

On Enhancing Deterministic Sequential ATPG

Khanh Viet Duong

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Electrical and Computer Engineering

Michael S. Hsiao

Dong S. Ha

Sandeep K. Shukla

February 1st, 2011

Blacksburg, Virginia

Keywords: Automatic Test Pattern Generation , Logic Testing , Sequential Circuits

Copyright © 2011, Khanh Viet Duong

On Enhancing Deterministic Sequential ATPG

Khanh Viet Duong

ABSTRACT

This thesis presents four different techniques for improving the average-case performance of deterministic sequential circuit Automatic Test Patterns Generators (ATPG). Three techniques make use of information gathered during test generation to help identify more unjustifiable states with higher percentage of “don’t care” value. An approach for reducing the search space of the ATPG was introduced. The technique can significantly reduce the size of the search space but cannot ensure the completeness of the search. Results on ISCAS’85 benchmark circuits show that all of the proposed techniques allow for better fault detection in shorter amounts of time. These techniques, when used together, produced test vectors with high fault coverages. Also investigated in this thesis is the Decision Inversion Problem which threatens the completeness of ATPG tools such as HITEC or ATOMS. We propose a technique which can eliminate this problem by forcing the ATPG to consider search space with certain flip-flops untouched. Results show that our technique eliminated the decision inversion problem, ensuring the soundness of the search algorithm under the 9-valued logic model.

Dedication

~ To my family ~

Acknowledgments

I would like to thank my advisor, Dr. Michael Hsiao for his direction, support and motivation throughout this work. I would also like to thank Dr. Dong S. Ha and Dr. Shukla for serving on my thesis committee.

Content

Chapter 1	Introduction.....	1
1.1	Previous Work	2
1.2	Thesis Outline	5
Chapter 2	Preliminaries.....	6
2.1	Single Stuck-at Fault Model.....	6
2.2	PODEM Algorithm	8
2.2.1	5-valued Logic System	9
2.2.2	Fault Excitation.....	9
2.2.3	Backtracing	9
2.2.4	D-frontier	10
2.2.5	X-path Check	11
2.2.6	Backtracking	12
2.2.7	Fault Simulation.....	13
2.3	Sandia Controllability/Observability Analysis Program.....	14
2.4	Sequential ATPG.....	16
2.4.1	The Iterative Logic Array Model	16
2.4.2	Forward Time Processing	17
2.4.3	Reverse Time Processing.....	17
2.4.4	9-valued Logic System	18
2.4.5	State Space Reduction.....	19

Chapter 3	Techniques for Enhancing Sequential ATPG	21
3.1	Techniques for Enhancing Sequential ATPG	21
3.1.1	Unjustifiable State Relaxation	21
3.1.2	Simple Non-chronological Backtrack	23
3.1.3	Loop Control	25
3.1.4	Search Space Reduction	30
3.2	Experiment Results	32
3.2.1	Our ATPG	32
3.2.2	Experimental Results	35
Chapter 4	Decision Inversion Problem	44
4.1	The Decision Inversion Problem	44
4.2	X-Locking	48
4.3	Experimental Results	50
Chapter 5	Conclusion and Future Direction	52
5.1	Conclusion	52
5.2	Future Direction	53
Reference		54

List of Figures

Figure 2.1	Effect of Stuck-at faults	7
Figure 2.2	PODEM flow.....	8
Figure 2.3	An example backtrace operation.....	10
Figure 2.4	D-frontier examples	11
Figure 2.5	Backtracking illustrated.....	12
Figure 2.6	Sequential test generation: taxonomy	16
Figure 2.7	Synchronous sequential circuit.....	16
Figure 2.8	The iterative logic array model of a sequential circuit.....	17
Figure 2.9	Example for the need of 9-value logic	18
Figure 2.10	5-value logic vs. 9-value logic.....	19
Figure 3.1	Examples for unreachable state relaxation.....	22
Figure 3.2	Simple Non-chronological Backtracking.....	24
Figure 3.3	Non-chronological Backtrack Example.....	25
Figure 3.4	Example STG with state repetition	26
Figure 3.5	Single loop example for Loop Control	28
Figure 3.6	Loop Control with multiple state repetition instances.....	30
Figure 3.7	Our ATPG pattern generation flow.....	32
Figure 3.8	Fault propagation flow.....	33
Figure 3.9	ATPG justification flow.....	34
Figure 4.1	Example circuit for the Decision Inversion Problem.....	44
Figure 4.2	Unrolled example circuit – XX0 justification	45
Figure 4.3	Backtracing for $D=0$ through C	46
Figure 4.4	Example decision tree	47
Figure 4.5	Decision tree for an ATPG with X-locking implemented.....	49

List of Tables

Table 2.1	Controllability calculation rules of PI, output and branch	15
Table 2.2	Observability calculation rules if PO, input and stem	15
Table 3.1	ATPG results without any technique implemented	36
Table 3.2	ATPG results with Unjustifiable State Relaxation	37
Table 3.3	ATPG results with Simple Non-chronological Backtrack	38
Table 3.4	ATPG results with Loop Control	39
Table 3.5	ATPG results with Search Space Reduction	40
Table 3.6	ATPG results with all techniques	41
Table 3.7	Impact of each technique on sequential circuit test generation performance....	42
Table 3.8	Comparison of sequential ATPG results	42
Table 4.1	ATPG results with and without X-locking	51
Table 4.2	Extended ATPG results with and without X-locking	51

Chapter 1

Introduction

Under the rising demand for more computational power, modern VLSI technology continues to advance to meet industry expectations, resulting in computing hardware systems more complex than ever; while at the same time presenting growing challenges to post-silicon testing and verification. The need for post-silicon testing arose due to discrepancies in the manufactured chip caused by design errors, manufacturing defects or testing conditions. Although test generation is now feasible for almost any combinational circuit, the same cannot be said about sequential circuits. Since the cost of complete scan can be prohibitive in both area and performance degradation for some classes of circuits, such as high-performance processors, there is a need for efficient sequential ATPG. Simulation-based test generators can perform better than the deterministic type in the area of fault coverage, but they cannot practically handle random-resistant faults nor determine all untestable faults in modern day circuits. Since hybrid ATPG also utilize deterministic algorithm, there is always demand for improvement in deterministic ATPG. An example of a hybrid ATPG is to use test vectors generated to seed the genetic algorithms and information regarding untestable faults can be used to make the faults testable with minimum overhead.

In this thesis, aspects of unreachable state detection as well as loop detection within the state transition graph have been studied and new techniques have been proposed for enhancing the performance of deterministic sequential Automatic Test Pattern Generators (ATPG). We know that a deterministic ATPG exhibit worst case performance when it searches within non-solution areas. It can be propagating fault effects through non-

sensitizable paths or justifying unreachable states. Therefore, unreachable states detection is necessary in effectively pruning the search space. Loop detection is already needed since the generator can waste a lot of effort traversing in a loop in the state transition graph and might never reach a solution. Under the presence of a loop, however, one cannot always declare a state to be unreachable whenever the justification of said state returned non-solution. Instead of throwing that result away, we present a method for storing the result until it can be declared reachable or unreachable. An approach is also proposed for pruning the search space.

A second topic being studied in this thesis is the bit inversion problem. Due to the constraints imposed on the sequential ATPG that it must start the sequence with a fully unspecified starting state, the typical bit inversion scheme can lead the PODEM algorithm into falsely declaring a state to be unreachable. This can consequently lead to falsely declaring a fault to be untestable. This problem has been addressed only once in [Gouders91], by using the CONSEQUENT circuit model. This problem invalidates the completeness of the search process of ATPG tools which uses the 9-valued circuit model [Muth76] like HITEC [Niermann91]. We propose a method to attain complete search under the 9-valued circuit model: X value locking.

1.1 Previous Work

It has been known for over three decades that the problem of test generation on combinational circuits is an NP-complete problem [Ibarra75][Fujiwara82]. Due to its practical application, a large amount of work has been done in the area of combinational circuit test generation. The earliest notable work is the D-algorithm (DALG) reported by Roth [Roth66]. The algorithm provided an objective method in the search for a solution, but it is inefficient when a fault requires multiple paths to be sensitized simultaneously. PODEM [Goel81] by Goel improved on DALG by extending and limiting search decisions to primary inputs; thus it overcame DALG's inefficiency in solving circuits which implement

error correction and translation functions. PODEM was a significant improvement from DALG and is often used as a baseline even in modern ATPG algorithms.

After the introduction of the branch and bound search algorithm in PODEM, a number of techniques have been proposed in the literature to improve its performance. These techniques have been presented and implemented in many deterministic combinational ATPG systems: FAN [Fujiwara83], ATWIG [Trischler84], FAST [Abramovici86], TOPS [Kirkland87], SOCRATE [Schulz88], ATALANTA [Lee93], LEAP [Silva94] and ATOM [Hamzaoglu98]. These showed the tremendous amount of work and effort which has been invested in solving the combinational ATPG problem.

Comparing to the combinational circuit test generational problem, the problem of test generation for sequential circuit is much more complex [Marchok95]. Over the years, there have been numerous algorithms proposed in this area; they can be classified into three types: deterministic, simulation-based and hybrid. The earliest reported sequential ATPG was that of Seshu and Freeman [Seshu62]. But it was not until after the introduction of DALG, did deterministic ATPG make significant progress. One of the earliest algorithms for sequential circuit, also introduced by Roth, is the D-algorithm II (DALG II) [Roth67]. The algorithm extended upon the original DALG to handle sequential circuit; however, the algorithm does not take into account correctly the repeat effects of faults during a test sequence. In an attempt to design a complete algorithm, a number of different models for sequential circuit has been proposed; they include the 9-Valued Circuit Model [Muth76], the Split Model [Cheng88] and the CONSEQUENT circuit model [Gouders91].

Ever since DALG II, different approaches have been proposed for sequential circuit test generators. Instead of forward time processing, EBT [Marlett78] and BACK algorithm [Cheng88] consider only reverse time processing through the iterative logic array. EBT preselects a sensitization path and try to justify it. The approach misses faults that require multiple-path sensitization and is also impractical due to the number of available paths. Alternatively, the BACK algorithm pre-selects a primary output and performs backward justification using a drivability heuristic. GENTEST [Cheng89] tried to improve upon BACK

by considering multiple-path sensitization but remain unnecessarily complex and inefficient as the algorithm works exclusively backward in time. ESSENTIAL [Auth91] tried to overcome the some of the disadvantages by collecting information during the preprocessing phase and during the test generation phase to prune the search space.

On the other hand, HITEST [Bending84] and STALLION [Ma88] made use of PODEM, processing both forward and backward. HITEST is a knowledge-based interactive test generation system, which uses PODEM to processing one timeframe a time, treating memory elements as pseudo inputs and outputs. The down side of HITEST is that it does not consider the fault effect reaching the current state. STALLION uses PODEM to propagate the fault effect to a primary output and then uses backward justification to find the sequence that can bring the circuit from an unspecified state to the initial state found during the fault effect propagation phase. STALLION, however, assumes the circuit to be fault-free during backward justification; thus it is inefficient and incomplete. Then we have test generators like FASTEST [Kelsey89] which operate exclusively in forward time. The generator utilizes the Initial Timeframe Algorithm to estimate the number of timeframes to begin test generation with and the timeframe to attempt to excite the fault based on Controllability and Observability [Goldstein80]. FASTEST correctly addresses the problem of fault effect reach the current state; but wastes effort and resources when overestimating or underestimating the number of timeframes needed.

More recently developed is the popular, and often compared to, sequential circuit test generation system HITEC [Niermann91] by Niermann. HITEC is a complete test generation test package for sequential circuits based on the implicit enumeration of PODEM and uses dominators and mandatory assignments of FAN, TOPS and SOCRATE. The algorithm also makes use of previously generated initialization sequences and information collected during fault simulation to guide the test generator. HITEC was able to achieve high fault coverage using only the gate level description of the circuit.

A few other recently developed deterministic sequential ATPG are MOSAIC [Dargelas97], which looked for a different approach in both fault simulation and detection using its own

256-Valued system derived from CONSEQUENT; and ATOMS [Hamzaoglu00], which combined the techniques in HITEC and ATOM to create an efficient deterministic sequential ATPG and was able to achieve very high fault coverage in a short amount of time compared to HITEC.

Although many of the deterministic branch-and-bound test generators described above achieve very high fault coverage, this type of sequential ATPG remains computationally intensive. Alternatives to deterministic ATPG include simulation based ATPG (CRIS [Saab92], GATTO [Prinetto94], STRATEGATE [Hsiao97], GATEST [Rudnick97], IGATE [Hsiao98]) , which often uses genetic optimization to arrive at a suitable test set, and hybrid ATPG ([Saab94], GA-HITEC [Rudnick95], ALT-TEST [Hsiao96], MIX [Xijiang98], MIX-PLUS[Xijiang99]), which combines simulation based algorithms with deterministic algorithm to achieve high fault coverage in often shorter time.

The contributions of this thesis include the following:

- Four techniques for improving the average-case performance of deterministic sequential ATPG
- A technique for solving the Bit Decision Problem

1.2 Thesis Outline

An Outline of the rest of the thesis is as follows:

- Chapter 2 outlines the background and basic concepts necessary for understanding deterministic combinational and sequential ATPG
- Chapter 3 explains the four enhancements: Unjustifiable States Relaxation, Simple Non-chronological Backtrack, Loop Control and Search Space Reduction.
- Chapter 4 details the Decision Inversion Problem and the proposed solution
- Chapter 5 concludes the work with an overview and presents a recommendation for future work.

Chapter 2

Preliminaries

Physically testing the manufactured circuit chip, entails stimulating the inputs to the system, measuring the response and comparing the results response to the expected response to ascertain the correctness of the circuit's behavior. In digital systems, a test generally includes a pre-designed set of stimuli, *test patterns*, and the corresponding responses, *test responses*. As it is very difficult to mathematically model physical failures and fabrication defects, test evaluation is often carried under the context of a *fault model*. *Fault simulation* produces the response of the CUT in the presence of faults (faulty circuit). A fault is said to have been detected when the response of the faulty circuit differs from the expected response of the fault-free circuit under the same input stimuli. To measure the effective of a test, a commonly metric for any particular fault model is *fault coverage*, defined as the ratio between the number of faults detected and the total number of faults of the CUT.

2.1 Single Stuck-at Fault Model

The single stuck-at fault model is the one of the most commonly used logical fault models. The fault model is simple enough to create an efficient algorithm while at the same time can capture many different physical faults and has been shown to be effective in capturing a wide range of defects in the fabricated circuit. Other advantages for using the single stuck-at fault model include independence from fabrication technology and having the number of faults linear to the size of the CUT.

The single stuck-at fault model assumes there is at most one stuck-at fault present in the faulty-circuit. This fault is modeled by having the faulty node tied to either '1' (VDD, HI) or '0' (GND, LO). These types of fault are known as Stuck-at-1 (s-a-1) and Stuck-at-0 (s-a-0) respectively. Example 1 below demonstrates the effects of the two stuck-at faults.

