logic circuits made up of Boolean gates and functional blocks. It employs both event-driven and circuit leveling techniques. Functional block evaluation is much more costly in terms of CPU time compared to gate evaluation. A purely event-driven simulator will not be suitable: a functional block which has several inputs changed during the simulation may get evaluated several times due to different delays in signal paths. With the leveling technique, the functional block will be evaluated only after all its inputs have changed. The simulation algorithm is described as below :

It is important that the representation of the functional blocks allows fast evaluation. As noted in Section 4.3, the computation of observability for MIMO block inputs involves many functional block evaluations. There are many ways to describe/represent a functional block in logic simulation. The truth table is the most straightforward one that allows fastest evaluation using indexing technique. The main drawback of this representation is the memory requirement for storing the table : it is proportional to 2^n where n is the number of inputs to the functional block. Here the ON-set cover representation [2] of the functional block is used. This is a good compromise between memory requirement and evaluation speed. This table is

```
Logic Simulation Algorithm:
{
  present_level = 0;
  highest_level = max_level;
  while (cell-list is not empty) {
    if (cell_level is equal to present_level) {
      evaluate the cell;
      for (each output line of the cell) {
        if (value of line has changed) {
          schedule cells driven by this line ;
        }
      }
      remove cell from the list;
    }
    else {
      put cell back into the list;
    }
    present_level = present_level + 1;
    if (present_level > highest_level) {
      find new highest_level and lowest_level;
      present_level = lowest_level;
    }
  }
}
```

usually available for PLA-implementation block during the synthesis phase. Other logical descriptions of a functional block, i.e. Boolean expressions and truth table, can be readily transformed into this representation using CAD tools such as ESPRESSO II-MV and EQNTOTT developed in University of California at Berkeley. The ON-set cover representation is a compact form of describing a full truth table. A matrix representation of ON-set cover for a 4-to-1 multiplexer is shown in Figure 4.4. $S0$ and $S1$ are the selection inputs, D_i's are the data inputs and Z is the output.

The evaluation of the functional block is performed by comparing the input vector with the input part(cube) of each row of the matrix and setting the output according to the output part(cube).

S0	S1	D1	D2	D3	D4	Z
0	0	1	-	-	-	1
0	1	-	1	-	-	1
1	0	-	-	1	-	1
1	1	-	-	-	1	1

Fig. 4.4 ON-set cover table for a 4-to-1 multiplexer

4.5.3 Fault Coverage Estimator

This program functions to update all the counters after simulation of each vector and compute the controllabilities, observabilities and detection probabilities for all the logic faults and transistor faults.

4.6 Results

Fault coverage estimation results were obtained for three CMOS circuits and a 8-bit multiplier circuit. Results are summarized in Table 4.1 and Table 4.2. Circuit 1 and Circuit 2 in Table 4.1 are CMOS implementations of two benchmark circuits from [39]. The average differences between actual fault simulation results and FACE's estimates for the three transistor circuits vary from 0.62 percent to 2.51 percent.

The 8-bit multiplier circuit is represented in three different ways for fault coverage estimation to test the mixed-level capability of FACE. In configuration 1, the 8-bit multiplier is described as interconnections of twenty functional blocks. In configuration 2, some of the blocks are replaced by their gate-level equivalent circuits. Finally, some of the gate circuits are described at transistor level. Results in Table 4.2 show that good correlation between

Circuit	#Trans-istors	Primary Inputs/ Outputs	Number of Vectors	Number of Faults	Final Coverage		
					Fault Sim. %	FACE %	Average Difference
4-bit ALU	326	14/8	31	652	64.9	65.2	0.62
CKT 1	624	36/7	42	1248	62.2	62.6	2.51
CKT 2	1802	60/26	68	3604	71.2	71.7	1.71

TABLE 4.1 FACE results for three cmos circuits

estimate and actual coverage is obtained for all three configurations. For configuration 3, the coverage curve for transistor faults and logic faults are plotted in Figure 4.5 and 4.6. From the two graphs, it can be seen that FACE coverage estimates follow closely the fault simulation results for both transistor and logic faults.

Config.	#Trans.	#Gates	#Blocks	No. of Trans. Faults	No. of Logic Faults	No. of test vectors	Transistor Fault			Logic Fault		
							Fault Sim(%)	FACE %	Average Diff.	Fault Sim(%)	FACE FACE	Average Diff.
1	0	0	20	...	356	10	...	...	...	98.9	98.5	0.85
2	0	292	12	...	3230	194	...	...	...	95.1	93.0	0.83
3	470	203	12	940	2646	194	68.5	70.0	0.90	95.6	92.5	1.1

TABLE 4.2 FACE results for 8-bit multiplier in 3 configurations

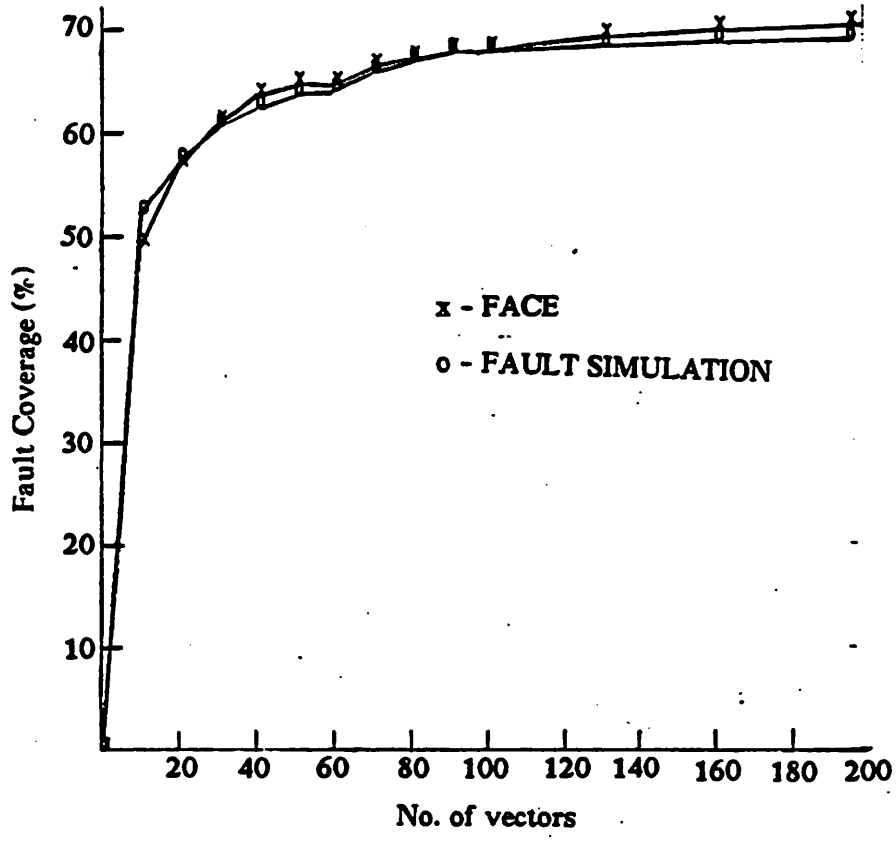


Fig. 4.5 Transistor fault coverage for a mixed-level 8-bit multiplier circuit

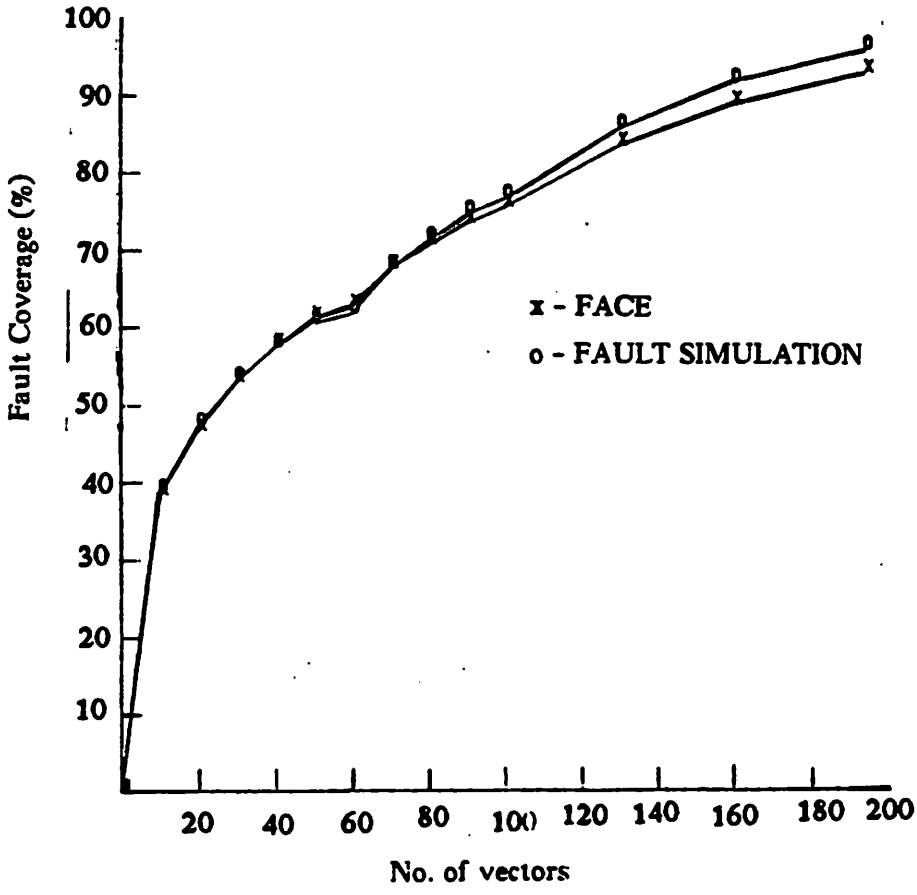


Fig. 4.6 Logic fault coverage for a mixed-level 8-bit multiplier circuit

4.7 Conclusions

Statistical techniques are applied in evaluation of test patterns for fault coverage estimation at mixed-level. Observability formulae are developed for MIMO functional blocks and methods are described for computing transistor fault probability. Results show that good match between fault coverage estimate and the actual value can be obtained with greatly reduced computational complexities. For a set of test vectors, fault coverage can be estimated for both transistor and logic faults in a mixed-level circuit.

CHAPTER 5

Fault Simulation for Multiple-Fault Detection

5.1 Introduction

The most popular fault model used for testing is the stuck-at fault model. For a circuit with n lines, there are a total of $2n$ stuck-at faults. If combinations of these faults may simultaneously occur, there are a total of $3^n - 1$ multiple-fault combinations. The number of faults can be reduced by applying fault equivalence [57] and dominance [58], and the concepts of checkpoint [59] and prime [60] faults. However, the total number of multiple-faults has been still considered unacceptably large. For this reason, it has been often assumed that at most a single fault can occur at one time in a circuit. However, there are situations where the single-fault assumption is inadequate and the multiple-fault model cannot be avoided [61] [4]. Another application of multiple-fault detection is in the area of logic synthesis. Redundant multiple-faults correspond to a set of multiple lines that can be removed from a circuit at the same time resulting in a reduction of the circuit size. This would not have been possible if the single stuck-at fault model have been used.

The traditional approach of test generation - generate a test vector for a chosen fault and evaluate the test vector through fault simulation of the total fault list - is impractical even for small circuits. To store explicitly all the fault combinations for a circuit of reasonable size would require a prohibitive amount of computer-memory storage. Algebraic approaches have been proposed for multiple-fault detection [62] [60]. But their complexity and ineffective use of topological information on the test circuit make their applications only practical for trivial circuits. A possible approach to multiple-fault detection is to convert a Single-Fault Detection Test Set (SFDTS) into a multiple-fault detection test set by augmenting the SFDTS with tests

for the undetected multiple faults. It has been observed that a SFDTS designed to detect only single stuck-at faults usually detects most multiple stuck-at faults. However, the identification of the undetected multiple-faults associated with a test set is itself a formidable task.

Conventional fault simulators are not suitable for the evaluation of test sets for multiple-fault detection. Explicit fault simulation of all multiple-fault combinations would require an unreasonable, if not prohibitive, amount of CPU time. An alternative approach to fault simulation for identifying undetected multiple-faults is to examine the masking relations [23] among faults associated with a test set. This approach involves cumbersome manipulation of Boolean equations and masking graphs and is only feasible for restricted type of circuits.

Identification of undetected multiple-faults can only be performed efficiently if fault combinations are evaluated implicitly. In [63], an approach based on effect-cause analysis was introduced. This method avoids the explicit analysis of the masking relations and utilizes the network topology to guide the multiple-fault diagnosis process. The new implicit multiple-fault simulation approach proposed in this chapter is a great improvement over [63]. In the new fault simulation approach, the problem of identification of undetected multiple-faults is considered as one of *implicit enumeration of the fault space*. Fault combinations producing the same output responses as the fault-free circuit for a test set are implicitly enumerated. A new *15-valued simulation method* is used to facilitate *vertical implication* that relates values of the same line for different test vectors. Using the 15-valued simulation method, the masking relations among faults are implicitly analyzed. Topological information is used, based on the concept of dominators [64], to facilitate the enumeration process.

This chapter is organized as follows. In the next section, previous work on fault simulation for multiple-fault is described. In Section 5.3, the simulation model for the enumeration of the fault space is described. The 15-valued simulation method is described in Section 5.4. The fault-space enumeration process is explained in Section 5.5. Procedures used to speed up

the enumeration process are discussed in Section 5.6. Results obtained for several examples are presented in Section 5.7.

5.2 Previous Work

The identification of undetected multiple-faults associated with a test set were mostly done by examining the masking relations among faults [62] [23] [60]. A fault f_j is said to mask a fault f_i for test t if the test t detects f_i but not the multiple-fault (f_i, f_j). This represents a fault masking condition for f_i associated with t. The identification procedure can be summarized as follows.

(1) For each test t_i in a test set T, find the set of faults F_{t_i} that can be detected by it. For each fault f in F_{t_i}, determine the fault masking conditions associated with f for t_i.

(2) Deduce from the fault masking conditions the set of multiple-faults not detected by the test set T. The deduction is performed by iteratively determining whether the fault f_j that masks the fault f_i is guaranteed to be detected by the test set T (GTBD). If this is true, then fault f_i is also GTBD and the masking condition is resolved. The set of masking conditions that cannot be resolved corresponds to the set of undetected multiple-faults.

Step 1 involves cumbersome manipulation of Boolean equations and the set of masking conditions which represents explicitly the set of undetermined multiple-faults can be very large. The examination of fault masking conditions in Step 2, performed using the masking graph [62] [23] or a tabular approach [60], is very inefficient.

5.3 Simulation Model

The set of basic faults used for multiple-fault fault simulation is the set of prime faults [60]. Any possible multiple stuck-at fault is equivalent to a multiple prime fault. The set of prime faults contains the following faults:

(1) For an AND or NAND gate, if an input of the gate is a fanout branch or a primary input line, then the input stuck-at-1 fault is included in the prime fault set. Similarly for an OR or NOR gate, an input stuck-at-0 fault is included. (Inverters and buffers are considered as lines unless their outputs are primary outputs, then they are treated as NAND/NOR and AND/OR gates)

(2) If all input stuck-at-1 faults of an AND or NAND gate are included as prime faults and the output of the gate does not fan-out, then the output stuck-at-0 fault is included. Similarly, the output stuck-at-1 fault for an OR or NOR gate is included.

The number of prime faults is significantly smaller than the total number of stuck-at faults in a circuit and any possible multiple stuck-at fault is equivalent to a multiple prime fault. One important feature of the prime fault is that only one of the stuck-at faults, i.e. stuck-at-1 or stuck-at-0, is considered for each prime fault site - each line in the circuit is in at most any of two states. The simulation model, similar to the fault injection model [58], used for enumeration of the fault space is constructed based on the prime faults as follows.

(1) If line l in a circuit is a prime fault site and it can only be in any of the two states, fault-free or stuck-at-0, then line l is replaced by the line model shown in Figure 5.1.

(2) Similarly, if line l is either fault-free or stuck-at-1, then an OR gate is used instead of an AND gate in the line model in Figure 5.1.

Every prime fault introduces a fault-input (FI) and a logic gate into the simulation model as shown in Figure 5.1. The value assigned to the fault-input FI represents the status of the line l. For example, if a 0 is assigned to FI in Figure 5.1, l is stuck-at-0 and a 1 at FI means l is fault-free. Any fault-input combination (vector) represents a multiple prime fault.

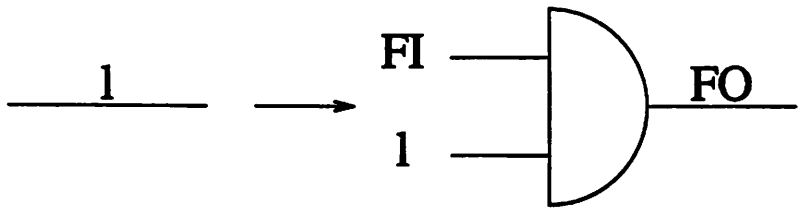


Fig. 5.1 Line model for a prime fault line l s-a-0

Given a test set and a fault-input vector, if the output responses of the simulation model is the same as the fault-free responses of the circuit, the fault-input vector corresponds to an undetected multiple-fault associated with the test set.

The simulation model can be used to determine whether a multiple-fault is detected by a test set. One multiple-fault and a single test vector is examined at a time, but the great number of multiple-faults makes it impossible to carry out this procedure explicitly. The undetected fault space should be enumerated implicitly and the whole test set should be examined simultaneously through vertical implications. Vertical implication is the process by which the value v_1 of a line l for an input vector t_1 implies that the value of l for another input vector t_2 is equal to v_2. Vertical implications are accomplished through the 15-valued simulation method described in the next section.