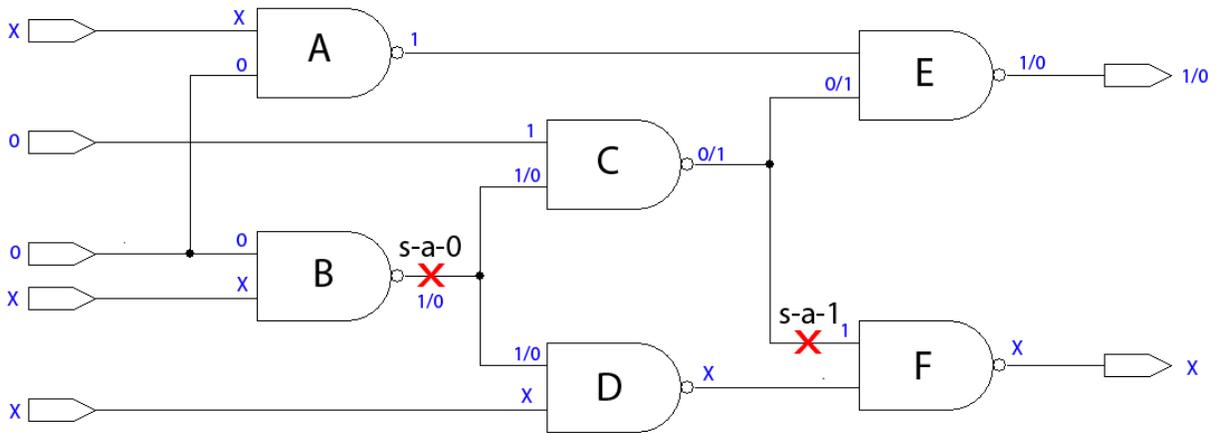


Figure 2.1 Effect of Stuck-at faults

Example 1. Present within the circuit in Figure 2.1 are two stuck-at faults: a s-a-0 at the output node of gate B and a s-a-1 at one of the input nodes of gate F. The presence of the s-a-0 at gate B causes it to feed a value of 1/0 ('1' when fault is not present, '0' when fault is present) to gate C and gate D. On the other hand, the s-a-1 fault at the input of gate F drive said input to 1 regardless of the value of the node it originated from.

Example 1 shows two different positions the fault can occur: at an output node where the fault affects all the branches going out from the node and at an input node where the fault affects that stem exclusively. The case where, under the same input stimuli, the value of a node in a *fault-free* circuit differs from the value of said node (the output of gate B, C, E) in the *faulty* circuit can be classified as a *fault effect*. The faults in example 1 are said to be detected as the same input vector applied to both the faulty and fault-free circuit propagated the fault effect to an observable primary output (PO) node. Since the Single Stuck-at fault model assumes only one fault being present at any time, the two faults in Figure 2.1 are only for demonstrative purpose.

2.2 PODEM Algorithm

PODEM [Goel81] was proposed to tackle the problem of multiple paths sensitization in the D algorithm. PODEM is a branch-and-bound algorithm which expands its decision tree around the Primary Inputs (PIs). PODEM maintains a D-frontier for fault propagation; but keeps no J-frontier, which is unlike DALG. Since PODEM uses logic simulation to assign values to internal nodes, no value conflicts will ever happen at any of the internal nodes due to simulation. Therefore, the bounding conditions include and are limited to either not excited target fault or an empty D-frontier. Figure 2.2 shows the basic flow of the PODEM algorithm.

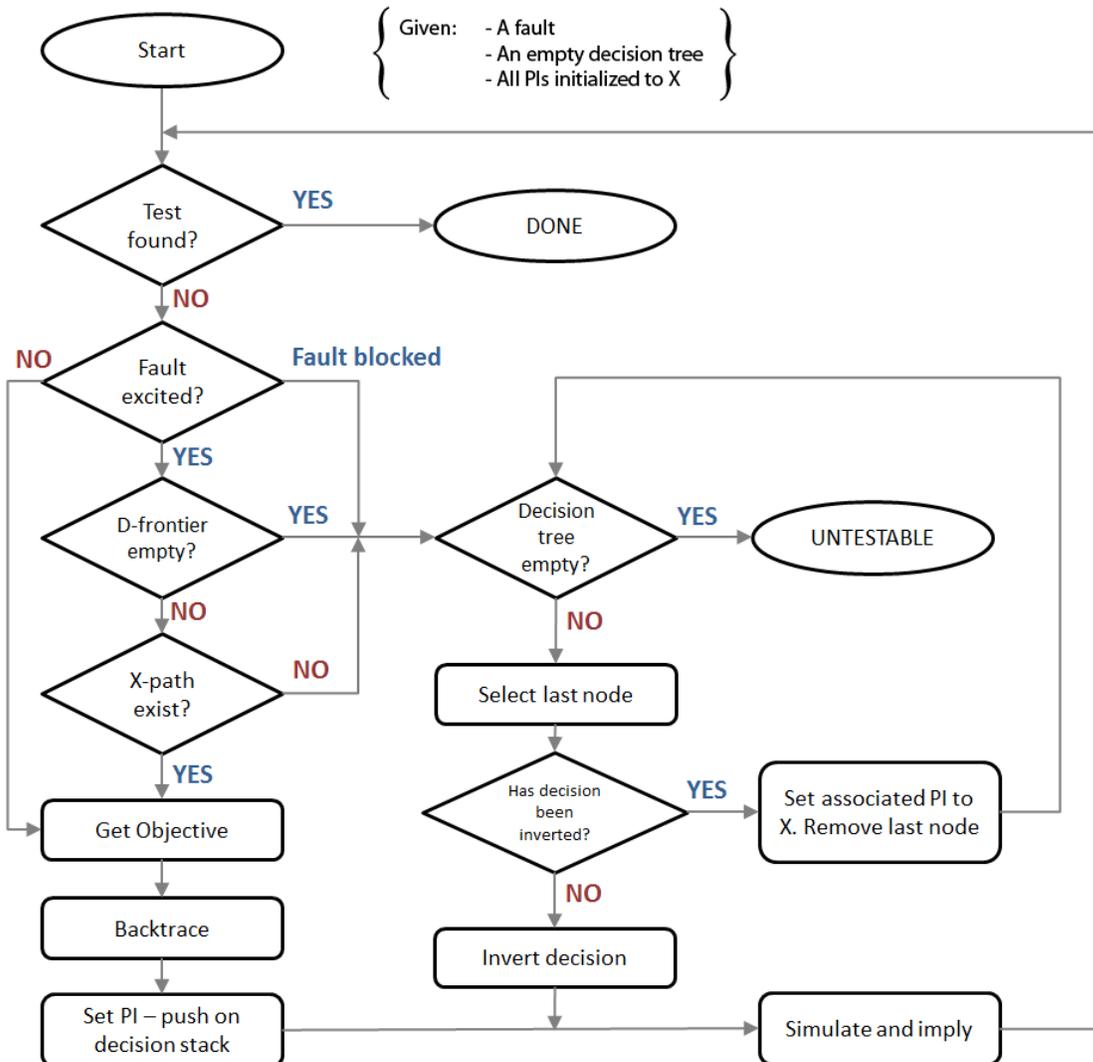


Figure 2.2 PODEM flow

2.2.1 5-valued Logic System

To correctly model the fault effect in a combinational circuit, along with DALG, Roth proposed a 5-valued logic system [Roth66]. All five values can be seen in Figure 1: 0, 1, X, D (1/0) and \bar{D} (0/1). D designates a value 1 a node in the fault-free circuit and a 0 for the same node in the faulty circuit. \bar{D} is the complement of D. For the purpose of propagating fault effects within a combinational circuit, since each PI can only take on a single value, there will not be a case where partial fault effects (0/X, X/0, 1/X and X/1) converging into D or \bar{D} . Thus the 5-valued system is sufficient for combinational circuit test generation.

2.2.2 Fault Excitation

Before a fault effect can be realized, the fault needed to be excited first. This is done by justifying the fault-free value of the node at the fault site to the opposing value of the stuck-at value. That is, if the fault is s-a-1 at the output node of gate A, the objective is to drive a value 0 to gate A. If the fault is s-a-0 at the input node of gate A from gate B, the objective is to drive a value 1 to gate B.

2.2.3 Backtracing

To arrive at a decision at a PI, the PODEM algorithm needs to trace backward from the objective node and value to a PI. The backtrace operation works by selecting an unspecified input to the objective and assigning the necessary value to drive the current objective gate to the needed value. A naïve backtrace algorithm is described below:

```
backtrace(gate, value)
if gate != a Primary Input
    if gate == NAND, NOR, XNOR or NOT
        value = NOT(value)
    from fanin of gate select an unspecified input
    return backtrace(input, value)
else
    return (gate, value)
```

Assuming that the initial objective is not violated or specified before the backtrace operation, the algorithm will always backtrace to an unspecified PI. Figure 2.3 shows a simple backtrace operation from an initial objective: $E = 0$ (s-a-1 fault excitation). For such reason, the backtrace operation can simply be implemented as a loop from the initial objective to the PI. The current algorithm is kept in a recursive form so that the programming stack can be exploited when the function is transferred to the sequential ATPG.

Starting from the output node of E:

- To set E to 0, one input must be set to 1. D is selected.
- To set D to 1, all input must be set to 1. For now, B is selected
- To set B to 1, the input must be set to 0. PI 2 is selected.
- Returning PI 2 = 0

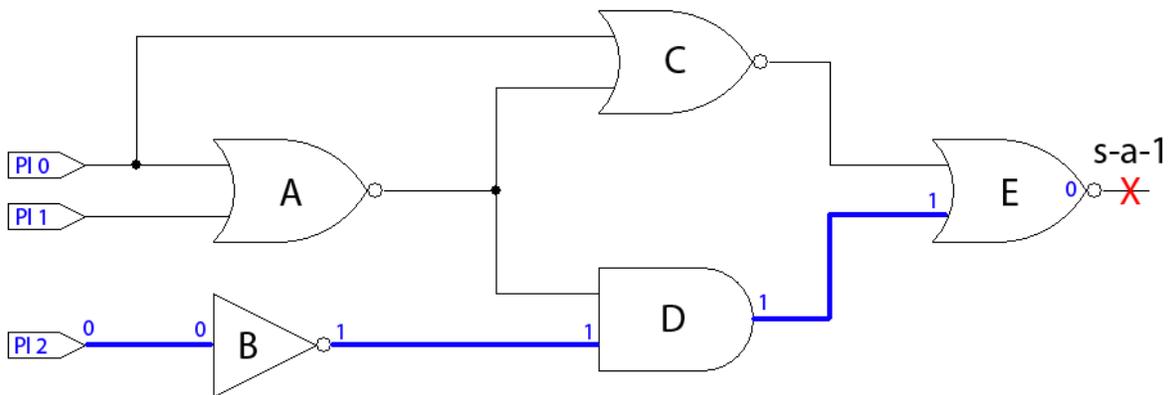


Figure 2.3 An example backtrace operation

2.2.4 D-frontier

The D-frontier comprises of all gates whose output values are X but have a D or Don at least one of their inputs. This also means that at least one of the inputs currently holds a “don’t care” value. PODEM keep track of the D-frontier during test generation. Once the fault has been excited, PODEM propagate fault effects toward the POs by advancing the D-frontier. Figure 2.4(a) shows an example D-frontier consisting of gate 2 and gate 3. The fault effect can be propagated toward gate 4 by assigning the non-controlling value 0 to the

input with “don’t care” at gate 2 (Figure 2.4(b)). The effect could be achieved by setting the input of gate 1 to 0; thus propagating a non-controlling value 1 to gate 3 and ultimately propagating the fault effect through gate 3 toward gate 4. In the case of Figure 2.4(b), the new D-frontier consists of gate 3 and gate 4.

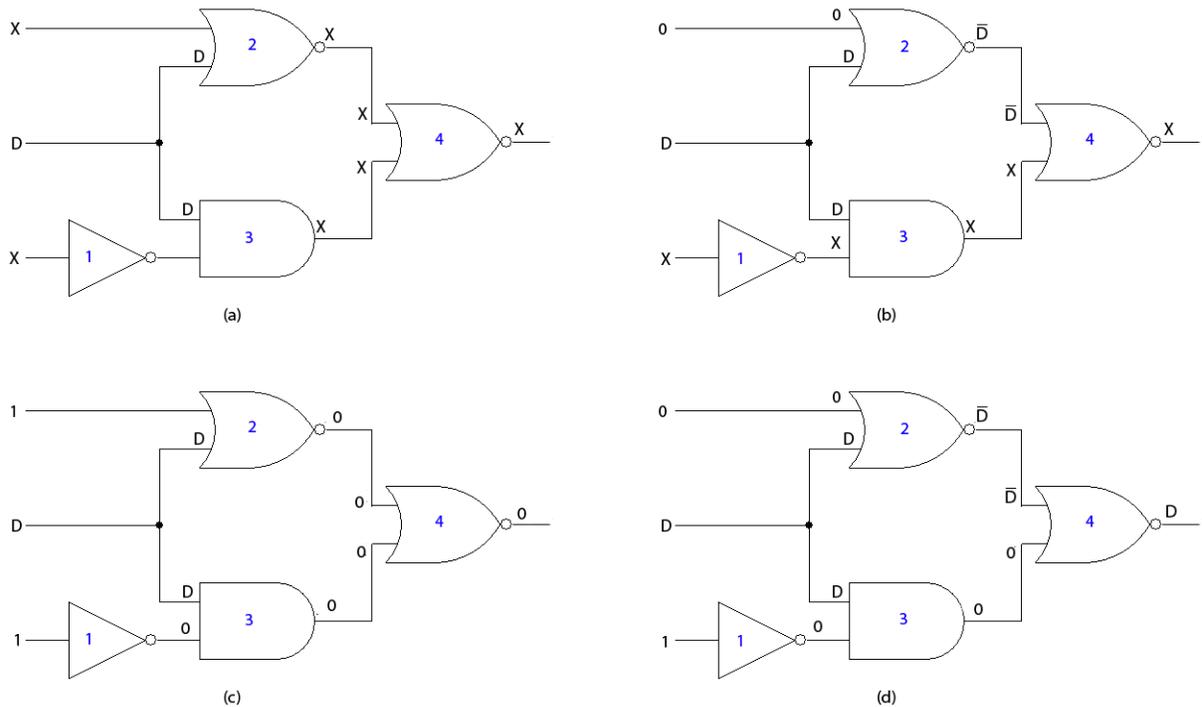


Figure 2.4 D-frontier examples

Figure 2.4(c) illustrates a situation where all the fault effects in the D-frontier are blocked. All the possible propagation paths for the fault effect have been assigned a value. The D-frontier in this case is empty. Clearly, the algorithm cannot propagate the fault effect if there is no D-frontier. If the fault has been excited, but no D or \bar{D} has been propagated to a PO yet and the D-frontier is empty, PODEM must backtrack. Figure 2.4(d) illustrates a situation where the D frontier has been propagated further past gate 4.

2.2.5 X-path Check

An X-path is a path between two nodes of a circuit in which the output values of all nodes along this path are unknown (X). To advance a gate, g , within the D-frontier, toward a PO, it is necessary that there exist at least one X-path from gate g to a PO. As the algorithm select

to propagate the fault effect through the easiest X-path, having no X-path for a D-frontier will eventually lead to an empty D-frontier. PODEM uses an X-path check procedure to catch conflicts before it happens, allowing itself to backtrack away from the non-solution area earlier.

2.2.6 Backtracking

When a conflict happens, the algorithm must backtrack as the current path segment will lead to no solution. The search should continue on a different branch on the previous decision point. Since a PI can only take on two different Boolean values, the simplest mechanism is to reverse the latest decision made. Implication and simulation done after this reversal will also effectively undo all implied assignments of the previous decision variable. Here, when the backtrack procedure is called, PODEM inverts the value of the most recent decision, blocking itself from exploring that particular branch again. If the latest decision has been reversed before then it can be concluded that no test can be found under the current decision tree. Therefore, the associated PI assignment is then undone back to don't care (X) and PODEM proceed to go back further in the decision tree. After PODEM backtrack on the very first decision, the fault is determined to be untestable.