5.4 15-valued Simulation

Given a test set and a fault-input vector, a line l in the simulation model for fault-space enumeration can assert one of the following 4 values for a test in a test set.

(1) A0: The value of line l is equal to a logical 0 for any test of the test set.

(2) A1: The value of line l is equal to a logical 1 for any test of the test set.

(3) 0: The value of line l is a logical 0 for the test t_1 and there exists at least one test t_2 in the test set for which the value of l is a logical 1.

(4) 1: The value of line l is a logical 1 for the test t_1 and there exists at least one test t_2 in the test set for which the value of l is 0.

The first two values are used to facilitate the handling and propagation of the effect of a fault. The last two values are similar to the conventional logical 0 and 1, but asserting any of the two values implies that there exists at least two tests, t_1 and t_2, such that the value of line l is 0 for t_1 and 1 for t_2 or vice versa.

The set of possible values for a fault-input of the simulation model is {A0, A1}, the two values corresponding to the two possible states of a prime fault site. The set of possible values for an internal line in the simulation model is {0, 1, A0, A1}. Based on the four possible values, a 15-valued alphabet, shown in Figure 5.2, is used for fault space enumeration. The sets defining members of the 15-valued alphabet are unordered. During the fault-space enumeration process, a line in the simulation model can assert any member of the 15-valued alphabet.

Based on the 15-valued alphabet, truth tables of primitive logic gates can be defined for forward implications during the fault space enumeration. The basic truth table for an inverter for the first four values of the 15-valued alphabet is shown in Figure 5.3 and the basic truth tables for 2-input AND and OR gates for the first four values are shown in Figure 5.4 and Figure 5.5 respectively. The complete truth tables containing all possible combinations of input values for the primitive gates can be constructed from the basic truth tables. For example, if

0	{0}
1	{1}
2	{A0}
3	{A1}
4	{0, 1}
5	{0, A0}
6	{0, A1}
7	{1, A0}
8	{1, A1}
9	{A0, A1}
10	{0, 1, A0}
11	{0, 1, A1}
12	{0, A0, A1}
13	{1, A0, A1}
14	{0, 1, A0, A1}

Fig. 5.2 A 15-valued alphabet

0	1
1	0
2	3
3	2

Fig. 5.3 Basic Truth Table for an inverter

the sets of possible values for the inputs of a 2-input AND gate are {0, A1} and {1, A0}, then the set of possible values for the gate output is {0, 1, A0}.

The 15-valued alphabet not only facilitates the handling and propagation of fault effects on the circuit, it also enables vertical implication to be carried out so that different values of the same line for different test vectors can be related. For example, if line l asserts a 0 for test t_1, then it is implied that line l cannot asserts A0 or A1 for any other test. Similarly, if l asserts A0 for a test t_1, then the values for the entire test set must be equal to A0. Vertical implication allows one to consider all the test vectors of a test set simultaneously when

	0	1	2	3
0	5	5	2	0
1	5	1	2	1
2	2	2	2	2
3	0	1	2	3

Fig. 5.4 Basic Truth Table for an AND gate

	0	1	2	3
0	0	8	0	3
1	8	8	1	3
2	0	1	2	3
3	3	3	3	3

Fig. 5.5 Basic Truth Table for an OR gate

	0	1	2	3
0	0	1	-	-
1	0	1	-	-
2	-	-	2	-
3	-	-	-	3

Fig. 5.6 The Basic Vertical Implication Table

examining the multiple-fault detectability of a test set. The basic vertical implication table for the first four values of the 15-valued alphabet is shown in Figure 5.6. Elements of the row heading represent the values of a line l for a test t_1. Elements of the column heading represent the current values of l for a test t_2. Entries in the table correspond to the new values of l for test t_2 vertically implied by t_1. A '-' entry means conflict. The complete vertical implication table for all possible combinations of values for t_1 and t_2 can be constructed

from the basic vertical implication table. For example, if the sets of possible values for t_1 and t_2 are {0, A1} and {1, A0} respectively, then the new sets of possible values for t_1 and t_2 are {0} and {1}.

5.5 Fault-Space Enumeration

Implicit multiple-fault simulation for a test set can be done using the simulation model and the 15-valued alphabet via fault-space enumeration. The fault-input space of the simulation model described in Section 5.2 represents the space of all possible combinations of multiple-faults. Any multiple-fault and its effect on the original circuit can be represented by a fault-input vector to the simulation model. Given a test set and a fault-input vector, if the output responses of the simulation model is the same as that of the fault-free circuit, the fault-input vector corresponds to an undetected multiple-fault associated with the test set. The set of undetected multiple-faults associated with a test set can be determined by implicitly enumerating the fault-input space.

Fault space enumeration is performed through the process of value justification similar to that of [63], which is an extension of the line justification technique in the D-Algorithm. Rather than a single test in the D-Algorithm, a test set is worked with in fault-space enumeration. Initially, all tests of the test set are placed at the primary inputs of the simulation model, all fault-inputs are assigned the set of possible values {A0, A1} and all other wires are assigned the set of values {0, 1, A0, A1} for each test. Implications induced by the value assignments at the primary inputs and fault-inputs are performed. The fault-free output values for all tests of the test-set are then placed at outputs of the simulation model and the corresponding induced implications are carried out. After all implications are done, there may be values at the outputs of gates that need to be justified. One of the unjustified values is chosen and a *decision* is made to assign a particular value to one of the inputs of the corresponding gate. After a decision is made, implications induced by it are performed. Decisions and implications are carried out until there are no values left unjustified. When all

values are justified, the current fault-input space represents a set of undetected multiple-faults.

There are three kinds of implications - forward, backward and vertical - performed during the fault-space-enumeration process. Forward and backward implications are also called horizontal implications. Forward implication on a gate is performed using the truth tables in Figures 5.3, 5.4 and 5.5. Vertical implications relating values of the same line for different tests are evaluated using the table in Figure 5.6. Backward implication on a gate is based on the truth tables in Figures 5.3, 5.4 and 5.5. Whenever the value of a line l for a test t_1 is changed, forward implications are checked for the outputs of all the gates l fans out to and backward implications are checked for the inputs of the gate driving l. Vertical implications are checked for values of l for other tests.

During the enumeration process, backtracking is invoked under two conditions: 1) Inconsistency arises during the implication process; 2) No values are left unjustified. Backtracking is the process of returning to the last decision point and trying an alternative decision, i.e. assigning other possible values to a line. The fault space is enumerated exhaustively, but implicitly, via the backtracking process. The flowchart of the fault-space-enumeration process is shown in Figure 5.7.

5.6 Procedures For Speeding-up Enumeration

Based on the simulation model and the 15-valued simulation method, the problem of identifying undetected multiple-faults can be viewed as a finite-space search problem. Search is conducted systematically on the fault-input space of the simulation model through branch-and-bound techniques. A decision tree is constructed and a backtracking procedure is applied, as described in Section 5.4, to search exhaustively but implicitly through the entire space.

At any point of the search, the current decision tree corresponds to a subspace of the entire fault-input space. Whenever a new decision is made, a new node is added to the decision tree - branching; and whenever it can be identified that the entire current subspace is either a solution (no values left to be unjustified) or not a solution (inconsistency arises),

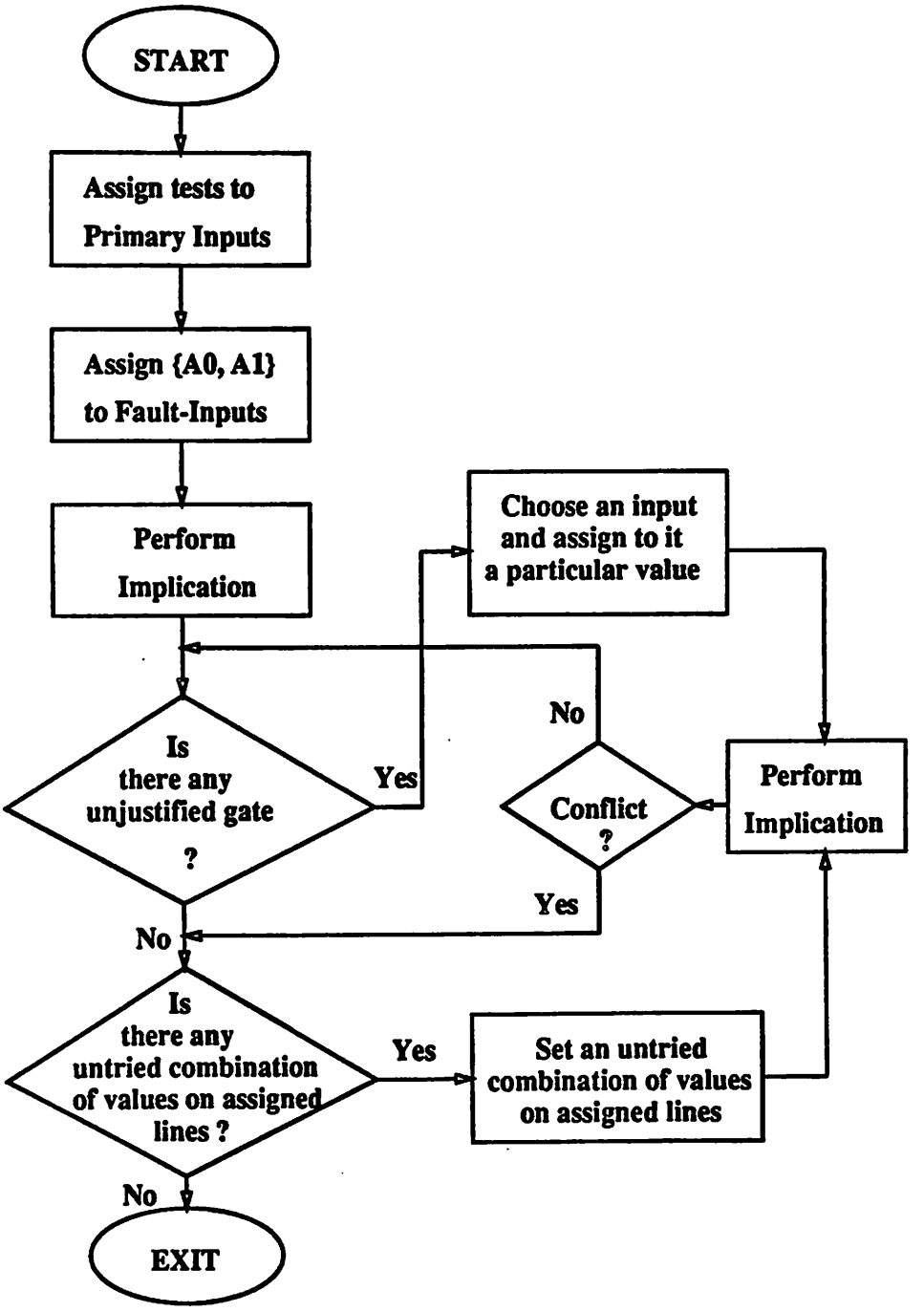


Fig. 5.7 Flowchart of fault-space-enumeration process

backtracking is invoked to leave the current subspace for an unenumerated area - bounding. The number of backtracks required to enumerate the entire search space greatly influences the efficiency of the enumeration process. In order to minimize the number of backtracks, the algorithm needs to be able to recognize as early as possible that the current subspace is either a solution or not a solution without making additional decisions. This can be done by utilizing intelligently the topological information and improved implication procedure to reduce the set of possible values a line can have for a test and detect inconsistency early.

5.6.1 Improved Horizontal Implication Procedure

One reason that the enumeration process may fail to recognize early whether a subspace is a solution or not is that the set of possible values assigned to a line for a test can be unnecessarily large. This is because during the backward implication process, all inputs to a gate are considered independent of each other, which in general is not true with the presence of reconvergent fanouts. This can be remedied using a *learning* procedure similar to the combinational test generation algorithm SOCRATES [10] as follows.

(1) Select one of the possible values a line l can have and assign it to l.

(2) Perform all implications induced by that assignment.

(3) If inconsistency arises during the implication process, delete that assignment value from the set of possible values l can have, perform all induced implications and goto 4. Else stop.

(4) If inconsistency arises from the induced implications, immediately backtrack on the decision tree. Else stop.

These three steps are applied to all lines of the simulation model during the enumeration process whenever a new decision is made. An occurrence of a conflict in (3) implies that l can never assert the assigned value and an occurrence of a conflict in (4) means the current subspace is not a solution and immediate backtracking can be carried out on the decision tree.

The selection of which value to assign to a line l is based on the following criterion to avoid excessive computations. The criterion is aimed to learn implications that cannot be deduced from backward implications performed on the circuits.

(1) If l is a point of reconvergence, assign both A0 and A1 to l.

(2) If l is driven by an OR or NAND gate and the set of values l can have is {0, 1} or {0, 1, A1}, assign 0 to l.

(3) If l is driven by an AND or NOR gate and the set of values l can have is {0, 1} or {0, 1, A0}, assign 1 to l.

5.6.2 Improved Vertical Implication Procedure

Using the 15-valued simulation method described in Section 5.3, the set of possible values assigned to a line can be reduced through vertical implication relating values of the same line for different tests. It is through vertical implication that fault maskings associated with a test set are examined and resolved implicitly. The vertical implication process can be greatly improved by making use of the network topology. The improved vertical implication is based on the Complete Normal Path (CNP) theorem in [63] and the concept of dominators [64].

Definition 5.1: A *complementary path* (CP) is a path from a primary input to a primary output in the simulation model associated with a pair of tests t_1 and t_2 such that every line on this path has complementary values 0 and 1 for t_1 and t_2.

If the number of inversions between two lines l_1 and l_2 on a complementary path is even, then values of l_1 and l_2 are equal for both tests (and if the number of inversions is odd, then values of l_1 and l_2 are not equal for both tests).

The CNP theorem is restated as below and the proof can be found in [63].

Theorem 5.1: If a primary output o of the simulation model has complementary values 0 and 1 for a pair of test t_1 and t_2, then there exists at least one *complementary path* between some

primary inputs and o.

In Theorem 5.1, the relations between the values of lines for two tests, t_1 and t_2, that produce a pair of complementary values at a primary output is described. Any path between an input with complementary values and an output with complementary values for a pair of test t_1 and t_2 is a *possible complementary path* (PCP). Any line on a possible complementary path may have complementary values for the pair of tests t_1 and t_2.

Definition 5.2: A *complementary dominator* associated with a primary output o and a pair of tests t_1 and t_2 is a gate in the simulation model such that its removal breaks all possible complementary paths for that output.

A complementary dominator is a single-element cutset gate of the subcircuit consisting of all possible complementary paths to a primary output for a pair of tests.

Corollary 5.1: The outputs of a complementary dominator for a pair of tests must be complementary.

Proof: The proof follows from the fact that there exists at least one complementary path and a complementary dominator lies on every possible complementary path. □

The property of the complementary dominator can be used to improve the vertical implication process. For example, assume gate n is a complementary dominator for the pair of tests t_1 and t_2 and line l is driven by n. If l asserts a 0 for t_1, then the value of l for t_2 can be vertically deduced to be 1 using the property of a complementary dominator. Furthermore, if the number of inversions of all paths from a complementary dominator to the corresponding output are the same, the values of l for t_1 and t_2 can be uniquely determined. For example, if the values of the primary output for tests t_1 and t_2 are 0 and 1, and the inversions of all paths from a complementary dominator to that output is even, then the values of the output of the complementary dominator for tests t_1 and t_2 must be 0 and 1. This is because the complementary values at the primary output must have come from the

complementary dominator gate. During the enumeration process, sets of complementary dominators for all primary outputs and each pair of tests in the test set are found whenever a new decision is made. The set of complementary dominators for a primary output associated with a pair of tests is computed by deriving a subcircuit that contains all the possible complementary paths for that output and find the set of single-element cutset gates for the subcircuit.

5.7 Results

An experimental version of the new implicit fault simulation approach, called MULTI, has been implemented in C on the DEC computer VAX8650. The characteristics of the test circuits used are summarized in Table 5.1. The 5th column indicates the number of prime faults. The number of redundant single-faults is shown in the 6th column and the number of redundant multiple-faults is shown in the 7th column. The number of multiple-faults considered for each example is 2^n, where n is number of prime fault in the circuit. The first example is taken from [4]. It is single-fault irredundant but has redundant multiple-faults. The second example is from [65] and has one redundant single-fault and several redundant multiple-faults. The last example is an optimized version of the C880 circuit from the ISCAS benchmark sets [39].

Results obtained for each example for a test set are shown in Table 5.2. In the table, m and s stand for minutes and seconds respectively. Most test sets are generated using single-fault test generation except for the last example, for which a few tests are added to the single-fault detection test set to ease the computational task (described later). As can be seen, fault simulations are completed for all examples within reasonable CPU time. Multiple-faults of all multiplicities are implicitly considered. Most of the undetectable multiple-faults in the table are proved to be redundant.

The amount of computation required for fault simulation is dependent both on the circuit structure and the test set. Circuits with reconvergent fanouts and a great number of redundant single-faults in general require more computations. The presence of reconvergent

Circuits	#inp	#out	#gate	#prime faults	#redundant prime faults	#redundant multiple faults
fried.ex	7	1	15	33	0	16
dand.ex	4	1	7	17	1	28
ALU1	14	8	88	166	0	0
ALU2	14	8	88	178	8	256
ex1	10	1	34	54	0	0
ex2	29	1	52	78	0	0
C880.opt	60	26	225	613	0	0

Table 5.1: Statistics for 7 example circuits

Circuits	#vector	CPU time	#undetected multiple faults
fried.ex	18	2.8s	16
dand.ex	12	1.2s	28
ALU1	28	30s	0
ALU2	32	50s	256
ex1	14	16s	0
ex2	53	186s	0
C880.opt	142	27m	0

Table 5.2: Implicit Fault Simulation Results

fanouts gives rise to functionally equivalent but structurally nonequivalent faults that have the same effects on the circuit but cannot be determined to be equivalent faults from the topological information on the circuit. Singly redundant faults may produce a large number of redundant multiple-faults. During the enumeration process, if the improved vertical implication procedure can be used to determine uniquely many values, then the total amount of computation is greatly reduced. This would be true if pairs of vectors in the test set are different only

in a small number of bits and producing different output values. A simple and straightforward way of reducing the computation, and at the same time, enhancing the multiple-fault detectability of a single-fault detection test set is to add a few tests that are different only in one bit from some tests in the test set but producing different output values from those tests. This is done for the last example in Table 5.2. It can be viewed as a tradeoff between the size of the test set and the computation expenditure.

5.8 Conclusions

A new implicit fault-simulation method for multiple-fault detection has been presented in this chapter. In the new fault-simulation approach, the problem of identification of undetected multiple-faults associated with a test set, is considered as one of implicit enumeration of the fault space using a simulation model. Fault combinations producing the same output responses as the fault-free circuit for a test set are implicitly enumerated. This method avoids the explicit analysis of the masking relations and utilizes the network topology to guide the enumeration process. A new 15-valued simulation method is used to facilitate vertical implication that relates values of the same line for different test vectors. Through the 15-valued simulation method, the masking relations among faults for a test set are implicitly analyzed. Topological information is used, based on the concept of dominators, to improve the vertical implication process. Horizontal implication process is improved by a learning procedure.

CHAPTER 6

Synthesis For Testability

6.1 Introduction

Logic Design and testability have traditionally been addressed as two separate entities. Given the specifications of a digital circuit, logic design tools can be used to produce a feasible design that meets performance and area goals. The design may then be modified later to meet the test requirements. Testability problem is generally solved independently posterior to the design process. Design-for-test techniques, such as scan-based [16] [17] methods, have been developed to facilitate post-design test generation by reducing the sequential test problem to one of testing combinational circuits. This further separates the design and test problems.

Logic synthesis has been the subject of many years of research in both academic and industrial laboratories. The algorithms developed in this arena have matured sufficiently to be practical for real circuit designs and are rapidly gaining acceptance in the design of complex digital circuits. A number of commercial systems are available to carry out logic synthesis. The final implementation obtained by these tools is almost independent of the initial description, freeing the designer from a long and tedious logic gate manipulation process.

Logic synthesis has a profound impact on test generation. Designs produced by synthesis systems [66] that disregard the testability aspect of the designs can be very difficult to test. The increasing use of logic synthesis tools also has escalated the need for automatic test generation tools since the logic produced by the synthesis program may have little direct resemblance to the initial description. It is also well known that there is a strong relationship between combinational logic synthesis and testing. The task of generating a suitable set of test patterns for a combinational circuit is closely related to the logic optimization problem. It

is therefore only natural to address testability simultaneously during the synthesis process such that circuits produced are highly testable with very little redundancy, if not ideally 100% testable.

One of the main objectives for synthesis system is to produce optimal logic designs. Untestable circuits with redundancies certainly are not optimal designs. In fact, optimal logic synthesis can in theory produce fully testable combinational logic designs. In reality, "perfect" optimization is not achievable within reasonable computer time, but the heuristic algorithms used in the logic synthesis process can reach very good solutions that have very little redundancy and are highly testable. The later task of automatic test pattern generation can be significantly sped up for circuits with very little redundancy since the cost of trying to generate tests for redundant faults can be more than 90% of the total test generation time. For combinational circuits, there exists logic optimization algorithms [24] that not only can ideally guarantee fully testable design, but also produce as a by-product a set of test patterns with 100% fault coverage for all single stuck-at faults in the design. On the other hand, the relationship between synthesis and testability for sequential circuits is not understood so well as in the combinational case. And to date no sequential logic synthesis algorithms exist that will guarantee that the resulting sequential circuit is fully testable without resorting to post-design design-for-test techniques such as scan-based methods.

Test generation algorithms can in theory be used in conjunction with a sequential synthesis system to remove all the redundancies in sequential machines resulting in a fully testable design. However, in general, this method requires exorbitant amounts of CPU time. Another solution to the sequential testability problem is the use of scannable memory elements, i.e. complete scan design, in a synthesis system. This approach avoids the sequential testability problem by transforming it into a combinational one. It is a successful but short-term solution that solves the testability problem at the expense of the area and performance of the design. Both solutions also represent an afterthought operation in the design process to

guarantee testability, i.e. removing redundancies that are unconsciously introduced during the design process.

Synthesizing a sequential circuit from a State Transition Graph description involves the steps of state minimization, state assignment and logic optimization. Each step has a profound effect on the testability, area and performance of the final implementation. In this chapter, sequential synthesis procedures for producing highly testable and easily testable Moore or Mealy finite state machines implemented by multi-level logic or PLA are presented.

Basic definitions and terminologies used are given in Section 6.2. Various types of redundant faults in sequential circuits implemented by multi-level logic are described in Section 6.3. For multi-level implementation, two synthesis approaches are described. In the first approach given in Section 6.4, tailored steps of state minimization, state assignment and logic optimization are carried out to produce highly if not fully testable sequential machines. And in the second approach presented in Section 6.5, the synthesized machine is made easily testable by the use of extra logic and a constrained state assignment. For PLA-based finite state machines, a synthesis procedure based on constrained state assignment that ensures testability for all combinationally irredundant crosspoint faults is described in Section 6.6.

6.2 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 $\bar{a}$). A **cube** is a set C of literals such that $x \in C$ implies $\bar{x} \notin C$ (e.g., $\{a,b,\bar{c}\}$ is a cube, and $\{a,\bar{a}\}$ 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, $\{\{a\},\{b,\bar{c}\}\}$ is an expression consisting of the two cubes $\{a\}$ and $\{b,\bar{c}\}$. 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 minterm m_1 is said to **dominate** another minterm m_2 (written as $m_1 \supset m_2$) if for each bit position in the second minterm that contains a 1, the corresponding bit position in the first minterm also contains a 1.

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.** A **R-reachable** finite state machine has a STG such that input sequences exit which

place the machine in any of the 2^{N_b} states, beginning from the reset state. Given a fault F, the State Transition Graph of the machine with the fault is denoted G^F. Two states in a State Transition Graph G are **equivalent** if all possible input sequences when the machine is initially in either of the two states produce the same output response.

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.

A finite state machine is assumed to be implemented by multi-level combinational logic or PLA and feedback registers. The fault model assumed for multi-level implementation is **single stuck-at** on circuit line and the fault model for PLA-based implementation is crosspoint fault. Tests are generated for stuck-at faults in the combinational logic part or crosspoint fault in PLA.

A primitive gate in a multi-level 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.

In a sequential circuit, a fault may be **redundant**, i.e. untestable. There are two kinds of redundancies in a sequential circuit. The first kind is deemed **combinationally redundant** – the effect of the fault cannot be excited or propagated to the *primary outputs or the next state lines*, beginning from any state, with any input vector. A **sequentially redundant** fault is a fault which can be excited by some input vector but its effect cannot be propagated to the *primary outputs.*

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.

A **multiple F-type** fault for a line L, (which is the output of a gate and not a primary output), in a combinational network corresponds to a multiple fault condition on the fanout branches of line L. The multiple fault depends on the types of gates that L feeds into. For example, if a line L_1 has three fanout branches a, b, c, that feed into AND, OR, AND gates respectively, then the multiple F-type fault for L_1 is a stuck-at-1, b stuck-at-0 and c stuck-at-1. If the multiple F-type fault for a line is redundant, it means that the line (and all its fanout branches) can be bodily removed.

6.3 Origin of Redundant Faults in Sequential Circuits

There are two classes of redundant faults in a sequential circuit implemented by multi-level logic, namely, combinationally and sequentially redundant faults. Combinationally redundant faults (*CRF* s) 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. *CRF* s cannot be detected even if all the memory elements of the sequential circuit are made scannable. Sequentially redundant faults (*SRF* s), on the other hand, are related to the temporal characteristics of the sequential circuit. Although *SRF* s alter the combinational logic function of the circuit and hence the State Transition Graph (STG) representing the sequential circuit, they cannot be detected without making some of the latches scannable.

A definition of sequentially redundant faults is given as follows [29]:

(1) An equivalent-SRF is a fault which causes only interchange and/or creation of equivalent states in the STG of the finite state machine.

(2) An invalid-SRF does not corrupt any fanout edge of a valid state reachable from the reset state.

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

An example will be used to illustrate the existence of sequentially redundant faults.

The State Transition Graph (STG) of a finite state machine is shown in Figure 6.1. 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 6.2. The fault $w1$ stuck-at-0 (s-a-0) changes the original STG to the one shown in Figure 6.3. The corrupted edge is shown via a dotted line. Since 010 and 110 are equivalent states in the original STG, the fault $w1$ s-a-0 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 6.4. The fault creates an extra state 111, that was originally an invalid state which is equivalent to the true state 110. Therefore the fault $w2$ is also sequentially redundant. The corrupted edge is shown in dotted lines and the added edges shown in dashed lines.

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 an invalid-SRF. An isomorph-SRF may change the original machine to the one shown in Figure 6.5. 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.

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 6.1 Original Finite State Machine

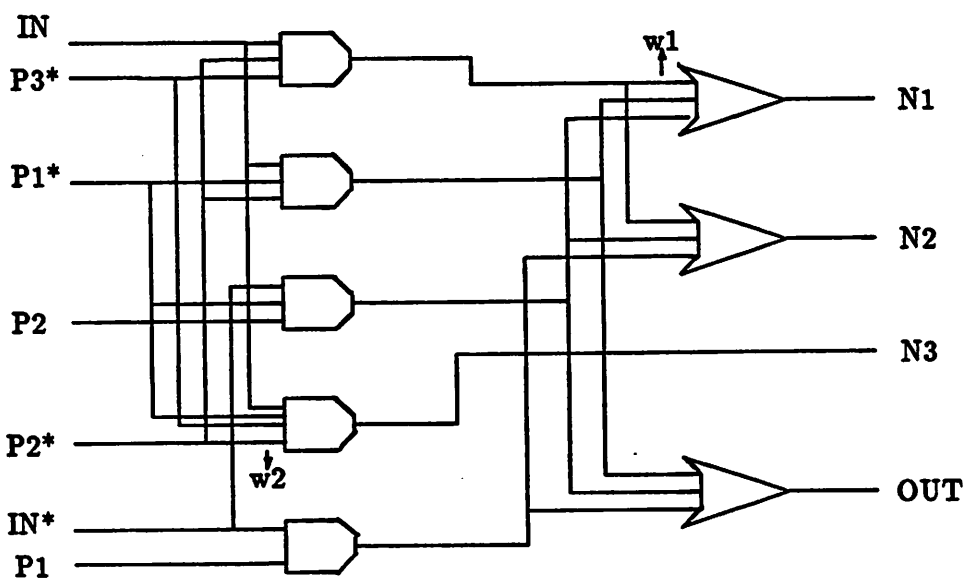


Figure 6.2 Combinational Logic of FSM

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 6.3 Faulty FSM with w1 s-a-0

Theorem 6.1: A redundant fault in a finite state machine is either a CRF or an equivalent-SRF or an invalid-SRF or an isomorph-SRF [29].

Proof (by contradiction): Assume a fault, F, is a redundant fault but not a *CRF* or equivalent-SRF or invalid-SRF or isomorph-SRF. Since F is not a *CRF* or an invalid-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 an equivalent-SRF or an isomorph-SRF, the fault effect on the next state lines will not cause an interchange or creation of 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. □

Theorem 6.1 guarantees that a fully testable finite state machine results if it can be ensured that none of these 4 kinds of redundancies described above exist in the synthesized

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 6.4 Faulty FSM with w2 s-a-1

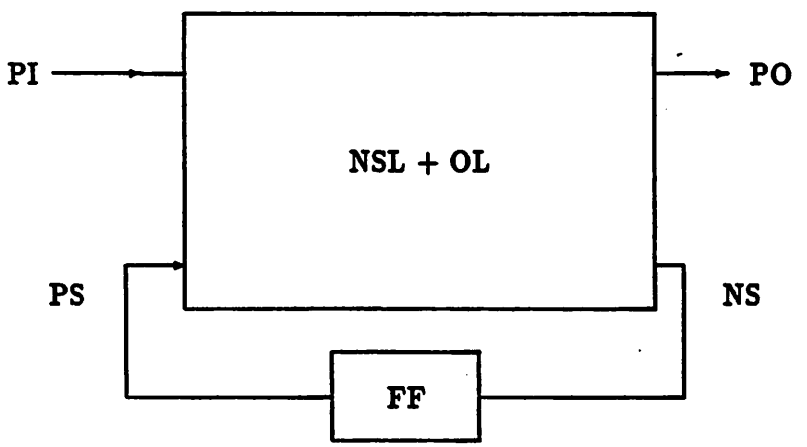


Figure 6.5 Faulty FSM with an isomorph-SRF

machine. Steps in the synthesis procedure designed in the next section are designed to

achieve this goal.

6.4 Irredundant Fully Testable Sequential Machines

A general model for a Mealy finite state machine is shown in Figure 6.6. It is realized by a multi-level 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.

The optimal synthesis procedure is first described in Section 6.4.1. In Section 6.4.2, it is proved that the resulting machine has no CRFs, invalid-SRFs or isomorph-SRFs. Experimental results indicate that the machine has very few redundancies. In Section 6.4.3, a modified synthesis procedure using extended don't care sets in repeated combinational logic minimization which ensures that equivalent-SRFs do not exist in the synthesized machine is presented. The synthesized machine is thus made fully testable. In Section 6.4.4, how finite automata represented at the truth table or at the logic-level can be made fully testable is briefly discussed. In Section 6.4.5, the effects of redundancy removal on the state encoding of

---

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 6.6 General Sequential Machine Model

---

the machine is discussed. Preliminary results, which indicate that these procedures are viable for medium-sized circuits, are given in Section 6.4.6.

6.4.1 The Synthesis Procedure

The procedure consists of the steps of state minimization, state assignment and combinational logic optimization [29]. These steps are described in the sequel.

(1) **State Minimization:** Given an original State Transition Graph specification G^O, a state minimum representation, G^M, is obtained using algorithms similar to those proposed in [67]. 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:** 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 [68] and [69] can be used, which find a state assignment that heuristically minimizes the area of the combinational network after optimization. However, the state assignment algorithm may have to *explore a certain number of possible state assignments* in order to ensure a locally optimal solution (see Definition 6.1).

(3) **Combinational Logic Optimization:** Given the encoded machine, which is now a combinational logic specification, a prime and irredundant combinational logic network which implements both the next state logic and output logic functions is synthesized. 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 [24]. The multi-level network has to be irredundant for a certain class of multiple stuck-at faults as well (see Lemma

6.2).

There are 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).

6.4.2 Correctness of Procedure

It can be proved that the sequential machine synthesized by the procedure of the previous section is irredundant for all CRFs, invalid-SRFs and isomorph-SRFs.

The following theorem follows from the definition of state minimality. It is given in [70].

Theorem 6.2: 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 [29].

Next, it will be shown 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 two-level, prime and irredundant. Isomorphic faulty and true State Transition Graphs imply that the fault has no other effect than interchanging the codes of the states of the machine.

Lemma 6.1: Stuck-at faults on the primary input (PI), primary output (PO), present state (PS) and next state (NS) lines cannot produce a faulty State Transition Graph G^F that is isomorphic to G [29].

Proof: Consider a primary input fault F. Without loss of generality, assume that it is a stuck-at-1 fault on the 1st primary input line. The effect of this fault is to cause all input vectors i_k such that $i_{k[1]}$=0 to become, in effect, i_l where $i_{l[1]}$=1 $i_{l[i]}$=$i\_k[i]$, $2 \le i \le N_i$. Since F is combinationally irredundant, there will exist an input vector pair (i_1, i_2) where $i_{1[1]}$=0, $i_{2[1]}$=1 $i_{1[i]}$=$i_{2[i]}$, $2 \le i \le N\_i$ such that $n(i_1, q) \neq n(i_2, q) \parallel o(i_1, q) \neq o(i_2, q)$ for some

q (Else, i_1 can be replaced by $i_1 \cup i_2$ in the combinational truth table). First, consider the case where the fanout states are different for i_1 and i_2. If in G, $n(i_1, q)=q_2$ and $n(i_2, q)=q_3$, then in G^F we have $n(i_1, q)=n(i_2, q)=q_3$. For G^F to be equivalent to G, we need $q_3 \in G^F \equiv q_2 \in G$ and $q_3 \in G^F \equiv q_3 \in G$ (since there is a corrupted and uncorrupted edge from q to q_3 in G^F). This requires $q_3 \in G \equiv q_2 \in G$, which is a contradiction. The second case where the primary outputs of i_1 and i_2 are different is simpler. We have two edges from a state in G that assert different outputs and go to the same next state, merging in G^F. This means G^F cannot be isomorphic to G.

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.

If all stuck-at faults on present state lines are combinationally irredundant, for any present state line i, there are two states q_1 and q_2 whose codes differ in bit i alone. q_2 and q_1 merge in G^F due to a fault on present state line i. Hence, $||G^F|| < ||G||$ and isomorphism cannot occur.