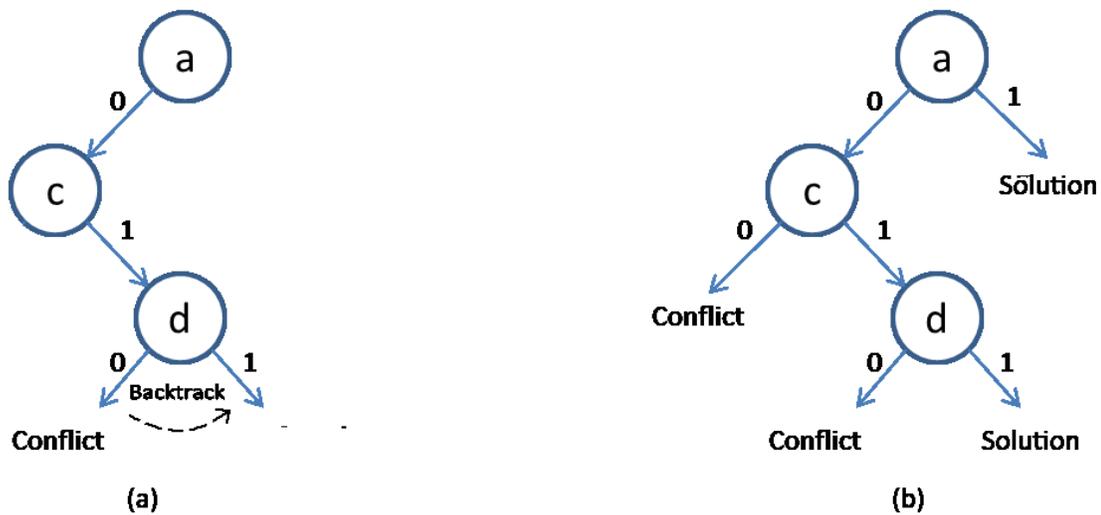


Figure 2.5 Backtracking illustrated

Consider the decision illustrated in Figure 2.5(a). Before deciding on d , previous decisions are $a=0$ and $c=1$. The decision $d=0$ led to a conflict; thus PODEM must backtrack. The decision on d is reverse to $d=1$ and the search continues under $a=0$, $c=1$ and $d=1$. If this path leads to conflict as well, since the decision on d has already been reversed, it means that no solution can be found under $a=0$ and $c=1$. The assignment on d is undone into $d=X$. PODEM will then attempt to reverse $c=1$ to $c=0$. If a conflict is encountered under $a=0$ and $c=0$, the search will continue under $a=0$. If PODEM backtrack beyond this point then the fault is untestable. Note that in this process, PODEM implicitly explores the entire input space. If a solution exists, it will guarantee to find it given sufficient time. Likewise, if no vector exists, it will complete the search and conclude that no solution exists. One can see that the search space for PODEM is exponential to the number of primary inputs.

2.2.7 Fault Simulation

Fault simulation is simulation in the presence of a fault. Fault simulation is required in test generators to correctly model the effect of the fault on the CUT. For every test vector found, fault simulation can be performed on all undetected faults. If any faults are detected by this vector, these faults can be *dropped*, since there would no need to perform test generation on those faults. *Fault dropping* improves the performance of the fault simulation process as well as the overall test generation process.

2.3 Sandia Controllability/Observability Analysis Program

Consider the decision tree illustrated in Figure 2.5(b) as an example, let us assume that our search algorithm is naïve enough to always take the left branch. Searching using this algorithm will lead us to two conflicts before arriving at a solution with $a=0$, $c=1$ and $d=1$. Had the algorithm been more intelligent, the search could have avoided the two conflicts or even find a solution at very first decision. To avoid searching randomly or naively, a deterministic ATPG would benefit from some kind of testability measure to guide its decision.

The Sandia Controllability/Observability Analysis Program (SCOAP) [Goldstein80] is one of many topology-based testability analysis that can provide an useful heuristic to be used in the test generation process. For each signal s in a logic circuit, SCOAP calculates six numerical values representing the signal's testability measures:

- $CC0(s)$ – combinational 0-controllability of s
- $CC1(s)$ – combinational 1-controllability of s
- $CO(s)$ – combinational observability of s
- $SC0(s)$ – sequential 0-controllability of s
- $SC1(s)$ – sequential 1-controllability of s
- $SO(s)$ – sequential observability of s

The controllability measures reflect the difficulty in setting the signal s to the require logic value ($CC0$ and $SC0$ for logic 0; $CC1$ and $SC1$ for logic 1) from the PIs and the observability measures reflect the difficulty in propagating the logic output of signal s to one of the POs.

The controllability measures of all signals are calculated before the observability signals are computed. In a breadth-first manner, controllability measures are calculated for signals from PIs to POs according to the rules in Table 2.1. More specifically, the controllability of a gate g is calculated only after the controllabilities of all of its inputs have been calculated. As an initial condition, the controllability values for all PIs are set to 0. A 1 is added to the combinational controllability measures when we move from one level of logic gate to another and a 1 is added to the sequential controllability measure when we pass through a

storage element. The higher the controllability measure is, the more difficult it is to control the signal from the PIs.

	CC0	CC1	SC0	SC1
PI	1	1	0	0
AND	$\min\{\text{input CC0}\} + 1$	$\Sigma (\text{input CC1}) + 1$	$\min\{\text{input SC0}\}$	$\Sigma (\text{input SC1})$
OR	$\Sigma (\text{input CC0}) + 1$	$\min\{\text{input CC1}\} + 1$	$\Sigma (\text{input SC0})$	$\min\{\text{input SC1}\}$
NOT	Input CC1 + 1	Input CC0 + 1	Input SC1	Input SC0
NAND	$\Sigma (\text{input CC1}) + 1$	$\min\{\text{input CC0}\} + 1$	$\Sigma (\text{input SC1})$	$\min\{\text{input SC0}\}$
NOR	$\min\{\text{input CC1}\} + 1$	$\Sigma (\text{input CC0}) + 1$	$\min\{\text{input SC1}\}$	$\Sigma (\text{input SC0})$
BUFFER	Input CC0 + 1	Input CC1 + 1	Input SC0	Input SC1
XOR	$\min\{\text{CC1}(a)+\text{CC1}(b), \text{CC0}(a)+\text{CC0}(b)\} + 1$	$\min\{\text{CC0}(a)+\text{CC1}(b), \text{CC1}(a)+\text{CC0}(b)\} + 1$	$\min\{\text{SC1}(a)+\text{SC1}(b), \text{SC0}(a)+\text{SC0}(b)\}$	$\min\{\text{SC0}(a)+\text{SC1}(b), \text{SC1}(a)+\text{SC0}(b)\}$
XNOR	$\min\{\text{CC0}(a)+\text{CC1}(b), \text{CC1}(a)+\text{CC0}(b)\} + 1$	$\min\{\text{CC1}(a)+\text{CC1}(b), \text{CC0}(a)+\text{CC0}(b)\} + 1$	$\min\{\text{SC0}(a)+\text{SC1}(b), \text{SC1}(a)+\text{SC0}(b)\}$	$\min\{\text{SC1}(a)+\text{SC1}(b), \text{SC0}(a)+\text{SC0}(b)\}$
DFF	Input CC0 + 1	Input CC1 + 1	Input SC0	Input SC1
Branch	Stem CC0	Stem CC1	Stem SC0	Stem SC1

Note: a and b are inputs of an XOR or XNOR gate.

Table 2.1 Controllability calculation rules of PI, output and branch

The observability measures are also calculated in a breadth-first manner but moving from POs toward PIs instead. Table 2.2 lays out the rules for calculating observability measures. The same rule for adding a 1 to the values applies here as with the controllability measures. As initial conditions, the observability values for all POs are set to 0. The higher the observability of a signal is, the harder it is to observe the logic value of the signal at any PO.

	Combinational Observability	Sequential Observability
PO	0	0
AND/NAND	$\Sigma (\text{output CO, CC1 of other input}) + 1$	$\Sigma (\text{output SO, SC1 of other input})$
OR/NOR	$\Sigma (\text{output CO, CC0 of other input}) + 1$	$\Sigma (\text{output SO, SC0 of other input})$
NOT/BUFFER	Output CO + 1	Output SO
XOR/XNOR	$a: \Sigma (\text{output CO, } \min\{\text{CC0}(b), \text{CC1}(b)\}) + 1$	$a: \Sigma (\text{output SO, } \min\{\text{SC0}(b), \text{SC1}(b)\})$
	$b: \Sigma (\text{output CO, } \min\{\text{CC0}(a), \text{CC1}(a)\}) + 1$	$b: \Sigma (\text{output SO, } \min\{\text{SC0}(a), \text{SC1}(a)\})$
DFF	Output CO	Output SO + 1
Stem	$\min\{\text{branch CO}\}$	$\min\{\text{branch SO}\}$

Note: a and b are inputs of an XOR or XNOR gate.

Table 2.2 Observability calculation rules if PO, input and stem

2.4 Sequential ATPG

2.4.1 The Iterative Logic Array Model

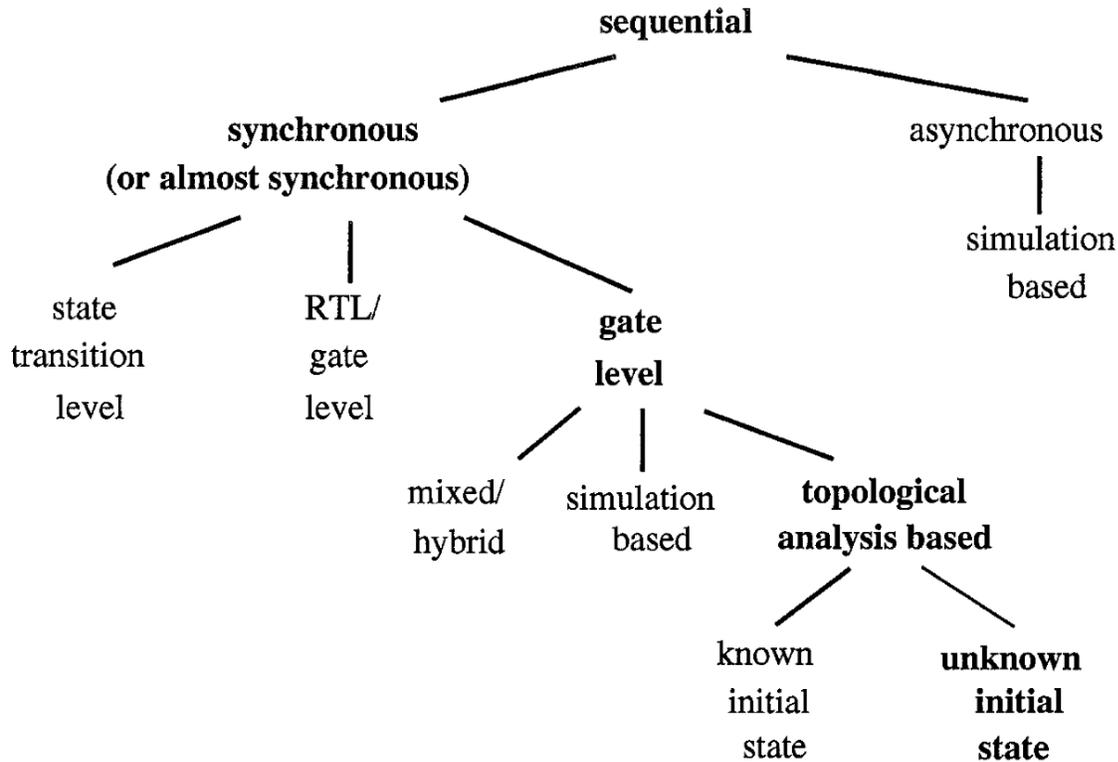


Figure 2.6 Sequential test generation: taxonomy

Figure 2.6 shows the many different sequential test generation approaches [Cheng96]. The approach discussed in this thesis target synchronous circuits. A synchronous sequential circuit can be viewed as a combination of a combinational sub-circuit and memory elements (Figure 2.7). Due to their simplicity, many sequential test generators use fundamental combinational algorithm as its underlying baseline.

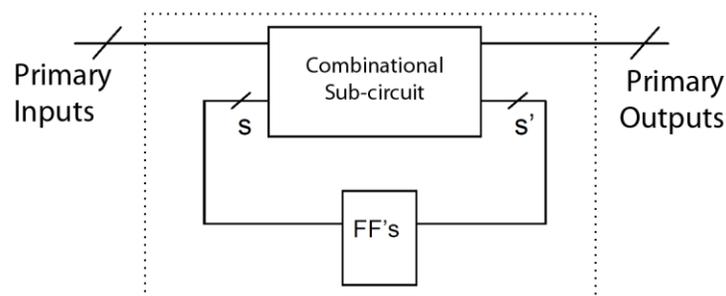


Figure 2.7 Synchronous sequential circuit

The iterative logic array (ILA) model is a combinational model for sequential circuits, constructed by having feedback signals for previous time copies of the circuit generating signals for current and future time copies. Figure 2.8 shows how this model is constructed. Each rectangle represents a time-frame, a copy of the combinational portion of the circuit. When considering any particular time copy, each output from memory elements can be treated as *Pseudo Primary Input* (PPI) and each input to them can be treated as *Pseudo Primary Output* (PPO). The upside of this model is that topological analysis algorithms can be used to generate a test through multiple copies of the combinational sub-circuit. On the other hand, a single stuck-at fault in the sequential circuit will correspond to a multiple stuck-at fault where each time-frame contains a stuck-at fault.

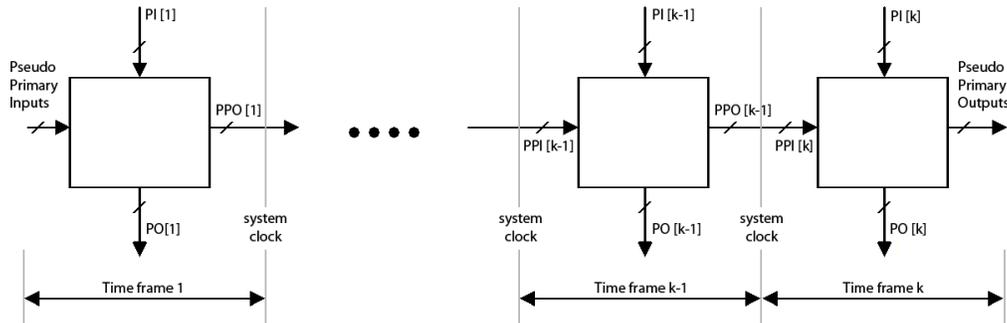


Figure 2.8 The iterative logic array model of a sequential circuit

2.4.2 Forward Time Processing

Forward Time Processing (FTP) is a process in which the fault is activated and propagated to a PO starting from time-frame 0. If the fault effect could not be propagated to any PO within the time-frame but is propagated to PPOs, FTP can be performed on the next time-frame to attempt to propagate the fault effect further. An ATPG can work using FTP exclusively by limiting search in time-frame 0 to PIs exclusively.

2.4.3 Reverse Time Processing

Reverse Time Processing (RTP) is the process in which the algorithm backtraces from a set of objectives in an effort to justify a state or a fault sensitization path. An ATPG can work

using RTP exclusively by pre-selecting a sensitization path or a PO and try to justifying the fault effects on the pre-selected PO.

2.4.4 9-valued Logic System

Since a fault is present in every single time-frame of the sequential circuit, the five-valued logic used in the D-algorithm is neither sufficient nor appropriate in this case [Muth76]. The nine-valued model is proposed to take into account the possible repeated effects of the fault in the iterative logic array model. Each of the nine values is comprised of an ordered pair of binary values of the fault-free circuit and the faulty circuit. As a node in a circuit could take 1, 0 or X (don't care), the nine values are: 0/0, 0/1, 0/X, 1/0, 1/1, 1/X, X/0, X/1 and X/X.