The argument for the next state line faults is similar to the argument for the present state line faults. □

Theorem 6.3: If the two-level 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. Also, if a prime and irredundant multi-level circuit is synthesized using only algebraic factoring techniques from a prime and irredundant two-level network, then any fault F in the circuit cannot produce a G^F that is isomorphic to G [29].

Proof: By Lemma 6.1, faults on the PI and PS lines need not to be considered. In a two-level network, faults on the intermediate lines and outputs, have the property that they either pro-

duce a D or a $\bar{D}$ at the outputs of the network, *uniformly* for all test vectors that detect the fault. Isomorphism implies an interchange of codes of multiple states. Without loss of generality, assume a two-way swap, between the codes of $q_1, q_2 \in G$ to produce G^F isomorphic to G. An edge e_1 exists from some state s_1 that goes to q_2 in G^F instead of q_1 in G. Similarly, an edge e_2 from some state s_2, that goes to q_1 in G^F instead of q_2 in G exists. In the combinational sense, if e_1 produces a D at some next state line where q_1 and q_2 differ, e_2 has to produce a $\bar{D}$ at that line. This is not possible in a two-level network for faults on intermediate lines and/or outputs. Therefore, isomorphism cannot occur.

The same argument holds for a multi-level network produced by algebraically factoring a two-level network. □

In a general multi-level network, however, the faults in the intermediate lines may produce both a D as well as a $\bar{D}$ at any particular output, due to reconvergent fanout paths with differing numbers of inversions. The arguments of Theorem 6.3 do not hold, when Boolean operations are used in multi-level combinational logic synthesis.

Lemma 6.2: If a prime and irredundant multi-level network C is irredundant for multiple F-type faults for each line in the network that is the output of a gate and not a primary output, then for any single stuck-at fault, F, in C, there will exist an input vector pair (i_1, i_2) such that i_1 is a test vector for the fault and i_2 is not, and i_1 produces the same output in C^F as i_2 does in C [29].

Proof: Consider a prime and irredundant multi-level circuit 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 $IV(j)(i)$, $1 \le i \le N_j$. The combinations of $IV(j)$ that are caused by some primary input combination are denoted $IV(j)^{CA}$ and the combinations that never appear are

denoted $IV(j)^{DC}$.

Without loss of generality, consider the s-a-0 and s-a-1 faults on $IV(1)(1)$. Some $iv_1 \in IV(1)^{CA}$ has to detect the s-a-0 fault and some $iv_2 \in IV(1)^{CA}$ has to detect the s-a-1 fault. Obviously, $iv_1[1] = 1$ and $iv_2[1] = 0$. If for any $iv_1 \in IV(1)^{CA}$ that detects the s-a-0 fault, there is a $iv_3 \in IV(1)^{CA}$ such that $iv_3[1] = 0$, $iv_3[i] = iv_1[i]$, $2 \leq i \leq N_j$, then there exists a complementary PI vector pair (i_1, i_3) corresponding to (iv_1, iv_3) with i_1 detecting the s-a-0 fault and producing a faulty output equal to the true output of i_3 which does not detect the fault. Furthermore, (i_3, i_1) will be a complementary PI vector pair for the s-a-1 fault.

Now, consider the case of $iv_3 \in IV(1)^{DC}$ for all $iv_1 \in IV(1)^{CA}$ that detect the s-a-0 fault. By the argument above, if for any $iv_2 \in IV(1)^{CA}$ that detects the s-a-1 fault, there is a $iv_4 \in IV(1)^{CA}$ such that $iv_4[1] = 0$, $iv_4[i] = iv_2[i]$, $2 \leq i \leq N_j$, then (iv_2, iv_4) constitutes a complementary pair for the s-a-1 fault and (iv_4, iv_2) constitutes a complementary pair for the s-a-0 fault.

The last case needs to be considered is $iv_3 \in IV(1)^{DC}$ for all $iv_1 \in IV(1)^{CA}$ that detect the s-a-0 fault and $iv_4 \in IV(1)^{DC}$ for all $iv_2 \in IV(1)^{CA}$ that detect the s-a-1 fault on $IV(1)(1)$. For any $iv_k \in IV(1)^{CA}$ that does not detect the s-a-0 or s-a-1 fault, there exists iv_l such that $iv_l[1] = \overline{iv_k[1]}$, $iv_l[i] = iv_k[i]$, $2 \leq i \leq N_j$, producing the same output as iv_k in the true or faulty circuit. $IV(1)^{CA}$ can then be represented using $IV(1)^{DC}$ as a set of cubes, $iv_1 \cup iv_3$, $iv_2 \cup iv_4$, .. $iv_k \cup iv_l$, where the first bit in each cube is a don't care. This means the line $IV(1)(1)$ can be bodily removed, i.e. the multiple F-type fault corresponding to $IV(1)(1)$ is redundant, which is a contradiction. Therefore, a complementary vector pair has to exist for the stuck-at faults on $IV(1)(1)$ and other $IV(1)(k)$. A similar argument can be made for the intermediate lines corresponding to the inputs to the g_{ji} □.

Using Lemma 6.2, the following theorem, which restricts the occurrence of isomorphism in sequential machines, implemented by prime and irredundant multi-level networks that are also irredundant for multiple F-type faults in the network, can be proved. Q denotes the set

of states in G^M.

Theorem 6.4: If a set of states $Q_I \in Q$ is such that each state in Q_I has the property that its fanout edges assert distinct outputs from all other states in Q or has fanout next states in $Q-Q_I$, which are distinct from the fanout states of all other states in Q, or possesses distinct combinations of outputs and fanout next states, then a fault cannot produce an isomorphic machine causing only interchange of states within Q_I [29].

Proof: The case of $\|Q_I\|=2$ and where fanout edges from state s_1 assert a set of distinct outputs O_1 and fanout edges from the second state s_2 assert a set of distinct outputs O_2 will be proved first. Assume there exists a fault F that produces an isomorphism between these states. In the isomorph G^F, fanout edges from s_1 (s_2) will assert O_2 (O_1). However, by Lemma 6.2, an uncorrupted edge asserting some $o \in O_1$ or $o \in O_2$ has to exist in G^F. This edge can only come from s_1 or s_2, respectively. This means that in the faulty machine, either s_1 or s_2 asserts outputs from both O_1 and O_2, implying that G^F is not isomorphic to G. The argument is easily generalized to $\|Q_I\|>2$.

A similar argument can be made for states s_1, s_2 with distinct next state fanouts or distinct combinations of outputs and next state fanouts. □

Thus, a sequential machine with a G^M where all states possess distinct combinations of outputs and fanout states cannot have faults that cause isomorphism, whether the combinational logic is implemented in two-level or general multi-level form.

Definition 6.1: A state assignment of G^M is deemed to be **locally optimal** with respect to a subset of states $Q_I \in G^M$, if interchanging the codes of $q \in Q_I$ does not produce a better logic implementation after optimization.

The state assignment is locally rather than globally optimal in the sense that interchanging the code of $q_1 \in Q_I$ with $q_2 \notin Q_I$ could produce a better logic implementation. In a multi-level implementation, if there exist states in G^M that do not satisfy the condition of Theorem

6.4, then in order to ensure that a redundant fault does not cause isomorphism, the state assignment of G^M has to be *locally optimal*, with respect to interchanging the codes of these states. For a two-level implementation, *any* state assignment is locally optimal, with respect to *all* states in G^M.

Theorem 6.5: If G^M contains 2^{N_b} valid states where N_b is the number of latches in S^M, S^M is fully testable, if the prime and irredundant combinational network is implemented in two-level form, or if a locally optimal state assignment has been found, as per Definition 6.1, across all states that do not satisfy the condition of Theorem 6.4 [29].

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 6.3 this is not possible in a two-level implementation. In a multi-level implementation, if G^F is isomorphic to G, the sets of states satisfying the condition of Theorem 6.4 cannot be involved in the isomorphism. If isomorphism occurs due to F, it has to involve a set of states, Q_I, not satisfying the condition of Theorem 6.4. The isomorphism produces a G^F equivalent to G, with a *better* implementation (after optimization) than that of G (with at least one less line). However, this contradicts the fact that the initial state assignment for G^M that produced G is locally optimal under the exchange(s) of the codes of states in Q_I. 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, provided the state assignment is locally optimal. 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 6.3: An invalid state in the State Transition Graph is never required to detect a fault in S^M [29].

Proof: All unused state codes may be 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 *under this don't care set*, there always exists a valid state that detects any fault (and provides the initial propagation to the next state lines or primary outputs) that the invalid state detects. □

The preceding results will be used to prove the partial irredundancy theorem for machines whose G^M has $N_s < 2^{N_b}$ states.

Theorem 6.6: The sequential machine S^M produced by the synthesis procedure may contain only equivalent-SRFs.

Proof: By Lemma 6.3, no invalid-SRFs can exist. By Theorem 6.3, if S^M is implemented as a two-level network, no isomorph-SRFs can exist. If S^M is implemented as a multi-level network, then a locally optimal state assignment as per Definition 6.1, across all states that do not satisfy the condition of Theorem 6.4, is found. This guarantees that no isomorph-SRFs will exist. S^M does not contain any CRFs. Therefore, by Theorem 6.1, only equivalent-SRFs can exist. □

6.4.3 Eliminating Redundancies Via Extended Don't Care Sets

In this section, it will be shown how the testability of the synthesized machine S^M can be increased by removing possible equivalent-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* equivalent-SRF was illustrated in Figure 6.4 (Section 6.3). There exists an invalid state q which has identical fanout and hence is equivalent to some valid state v_1. An edge from v_2 to v_1 is corrupted by a F 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 equivalent-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. These redundancies occur because current state assignment algorithms do not use the freedom of state splitting (Section 6.5), so as to obtain an optimal solution. It is very difficult to extend state assignment algorithms in this direction and hence irredundancy will be ensured by specifying an extended don't care set in a repeated logic minimization procedure.

(1) State assignment and logic optimization are performed as before, with logic optimization using 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, corresponding to an encoded State Transition Graph. ip

(3) Given a valid state v_1, valid states $v_2, v_3, .. v_L$ that are equivalent to 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 $n(fanin(v_1)) = DC(vsub1, v_2, .. v_L, iv_1, iv_2, .. iv_K)$. $DC()$ implies that any (but at least one) 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'. This may make a previously invalid state valid.

(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 $G \leftarrow G''$, go to Step 2.

In the first iteration, there will not be valid states $v_2, .. v_L$ that are equivalent to any v_1, since the machine is a reduced one. However, after Step 3 above, some invalid states that are equivalent to v_1 may become valid.

Theorem 6.7: The procedure above converges, and the resulting machine after convergence will not have any simple equivalent-SRFs, invalid-SRFs or isomorph-SRFs.

Proof: The procedure converges when succeeding logic minimizations have produced the same result. Each logic minimization starts with the result of the previous logic minimization. Additional don't cares are provided. It is 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. Since the cost function is bounded from below, the sequence of logic minimizations must eventually converge, and on the last call, return an unchanged network, η. No isomorph-SRFs will exist in the prime and irredundant network η by Theorem 6.3 and Theorem 6.4. 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 invalid-SRFs can exist.

Finally, using the don't care sets corresponding to the equivalent states, it is ensured 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 equivalent-SRF. Therefore, F is testable or not a simple equivalent-SRF. □

More complicated equivalent-SRFs may exist, though experimental evidence indicates that this is extremely rare. In fact, a *single* case of an equivalent-SRF that is not of the form of the SRF of Figure 6.4 has yet to be found. 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 equivalent-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 a 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 for a single cycle propagation is that

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

This set of don't cares and associated constraints are found for the different state pairs that are not equivalent. Optimal use of these don't cares and associated constraints, generalized to multiple-cycle propagation, ensures full testability.

Theorem 6.8: Using the additional don't care set in the synthesis procedure will result in a fully testable machine.

Proof: By Theorem 6.7, no simple equivalent-SRFs, invalid-SRFs or isomorph-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 equivalent-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 6.6) 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 be 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.

6.4.4 Synthesis from Logic-Level Descriptions

In this section, it will be described 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 disjoint 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 min-terms) 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 6.4.2 and 6.4.3.

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 [71]. Given this (encoded) State Transition Graph, Steps 1-3 described above are carried out as before.

6.4.5 Effect of Redundancy Removal via Logic Minimization on State Encoding

If a combinationally redundant line is removed from a logic network (i.e. replaced with a 0 or a 1), 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.

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 [72] that state splitting may be required for an optimal state assignment. Unfortunately, the state assignment problem is difficult enough, 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. As mentioned in Section 6.4.2, when a machine has a two-level combinational logic implementation, any state assignment is locally optimal with respect to all the *used* state codes. However, the state assignment may be sub-optimal when considering the invalid or unused state codes. In the multi-level case too, a state assignment that is locally optimal under the valid (used) state codes may be sub-optimal when considering the invalid (unused) state codes. The replacement of a state code by an unused state code results in a "better" machine.

State assignment techniques (e.g. [69] [68]) do not take state splitting into account in their attempt to find locally or globally optimal solutions. In our experience, the occurrence of the first effect is much more frequent. 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 6.4.3, has the effect of changing a sub-optimal state encoding to a locally optimal encoding that corresponds to a fully testable machine.

6.4.6 Results

In this section, some preliminary results obtained using the synthesis procedures described in Section 6.4 are presented. 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.

Redundancies can be explicitly removed via the use of test pattern generation algorithms, to produce fully testable sequential circuits. However, redundant lines corresponding to redundant stuck-at faults can only be removed (replaced with a 0 or a 1) one at a time. Furthermore, removing a redundant line may introduce new redundancies and so *all* faults have to be checked for redundancy on *each* removal. These two techniques are compared to the synthesis of irredundant sequential circuits.

Some examples in the MCNC 1987 Logic Synthesis Workshop are chosen as test cases, whose statistics are given in Table 6.1. Beginning from a State Transition Graph description, G, the following steps were performed in the synthesis procedure.

(1) **State Minimization:** The machines were state minimized.

(2) **State Assignment:** Binary codes were assigned to the states in G using the program KISS [68]. 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.

(3) **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.

Tests were generated for the resulting sequential machine M whose combinational logic is implemented by C. Test generation was accomplished using the program STALLION [26]. The number of encoding bits used in state assignment (#lat), the number of gates in C (#gate)

EX	#inp	#out	#states	#edges
ex1	2	2	6	24
ex2	2	1	13	57
bbara	4	2	7	45
bbsse	7	7	13	55
s1	8	6	20	110
planet	7	19	48	118
dfile	2	1	24	96
styr	9	10	30	165
keyb	7	2	19	170
scf	27	54	128	168

Table 6.1: Statistics of Benchmark Examples

and the fault coverage obtained (fault cov.) by STALLION are given in Table 6.2. The CPU times for logic optimization (l.o. time), test generation (TPG time) and the number of test sequences (test seq.) generated are also given. All the undetected 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) and redundancy removal (r.r. time) are given in Table 6.2. The CPU times for state assignment and the initial state minimization were negligible and 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. The larger examples, **scf** and **planet** which have significantly more outputs than latches are fully testable.

The redundancy identification times in Table 6.2 represent the CPU times required to explicitly identify redundant lines in the given circuit. Explicitly removing these redundancies in order to obtain a fully testable circuits requires considerably more CPU time as indicated in Table 6.2 (r.r. time). This method is only feasible for small examples.

The number of test sequences generated for each example is comparable to the number of single test vectors generated via a Complete Scan Design approach. However, each test sequence has multiple test vectors (between 1-10) that have to be applied to the PI lines. In the Scan Design case, each test vector requires multiple clock cycles to be applied.

The examples of Table 6.2 with less than 100% fault coverage were re-synthesized using the extended don't care set as described in Section 6.4.3. The CPU time to check for

EX	#lat.	#edges	#fault cov.	l.o. time	TPG time	test seq.	%red fault	r.i. time	r.r. time
ex1	3	23	97.92	0.5s	2.0s	19	2.08	1.1s	2.0s
ex2	5	35	98.15	2.2s	41.8s	22	1.85	6.1s	1.8m
bbara	3	56	100.0	1.2s	104.8s	42	0.0	0.0s	0.0s
bbsse	4	91	100.0	2.1s	3.2m	46	0.0	0.0s	0.0s
s1	5	105	99.79	5.5s	303s	74	0.21	4.0s	303s
planet	6	193	100.0	10.5s	141.8s	80	0.0	0.0s	0.0s
dfile	6	77	97.80	6.2s	331.8s	62	2.20	41.8s	> 1h
styr	5	367	100.0	80.4s	42.1m	165	0.0	0.0s	0.0s
keyb	5	146	98.65	29.5s	21.2m	101	1.35	1.2m	> 1h
scf	8	402	100.0	121.4s	82.2m	136	0.0	0.0s	0.0s

Table 6.2: Synthesis Procedure Results

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
keyb	14.6s	2	27.8s	100.0	21.6m

Table 6.3: Results using Extended Don't Care Sets in Synthesis

equivalence between invalid and valid states (s.e. time), number of logic minimizations (#logic mini.), CPU time spent in logic minimization (l.o. time), the final fault coverage (fault cov.) using STALLION and the test generation time (TPG time) are indicated in Table 6.3. The CPU time required for the state equivalence checks and the extra logic minimization steps are less than sequential test generation and redundancy removal times (Table 6.2), indicating that the optimal synthesis procedure is more efficient than an explicit redundancy identification method. Using the simple don't cares (**Step 3** in Section 6.4.3) resulted in fully testable designs in all cases. An example has yet to be found where this is not the case.