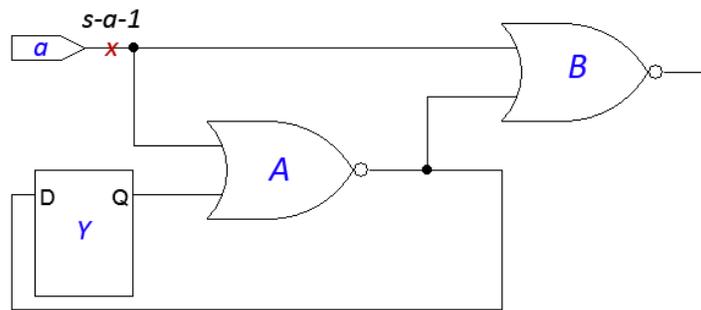


Figure 2.9 Example for the need of 9-value logic

Consider the circuit and the fault illustrated in Figure 2.9 as an example. Figure 2.10(a) illustrates a test generation effort using 5-value logic on an unrolled circuit. To activate the s-a-1 fault at the PI a , we have $a_0=0$, where the suffix denotes the time-frame. To propagate the fault effect through B to the PO, we need $A_0 = 1$. This requirement can be back-traced to $Y_0 = 1$, $A_{-1} = 0$ and $a_{-1} = 1$ (along with $Y_{-1}=1$). a_{-1} is impossible under the 5-value system due to the presence of the fault also in time-frame -1 on signal a . Thus the fault is untestable under the 5-value system.

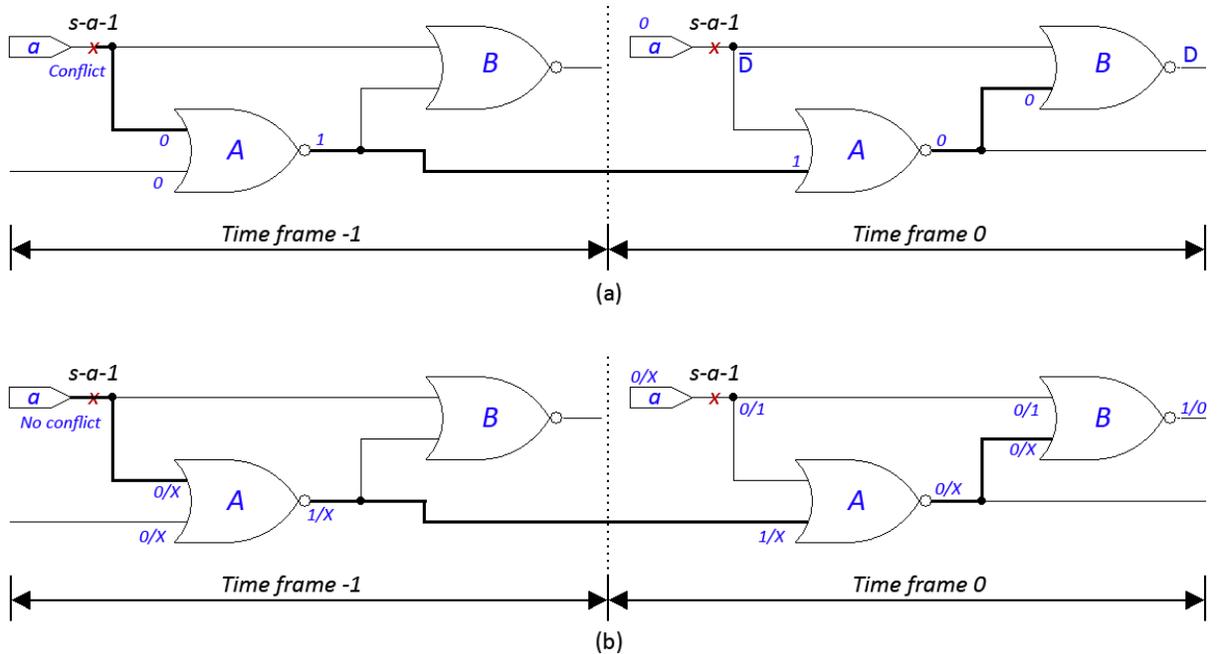


Figure 2.10 5-value logic vs. 9-value logic

As illustrated by Figure 2.10(b), a test for this fault exists under the 9-value logic system. The over specification of the 5-value system rendered the fault untestable. Under the 9-value logic system, we only require $a_0 = 0/X$ and $A_0 = 0/X$ as objectives. By following the same PODEM reasoning, a solution can be found.

2.4.5 State Space Reduction

Given the iterative logic array model, since a fault is present in every time-frame, deterministic test generation algorithms search the state space of nine logic values: 0/0, 0/1, 0/X, 1/0, 1/1, 1/X, X/0, X/1, X/X for each flip-flop in every time-frame. In [Hamzaoglu00], the authors proposed a new technique which reduces the search space by avoiding the assignment of the logic value 0/1 or 1/0 for any flip-flop in the excitation time-frame. The technique presented is based on the following theorem.

Theorem 0 *If a fault in a sequential circuit is testable, there exists a test sequence that detects this fault without assigning opposite good and faulty logic values, i.e., 0/1 or 1/0, to any flip-flop in the excitation time-frame.*

If a fault requires the assignment of 0/1 or 1/0 logic value to any flip-flop in the excitation time-frame, it means that this fault would have to be excited in a time-frame prior to this initial excitation time-frame. Hence, such a requirement of 0/1 or 1/0 to any flip-flop for a fault in the excitation time-frame makes this fault combinational redundant. A fault that is combinational redundant is also sequentially untestable. Thus, if a fault is sequentially testable, there exists a time-frame that can detect this fault without the assignment 0/1 or 1/0 logic value to any of the flip-flops in the time-frame. If a fault in a sequential circuit is untestable, to prove its un-testability, it is sufficient to search the state space of 0/0, 0/X, 1/1, 1/X, X/1, X/0, X/X logic value assignments for flip-flops in the excitation time-frame. Therefore, it is not necessary to consider the 0/1 or 1/0 logic values for any flip-flop in the excitation time-frame.

Chapter 3

Techniques for Enhancing Sequential ATPG

3.1 Techniques for Enhancing Sequential ATPG

3.1.1 *Unjustifiable State Relaxation*

Consider a state S , if there exist no input sequence that can bring the circuit from a fully unspecified state, where all flip-flops' values are unknown, to S , then S is *unjustifiable* or *unreachable*. On the other hand, S is said to be *justifiable* or *reachable* if there exists at least one such sequence. Knowledge of unjustifiable states is also useful as the ATPG can avoid wasting time exploring a non-solution space. Sequential ATPGs like HITEC and ATOMS keep a list of unjustifiable states for each fault. The ATPG would check the list before attempting to justify a state; therefore avoiding redundant and repeated justification effort.

Under a complete search, a state can be declared unjustifiable if the process backtracks beyond justifying the state during the justification phase, provided that the process did not abort the justification of any preceding state due to backtrack limit, time limit or sequence length limit. Interestingly, given a state that has been declared unjustifiable by the search process, not all the flip-flop objectives are responsible for the state being unjustifiable. In a sense, concerning the state's unjustifiability, the target state may have been over-specified. Using information obtained during the search process, a state can be relaxed further before being recorded into the unjustifiable state list.

Let us examine an example justification case illustrated in Figure 3.1(a). On the flip-flops array $Y_0Y_1Y_2$, we need to justify the state $1/X\ 0/X\ 1/X$. The required objectives are $Y_0 = 1/X$,

$Y_1=0/X$, $Y_2=1/X$ or simply $A=1/X$, $B=0/X$ and $C=1/X$ respectively. Assuming a single-objective backtrace search from the hardest to justify objective to the easiest to justify, the objectives' order is $A=1/X$, $B=0/X$ and $C=1/X$. The search shall start from $A=1/X$, backtracing back to $a=0/X$. A second backtrace leads to $b=1/X$. At this point, the first objective has been satisfied but the second objective has been violated (Figure 3.1(b)). The search backtracks and eventually finds no solution for the first objective; thus declares the state to be unreachable. Evidently, the state $1/X\ 0/X\ 1/X$ is unreachable; however, upon closer examination, $Y_0=1/X$ and $Y_1=0/X$ are sufficient to cause the state to be unreachable; objective $C=1/X$ was never "touched" (backtraced from nor violated). Thus, from the search, one can record $1/X\ 0/X\ X/X$ as unjustifiable. The new unreachable state is more desirable as it encompasses more sub-states than the originally deduced state.

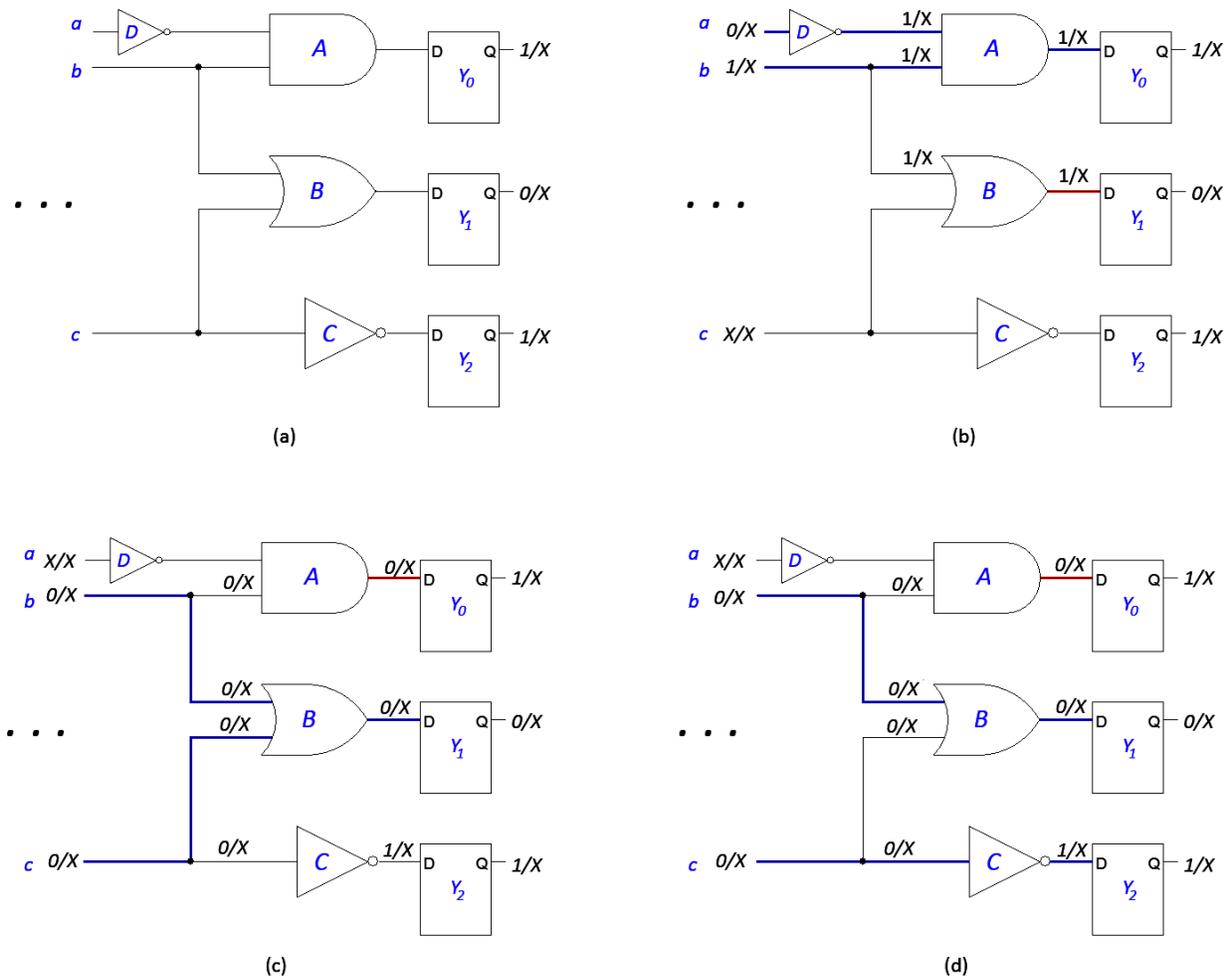


Figure 3.1 Examples for unreachable state relaxation

Figure 3.1(c) shows how objectives which were satisfied but never backtraced from or violated do not matter. The objectives order here is $B=0/X$, $C=1/X$ and $A=1/X$. Backtracing from $B=0/X$ leads us to $b=0/X$ and $c=0/X$. The objective $C=1/X$ has already been satisfied by the assignments to b and c and thus is skipped. But the objective at A has been violated. Backtracking will lead to the violation of $B=0/X$. $C=1/X$ would also be violated as the search backtrack but since the search considers B before C , the violation of $C=1/X$ is never recorded. Again, in this case, $Y_0=1/X$ and $Y_1=0/X$ are sufficient to cause the state to be unreachable.

We propose low cost techniques for identifying relax-able unjustifiable states. During justification, our technique simply keeps track of the objectives which have been backtraced from and/or violated. Upon declaring a state unreachable, before adding it to the unjustifiable states list, we mask out all flip-flops which have not been backtraced from or violated to X/X . The new state would allow for at least the same or more unreachable state detection.

It is worth noting that the order of the objectives can affect the resulting relaxed state. Consider the example in Figure 3.1(d) where objectives are considered from easiest to hardest, from $C=1/X$ to $B=0/X$ and then $A=1/X$. The search would have to backtrace from $C=1/X$ and $B=0/X$ to $b=0/X$ and $c=0/X$ before seeing a violation at gate A . Backtracking will lead to a violation of the objectives at B and C . Ultimately, the state $1/X\ 0/X\ 1/X$ must be declared unreachable but cannot be relaxed further since all three objectives have been violated during the search. Extending from this example, search algorithms that implement multiple objectives backtrace can only apply this technique with limited effect.

3.1.2 Simple Non-chronological Backtrack

Upon backtracking, the search reverses or resets the most recent PI or PPI assignment to an unknown value and then continues searching. Under single frame reverse time processing, backtrack is limited to the time-frame, the ATPG justifies the current state and

then moves on to justify the preceding state if necessary. When the ATPG backtracks and determines a state to be unreachable, it would backtrack to a decision made in the succeeding time-frame. However, only decisions made on flip-flops would affect the preceding time-frame.

Thus we propose a simple non-chronological backtracking technique where, upon moving up from failing to justify a state, we continue to backtrack until a decision at a flip-flop on the succeeding state has been reversed or reset. Effectively, we skip all branching at the inputs which would have no consequence on the preceding state.