6.5 Fully and Easily Testable Sequential Machines

In this section, a synthesis procedure of constrained state assignment and logic optimization to produce fully and easily testable finite state machines is presented. In contrast to the irredundant synthesis procedure in the previous section, extra logic is used to ensure fully testability and alleviate the difficulty in test generation for sequential machines. This work was also described by Devadas [73] and can also be found in [74].

The gate-count penalty incurred due to the constraints on the optimization is small. The performance of the synthesized design is usually *better* than a unconstrained design optimized for gate-count alone. The testing time for faults in the combinational logic is smaller than the testing time using a scan design methodology. The faults in the latches, however, are not guaranteed to be detected using these test sequences.

Results obtained on benchmark examples show that the area penalties incurred because of greater gate-count due to the constraints imposed during state coding and logic optimization are small. The performance of the resulting circuits is *better* than that of unconstrained designs optimized for minimum area (This is because one of the constraints imposed requires combinational logic partitioning in the machine).

The relationship between state assignment and testability is discussed in Section 6.5.1. In Section 6.5.2, the necessary conditions required for a fully testable Moore machine is

stated. Extensions to Mealy machines are made in Section 6.5.3. In Section 6.5.4, how an existing state assignment algorithm can be modified to produce a constrained encoding satisfying the testability criterion is discussed. The procedures described in Sections 6.5.2 and 6.5.3 are extended to handle cascades of state machines in Section 6.5.5. Results are presented in Section 6.5.6.

6.5.1 Relationship between State Assignment and Testability

State assignment can has a profound impact on the testability of a sequential machine [74]. The effect of state assignment on testability for a Moore finite state machines will be discussed here. However, the discussion can easily be extended to Mealy machines. A general model for a Moore finite state machine is shown in Figure 6.7. It is realized by two logic blocks, the Output Logic (*OL*) block and the Next State Logic (*NSL*) block, and registers. In a Moore machine, the outputs depend only on the present state (or the outputs of the registers) of the machine. The output logic block receives inputs from all, or a subset of the outputs of the memory elements. It is assumed that the machine has been synthesized from a State Transition Graph specification and that both the *OL* and *NSL* blocks are combinationally irredundant.

Faults in the *OL* block can be detected by justification sequences to any state which propagates the effect of the fault to the primary outputs. For any fault in the irredundant *NSL* block, there exists a state s and an input vector i which propagate the effect of the fault to the next state lines. A faulty next state q^F instead of the fault-free (true) next state q is obtained. q and q^F need to be distinguished at the primary outputs. If q and q^F have different outputs, the fault will be detected in the next clock cycle. If q and q^F are equivalent states in the faulty machine, the fault cannot be detected.

The codes of q and q^F will differ in as many bits as the number of next state lines that the fault has been propagated to and will be identical in the remaining bits. If it can be ensured via state assignment that any two states produced as a faulty fault-free pair are not

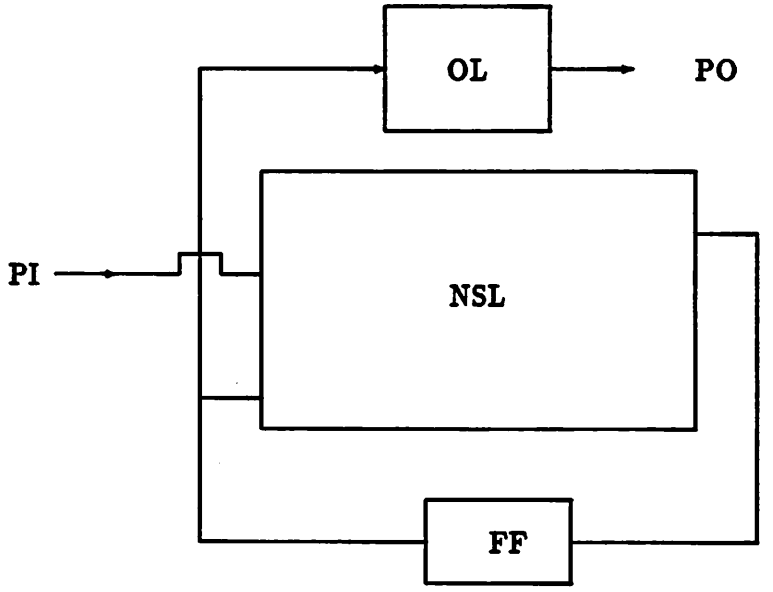


Figure 6.7 General Moore Machine Model

equivalent (in the faulty machine) then any fault which is propagated to the next state lines will always be detectable at the primary outputs. To do this, the faulty next state corresponding to a given fault and a fault-free next state must be restricted to belong to a small set of states.

6.5.2 Fully Testable Moore Machines

The conditions for a general Moore machine to be fully testable will be first presented.

Theorem 6.9: Given a n-latch logic-level implementation of a Moore machine (shown in Figure 6.7), if (1) the combinational logic blocks *OL* and *NSL* are irredundant (2) the machine is R-reachable i.e. all 2^n states are reachable from the reset state and (3) all 2^n states have distinct outputs, the machine is fully testable for all stuck-at faults in *OL* and *NSL*.

Proof: Consider a fault, F, in the *OL* block. Since the block is irredundant (Condition 1), a

state, s, exists which detects F. This state, s, can be reached from the reset state, R, of the machine via an input sequence, I, because the machine is R-reachable (Condition 2). State s will be reached on applying I from R regardless of F since F is in the *OL* block. Therefore, a sequence exists, namely I, which can detect F.

Now consider a fault, F in the *NSL* block. Again, since *NSL* is irredundant, a state, s, and an input i exist which propagate the effect of this fault to the next state lines. Instead of obtaining the true next state, q, a faulty next state q^F is obtained. q and q^F have distinct outputs (Condition 3). Therefore, at the next clock cycle the effect of F is propagated to the primary outputs. s have to be reached from R. A path always exists from s to R (Condition 2). However, this path may or may not have been corrupted by F. If the path has not been corrupted, F can be detected after reaching s and applying input i. If the path has been corrupted, it means that for some edge in the path, the next state reached was different due to F. In this case, the fault is detected even *before* reaching s, since two different states were reached in the faulty and fault-free machine. □

Stuck-at faults on the primary inputs and the present state lines can produce faulty next states that are greater than distance 1 from the true next states. However, it is proved in Section 6.4 that these faults always produce a faulty machine that is distinguishable from the true machine (these faults are usually the easiest to detect in the machine).

The implications of each of the conditions of Theorem 6.9 is now analyzed. Condition (1) is necessary because a redundant fault in *NSL* or *OL* cannot be detected in the sequential machine. Redundancies are sometimes introduced for performance reasons, but mostly they are due to unoptimized logic [24]. An irredundant logic network would have minimal gate-count. With recent advances in multi-level logic optimization, large networks can be made irredundant.

In general, STG specifications of machines have reset states and are R-reachable. However, a STG specification of a machine need not necessarily have $N_s=2^k$ states, $k=1, 2...$

Given the number of encoding bits to be used, n ($n \geq \lceil \log(N_s) \rceil$), the number of states in a STG can be raised to 2^n. These new states need to be reachable from the reset state to satisfy the R-reachability condition. Given a single unspecified transition edge (minterm or cube) from a single state in the original STG, edges can be added to the STG so as to ensure that all the added states are reachable (If the machine is completely specified, an extra input has to be added). Most STGs encountered in practical design have a large number of transitions that are not specified.

Condition 3 is obviously unacceptable, since if the STG specification does not satisfy it, it cannot be made to do so without changing the functionality of the machine. This condition is now relaxed.

Consider the logic-level implementation of the Moore machine shown in Figure 6.8. The *NSL* block has been realized as n distinct single-output circuits or *partitions*. Each distinct cone circuit and the output logic block receive inputs from all or a subset of the outputs of the registers. The following theorem shows that a *constrained state assignment* can ensure a fully testable circuit.

Theorem 6.10: Given a n-latch logic-level implementation of a Moore machine (shown in Figure 6.8), if (1) the combinational logic blocks *OL* and NSL_i, i=1, 2 .. n, are irredundant, (2) the machine is R-reachable and (3) the state encoding of the machine is such that each pair of states asserting the same output has codes of at least distance-2 apart from each other, the machine is fully testable.

Proof: The faults in the *OL* block are detected as before in Theorem 6.9. Consider a fault F in the *NSL* block. Without loss of generality, assume that F is in the first partition. The effect of the fault when detected is to produce a 0 (1) instead of a 1 (0) at the NSL_1. In either case, the faulty next state produced, q^F, will differ from the true state, q, in at most one bit. Since state assignment has guaranteed that all states asserting the same outputs have been

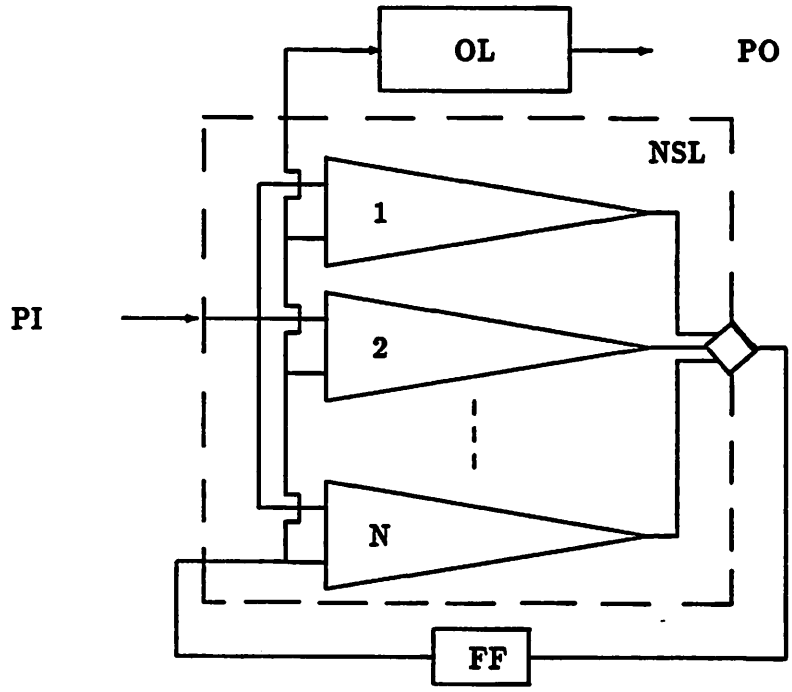


Figure 6.8 Partitioned Moore Machine Model

assigned at least distance-2 apart codes, q and q^F assert different outputs. This means that F is detected in the next clock cycle. □

A realization of a machine like the one shown in Figure 6.8 implies that logic cannot be shared between next state lines. Thus, a certain area penalty may be associated with such an implementation. The performance of the circuit does *not* suffer due to logic partitioning (and in fact may be improved). However, the implementation shown is an extreme case and can be generalized. A partition may contain more than one NSL_i. This means that the logic between these lines *can* be shared.

The number of *NSL* partitions required is related to the number of states asserting the same output in the original STG. It will be shown that the state assignment constraint (Condition 3 of Theorem 4.2) can be satisfied quite easily.

Lemma 6.4: Given k bits, $k \geq 2$, the 2^k possible codes can be split into 2 sets each of cardinality of 2^{k-1}, such that codes within each set are at least distance-2 apart.

Proof: The proof of the lemma is based on the method of mathematical induction. The lemma is obviously true for $k = 2$. Now assume the lemma is true for $k = n-1$. The 2^{n-1} codes can be split into 2 sets, A_{n-1} and B_{n-1}, each of cardinality of 2^{n-2} and codes within each set are at least distance-2 apart. Now, let $k = n$. Two sets, A_n and B_n, each of cardinality of 2^{n-1}, can be formed using elements of A_{n-1} and B_{n-1}. A_n is formed by appending a 0 to each element in A_{n-1} and a 1 to each element in B_{n-1}. B_n is formed by appending a 1 to each element in A_{n-1} and a 0 to each element in B_{n-1}. A_n and B_n are of cardinality of 2^{n-1} and codes within each set are at least distance-2 apart. □

The following result gives us the required number of partitions of the *NSL* lines as a function of the number of states with the same output.

Theorem 6.11: If at most k states exist in a State Transition Graph which produce the same outputs, $\lceil log(k) \rceil + 1$ separate partitions suffice to obtain a fully testable machine.

Proof: In the worst possible case, if there are 2^n states in the machine, there exists $\lfloor \frac{2^n}{k} \rfloor$ sets of states and the states within each set asserting the same output.

It is required that, for each set, no two of these k states are ever produced as a fault-free faulty pair due to a fault in *NSL*. This means that the codes assigned to any two of these states must differ in at least two next state lines belonging to two distinct partitions. By Lemma 6.4, the number of bits required to generate 2 sets of 2^{p-1} at least distance-2 apart codes is p. To generate 2 sets of k codes of at least distance-2 apart, $\lceil log(k) \rceil + 1$ partitions

is required. There are $n - (\lceil log(k) \rceil + 1)$ bits remaining. This means there can be

$$2^{n-\lceil log(k) \rceil - 1} \times 2 = \frac{2^{n-1}}{2^{\lceil log(k) \rceil}} \times 2 = \lfloor \frac{2^n}{k} \rfloor$$

sets each with k codes which differ in two next state lines belonging to two distinct partitions. □

There are thus three steps in producing combinational logic specifications for *OL* and *NSL* blocks from a State Transition Graph description. These steps are (1) raising the number of states in the State Transition Graph to 2^n, where n is the number of latches (2) obtaining constraints for the state assignment on the basis of state outputs and (3) state assignment obeying the constraint relations generated. A straightforward solution exists for Steps 1 and 2, however the optimality of the eventual implementation depends on the choices made during these steps. For example, in Step 1, transition edges connecting original states in the STG to the new states can be added in a variety of ways. The new states can be connected in a chain or separately connected from the original states. Similarly, if the number of required partitions is less than the number of next state lines, choices exist for next state lines that can be grouped together. Next state lines which can share logic maximally should be placed in the same partition. In Step 3, an optimal state assignment which minimizes combinational logic while meeting the distance constraints has to be found. This step is further discussed in Section 6.5.4.

After obtaining the combinational logic specifications, logic optimization algorithms which can ensure an irredundant logic network (e.g. [24]) can be applied. If redundancies are required in the logic, this synthesis procedure ensures that all combinationally irredundant faults are sequentially irredundant as well.

To generate tests for the sequential machine, test vectors are generated for all stuck-at faults in the *OL* and *NSL* combinational circuits. Then, justification paths are obtained from the STG using simple breadth-first search. It is guaranteed (by the theorems proved in this

section) that these paths concatenated with the test vectors applied to the primary inputs of the sequential machine will detect all stuck faults in the machine at the primary outputs.

This procedure has ensured that a faulty state is always propagated to the primary outputs in a single clock cycle via state assignment. This can, in fact, be generalized to multiple-vector propagation. That is, state assignment constraints can be derived which ensure that a faulty state is propagated to the primary outputs in at most P clock cycles ($P \geq 1$). A state assignment algorithm can construct an optimal encoding which satisfies these constraints. For large P, the constraints are less stringent but more difficult to state succinctly.

A re-statement of Condition 3 in Theorem 6.10 to ensure full testability via P-vector propagation sequences can be made. The re-statement for $P=2$ is given below.

The state encoding of the machine should be such that each pair of states asserting the same output should have codes at least distance-2 apart or for each pair of states, q_1 and q_2, which assert the same outputs and have uni-distant codes, the following should hold. Assume that q_1 and q_2 differ in bit i. An input combination should exist which drives the fault-free machine from q_1 and q_2 to states s_1 and s_2, respectively, such that

(1) s_1 and s_2 assert different output and

(2) s_2', which is the state that differs from s_2 in bit i alone, asserts a different output from s_1.

6.5.3 Fully Testable Mealy Machines

In a Mealy machine, the outputs depend on both the present state as well as the primary inputs. A model for a Mealy machine with each next state line realized as a separate circuit and with the output and next state logic separated is shown in Figure 6.9.

A theorem in direct correspondence to Theorem 6.10 for Mealy machines will be proved. First, the notion of O-equivalence for a pair of states of a Mealy machine is defined.

Definition 6.2: Two states in a Mealy machine are said to be **O-equivalent** if for any primary

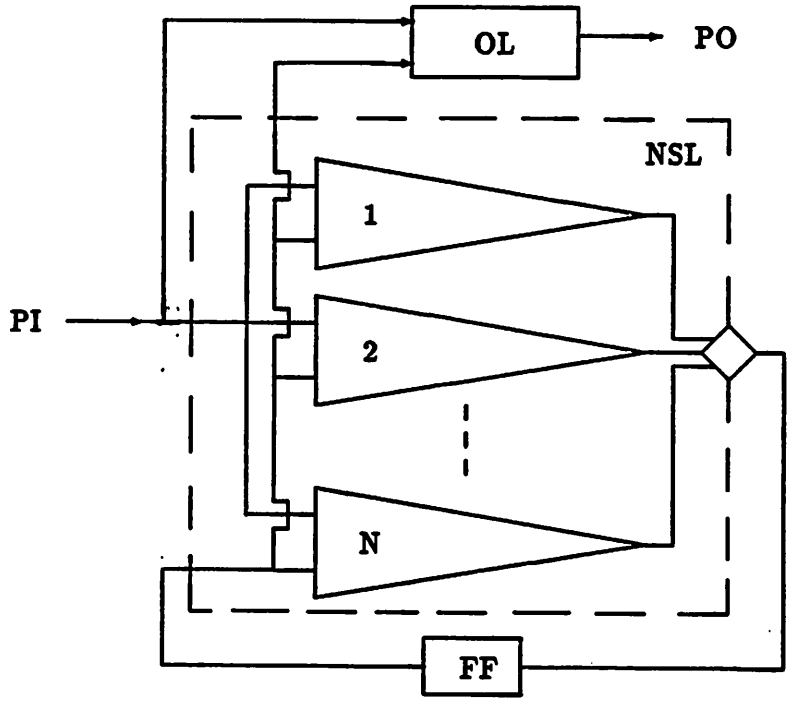


Figure 6.9 Partitioned Mealy Machine Model

input combination the two states produce the same output.

Theorem 6.12: Given a n-latch logic-level implementation of a Mealy machine (shown in Figure 6.9), if (1) the combinational logic blocks OL and NSL_i, i=1, 2 .. n, are irredundant (2) the machine is R-reachable i.e. all 2^n states are reachable from the reset state and (3) if the codes of states of the machine are such that each pair of O-equivalent states have codes of distance-2 from each other, the machine is fully testable.

Proof: Consider a fault F in the OL block. There exists a state, s and input i which detects this fault by Condition 1. R-reachability and the fact that F is in the OL block imply that

state s can be reached from R. F can thus be detected.

Consider a fault F in the *NSL* block. Without loss of generality, assume that F is in the first partition. Since this partition is irredundant, a state s and an input i_1 exist which can propagate the effect of the fault to the next state line. The effect of the fault when detected is to produce a 0 (1) instead of a 1 (0) at the NSL_1. In either case, the faulty next state produced, q^F, will differ from the true state, q, in at most one bit. Condition 3 guarantees that q and q^F are *not* O-equivalent since all O-equivalent states have distance-2 codes. This means that an input, i_2, exists which will produce a different output in the faulty machine (which is in q^F) from the fault-free machine (which is in q). However, s has to be reached from R. A path exists from s to R (Condition 2). However, this path may or may not have been corrupted by F. If the path has not been corrupted, F can be detected after reaching s and applying input i_1 followed by i_2. If the path has been corrupted, it means that for some edge in the path, the next state reached was different due to F. There exists a fault-free/faulty pair (q', q'^F). By the argument above, an input i_3 which produces a different output for q' and q'^F exists, thus detecting F. □

The synthesis procedure for obtaining a fully testable Mealy machine is the same as the procedure outlined for the Moore machine in Section 6.5.2. To generate tests for the machine, as before, all the combinational logic tests for the *OL* and *NSL* blocks are generated. The justification path to the state detecting the fault concatenated with the primary input part of the combinational test vector and the differentiating input vector (for the fault-free/faulty next state pair) constitutes the test sequence for a given fault.

A re-statement of Condition 3 in Theorem 6.12 to ensure full testability via P-vector propagation sequences can be made as in the Moore machine case. The re-statement for P=2 is given below.

The state encoding of the machine should be such that each pair of O-equivalent states should have codes at least distance-2 apart or for each pair of states, q_1 and q_2, which are O-

equivalent and have uni-distant codes, the following should hold. Assume that q_1 and q_2 differ in bit i. An input combination should exist which drives the fault-free machine from q_1 and q_2 to states s_1 and s_2 respectively, such that

(1) s_1 and s_2 are not O-equivalent and

(2) s_2', which is the state that differs from s_2 in bit i alone, is not O-equivalent to s_1.

6.5.4 Constrained State Encoding

State assignment is the process of assigning binary codes to the internal states of a finite automaton. The problem of optimal state assignment is to find an encoding of states which minimizes the combinational logic part of the sequential machine.

The combinational logic part of the sequential machine can be implemented using a Programmable Logic Array (PLA) or using multi-level logic. State assignment techniques targeting both these implementations have been proposed (e.g. [68] [69]). The program MUSTANG [69] produces a state assignment that heuristically minimizes the number of literals in the combinational logic *after* multiple-level logic optimization.

The technique used by MUSTANG is based on maximizing common factors in the logic in an attempt to reduce the gate-count of the network. A weighted graph whose nodes represent states of the machine is constructed. The weights between the edges in the graph reflect the "gains" in coding the corresponding states with uni-distant codes.

An embedding algorithm is used to assign binary codes to the states (nodes in the graph) so as to maximize the overall gain. The algorithm iteratively selects groups of states to be encoded. These states are given minimally-distant codes from the unassigned codes.

For the constrained state assignment problem in the synthesis procedure, the graph construction part remains the same. During embedding, when a group of states is selected, they are checked for distance-2 constraints. A minimally-distant set of codes satisfying these constraints is then assigned to the states. Detail discussion of the state encoding problem can be found in [69].

6.5.5 Fully Testable Cascaded State Machines

The procedures described in Sections 6.5.2 and 6.5.3 are relevant when a single finite state machine is being synthesized. In most industrial chip designs, cascaded state machine implementations are typical.

One sequential machine may drive another, as shown in Figure 6.10, and the intermediate inputs/outputs may not be directly observable. Primary inputs to the driven finite state machine (Π_2) and primary outputs from the driving machine (PO_1) may or may not exist. A procedure which does not assume the existence of either Π_2 or PO_1 will be described since their absence is the worst case of minimum controllability and observability.

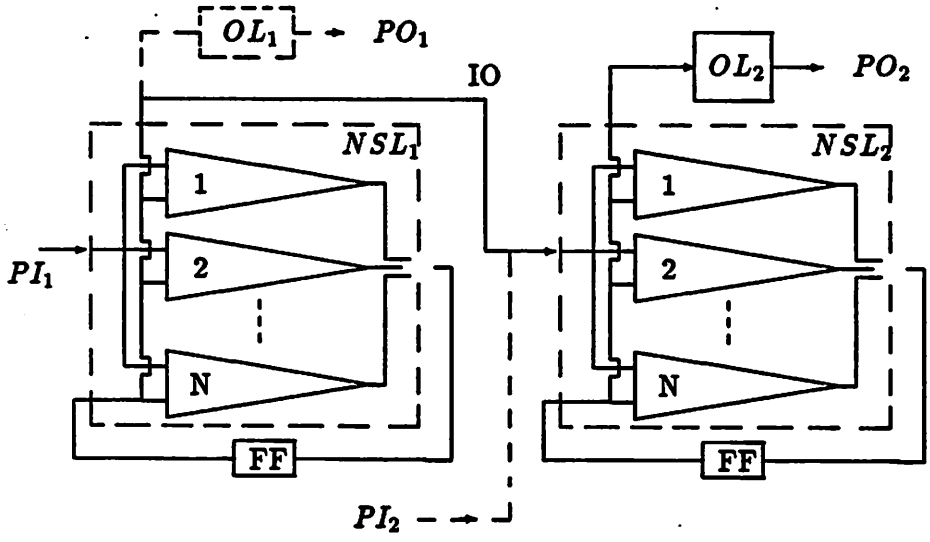


Figure 6.10 Cascaded Moore Machines

The synthesis procedure summarized in Theorem 6.10 will be extended for a cascade of Moore machines. The same can be done for Mealy machines.

It is assumed there exists a *single* reset line which resets both M_1 and M_2 to corresponding reset states, R_1 and R_2. If M_1 and M_2 run on the same clock, it is required that the clock signal to M_2 can be disabled by some means such that M_2 can remain in whatever state it is in regardless of the IO signal from M_1 (Figure 6.10). The following result summarizes the constrained synthesis procedure.

Theorem 6.13: Given the State Transition Graph descriptions of the driving and driven Moore machines with n_1 and n_2 latches, respectively, each with the partitioned architecture shown in Figure 6.10, if (1) all 2^{n_1} states in M_1 are reachable from R_1 and all 2^{n_2} states in M_2 are reachable from R_2 via some primary input sequence (2) all states in M_2 which assert the same outputs are given codes at least of distance-2 (3) all the combinational logic blocks are irredundant and (4) all states in M_1 (corresponding to IO) which for any present state in M_2 produce next states that assert the same output vector are given codes of at least distance-2 apart, then the cascade implementation is fully testable (controllable from Π_1 and observable from PO_2).

Proof: Consider a fault, F, in the OL block of M_2. By Condition 3, a state q_2 exists in M_2 which detects this fault. This state can be reached from R_2 by Condition 1. Thus, F can be detected by any justification sequence for q_2, since the sequence cannot be corrupted by F (F is in OL).

Next, consider a fault, F, in the NSL block of M_2. Without loss of generality, assume that the fault exists in the first partition. Again, a state q_2 in M_2 and an input vector io_1 corresponding to a state q_1 in M_1 exist which propagate the effect of this fault to the next state lines of M_2. M_2 can be placed in q_2 and then the clock signal to M_2 is disabled until M_1 is placed in q_1. F can be propagated to the NSL_1 of M_2 producing a fault-free faulty

state pair (s, s^F) in M_2. s and s^F differ in the first bit alone, and are hence of distance-1. By Condition 2, s and s^F have different outputs. Thus, F is detected at PO_2 in the next clock cycle if the justification sequence for q_2 is not corrupted by F. If the justification sequence for q_2 has been corrupted by F, it means that, somewhere along the justification path, the next states in the fault-free and faulty machine are different. In this case, F is detected even *before* reaching q_2, since a fault-free faulty state pair is always propagated to the primary outputs by the above reasoning.

Finally, consider a fault, F, in the *NSL* block of M_1. Without loss of generality, assume that it exists in the first partition. A state q_1 in M_1 is required to propagate the effect of F to NSL_1 of M_1. This state q_1 can be reached from R_1. Once again, the property of the fault-free faulty state pair, (s, s^F), in M_1 is that they are of distance-1. By Condition 4, whatever state M_2 is in, the fault-free faulty next state pair produced in M_2 due to the application of (s, s^F) will assert different outputs. The justification path for q_1 may be corrupted by F. In this case, F is detected even before reaching q_1. □

To obtain test sequences for the faults in the logic blocks of M_1 and M_2, combinational test vectors to propagate these faults to the next state lines or outputs, are found. Justification sequences for the states corresponding to these test vectors are found via breadth-first search on the State Transition Graph description of the machine. By Theorem 6.13, these sequences detect all single stuck-at faults in M_1 and M_2, observable at PO_2.

The implications of Conditions 1, 2 and 3 have been analyzed in Section 6.5.2. Experimental results indicate that the area penalties due to these constraints are small. Performance is enhanced rather than penalized by logic partitioning. The state assignment constraint for M_1 (Condition 4) has to be analyzed.

If a single machine were to be used to implement the cascade, it would essentially be a product of the two machines. The number of distinct output patterns is the same for the single composite machine or the cascade. The driven machine will have to assert all the output

patterns and therefore will have significantly fewer states asserting the same outputs relative to the single machine. Therefore, the constraint can be satisfied quite easily.

If M_2 is controllable from the primary inputs, i.e. if Π_2 (Figure 6.10) exists, Condition 4 can be re-stated as follows – all states in M_1 (corresponding to IO) which, for *any* present state in M_2, produce next states that assert the same output vector, PO_2, for *all input combinations* Π_2, have to be given codes at least of distance-2. The condition is more relaxed if M_2 is more controllable.

Similarly, if some outputs of M_1, namely PO_1 (Figure 6.10), are directly observable, then all the faults in the NSL block of M_1 can be detected at PO_1 by obeying state assignment constraints in M_1 similar to Condition 2 (rather than the constraints specified by Condition 4).

All the next state lines in M_1 and M_2 need not be realized as separate circuits, some of them can be merged. That is a partition may contain more than one NSL_i. The number of distinct partitions required is a function of how many states have to be at least distance-2 from each other (Section 6.5.2). The procedure can be extended to handle multiple cascades of machines, which may be Moore or Mealy machines.

6.5.6 Results

Results obtained on five State Transition Graphs from the MCNC 1987 Logic Synthesis Workshop benchmark set, whose statistics are given in Table 6.4, are given in Table 6.5. First, the machines were encoded and optimized disregarding testability. The number of gates in the machine, the fault coverage obtained and the test generation time are given in Table 6.5 under the column labeled OPTIMIZE. Test generation was accomplished using an efficient test generation algorithm that was recently proposed [26]. In Table 6.5, m stands for CPU-minutes and s for CPU-seconds on a VAX 11/8650. All machines were resynthesized using the procedure described in Sections 6.5.2 and 6.5.3. Again, the number of gates, fault coverage obtained and the test generation time are given. Sequential test generation for these

circuits was faster because combinational test generation and breadth-first search suffice to produce the test sequences. The CPU times required for all the examples in both synthesis procedures are virtually identical, and the maximum CPU time is less than 6 minutes. The example scf is a Moore machine, the others are Mealy machines.

It can be observed from Table 6.5 that the number of gates needed for testable designs is usually larger than that of unconstrained optimized designs. The gate-count penalties incurred are due to three reasons : (1) the constraints imposed during state assignment (2) the addition of extra edges to the STG to obtain R-reachability and (3) logic partitioning constraints. Empirical evidence has shown that (3) is easily the most significant factor – the next

EX	#inp	#out	#states	#latches	#edges
sse	7	7	13	4	59
tbk	6	3	16	4	787
scf	27	54	97	7	168
dfile	2	1	24	5	99
planet	7	19	48	6	118

Table 6.4: Statistics of Benchmark Examples

EX	I - OPTIMIZE			II - TESTABLE		
	#gates.	fault cov.	tpg time	#gates	fault cov.	tpg time
sse	91	84.57	69.9s	129	100.0	5.2s ne
tbk	181	98.57*	72.1s	231	98.57*	4.1s ne
scf	502	96.14	83.1m	541	100.0	71s ne
dfile	124	96.94	104s	144	100.0	2.0s ne
planet	417	98.82	373s	449	100.0	14s ne

* OL block was not combinationally irredundant

Table 6.5: Synthesis for Testability Results

state lines may have to be realized as separate circuits. Additionally, for a Mealy machine, unlike in an unconstrained design, the next state and the output logic have to be separated.

Logic partitioning is extensively used to gain higher performance. A Mealy machine with separate next state and output logic blocks can be clocked faster than a machine with a single lumped block of logic. This is the case in the example designs of Table 6.4 as well. Thus, the fully testable machines produced by logic partitioning may provide a better gate-count/performance trade-off.

The number of gates in a circuit is, in general, indicative of the area required to implement the circuit. However, in some cases, this measure of area may not be very accurate. To obtain accurate estimates of circuit areas, the synthesized examples of Table 6.4 and 6.5 were placed and routed using the TimberWolf standard cell placement and routing package [75]. The areas of the resulting designs after place and route for the unconstrained and constrained cases are given in Table 6.6. For each example, the areas of the designs have been normalized to that of the unconstrained design.

In Table 6.6, some constrained designs are about the same size or smaller than the corresponding unconstrained ones. Logic partitioning, in these cases, has decreased routing complexity to the extent of balancing the increase in the number of logic gates. The cost

EXAMPLE	I - OPTIMIZE		II - TESTABLE	
	#gates	area	#gates	area
sse	91	1.0	129	1.34
tbk	181	1.0	231	1.10
scf	502	1.0	541	1.01
dfile	124	1.0	144	0.98
planet	417	1.0	449	0.86

Table 6.6: Areas of Standard Cell Designs

function used in multi-level logic optimization is the number of literals (transistors) [3], which may be a poor estimate of the circuit area.

The number of test sequences required varied between 30-70 for these examples. The number of test sequences can be reduced by applying combinational test compaction strategies after generating all test vectors for the combinational logic blocks. The average length of a sequence was 5. Since the test vectors only access the primary inputs and only the primary outputs are observed, each vector can be applied in one clock cycle.

6.6 Easily Testable PLA-based Finite State Machine

Programmable Logic Arrays (PLAs) are used extensively in the design of complex VLSI systems. Sequential functions can be realized very efficiently by adding feedback registers to the PLA. Numerous programs for the optimal synthesis of PLA-based finite state machines have been developed (e.g. [68] [69]). Test generation and design-for-testability techniques for PLA structures have been active areas of research.

Due to a PLA's dense layout, PLA faults other than conventional stuck-at faults can occur easily and must be modeled. An extended model, the *crosspoint fault* model, has been proposed in [76] and [77]. The crosspoint-oriented test set covers many of the frequently occurring physical faults, including shorts between lines. Several PLA test generation techniques aimed at the crosspoint fault model have been proposed (e.g. [78] [79]). In particular, an exact and efficient technique which guarantees maximum fault coverage and identification of all redundant faults was proposed in [80].

Design-for-testability techniques (e.g. [81]) for PLAs require controllability of all inputs and observability of all outputs of the PLA. Synthesis approaches to producing easily testable sequential machines, without requiring direct access to the inputs/outputs of the circuit's memory elements, have not been aimed at the crosspoint fault model.

In this section, a synthesis procedure, which beginning from a State Transition Graph description of a sequential machine, produces an optimized easily testable PLA-based logic

implementation is outlined [28]. A procedure of constrained state assignment and logic optimization which guarantees testability for all combinationally irredundant crosspoint faults in a PLA-based finite state machine is proposed. No direct access to the flip-flops is required. The test sequences to detect these faults can be obtained using combinational test generation techniques alone. This procedure thus represents an alternative to a Scan Design methodology. Empirical results shows that the efficacy of this procedure – the area/performance penalties in return for easy non-scan testability are small.

6.6.1 Crosspoint Faults

The following faults are considered in the crosspoint fault model.

(1) Growth/Missing contact faults in the input plane

(2) Shrinkage/Extra contact faults in the input plane

(3) Appearance/Extra contact faults in the output plane

(4) Disappearance/Missing contact faults in the output plane

(5) Output stuck-at-one faults

Except for fault type 5, essentially two types of faults are present, namely, the *missing contact* and *extra contact* faults. In the input plane, an additional contact on a row reflects an additional constraint placed on the cube corresponding to the row and has the effect of shrinking the set of vertices covered by the cube. On the contrary, a missing contact in the input plane removes a constraint and thus expands the set of vertices covered by the cube. A missing contact on the ith column of the output plane reflects a removal of a cube from the ON-set cover of the ith output function. The effect is then the shrinkage of the ON-set of that ith output function. By the same token, an extra contact in the output plane adds an additional cube to the output ON-set cover and thus enlarges the ON-set. In the sequel, to adopt a unified point of view on these faults, fault types 1 and 3 are called *GROWTH* faults and fault types 2 and 4 are called *SHRINKAGE* faults.