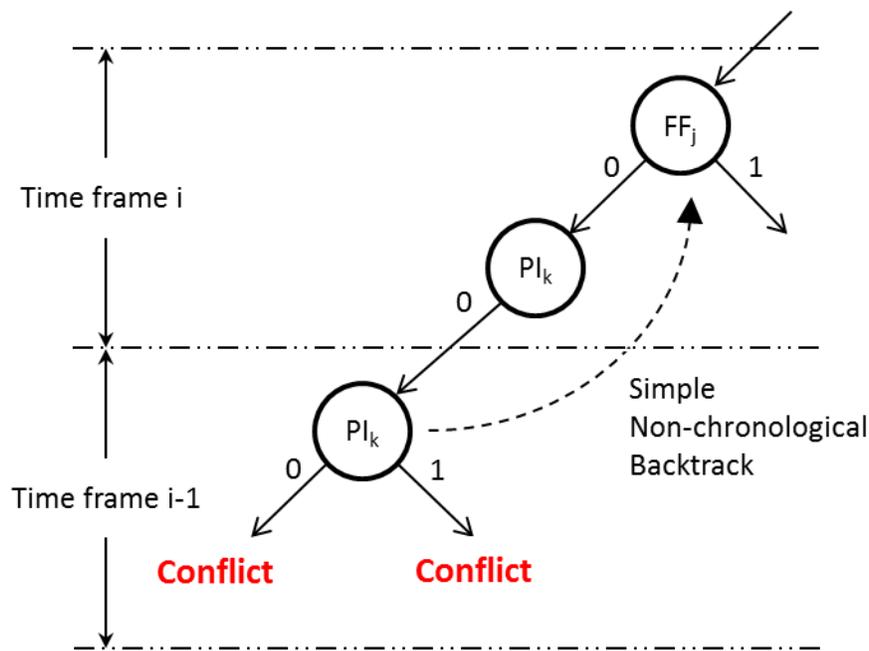


Figure 3.2 Simple Non-chronological Backtracking

Figure 3.2 illustrates how this technique can be achieved. Upon returning from time-frame $i-1$, since reversing PI_k in time-frame i would have no effect on the justification objectives of the state subsequently in time-frame $i-1$, we continue backtracking to reversing FF_j . This technique allows us to avoid searching the non-solution space and save time. If used in conjunction with the unjustifiable state relaxation technique outlined in the previous section, information from the search in the lower time-frame ($i-1$) can be used. The ATPG can then backtrack until it reaches a decision, in time-frame i , involving a PPO whose flip-

flop objective has been backtraced from or violated during the justification of time-frame (i-1)

In an ATPG that utilized both FTP and RTP, the technique is particularly useful when the ATPG's justification phase backtracks to either the excitation time-frame or one of the propagation time-frames (from time-frame 0 and up). Figure 3.3 illustrates the potential amount of effort that could be saved with this technique. In this example, the time-frame -1 has been determined to be unjustifiable and no decision was made at any PPI during the justification of time-frame 0. This means that the flip-flops responsible for time-frame -1 were set during the fault propagation phase at time-frame 0. Our technique allows the ATPG to backtrack all the way to the propagation phase of time-frame 0, skipping all the decision during the justification and propagation of time-frame 1 and 2 which would lead to time-frame -1 being unreachable regardless.

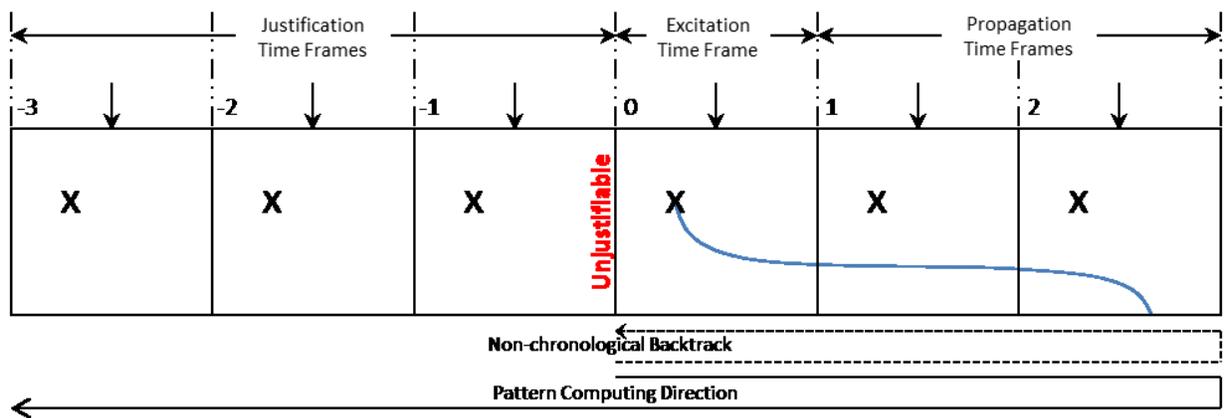


Figure 3.3 Non-chronological Backtrack Example

3.1.3 Loop Control

As discussed earlier, to avoid redundant searches, sequential ATPG backtracks whenever it encounters an *illegal state*. A state is deemed illegal by one of two criteria. The first criterion is the state has been determined to be unjustifiable through the search discussed earlier. The second criterion is state repetition, i.e., the current state is a sub-state of another state visited earlier, forming a loop in the state transition graph. If the test generator continues to circle the loop, the state will keep repeating and eventually the

justification process will be deadlocked. Among the two types of illegal states, only unjustifiable state can be reused. Therefore, it is advantageous to discover and record as many unjustifiable states as possible. In deterministic sequential ATPG such as HITEC and ATOMS, before attempting to justify any time-frame, the ATPG checks the state against two unreachable states lists. One of them is a global unreachable states list learned thus far, which can be used for every faults, and the other one is a list applicable only to the current fault. Upon discovery of an unreachable state, it is added to the global unreachable states list if all known values it contains are for the good circuit portion only (all unknown in the faulty circuit portion); otherwise, the state is added to the current fault's own unreachable list.

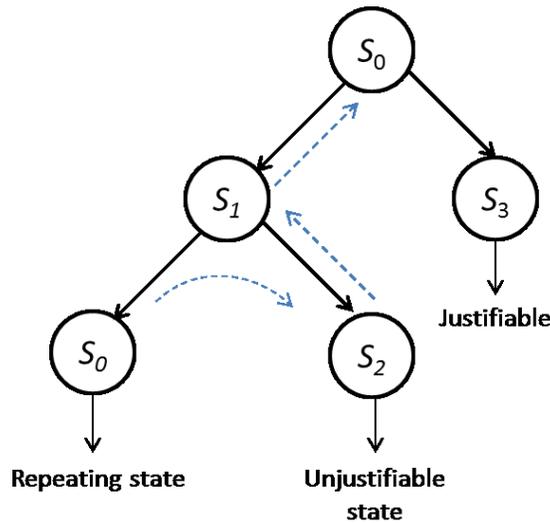


Figure 3.4 Example STG with state repetition

Let us consider the simple state transition graph in Figure 3.4 as an example. The graph shows the states explored by the ATPG. The dashed lines represent the flow of backtrack transitions. Starting from trying to justify state S_0 , the ATPG justified it with a preceding state S_1 and proceeds to justifying the state S_1 . The justification of S_1 leads the ATPG to state S_0 , upon which the test generator backtracks again. If S_0 is a necessary condition to reach S_1 , S_0 may be an illegal state due to state repetition, but we do not know for sure at this point. Note that S_0 is illegal only under the presence of S_0 in one of the succeeding justification time-frames. Another attempt to justify S_1 resulted in another preceding state S_2 , which

proved to be unjustifiable. In this situation, S_2 is identified as an unreachable state. As the ATPG backtrack through state S_1 to S_0 , however, S_1 cannot be declared unreachable since S_0 has not yet been proven unreachable. As show in the figure, S_0 is actually reachable through another state S_3 . Therefore S_1 is reachable through S_0 and declaring it unreachable would have been erroneous. Thus, the ATPG cannot declare a state unreachable simply if one of the preceding state was deemed illegal due to state repetition.

We propose a new technique which would allow for better unreachable state identification under the presence of state repetition. Our technique keeps a separate temporary list of illegal states aside from the global unreachable states list and the faults' individual unreachable states lists. States which have been backtracked through but encountered state repetition during the justification search are added to this list. Repeated states are also added to this temporary illegal states list. The ATPG checks this list along with the other two unreachable states lists for illegal states before attempting to justify a state. If the ATPG ever backtrack to the time-frame earliest of the loops encountered, the states in this temporary list are added the other two lists. If the ATPG found a test sequence before that, the temporary list is cleared.

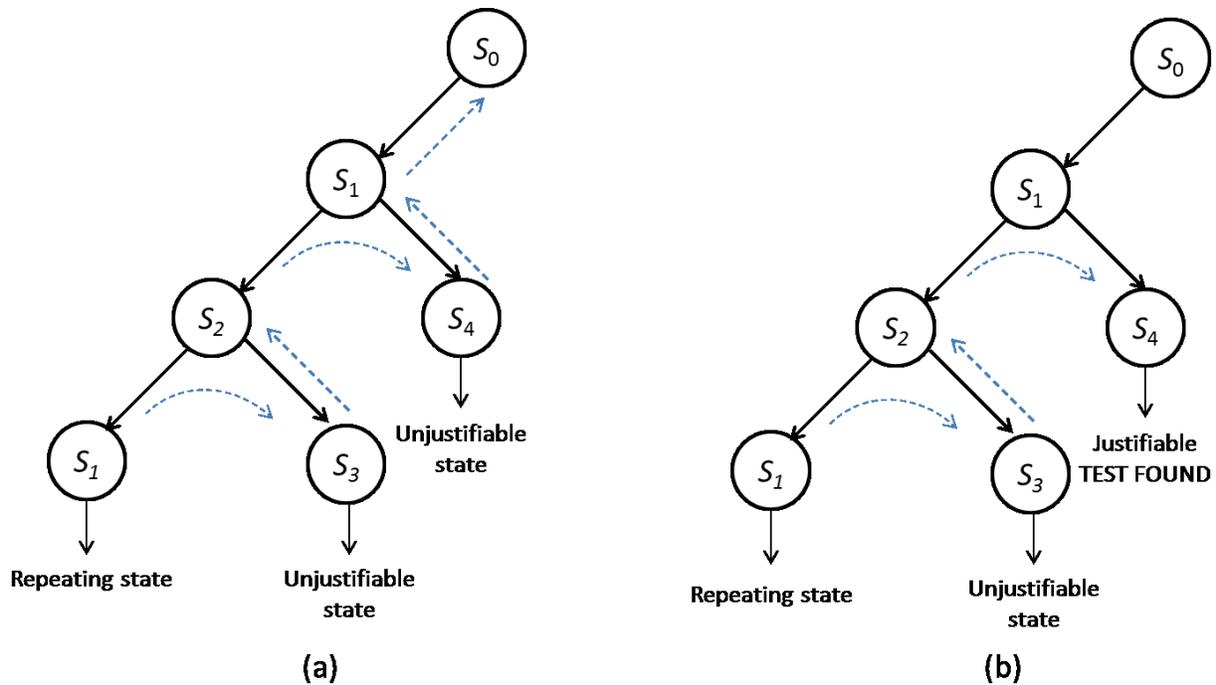


Figure 3.5 Single loop example for Loop Control

To illustrate how our technique works, let us consider the example shown in Figure 3.5(a). Again the solid arrows represent state transitions and the dashed arrows represent the flow of the ATPG's backtrack transition. The ATPG encounters S_1 when it is trying to justify S_2 . S_1 is illegal in this justification process as long as it is also present in one of the preceding time-frames. Thus S_1 is added to the temporary illegal states list that our technique maintains. As the ATPG backtracks from S_3 to S_2 , S_3 is added to one of the two main unreachable states lists since state S_3 being illegal does not depend on the state of any preceding time-frame. When the ATPG backtracks from S_2 to S_1 , however, S_2 cannot be added to the unreachable states lists so long as S_1 remains not unjustifiable. But, at the same time, S_2 is illegal as long as S_1 is present in one of the preceding time-frames. Therefore S_2 is also added to the temporary illegal states list. It is clear at this point that as long as we are trying to find justifying sequence for S_1 , trying to justify any states in the temporary illegal states list would only lead to unnecessary redundant justification effort. When we backtrack to S_1 , we are presented with two different situations. We could find S_1 justifiable through S_4 as illustrated in Figure 3.5(b). If that is the case, since all the states in the temporary illegal states list depend on S_1 being unreachable, the list must be cleared

when S_1 is found reachable. On the other hand, if we cannot find a justifying sequence for S_1 and have to backtrack from S_1 back to S_0 , it is sufficient to conclude that all the states in our temporary illegal states are in fact unreachable and thus can be added to the two unreachable states lists. The temporary illegal states list would be cleared and the search continues.

Figure 3.6 illustrates an example to show how our technique handles multiple instances of state repetition. The small number above each states in the STG represents the time-frame in which the ATPG is currently trying to justify the state. When ATPG encounters S_2 at time-frame -4, S_2 at time-frame -2 is identified as the time-frame through which when the ATPG backtracks, all the states in the temporary illegal states list would be added to the two unreachable states lists. However, when the process encounters S_1 at time-frame -5, S_1 at time-frame -1 is identified as the top time-frame which has been repeated, replacing the previously selected S_2 at time-frame -2. When the ATPG backtracks through the very top state S_2 , even though the justification process encountered S_1 only on the right branch, S_3 on the left branch is still only temporarily illegal under the presence of S_1 since S_3 depends on S_2 and S_2 depends on S_1 . Our technique correctly handles the temporary illegal states by waiting until backtracking through time-frame -1 before merging the temporary illegal states list with the two unreachable states lists. The case of justifying S_4 at time-frame -3 after S_6 at time-frame -2 illustrates how our technique can help avoid considerable amount of unnecessary search. Without our technique, the ATPG would have no preceding states to compare it to and would waste considerable time searching for a solution at it did in the right branch of S_2 at time-frame -2.

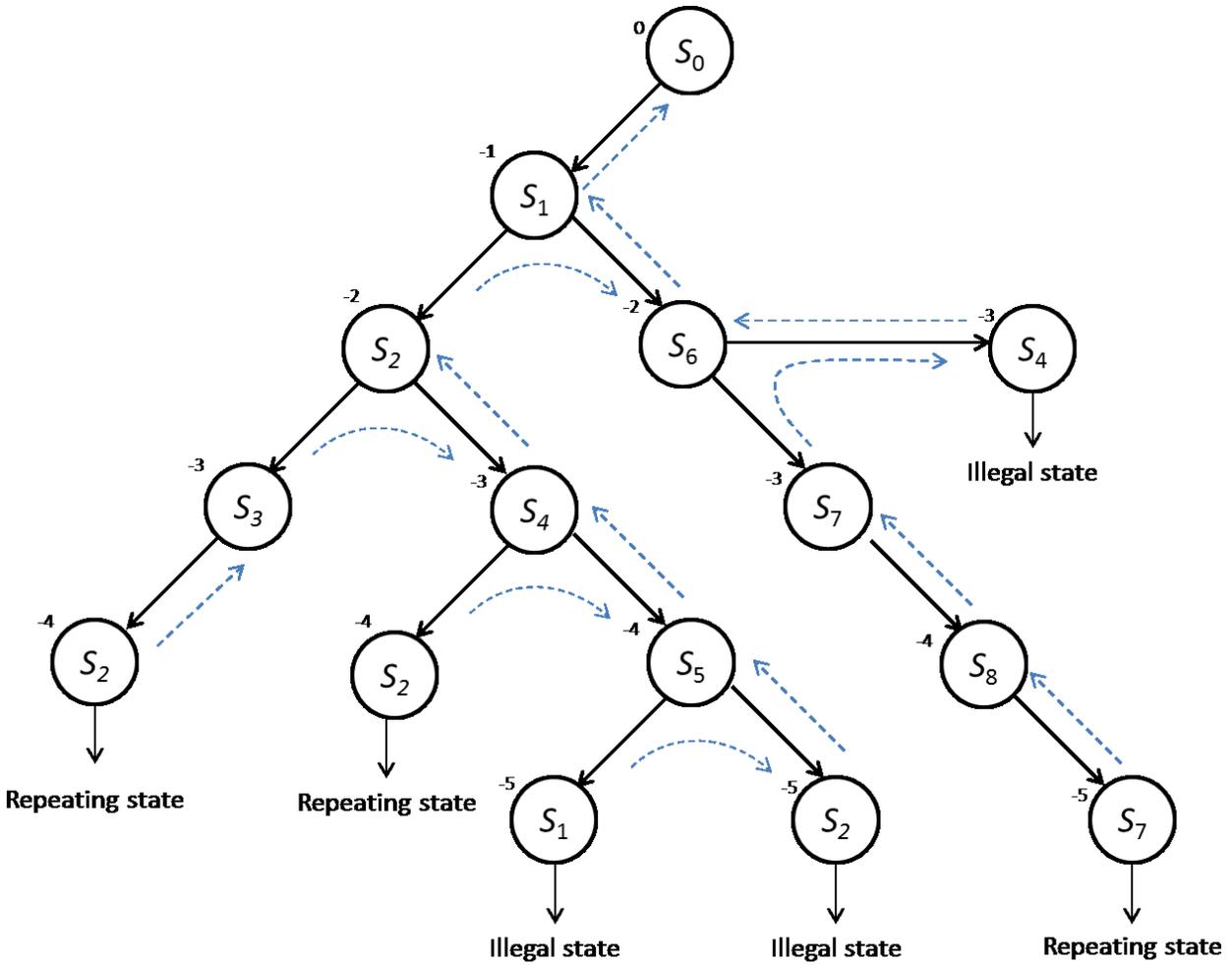


Figure 3.6 Loop Control with multiple state repetition instances

In the case of state repetition of S_7 at time-frame -3 and -5, technically S_7 and S_8 can be declared unreachable as soon as the process backtracks through S_7 in time-frame -3. The problem of keeping track of all the loops in the STG and their intersection is complex. It is not yet tackled in this thesis but should be explored in future works.