6.6.2 Easily Testable PLA-based Finite State Machines

Synthesizing a logic-level implementation of a finite state machine from a State Transition Graph description involves the steps of state minimization, state assignment and logic optimization. All three steps have a profound effect on the testability of the resulting logic implementation.

In order to detect a fault in a sequential machine, the machine has (1) to be placed in a state that can excite the fault and (2) the effect of the fault has to be propagated to the primary outputs. State assignment does not affect the first step, i.e. state justification but can have a profound effect on the second step of fault propagation.

PLA-based Mealy finite state machines will be considered, since a Mealy machine can be viewed as a more general case of a Moore machine. The PLA implements both the output logic and next state logic functions. Combinationally irredundant crosspoint faults in the PLA will be focused on – combinationally redundant faults cannot be made testable in a sequential machine even using full Scan Design or via state assignment.

For any combinationally irredundant crosspoint fault, a present state, s, and a primary input vector, i, exist, which can propagate the effect of the fault to the next state lines (NS) or the primary outputs (PO). If the effect of the fault is propagated to PO, then the fault can be detected in the non-scan sequential machine via a justification sequence for s. That is, when the machine is in s, applying i will detect the fault. On the other hand, if the effect of the fault is propagated to NS but not PO, then a faulty next state q^F instead of the fault-free (true) next state q is obtained. q and q^F need to be distinguished at the primary outputs. If q and q^F are equivalent states in the faulty machine then the fault cannot be detected.

Depending on the type of crosspoint fault under test, the codes of q and q^F will have certain relationships. If it can be ensured via state assignment that any two states produced as a faulty fault-free pair are not equivalent (in the faulty machine) then any fault which is propagated to the next state lines will always be detectable at the primary outputs. The synthesis

procedure to be described in the next section does precisely this, in order to ensure testability for all combinationally irredundant crosspoint faults in the sequential machine.

6.6.2.1 The Synthesis Procedure

Definition 6.3: Two minterms m_1 and m_2 are said to be mutually-dominant if $m_1 \supset m_2$ or $m_2 \supset m_1$. Two minterms m_1 and m_2 which are not mutually-dominant are said to be mutually-nondominant if $m_1 \neq m_2$.

Lemma 6.5: For any kind of irredundant crosspoint fault in a PLA, the faulty output vector and the true output vector are mutually-dominant.

Proof: Consider a fault, F, in the PLA. If the fault is a *GROWTH* fault, then F adds to the ON-set of some outputs, but does not subtract from the ON-set of any output. Therefore, if F is detected by some input vector i, then for some subset of the outputs whose true value is 0 for i, the faulty value is 1. Outputs whose true value is 1, remain at 1. This means that o^F, the faulty output vector for i, dominates o, the true output vector for i.

If F is a *SHRINKAGE* fault, then F subtracts from the ON-set of some outputs, but does not add to the ON-set of any output. Therefore, if F is detected by some input vector i, then for some subset of the outputs whose true value is 1, the faulty value is 0. Outputs whose true value is 0, remain at 0. This means that $o \supset o^F$.

Finally, a crosspoint fault of type 5, namely an output stuck-at-one fault if detected will produce a o^F which differs in one bit from o (a 1 instead of a 0). Again, $o^F \supset o$. □

A procedure of constrained state assignment, which ensures that faulty fault-free next state pairs are always propagated to the primary outputs within one clock cycle, is summarized in Theorem 6.14 below.

Theorem 6.14: Given a n-latch logic-level implementation of a PLA-based finite state machine, if (1) the machine is R-reachable and (2) if the state encoding of the machine is

such that each pair of states which do not produce mutually-nondominating primary outputs for at least one primary input vector are assigned mutually-nondominating codes, the machine is testable for all combinationally irredundant crosspoint faults.

Proof: Consider a fault F in the PLA. Since the fault is combinationally irredundant a primary input vector i_1 and a present state s exist which detect the fault at either the primary outputs or at the next state lines. If the fault is detectable at the next state lines, then the faulty next state produced q^F instead of the true next state q are mutually-dominating by Lemma 6.5. By Condition 2, a primary input vector i_2 will exist which will distinguish q^F and q in the next cycle, since states which cannot be distinguished in the true machine are given mutually-nondominating codes and never allowed to appear as faulty fault-free pairs. It is known that s can be justified in the true machine because of Condition 1. However, the justification sequence may have been corrupted by the fault. Also, we have to show that the distinguishing vector i_2 also holds in presence of the fault.

The distinguishing vector i_2 is such that q and q_F produce mutually-nondominating primary output vectors o_1 and o_2 on applying i_2 in the true machine. o_2 may be corrupted due to the fault, the vector produced may be $o_2{}^F \neq o_2$. By Lemma 6.5, $o_2{}^F$ and o_2 are mutually-dominating. Therefore, o_1 has to be different from $o_2{}^F$, since o_1 and o_2 are mutually-nondominating. This means that the distinguishing vector i_2 holds in faulty conditions as well. Note that if o_1 and o_2 were distinct but mutually-dominating this is not the case.

There exists a justification sequence for s, namely I. This path may or may not be corrupted due to F. If the path is not corrupted, F can be detected by applying i_2 on reaching s. If the path is corrupted, it means that for some edge in the path, F has been propagated to the primary outputs or next state lines. If F is propagated to the primary outputs, F is detected even before reaching s. Else, if F has been propagated to the next state lines, a faulty and fault-free next state pair n and n^F is obtained. n and n^F can be distinguished with some input vector i_3 even under faulty conditions. □

Condition 2, which requires mutually-nondominating codes to be assigned to some state pairs can be satisfied quite easily.

Lemma 6.6: Given n bits, there are $C^n_1, C^n_2, .. C^n_{n-1}$ sets of mutually-nondominating codes, where $C^n_p = \frac{n!}{(n-p)!\,p!}$. The maximum number of mutually-nondominating codes given an n bits is $C^n_{n/2}$ if n is even and $C^n_{(n+1)/2}$ if n is odd.

The above lemma considers mutually-nondominating codes with the same number of 1s in each code. Given n bits, there are C^n_p distinct codes, such that each code has p 1s. All these codes are mutually-nondominating (each code will have a 1 in a position where another does not). For example, given a 3 bits, there are following sets of multiple mutually-nondominating codes, (001, 010, 100) and (011, 101, 110), whose cardinalities correspond to C^3_1 and C^3_2 respectively.

In general, State Transition Graph specifications of machines have reset states. However, a STG specification of a machine need not necessarily have $N_s=2^k$ states, $k=1, 2..$ etc. Given the number of encoding bits to be used, n ($n \geq \lceil \log(N_s) \rceil$), the number of states in a STG can be raised to 2^n. These new states need to be reachable from the reset state to satisfy the R-reachability condition. Given a single unspecified transition edge (minterm or cube) from a single state in the original STG, edges can be added to the STG so as to ensure that all the added states are reachable (If the machine is completely specified, an extra input has to be added). Most STGs encountered in practical design have a large number of transitions that are not specified. It should be noted that these extra states may be equivalent to other previously existing states in the STG. State minimality is not required as a condition for easy testability, but we require all states to be reachable.

There are thus three steps in producing a PLA logic specification for the output logic and next state logic functions. This specification is then optimized using a two-level logic minimizer like ESPRESSO [2]. These steps are (1) raising the number of states in the State

Transition Graph to 2^n, where n is the number of latches (2) obtaining constraints for the state assignment on the basis of state fanouts and (3) state assignment obeying the constraint relations generated. A straightforward solution exists for Step 1, however the optimality of the eventual implementation depends on the choices made during this step. For example, in Step 1, transition edges connecting original states in the STG to the new states can be added in a variety of ways. The new states can be connected in a chain or separately connected from the original states. In Step 3, an optimal state assignment which minimizes combinational logic while meeting the dominance constraints has to be found. This step is further discussed in Section 6.6.3.

To generate tests for the sequential machine, test vectors are generated for all irredundant crosspoint faults using a program like PLATYPUS [80]. Then, justification paths are obtained from the STG using simple breadth-first search. These paths concatenated with the test vectors applied to the primary inputs of a non-scan sequential machine will detect all the crosspoint faults in the machine so as to be observable at the primary outputs.

This procedure has ensured that a faulty state is always propagated to the primary outputs in a single clock cycle via state assignment. This can, in fact, be generalized to multiple-vector propagation. That is, state assignment constraints can be derived which ensure that a faulty state is propagated to the primary outputs in at most P clock cycles ($P \geq 1$). A state assignment algorithm can construct an optimal encoding which satisfies these constraints. For large P, the constraints are less stringent but more difficult to state succinctly.

A re-statement of Condition 2 in Theorem 6.14 to ensure testability via P-vector propagation sequences can be made. The re-statement for $P=2$ is given below.

Definition 6.4: Two states q_1 and q_2 are said to be m-distinguishable if a primary input vector exists which produces two mutually-nondominating primary outputs o_1 and o_2 when the machine is in q_1 and q_2 respectively.

The state encoding of the machine should be such that each pair of states which cannot be m-distinguishable should be assigned mutually-nondominating codes *or* the following should hold for any pair of states (q_1, q_2) which are not m-distinguishable and have mutually-dominating codes. An input combination should exist which drives the fault-free machine from q_1 and q_2 to states s_1 and s_2 respectively, such that

(1) s_1 and s_2 are m-distinguishable and

(2) If $q_2 \supset q_1$, then for all $s_2' \supset s2$, s_2' should be m-distinguishable from s_1. Similarly, if $q_2 \subset q_1$, then for all $s_2' \subset s2$, s_2' should be m-distinguishable from s_1.

6.6.2.2 Combinationally Redundant Crosspoint Faults

A two-level or multi-level circuit can be made irredundant for all single stuck-at faults. Such circuits are called prime and irredundant circuits. Logic minimization programs like ESPRESSO can ensure prime and irredundant two-level covers. However, since the crosspoint fault model is a superset of the stuck-at fault model, PLAs implementing prime and irredundant covers may not be testable for all possible crosspoint faults. However, typically a large percentage of crosspoint faults can be made testable via optimization [80]. All crosspoint faults of type 1, 4 and 5 can be guaranteed to be testable via logic minimization.

6.6.3 Constrained State Encoding

Constrained state encoding of the synthesis procedure is performed using the state assignment program MUSTANG [69]. The technique used by MUSTANG is based on maximizing common factors in the logic in an effort to reduce the area of the network. A weighted graph whose nodes represent each state of the machine is constructed. The weights between the edges in the graph reflect the "gains" in coding the corresponding states with uni-distant codes.

An embedding algorithm is used to assign binary codes to the states (nodes in the graph) so as to maximize the overall gain. The algorithm iteratively selects groups of states to be encoded. These states are given minimally-distant codes from the unassigned codes.

For the constrained state encoding problem in the synthesis procedure, the graph construction part remains the same. During embedding, when a group of states is selected, they are checked for mutual-nondominance constraints. A minimally-distant set of codes satisfying these constraints is then assigned to the states. The more complex, but less stringent, constraints given by multiple-vector propagation can also be accommodated.

6.6.4 Results

Results obtained on five State Transition Graphs from the MCNC 1987 Logic Synthesis Workshop benchmark set, whose statistics are given in Table 6.4, are given in Table 6.8. First, the machines were encoded and optimized disregarding testability. The number of product terms in the PLA, the fault coverage obtained and the test generation time are given in Table 6.8 under the column labeled OPTIMIZE. In Table 6.8, m stands for CPU-minutes and s for CPU-seconds on a VAX 11/8650. Then, each of the machines were synthesized using the procedure described in Section 6.6.2. Again, the number of product terms, fault coverage obtained and the test generation time are given. The example scf is a Moore machine, the others Mealy machines. In all cases, single-vector propagation constraints were placed on the state assignment program.

For the optimized machine, sequential test generation was accomplished as follows:

(1) A present state and a primary input vector which propagates the effect of the fault to the primary outputs or the next state lines is found, if such a vector exists, using PLATYPUS [80].

(2) A fault-free justification sequence for the required present state is found via breadth-first search on the State Transition Graph of the machine.

(3) If the fault has been propagated to the next state lines, then a fault-free distinguishing sequence is found for the true and faulty states in the State Transition Graph. Such a sequence may not exist if the true and faulty states are equivalent. If this is the case, a new test vector is generated which produces a different true and faulty state pair, if

possible.

(4) The justification sequence, the combinational test vector and the distinguishing sequence are concatenated to produce a possible test sequence for the fault. The sequence may not be valid because the justification and/or distinguishing sequence may be invalid under fault conditions. The sequence is fault simulated on the circuit to check if the fault is indeed detected at the primary outputs.

(5) If the sequence does not detect the fault, a different distinguishing sequence for the true-faulty state pair is tried, if possible.

A combinationally irredundant fault may not be detected using this procedure because (a) a distinguishing sequence may not exist for a true-faulty state pair since no constraints have been placed on the state assignment and (b) even if a distinguishing sequence exists, it may not hold under fault conditions.

The constrained synthesis procedure ensures that distinguishing sequences always exist and always hold under fault conditions. Test generation for the testable machine was accomplished as follows:

(1) Same as Step 1 described above.

(2) Same as Step 2 described above.

(3) A single distinguishing vector which produces mutually-nondominating outputs is found for each true-faulty state pair (such a vector is guaranteed to exist).

(4) The justification sequences are checked to see if they are valid under fault conditions. If the sequence is valid, a test sequence is constructed by concatenating the sequence with the combinational test vector and the distinguishing vector. If the justification sequence is invalid, and is not a test sequence for the fault by itself, a new distinguishing vector (which is guaranteed to exist) is found for the true-faulty state pair that is generated by the first corrupted edge in the sequence. The shortened justification

sequence concatenated with the distinguishing vector constitutes a test sequence for the fault.

Sequential test generation for the testable machine is faster because typically more than one distinguishing sequence has to be tried to produce a test sequence for the fault in the optimized machine. Also, rather than having to fault simulate the entire test sequence in the optimized machine, only the justification sequences have to be fault simulated in the testable machine.

In all cases, the maximum possible fault coverage was achieved in the testable machine, i.e. all combinationally irredundant crosspoint faults are detectable in the sequential machine. The area penalties incurred are due to two reasons: (1) the constraints imposed during state assignment (2) the addition of extra edges to the STG to obtain R-reachability. As can be seen the area penalties are quite small, and compare favorably to Scan Design approaches. The gain in fault coverage and test generation times more than offsets the area penalty.

The number of test sequences required varied between 70-300 for these examples. The average length of each sequence was 5. Since the test vectors only access the primary inputs and only the primary outputs are observed, each vector can be applied in one clock cycle.

EX	I - OPTIMIZE			II - TESTABLE		
	#prod.	fault cov.	tpg time	#prod.	fault cov.	tpg time
sse	33	89.56	6.4s	36	92.12	3.6s
tbk	56	90.21	22.1s	61	95.83	15.8s
scf	145	93.31	6.2m	154	96.07	3.3m
dfile	51	94.12	16.2s	54	98.81	6.1s
planet	97	91.72	93s	104	95.67	59.5s

Table 6.8: Synthesis for Testability Results

6.7 Conclusions

Sequential synthesis procedures that produces fully and/or easily testable logic implementation of a Moore or Mealy finite state machine from a State Transition Graph description of the machine are presented. The combinational part of the synthesized machine is implemented by multi-level logic or PLA. No direct access to the memory elements is required for testing the synthesized sequential machine.

For optimized fully testable synthesis, 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. The synthesis procedure has no associated area/performance overhead unlike Scan Design methodologies.

For easily testable synthesis, the synthesis procedure involves constrained state assignment and logic optimization. Testing is made significantly easier at the possible expense of extra logic; though experimental results have shown that the area penalty incurred due to the constrains on the optimization is small. Test sequences can be obtained using combinational test generation techniques alone for multi-level logic implementation. For PLA-based implementation, PLA test pattern generation technique and breadth-first search on the State Transition Graph can be used. alone.

CHAPTER 7

Conclusions

In this dissertation, the problem of logic validation has been addressed. Different tools required to ensure that a chip is free of design errors and thoroughly tested were reviewed and new algorithms for implementing these tasks were presented.

Logic verification, the process of checking the Boolean equivalence of two circuits, was presented in Chapter 2. New algorithms based on enumeration-and-simulation approach for combinational logic verification were described. The equivalence of two logic circuits is checked by generating the ON-set and OFF-set covers of one circuit in cube form and simulating the covers on the other one. Logic verification can be a very CPU-intensive task. This makes an efficient parallel implementation highly desirable and so new, parallel logic verification schemes were also described in Chapter 2. A dynamic scheduling scheme using a PODEM-based enumeration algorithm was presented. Circuits of reasonable size have been successfully verified. Circuits with covers of enormous size, i.e. parity checking circuits, in general may require an inordinate amount of CPU time using the enumeration-and-simulation approach. New and interesting approaches [82] [83] based on Binary Decision Diagrams [84] for logic verification will be a rewarding research area.

In Chapter 3, the problem of sequential test generation was dealt with. The difficulties of generating tests for sequential circuits are mainly due to the limited and constrained access to circuits with memory elements. In general, the controllability and observability of a sequential circuit is reduced greatly compared to a combinational one. Scan Design can be used to transform the sequential test generation problem into that of the easier combinational test generation. However, Scan Design may result in unnecessary area/performance penalty.