3.1.4 Search Space Reduction

We propose a new technique for reducing the size of the search space which needs to be explored by a sequential deterministic test generator. This technique reduces the justification search space by avoiding to consider the logic values 0/1 and 1/0 for any flip-flop in the excitation time-frame and in any preceding justification time-frames. The search

space for the justification process in the justification time-frames therefore consist of only 7 values: 0/0, 0/X, 1/1, 1/X, X/0, X/1 and X/X.

It has been shown that a deterministic test generator can benefit greatly from a heuristic during its search. However, the justification of a value 1/0 or 0/1 at a flip-flop requires the search to trace back to a fault site in one of the preceding time-frames, of which there are numerous. The current heuristic, SCOAP, lacks the capability to guide the backtrace process to a fault site. At the same time, since the current fault model is the single stuck-at fault model, a backtrace to the fault site does not always guarantee a decision at a PI or PPI (e.g., backtrack objective is 1/0 or X/0 for a s-a1 fault). Thus we decided that it might be more beneficial to forego searching the search space under a 1/0 or 0/1 decision at a flip-flop at time-frames less than or equal to 0. Since the search space varies exponentially with the number of values a PPI can take, in this case, our technique allows the search to avoid considerable amount of unnecessary search if the fault is testable and there exist a sequence that can detect the fault without assigning opposing good and faulty values to a flip-flop in the excitation and state-justification time-frames.

Since there are cases where the justification of a time-frame requires the assignment of 1/0 or 0/1 in a flip-flop in the previous time-frame, this is not a complete technique. Unreachable states found, however, can still be counted as pseudo unreachable states. That is under the search space constraint imposed by our technique, those states are unreachable.

Note that all of the non-negative time-frames have been constrained during excitation and propagation of the fault, unjustifiable states can only be found during the justification of negative numbered time-frames. That is failing to justify time-frame 1 does not imply that the current state of time-frame 1 is unjustifiable. Using our proposed technique, however, the search space of the propagation time-frames includes all 9 values of the 9-valued logic model while the search space of the excitation time-frame and the justification time-frames includes only 7 values (excluding 1/0 and 0/1). Hence, all pseudo unreachable states found

while using our technique can only be used to compare against time-frame $i \leq 0$, i.e., these states are only unreachable under a 7-value justification search space.

3.2 Experiment Results

3.2.1 *Our ATPG*

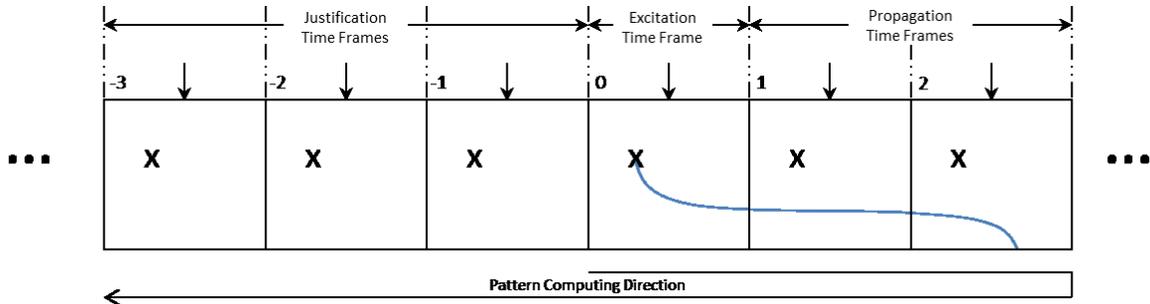


Figure 3.7 Our ATPG pattern generation flow

We implemented our own sequential ATPG to test the effect of each of our proposed techniques. Our ATPG implements both forward time processing and reverse time processing. We excite the fault in time-frame 0. The ATPG would then try to propagate the fault effect to the PO of one of the non-negative time-frames. Finally, from the time-frame in which the fault effect is observed at a PO, the ATPG attempt to find a justification sequence from a fully unknown state. Figure 3.7 shows the computing direction of our APTG. During FTP, the circuit is rolled out multiple time-frames at a time. The higher number the time-frames are rolled out, the easier it is to justify the propagation time-frames; however, it also makes it harder for use to implement forward time state repetition without incurring considerable execution time trade-off. We found that unrolling the 10 time-frames at a time produced reasonable performance. For RTP, the time-frames are processed one time-frame at a time. We also implemented basic unreachable state recognition and state repetition recognition for this core ATPG.

frames in the unrolled sets incurs costly performance overhead. Thus it is performed only before unrolling the next set of time-frames.

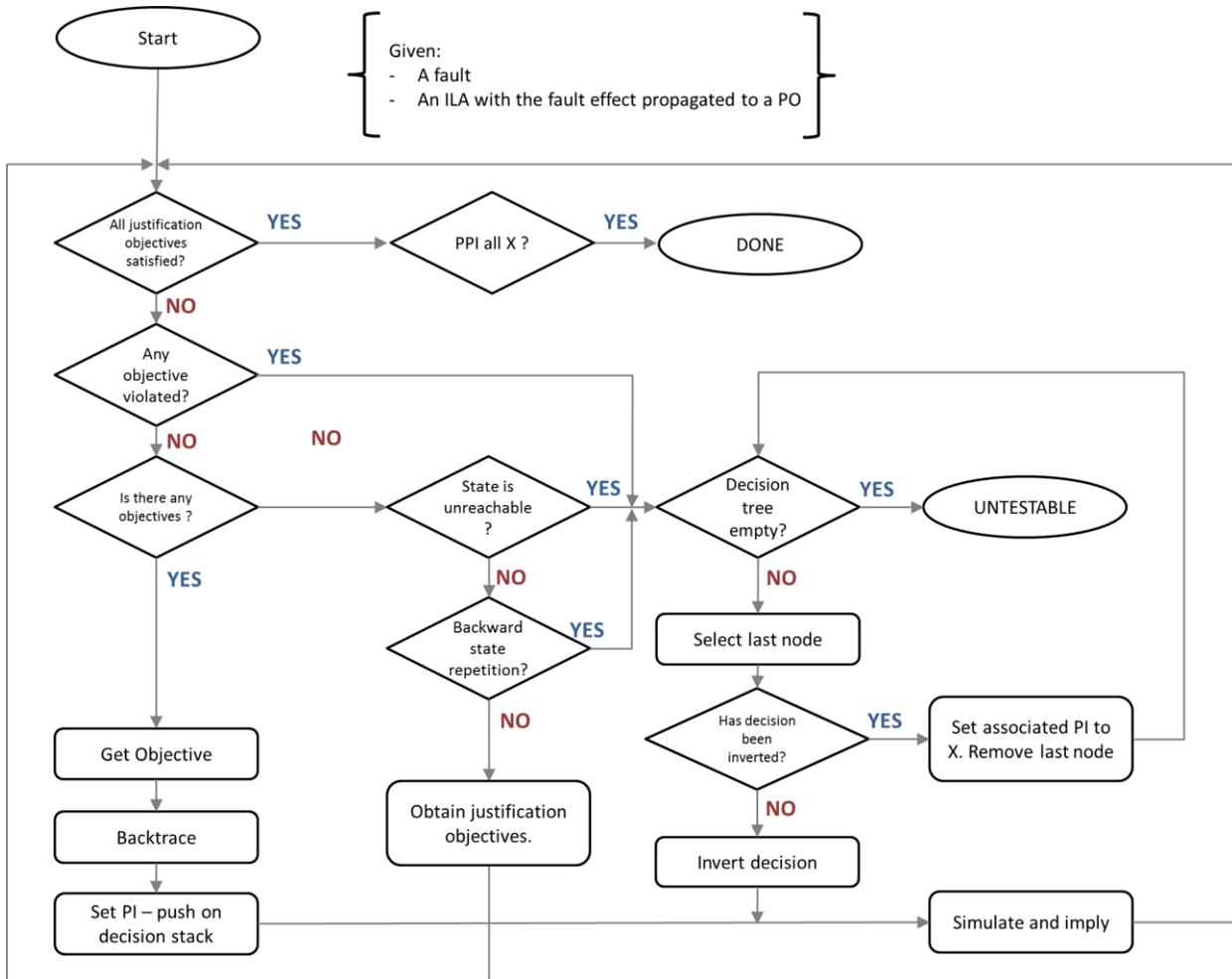


Figure 3.9 ATPG justification flow

Figure 3.9 details the flow of the algorithm during the fault justification phase. The justification phase starts after the propagation phase and requires that the fault has been excited and propagated to a PO. The ATPG checks for state repetition and unreachable state before attempting to justify a state. If the state passes all the conditions, then all justification objectives are collected. If any objective is violated then the ATPG must backtrack. If the ATPG backtracks past the starting point of the justification phase then the ATPG continues backtracking in the propagation phase. If all the justification objectives are all fulfilled then the state has been justified. The ATPG would then either attempt to justify

the preceding state, if a decision has been made at a flip-flop of this time-frame, or to return the justification process as successful.

In conjunction with our ATPG, we also implemented a simple fault simulator. Upon discovering a test pattern, all remaining untested faults are simulated with this pattern. If any fault is detected, we do not have to find a test pattern for it later and can now remove it from the fault list. We initially start with a backtrack limit of 100. This limit is increased tenfold to attempt to target again for the aborted faults. This increase in the backtrack limit is repeated until either a fault coverage is 100% or until it reaches an user defined limit. Once the test generation is completed, the fault simulator is invoked one last time to simulate all of the remaining untested faults. This time, all of the previously found sequences are appended together into one large vector sequence to maximize its fault detecting potential. Our simulator is very simple without any advance techniques; our execution time could suffer when the number of vectors generated is large and/or the number of faults is large.

3.2.2 Experimental Results

This section presents the results for the ISCAS'89 benchmark circuits. The programs were written using C++ and were simulated on a 2.66Ghz, Core 2 Duo, 4GB RAM machine, running CentOS 2.16.0. For each circuit, the backtrack limit is set to 10000. A maximum propagation time-frame expansion of 250 and a maximum justification time-frame expansion of 250 are used for circuits with less than 2000 gates, whereas a maximum backtrack limit of 1000 are used for circuits with more than 2000 gates.

Circuit	#Det	#Unt	#Abt	Time (s)	Vectors
s298	250	10	48	32.07	136
s344	319	5	18	10.45	212
s349	320	7	23	14.99	224

s382	84	0	315	309.09	39
s386	308	63	13	10.38	357
s400	87	6	333	311.76	37
s444	64	14	396	340.62	46
s526	72	1	482	330.81	59
s641	394	2	71	24.64	459
s713	476	42	63	23.15	514
s820	502	18	330	181.86	280
s832	492	30	348	204.40	251
s1196	1239	3	0	14.47	659
s1238	1283	72	0	12.73	663
s1423	631	9	875	1267.08	222
s1488	1010	20	456	461.04	147
s1494	1002	30	474	436.66	136
s5378	3182	111	1310	1165.12	3041

Table 3.1 ATPG results without any technique implemented

Table 3.1 presents the results from our base-line ATPG without any additional techniques implemented. The second column gives the total number of faults detected by the generated vectors. Column 3 give the number of untestable faults found and column 4 reports the number of faults that were neither detected nor found untestable. The last two columns present the execution time and total number of vectors generated. The execution time for our base-line ATPG includes the time for pre-processing the circuit and fault simulation time. Since our fault simulator is a very basic one, its effect on the ATPG's run time is acute for large circuits such as *s5378*.

Circuit	#Det	#Unt	#Abt	Time (s)	Vectors
s298	249	14	45	33.96	159
s344	319	5	18	9.78	189
s349	320	7	23	13.89	192

s382	84	3	312	299.17	39
s386	308	64	12	9.34	357
s400	87	9	330	323.67	37
s444	64	14	396	358.47	53
s526	283	1	271	282.96	242
s641	394	2	72	24.59	459
s713	476	42	63	23.07	514
s820	502	18	330	147.88	280
s832	492	30	348	175.65	251
s1196	1239	3	0	10.17	659
s1238	1283	72	0	12.93	663
s1423	631	9	875	1106.99	222
s1488	1010	20	456	419.25	147
s1494	1002	30	474	412.38	136
s5378	3182	221	1200	1340.21	3041

Table 3.2 ATPG results with Unjustifiable State Relaxation

Table 3.2 presents the results from our core ATPG with Unjustifiable State Relaxation implemented. While the ATPG detected one less fault for s298, for all the circuits, the number of faults detected are equal but the number of untestable faults detected are at least equal to or greater than the results from Table 3.1. Circuits with a high number of flip-flops can benefit greatly from this technique as there is a higher chance that a flip-flop is not responsible for a state being unjustifiable as evident from the result of s5378. The number of untestable faults detected for this circuit nearly doubled its result in Table 3.1.

Circuit	#Det	#Unt	#Abt	Time (s)	Vectors
s298	250	15	43	28.03	136
s344	319	6	17	9.07	212
s349	320	8	22	13.24	224

s382	98	4	297	224.25	77
s386	308	64	12	9.79	357
s400	103	11	312	252.16	75
s444	105	18	351	262.29	115
s526	302	6	247	198.51	705
s641	394	41	32	15.42	459
s713	476	58	47	19.10	514
s820	564	18	268	138.12	324
s832	549	30	291	159.68	303
s1196	1239	3	0	14.08	659
s1238	1283	72	0	17.54	664
s1423	631	10	874	1139.96	222
s1488	1028	20	438	444.24	162
s1494	1108	30	368	402.54	213
s5378	3182	111	1310	1288.72	3041

Table 3.3 ATPG results with Simple Non-chronological Backtrack

Table 3.3 presents the results from our ATPG with Simple Non-chronological Backtrack implemented. Using Simple Non-chronological Backtrack (SNB) has enabled us to detect more faults and even more untestable faults, while at the same time reduce the overall runtime. It goes to show that for a relatively simple technique, SNB was a good addition to the available techniques for deterministic sequential ATPG.

Circuit	#Det	#Unt	#Abt	Time (s)	Vectors
s298	263	16	29	29.85	188
s344	319	7	16	10.29	212
s349	320	9	21	14.61	224

s382	84	0	315	304.52	39
s386	308	63	13	10.04	357
s400	87	6	333	356.08	37
s444	64	14	396	328.19	46
s526	283	1	271	293.45	242
s641	394	61	12	8.236	459
s713	476	104	1	4.414	514
s820	613	19	218	150.681	408
s832	606	31	233	136.563	418
s1196	1239	3	0	9.734	659
s1238	1283	72	0	12.791	663
s1423	631	9	875	1049.434	222
s1488	1128	22	336	361.922	253
s1494	1124	32	350	357.227	242
s5378	3182	111	1310	1211.73	3041

Table 3.4 ATPG results with Loop Control

Table 3.4 gives the results from our ATPG with Loop Control implemented. Our ATPG with Loop Control returned a greater number of detected faults and untestable faults than with SNB. When generating test patterns for *s641*, *s713*, *s820* and *s832*, our ATPG encounters many loops, thus benefited greatly from Loop Control.

Circuit	#Det	#Unt	#Abt	Time (s)	Vectors
s298	265	24	19	17.612	271
s344	327	5	10	6.708	272
s349	331	7	12	10.42	260
s382	84	1	314	312.673	39
s386	314	69	1	1.7	414
s400	287	7	132	171.163	216
s444	323	15	136	177.778	224
s526	72	9	474	341.564	59
s641	404	2	61	18.033	505
s713	476	42	63	21.215	520
s820	547	35	268	96.517	391
s832	589	51	230	88.498	465
s1196	1239	3	0	9.952	659
s1238	1283	72	0	12.682	663
s1423	631	9	875	1380.26	222
s1488	1338	40	108	129.495	1290
s1494	1345	51	110	131.415	1234
s5378	3196	106	1301	726.777	3039

Table 3.5 ATPG results with Search Space Reduction

Table 3.5 gives the results from our ATPG with Search Space Reduction implemented. Compared to the other three techniques, our ATPG with Search Space Reduction (SSR) detects the highest number of faults. It is worth noting that although our ATPG with SSR cannot guarantee that the untestable faults returned are valid; however, all returned untestable faults do not conflicts with any faults detected by our ATPG by any of the other three techniques or any combinations thereof.

Table 3.6 presents the results from our ATPG with all techniques implemented. The combined results from all experiments are presented in Table 3.7. The columns in the table present the total number of detected faults, the total number of faults proven to be

untestable, the total number of aborted faults, and the total test generation time in seconds for the ISCAS89 sequential benchmark circuits tested. The results show that each technique helps improve the performance of the test generator. With all techniques implemented, not only was the number of faults detected highest, but the test generation time was also reduced considerably.

Circuit	#Det	#Unt	#Abt	Time (s)	Vectors
s298	265 (1)	36	7	8.79	334
s344	327	9	6	5.56	253
s349	331	11	8	8.95	250
s382	98	14	287	353.74	77
s386	314	70	0	0.96	414
s400	287	20	119	198.07	254
s444	323	25	126	207.59	300
s526	302	16	237	258.85	705
s641	404	62	1	2.35	505
s713	476	105	0	4.19	522
s820	814	36	0	26.19	1596
s832	818	52	0	28.11	1674
s1196	1239	3	0	9.81	659
s1238	1283	72	0	12.60	664
s1423	631	10	874	1014.95	222
s1488	1443	42	1	63.82	1782
s1494	1452	53	1	57.70	1733
s5378	3197	252	1154	1287.11	3067

Table 3.6 ATPG results with all techniques

ATPG	#Det	#Unt	#Abt	Time (s)
w/o any techniques	11715	443	5555	5151.32
w/ Unjustifiable State Relaxation	11925	564	5225	5004.36
w/ Simple Non-chronological Backtrack	12259	525	4929	4636.74
w/ Loop Control	12404	580	4729	4649.762
w/ Search Space Reduction	13051	548	4114	3654.462
w/ all techniques	14004	888	2821	3549.415

Table 3.7 Impact of each technique on sequential circuit test generation performance

Circuit	Our ATPG			HITEC		
	#Det	#Unt	#Abt	#Det	#Unt	#Abt
s298	265	37	6	265	26	17
s344	329	13	0	324	11	7
s349	335	15	0	332	13	5
s382	240	20	139	301	9	89
s386	314	70	0	314	70	0
s400	289	27	110	341	17	68
s444	326	32	116	373	25	76
s526	369	26	160	316	23	216
s641	404	63	0	404	63	0
s713	476	105	0	476	105	0
s820	814	36	0	813	37	0
s832	818	52	0	818	52	0
s1196	1239	3	0	1239	3	0
s1238	1283	72	0	1283	72	0
s1423	631	10	874	723	14	778
s1488	1444	42	0	1444	41	1
s1494	1453	53	0	1453	52	1
s5378	3197	252	1154	3231	217	1155

Table 3.8 Comparison of sequential ATPG results

The results from Table 3.8 demonstrate the effectiveness of our techniques. We compare our results with those reported by HITEC. Under each label, there are three columns presenting the number of faults detected, the number of untestable faults identified and the number of faults aborted. In this experiment, the ATPG will keep running until either fault coverage reaches 100%, backtrack limit reaches 1,000,000 or the run time of the latest iteration exceed 30 minutes. Since Search Space Reduction is used in this experiment, untestable faults found by our ATPG are not valid. However, we have compared our list of untestable faults with the list of faults detected by STRATEGATE [Hsiao97] and found no conflict. Therefore the numbers of untestable faults found by our ATPG are also reported here. The results are encouraging.

Chapter 4

Decision Inversion Problem

4.1 The Decision Inversion Problem

During test generation, suppose a decision leads to a conflict, all side effects of the decision are undone, the decision is inverted, and the ATPG continues the search with this inverted decision. The action effectively blocks out all the backtracing paths that would lead to this conflict again, allowing the ATPG to select alternative paths from the remaining paths. This is perfectly okay in combinational test generation since the complete search space for the single time-frame that is needed is available to the ATPG. It is not so for sequential test generation. The one constraint that limits test generation for sequential ATPG is that the test sequence must start from a fully unknown state. This constraint cannot be easily addressed since sequential ATPG cannot determine which time-frame would be the start of the test sequence. In all 9-valued deterministic sequential test generators, the constraint manifests itself in a problem that may invalidate the soundness of the search.

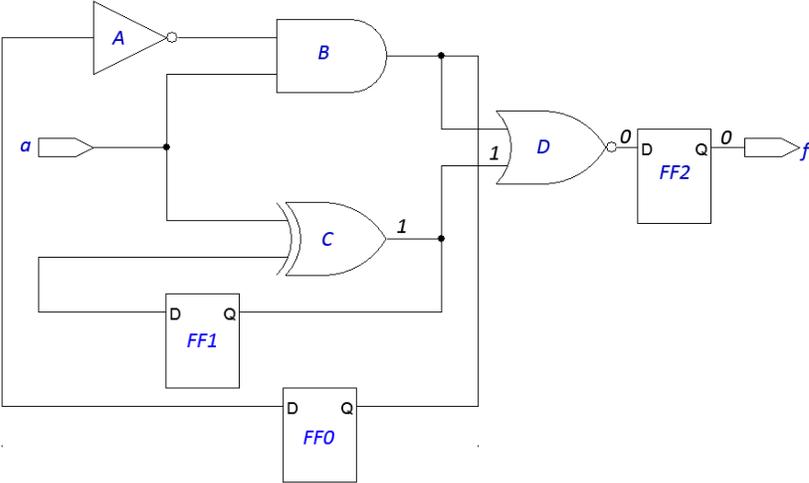


Figure 4.1 Example circuit for the Decision Inversion Problem

Let us consider a simple justification problem in the circuit illustrated in Figure 4.1. The state we want to justify is $FF0\ FF1\ FF2 = XX0$. The state $XX0$ is indeed justifiable by the two vectors “0”, “1” as illustrated in Figure 4.2. The justification objective here is $D_0 = 0$. As we can see here, C is an XOR gate in a close feedback loop with $FF0$. It is clear from examination by hand that C is not initializable from a fully unknown state if $FF0$ starts in an unknown value. At gate D , the ATPG would find a justification sequence for the state if it selects to backtrack through gate B instead of gate C .

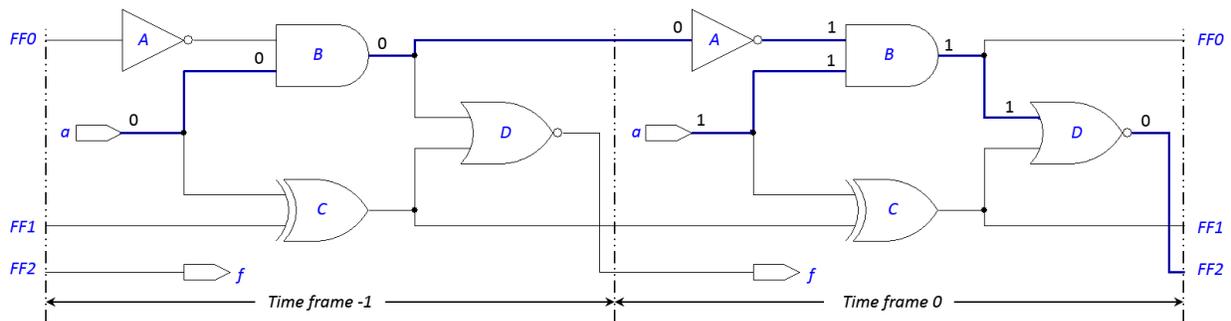


Figure 4.2 Unrolled example circuit – $XX0$ justification

However, heuristics such as SCOAP might suggest the ATPG to prefer justifying through C rather than B . Figure 4.3 shows a justification attempt by backtracing through gate C . At time-frame -2, we would have to backtrack for $C_{-2}=1$ again. Either combination for a_{-2} and $FF1_{-2}$ ($[1,0]$ and $[0,1]$) would lead to state repetition of $X1X$ (time-frame -1) or $X0X$ (time-frame 0) and other two combinations of a and $FF1$ ($[1,1]$ and $[0,0]$) would cause a violation of the objective $C_{-2}=1$. The ATPG would have to backtrack and invert the decision $FF1_{-1}=1$ to $FF1_{-1}=0$. This would cause a conflict with the objective $C_{-1}=0$ and the ATPG must backtrack to invert the decision $a_{-1}=1$ to $a_{-1}=0$. Unfortunately, after backtracing, we again encounter the state repetition of $X0X$ (time-frame 0). Similar to all the preceding time-frames, after backtracking to time-frame 0 and inverting the decision $FF1_0=0$ to $FF1_0=1$, we have a conflict with the objective $C_0=1$. Therefore the ATPG must backtrack again and invert the decision $a_0=1$ to $a_0=0$. This inverted decision will then force the ATPG to backtrack for $D=0$ through gate C again. The ATPG would backtrack to $FF1_0=1$ and would try to justify $X1X$ one more time.

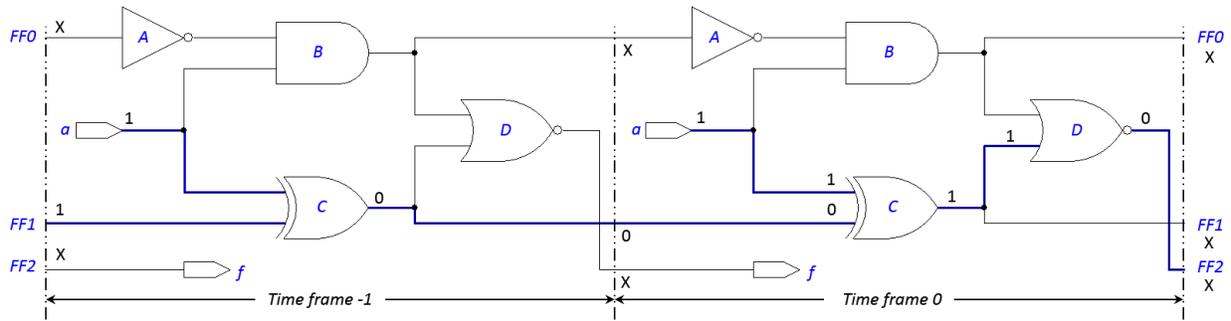


Figure 4.3 Backtracing for $D=0$ through C

If our previous technique (loop controlling) has been implemented, $X1X$ and $X0X$ would have been added to the unreachable states list upon backtrack to time-frame 0. Since both $X0X$ and $X1X$ are in the unreachable states list, we cannot continue. Even though the objective can still be satisfied by setting $FF0_0$, the justification requirement for $FF1_0$ will still cause the ATPG search to fail. Thus the ATPG must ultimately backtrack to reset a to unknown and return state $XX0$ as unjustifiable. As demonstrated in Figure 4.2, state $XX0$ is actually justifiable. Thus demonstrated here, the search process is not sound. Figure 4.4 shows the decision tree for the justification process just described.

More detrimental to the soundness of the ATPG, if the ATPG implement techniques to merge unreachable states list, $X0X$ and $X1X$ would merge to XXX and cause all subsequent attempt to justify any other state to fail. Even if our loop controlling technique is not implemented in this case, C remains uninitializable from a fully unknown state. Hence, instead of calling these two states as unreachable, popular techniques simply call $FF1$ *uninitializable* and avoids backtracing or assigning a value to it. The ATPG would still have to backtrack to invert $a=1$ to $a=0$ in time-frame 0, try to justify $X0X$ again, backtracks to reset a to unknown and return $XX0$ as an unjustifiable state.