A new approach based on state-space enumeration for sequential test generation was described in Chapter 3. The new approach involves extracting a partial State Transition Graph and using it in conjunction with fault excitation-and-propagation and state-justification algorithms in generating tests. Tests have been successfully generated for sequential circuits with thousands of gates and tens of latches. For very large sequential circuits, an Incomplete Scan Design approach that can be used together with the sequential test generation process was presented. This approach enables the detection of all detectable single stuck-at faults using significantly lower number of scan latches than Complete Scan Design in many cases. The excitation-and-propagation algorithm in the sequential test generator is based on the PODEM algorithm. Research for more efficient methods that will make better use of the topological information of the circuits to guide the search for the excitation-and-propagation sequence will further speed-up the test generation. More compact forms of representing the sequential behavior of the sequential circuits, instead of using State Transition Graph, may lead to a better state-justification method.

A fault simulator is an essential CAD testing tool. The problems of fault simulation for both single stuck-at and multiple stuck-at fault models are addressed in Chapter 4 and Chapter 5. In Chapter 4, a statistical fault simulator for mixed-level circuits using the single stuck-at fault model is described. Formulae for computing the controllabilities and observabilities of each wire in a network for a test set are presented. Based on the computed probabilistic measures, an accurate projection of the single stuck-at fault coverage for the test set can be obtained with greatly reduced computational complexity compared to deterministic fault simulation. Experiments with several example circuits have provided estimated fault coverage results close to the actual ones.

Fault simulation for multiple faults is much harder than for single faults due to the sheer number of multiple-fault combinations that can exist. A novel implicit fault-simulation approach was presented in Chapter 5. The problem of fault simulation is transformed into an

implicit fault-space enumeration one using a simulation model. A 15-valued simulation method is used to enable vertical implications to be performed. Procedures to speed up the implication processes, both horizontal and vertical, are described. To improve the vertical implication process, the topological information of a circuit is utilized. Results for circuits of size greater than two hundred gates have been obtained within reasonable CPU times. Further work is to research on test generator which generate compact test sets that not only cover most multiple-faults but also do not require excessive CPU time for fault simulation.

Circuits with redundancies can make the test generation task enormously more difficult for both combinational and sequential circuits. The cost of trying to generate tests for redundant faults can be more than 90% of the total test generation time. This makes synthesis methods that can ensure fully testable circuits very attractive. In Chapter 6, synthesis-for-testability procedures that ensure fully and/or easily testable sequential machines were presented. The synthesized machines can either be PLA-based or multi-level logic implementations. The input descriptions of the machines are either State-Transition Graphs or gate-level descriptions. These synthesis procedures involve steps of state assignment, state minimization and logic optimization to produce sequential machines that can be fully tested without direct access to the memory elements. Single machines are considered in the synthesis procedures. Further interesting work is to extend the synthesis procedures for interconnected sequential machines.

REFERENCES

References

1. A. Sangiovanni-Vincentelli, An Overview of Synthesis Systems, *Proc. Custom Integrated Circuits Conference*, Rochester, May 1985, 221-225.

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

3. R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, MIS: A Multiple Level Logic Optimization System, *IEEE Transactions on CAD CAD-6*, (November 1987), 1062-1081.

4. M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, 1976.

5. J. P. Roth, Diagnosis of Automata Failures: a calculus and a method, *IBM journal of Research and Development 10*, (July 1966), 278-291.

6. P. Goel, An Implicit Enumeration Algorithm to generate tests for combinational logic circuits, *IEEE Transactions on Computers C-30*, (March 1981), 215-222.

7. O. H. Ibarra and S. K. Sahni, Polynomially complete fault detection problems, *IEEE Transactions on Computers C-24*, (March 1975), 242-249.

8. H. Fujiwara, FAN: A Fanout-Oriented Test Pattern Generation Algorithm, *Proc. of ISCAS 85*, , June 1985, 671-674.

9. T. Kirkland and M. R. Mercer, A Topological Search Algorithm for ATPG, *Proc. of Design Automation Conference*, Miami Beach, FLA, July 1987, 502-508.

10. M. H. Schulz, E. Trischler and T. M. Sarfert, SOCRATES: A Highly Efficient Automatic Test Pattern Generation System, *IEEE Transactions on CAD 7*, (January 1988), 126-137.

11. M. A. Breuer, A Random and an Algorithmic technique for fault detection and Test generation for sequential circuits, *IEEE Transactions on Computers C-20*, (November 1971), 1366-1370.

12. R. Marlett, EBT: A Comprehensive Test Generation Technique for highly sequential circuits, *Proc. of 15th Design Automation Conference*, Las Vegas, June 1978, 332-338.

13. S. Shteingart, A. W. Nagle and J. Grason, RTG: Automatic Register Level Test Generator, *Proc. of 22nd Design Automation Conference*, Las Vegas, June 1985, 803-807.

14. H. D. Schnurmann, E. Lindbloom and R. G. Carpenter, The Weighted Random Test-Pattern Generator, *IEEE Transactions on Computers C-24*, (July 1975), 695-700.

15. S. Nitta, M. Kawamura and K. Hirabayashi, Test Generation by Activation and Defect-Drive (TEGAD), *INTEGRATION, the VLSI Journal 3*, (1985), 2-12.

16. E. B. Eichelberger and T. W. Williams, A logic design structure for LSI testability, *Proc. 14th Design Automation Conference*, New Orleans, June 1977, 462-468.

17. V. D. Agarwal, S. K. Jain and D. M. Singer, Automation in Design for Testability, *Proc. of Custom Integrated Circuits Conference*, Rochester, NY, May 1984.

18. D. B. Armstrong, A Deductive Method of Simulating Faults in Logic Circuits, *IEEE Transcations on Computers C-21*, (May 1972), 464-471.

19. E. G. Ulrich and T. Baker, Concurrent Simulation of Nearly Identical Digitial Networks, *Computer 7*, (April 1974), 39-44.

20. P. Goel, Test generation costs analysis and projections, *Proc. 17th Design Automation Conference*, , June 1980, 77-84.

21. S. J. Jain and V. D. Agrawal, STAFAN: An Alternative to Fault Simualtion, *Proc. 21st Design Automation Conference*, , June 1984, 18-23.

22. H. K. Ma and A. L. Sangiovanni-Vincentelli, Mixed-Level Fault Coverage Estimation, *Proc. 23rd Design Automation Conference*, Las Vegas, June 1986, 553-559.

23. F. J. O. Dias, Fault Masking in Combinational logic circuits, *IEEE Transactions on Computers C-24*, (May 1975), 476-482.

24. K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. L. Sangiovanni-Vincentelli and A. R. Wang, Multi-level logic minimization using implicit don't cares, *IEEE Transactions on CAD 7*, (June 1988), 723-740.

25. H. T. Ma, S. Devadas and A. L. Sangiovanni-Vincentelli, Logic Verification Algorithms and their Parallel Implementation, *Proc. 24th Design Automation Conference*, Miami Beach, June 1987, 283-290.

26. H. T. Ma, S. Devadas, A. R. Newton and A. L. Sangiovanni-Vincentelli, Test Generation for Sequential Circuits, *IEEE Transactions on CAD 7*, (October 1988), 1081-1093.

27. S. Devadas, H. T. Ma, A. R. Newton and A. Sangiovanni-Vincentelli, Synthesis and Optimization Procedures for Fully and Easily Testable Sequential Machines, *Proc. of Int'l Test Conference*, Washington D. C., September 1988.

28. S. Devadas, H. T. Ma and A. R. Newton, Easily Testable PLA-Based Finite State Machines, *Dig. 19th Int. Symp. FTC*, Chicago, June 1989, 102-109.

29. S. Devadas, H. T. Ma, A. R. Newton and A. Sangiovanni-Vincentelli, Irredundant Sequential Machines Via Optimal Logic Synthesis, *U. C. Berkeley Internal Memo No. UCB/ERL M88/52*, Berkeley, August 1988.

30. J. P. Roth, *Computer Hardware Testing and Verification*, Computer Science Press, Potomac, Maryland, 1980.

31. J. P. Roth, VERIFY: an algorithm to verify a computer design, *IBM Technical Disclosure Bulletin 15*, (1973), 2646-2648.

32. J. P. Roth, Hardware Verification, *IEEE Transactions on Computers C-26*, (1977), 1292-1294.

33. W. E. Donath and H. Ofek, Automatic identification of equivalence points for Boolean Logic Verification, *IBM Technical Disclosure Bulletin 18*, (January 1976), 2700-2703.

34. G. Odawara, M. Tomita, O. Okuzawa and T. Ohta, A Logic Verifier based on Boolean Comparison, *Proc. 23rd Design Automation Conference*, Las Vegas, June 1986, 208-214.

35. R. E. Bryant, Symbolic Verification of MOS Circuits, *1985 Chapel Hill Conference on VLSI*, Chapel Hill, December 1985, 419-438.

36. R. S. Wei, Logic Verification and Test Generation for VLSI Circuits, *Ph.D Dissertation*, Berkeley, September 1986.

37. D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison and D. Ravenscroft, The Boulder Optimal Logic Design System, *Proc. of Int'l Conference on Computer-Aided Design*, Santa Clara, November 1987, 62-65.

38. H. T. Ma, S. Devadas and A. L. Sangiovanni-Vincentelli, Logic Verification Algorithms and their Parallel Implementation, *IEEE Transactions on CAD 8*, (February 1989), 181-189.

39. F. Brglez and H. Fujiwara, A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran, *Proc. 1985 IEEE Int. Symp. Circuits and Systems*, Kyoto, Japan, June 5-7, 1985.

40. S. C. S. Inc., Balance 8000 Guide to Parallel Programming, , July 31, 1985.

41. F. C. Hennie, Fault detecting experiments for sequential circuits, *Proc. of 5th Annual Symp. on Switching Circuit Theory and Logical Design*, Princeton, N. J., November 1964, 95-110.

42. W. G. Bouricius and al, Algorithms for Detection of Faults in Logic Circuits, *IEEE Transactions on Computers C-20*, (November 1971), .

43. A. Miczo, The Sequential ATPG: A Theoretical Limit, *Proc. of 1983 International Test Conference*, Philadelphia, PA, October 1983, 143-147.

44. S. Mallela and S. Wu, A Sequential Test Generation System, *Proc. of International Test Conference*, Philadelphia, PA, October 1985, 57-61.

45. H. T. Ma, S. Devadas, A. R. Newton and A. Sangiovanni-Vincentelli, An Incomplete Scan Design Approach to Test Generation for Sequential Machines, *Proc. of Int'l Test Conference*, Washington D. C., September 1988.

46. M. Hill and al, Design decisions in SPUR, *IEEE Computer 19*, (November 1986), 8-22.

47. K. Cheng and V. D. Agrawal, CONTEST: A Concurrent Test Generator for Sequential Circuits, *Proc. of 25th Design Automation Conference*, Anahiem, CA, July 1988 (to appear).

48. V. D. Agrawal and K. Cheng, A Complete Solution to the Partial Scan Problem, *Proc. of Int'l Testing Conference*, , September 1987.

49. L. M. Huismam and V. Iyengar, Letters to the Editor, *IEEE Design & Test*, , August 1985, 6.

50. S. J. Jain and V. D. Agrawal, Authors' reply, *IEEE Design & Test*, , August 1985, 7-8.

51. C. Timoc, M. Buehler, T. Griswold, C. Pina, F. Stott and L. Hess, Logical Models of Physical Failures, *Proc. of International Test Conference*, , October 1983, 546-553.

52. S. J. Jain and V. D. Agrawal, Test Generation for MOS Circuits Using D-Algorithm, *Proc. 20th Design Automation Conference*, , June 1983, 64-70.

53. M. K. Reddy, S. M. Reddy and P. Agrawal, Transistor Level Test Generation for MOS Circuits, *Proc. 22nd Design Automation Conference*, , June 1985, 825-828.

54. R. M. Apte, N. S. Chang and J. A. Abraham, Logic Function Extraction for NMOS Circuits, *Proc. IEEE International Conference on Circuits & Computers*, , September 1982, 324-327.

55. D. Saab and I. Hajj, A Logic Expression Generator for MOS Circuits, *Proc. IEEE International Conference on Circuits & Computers*, , September 1982, 328-331.

56. C. Y. Lo, H. N. Nham and A. K. Bose, A Data Structure for MOS Circuits, *Proc. 20th Design Automation Conference*, , June 1983, 619-624.

57. F. W. Clegg and E. J. McCluskey, Fault Equivalence in Combinational Logic Networks, *IEEE Transactions on Computers C-20*, (November 1971), 1286-1293.

58. D. R. Schertz and G. Metze, A New Representation of Faults in Combinational Digitial Circuits, *IEEE Transactions on Computers C-21*, (August 1972), 858-866.

59. D. C. Bossen and S. J. Hong, Cause-Effect Analysis for Multiple Fault Detection in Combinational Networks, *IEEE Transactions on Computers C-20*, (November 1971), 1252-1257.

60. C. W. Cha, Multiple Fault Diagnosis in Combiantional Networks , *Proc. 16th Design Automation Conference*, San Diego, June 1979, 149-155.

61. A. D. Fricdman, Fault Detection in redundant circuits, *IEEE Transactions on Computers C-16*, (Feb. 1967), 99-100.

62. M. Fridrich and W. A. Davis, Minimal Fault Tests for combinational logic circuits, *IEEE Transactions on Computers C-23*, (August 1974), 850-859.

63. M. Abramovici and M. A. Breuer, Multiple Fault Diagnosis in Combinational Circuits Based on an Effect-Cause Analysis, *IEEE Transactions on Computers C-29*, (June 1980), 451-460.

64. R. Tarjan, Finding Dominators in Directed Graphs, *SIAM Journal of Computing Vol. 3*, (1974), 62-89.

65. R. Dandapani and S. M. Reddy, On the Design of Logic Networks with Redundancy and Testability Considerations, *IEEE Transactions on Computers C-23*, (November 1974), 1139-1149.

66. J. Darringer, D. Brand, J. Gerbi, W. Joyner and L. Trevillyan, LSS: A System for production logic synthesis, *IBMJRD 28*, (September 1984), 537-545.

67. M. C. Paul and S. H. Unger, Minimizing the number of states in Incompletely Specified Sequential Circuits, *IRE Transactions on Electronic Computers EC-8*, (September 1959), 356-357.

68. G. D. Micheli, R. K. Brayton and A. Sangiovanni-Vincentelli, Optimal state assignment of finite state machines, *IEEE Transactions on CAD CAD-4*, (July 1985), 269-285.

69. S. Devadas, H. T. Ma, A. R. Newton and A. Sangiovanni-Vincentelli, MUSTANG: State Assignment of Finite State Machines Targeting Multi-Level Logic Implementations, *IEEE Transactions on CAD 7*, (December 1988), 1290-1300.

70. F. J. Hill and G. R. Peterson, *Introduction to switching theory and logical design*, John Wiley and Sons, 1981.

71. S. Devadas, H. T. Ma and A. R. Newton, On the Verification of Sequential Machines At Differing Levels of Abstraction, *IEEE Transactions on CAD 7*, (June 1988), 713-722.

72. J. Hartmanis and R. E. Stearns, Some Dangers in the State Reduction of Sequential Machines, *Information and Control 5*, (September 1962), 252-260.

73. S. Devadas, Techniques for Optimization-based Synthesis of Digital Systems, *U. C. Berkeley Internal Memo No. UCB/ERL M88/54*, Berkeley, August 1988.

74. S. Devadas, H. T. Ma, A. R. Newton and A. L. Sangiovanni-Vincentelli, A Synthesis and Optimization Procedures for Fully and Easily Testable Sequential Machines, *IEEE Transactions on CAD 8*, (October 1989), 1100-1107.

75. C. Sechen and A. Sangiovanni-Vincentelli, The TimberWolf Placement and Routing Package, *IEEE Journal of Solid-State Circuits and Systems SC-20*, (April 1985), 510-522.

76. C. W. Cha, A testing strategy for PLAs, *Proc. 15th Design Automation Conference*, , June 1978.

77. S. J. Hong and D. J. Ostapko, Fault analysis and test generation for programmable logic arrays(PLA's), *IEEE Transcations on Computers C-28*, (September 1979), 617-626.

78. S. J. Hong and D. J. Ostapko, FITPLA: A Programmable Logic Array for Function Independent Testing, *Dig. 10th Int. Symp. FTC*, , 1980, 131-136.

79. E. B. Eichelberger and E. Lindbloom, A heuristic test-pattern generator for programmable logic arrays, *IBM J. Res. Develop. 24*, (Jan 1980), 15-22.

80. R. S. Wei, PLATYPUS: A PLA Test Pattern Generation Tool, *IEEE Transactions on CAD 5*, (October 1986), 633-644.

81. H. Fujiwara and K. Kinoshita, A Design of Programmable Logic Arrays with Universal Tests, *IEEE Transaction on Computers C-30*, (November 1981), 823-828.

82. M. Fujita, H. Fujisawa and N. Kawato, Evaluation and Improvements of Boolean Comparison Methods Based on Binary Decision Diagrams, *Proc. of Int'l Conference on Computer-Aided Design (ICCAD)*, Santa Clara, November 1988, 2-5.

83. S. Malik, Logic Verification using Binary Decision Diagrams in a Logic Synthesis environment, *Proc. of Int'l Conference on Computer-Aided Design (ICCAD)*, Santa Clara, November 1988, 6-9.

84. R. E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers C-35*, (August 1986), 677-691.