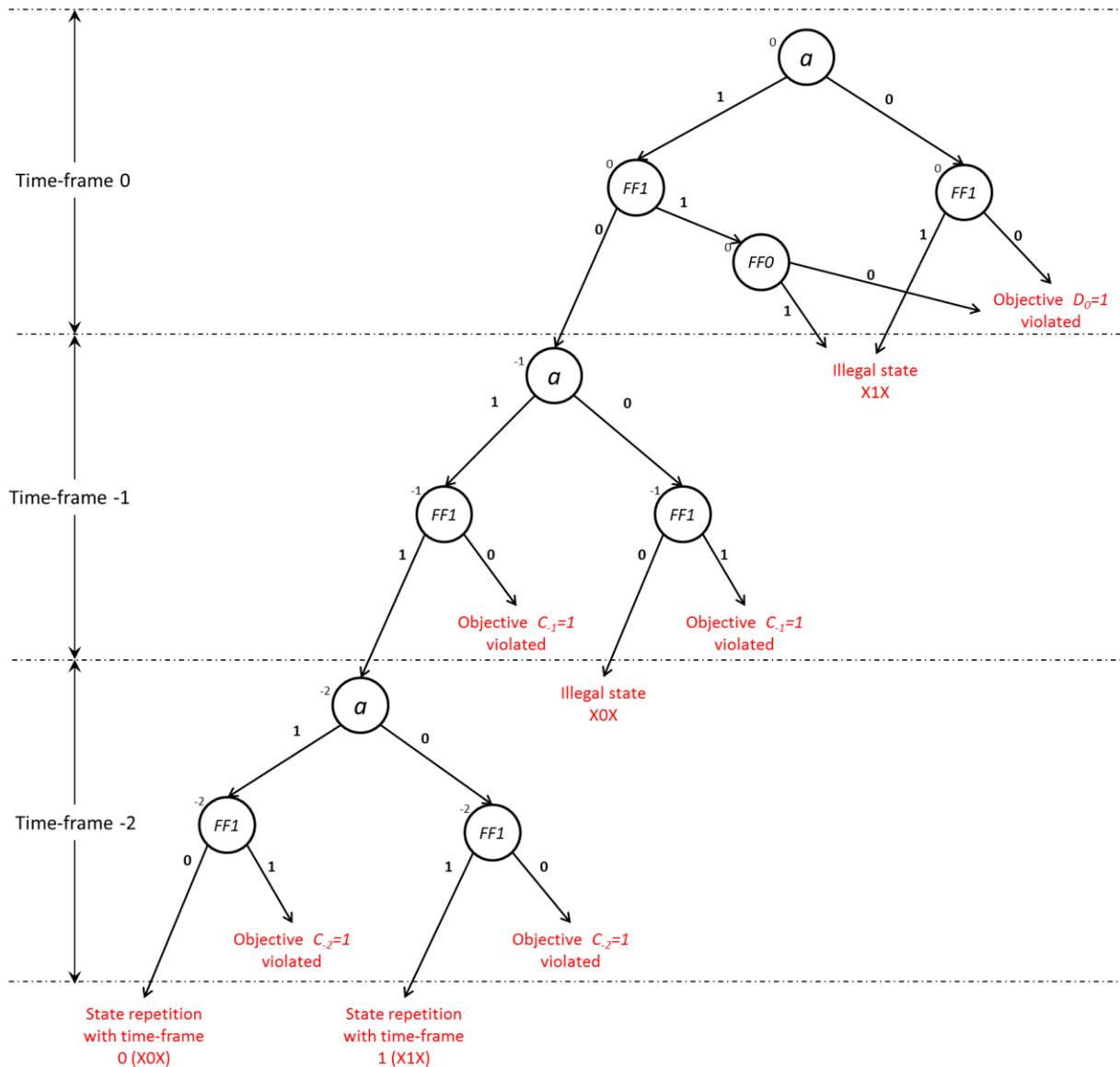


Figure 4.4 Example decision tree

Note that the problem concerning ourselves here is the inversion of a_0 and $FF1_0$, and not the fact that $FF1$ is uninitializable. Suppose that $FF1$ has been identified as uninitializable beforehand and if the ATPG simply avoids making a decision at $FF1_0$ and backtraces through B_0 then all is well. On the other hand, if the ATPG still backtraces to $FF1_0$ and return a conflict instead, it would still go through all the same branches as it did in time-frame 0 or Figure 4.4 and ultimately return the state XX0 as unreachable.

It has been quite a long time since the introduction of the 9-valued logic model for sequential circuit test generation. However, newer ATPG such as ATOMS and more popular ATPG such as HITEC are still implementing the 9-value logic model without addressing this decision inversion problem. Since completeness is one of the few advantages deterministic ATPGs have over simulation-based ATPG, the decision inversion problem is an important one.

4.2 X-Locking

One approach to solve this problem is to employ a different logic system. Gouders *et al* [Gouders91] proposed the CONSEQUENT logic model with an extra bit U added, representing the relation between the good and the faulty machine. Similar but capable of handling the Z state as well, MOSAIC [Dargelas97] implements the 256-valued system. The approach can solve the decision inversion problem but requires us to switch the underlying logic system. This requirement would force us to reexamine the implementation of all previously implemented techniques. Thus it is slightly more desirable if a solution can be found for the 9-valued circuit model.

We observe that the cause comes down to a certain number of flip-flops in the circuit being not initializable from a fully unknown state. If these flip-flops can be identified, a simple solution would be to block decisions on these flip-flops until all the possible combinations on other PPIs and PIs has been tried. It would guarantee that if there is a solution, it would have been found. Without the knowledge of these flip-flops, the best we can do is blocking out all flip-flops and search the PIs first. As demonstrated in the previous section, the solution can involve one or more flip-flops, thus this approach is incomplete. We can also attempt to relax the state before backtracing to another decision or before trying to justify a state to shake off all the unnecessary flip-flops which could be long to this un-initializable from unknown set; however, the case presented in the previous section shows an example where no relaxation is possible.

We propose a technique for solving this problem called X-locking. The idea is to simply systematically lock out all the decisions on all relevant flip-flops. Basically, our technique keeps track of all decisions at flip-flops. After both logics '1' and '0' have been tried out at a flip-flop, we reset the flip-flop back to unknown; but instead of backtracking to the next decision in the decision tree, we lock this flip-flop at unknown and try another search with this flip-flop blocked out. Essentially, we have added a fourth state "locked" to the current logic system ('0', '1', 'X'). Figure 4.5 shows the new decision tree for the circuit in Figure 4.1 with our technique implemented. We can see the solution from Figure 4.2 has been found.

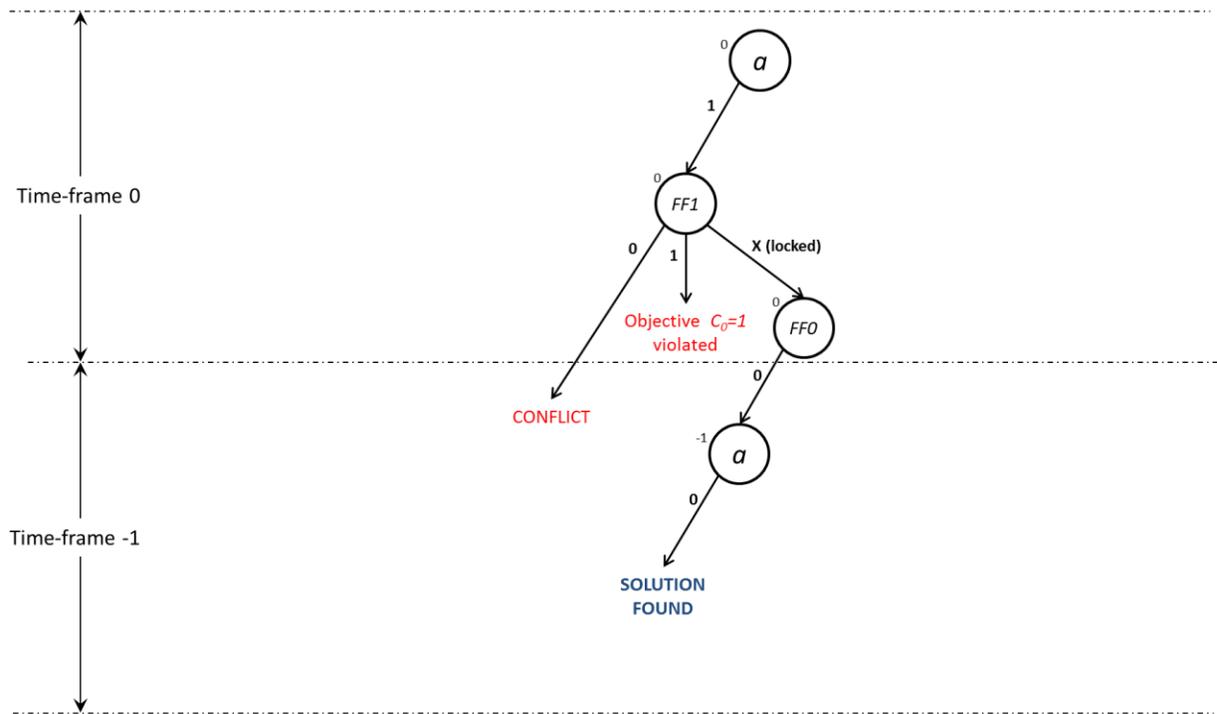


Figure 4.5 Decision tree for an ATPG with X-locking implemented

The advantage for using our technique is that the search is now sound, that is the search will find a solution if it exist, and we are still using the 9-value logic system. However, the search space also been increased exponentially. At the same time, since the ATPG relies on the inversion of decision to lock explored backtracing paths, without a way to propagate the "locked" state of a flip-flop, backtracing performance will suffer. An alternative is to use the backtracing method of the D-algorithm but it is known to be insufficient for certain classes of circuits.

4.3 Experimental Results

To test the effect of our technique, we compare the experiments results of our ATPG with and without X-locking. In this experiment, our core ATPG implementation is the same as with Chapter 3. There techniques from the previous chapter were also implemented: Unjustifiable State Relaxation, Simple Non-chronological Backtrack and Loop Control. Search Space Reduction was not implemented because it invalidates the soundness of the search process. For the experiment without X-locking, the maximum backtrack limit was set to 10000 for circuits with less than 2000 gates and 1000 for circuits with more than 2000 gates. The propagation time-frame expansion limit was set to 250 for all circuits, and the justification time-frame expansion limit was set to 250. Since the ATPG must explore a much larger search space with X-locking implemented, the maximum backtrack limit was doubled for the experiment with X-locking. In both experiment, all false positive untestable faults are caught by the final simulation run. Thus these faults are recorded before the final simulation and are not counted in the final untestable faults count.

Table 4.1 demonstrated the effect of X-locking on sequential ATPG. With X-locking, there are no false positive on untestable faults but the run time increase significantly. An experiment was performed comparing the number of false untestable faults found with and without X-locking, the results are presented in Table 4.2. In this experiment, only circuits with false position untestable faults are reported and test generation is run with a maximum backtrack limit of 10,000 for the ATPG without X-locking and 20,000 for the ATPG with X-locking. Again, we can see that X-locking helped eliminating all false positive untestable faults but require a significantly longer runtime to achieve comparable test results.

Circuit	w/o X-locking				w/ X-locking			
	#Det	#Unt	#False	Time (s)	#Det	#Unt	#False	Time (s)
s298	246	22	40	23.86	247	8	53	290.92
s344	319	9	14	8.36	315	7	20	25.16
s349	320	11	19	12.26	315	9	26	5798.10
s382	98	10	291	290.62	98	4	297	697.10
s386	308	64	12	8.65	303	59	22	29.29
s400	103	16	307	317.22	103	7	316	687.10
s444	105	21	348	303.04	105	17	352	591.53
s526	302	7	246	201.84	302	7	246	548.84
s641	394	62	11	8.01	395	62	10	21.7
s713	476	105	0	4.19	476	99	6	21.84
s1196	1239	3	0	9.75	1239	3	0	9.71
s1238	1283	72	0	12.65	1283	72	0	12.66
s1488	631	10	874	1022.92	631	7	877	3994.71
s1494	1128	22	336	346.94	1128	14	344	995.41
s5378	1124	32	350	367.14	1124	27	355	880.23

Table 4.1 ATPG results with and without X-locking

w/o X-locking					
Circuit	#Det	#Unt	#Abt	Time (s)	#False
w/ All techniques but Search State Reduction and X-locking					
s386	308	64	12	8.657	2
w/ All techniques but X-locking					
s298	265	36	7	8.799	1
w/ X-locking					
w/ All techniques but Search State Reduction					
s386	303	59	22	29.296	0
w/ All techniques					
s298	265	21	22	560.79	0

Table 4.2 Extended ATPG results with and without X-locking

Chapter 5

Conclusion and Future Direction

5.1 Conclusion

Although large amount of work has been conducted in the past to improve the performance of deterministic sequential test generators and significant progress has been made toward generating high quality test patterns, sequential test generation remains a time consuming process. As feature size continues to decrease, digital integrated circuits continue to increase in size and complexity. Therefore new techniques are required to generate better tests in shorter time.

In this thesis, we presented four new techniques for improving the average-case performance of the iterative logic array basic deterministic sequential circuit test generation algorithms for single stuck-at faults. Among the new concepts presented in this thesis are the use of test generation knowledge to identify more unjustifiable states and less specified unjustifiable states. To assess the effectiveness of the proposed techniques, we have developed our own deterministic sequential ATPG and incorporated these techniques into our test generator. As demonstrated through individual experiments, ATPG with our techniques detects more faults and identify more untestable faults while at the same time significantly reduce the test generation time. Under the same condition, with all techniques implemented, our ATPG can execute is less than a third of the original ran time. Compared to the results of HITEC, the results are encouraging. We expect that these techniques will also improve the performance of hybrid sequential circuit test generation systems.

We have also presented an approach for solving the decision inversion problem for 9-valued logic based deterministic sequential circuit test generation algorithms. Upon investigating the problem, it was determined that, to ensure the soundness of the algorithm, the ATPG would need to consider a flip-flop's value locked at unknown along with logic 1 and logic 0. We validated our approach by comparing the number of false positive untestable faults identified by our ATPG with and without the approach in considerable. It is found that the ATPG does not falsely identify any fault as untestable with our technique implemented. However, although the numbers of faults detected are comparable, the ATPG's execution time suffered greatly due the increase in the size of the search space. On the other hand, the decision inversion problem is eliminated.

5.2 Future Direction

Our current technique for solving the decision inversion problem is crude and incur large penalty to the test generator's execution time. However, we believe that the number flip-flops that need to be X-locked can be narrowed down considerably. A flip-flop being uninitializable from a fully unknown state is but one of the criteria for X-locking the flip-flop. The number of criteria, however, does not seem to be numerous. If investigated further, there is a possibility for discovery of the sufficient and necessary criteria for X-locking a flip-flop. Thus, there is potential for reducing the cost in execution time paid for the soundness of the PODEM algorithm in a 9-valued logic based deterministic sequential ATPG.

Reference

[**Abramovici86**] Abramovici, M.; Kulikowski, J.J.; Menon, P.R.; Miller, D.T.; , "SMART And FAST: Test Generation for VLSI Scan-Design Circuits," *Design & Test of Computers, IEEE* , vol.3, no.4, pp.43-54, Aug. 1986[**Auth91**] Auth, E.; Schulz, M.H.; , "A test-pattern-generation algorithm for sequential circuits," *Design & Test of Computers, IEEE* , vol.8, no.2, pp.72-85, Jun 1991

[**Bending84**] Bending, Michael J.; , "Hitest: A Knowledge-Based Test Generation System," *Design & Test of Computers, IEEE* , vol.1, no.2, pp.83-92, May 1984

[**Cheng88**] Cheng, W.-T.; , "The BACK algorithm for sequential test generation," *Computer Design: VLSI in Computers and Processors, 1988. ICCD '88., Proceedings of the 1988 IEEE International Conference on* , vol., no., pp.66-69, 3-5 Oct 1988

[**Cheng89**] Cheng, W.-T.; Chakraborty, T.J.; , "Gentest: an automatic test-generation system for sequential circuits," *Computer* , vol.22, no.4, pp.43-49, Apr 1989

[**Cheng96**] Cheng, K.-T.; , "Gate-level test generation for sequential circuits," *ACM Transactions on Design Automation of Electronic Systems*, vol.1, no.4, pp.405-442, Oct 1996

[**Dargelas97**] Dargelas, A.; Gauthron, C.; Bertrand, Y.; , "MOSAIC: a multiple-strategy oriented sequential ATPG for integrated circuits," *European Design and Test Conference, 1997. ED&TC 97. Proceedings* , vol., no., pp.29-36, 17-20 Mar 1997

[**Fujiwara82**] Fujiwara, H.; Toida, S.; , "The Complexity of Fault Detection Problems for Combinational Logic Circuits," *Computers, IEEE Transactions on* , vol.C-31, no.6, pp.555-560, June 1982

- [**Fujiwara83**] Fujiwara, H.; Shimono, T.; , "On the Acceleration of Test Generation Algorithms," *Computers, IEEE Transactions on* , vol.C-32, no.12, pp.1137-1144, Dec. 1983
- [**Goel81**] Goel, P.; , "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *Computers, IEEE Transactions on* , vol.C-30, no.3, pp.215-222, March 1981
- [**Goldstein80**] Goldstein, L.H.; Thigpen, E.L.; , "SCOAP: Sandia Controllability/ Observability Analysis Program," *Design Automation, 1980. 17th Conference on* , vol., no., pp. 190- 196, 23-25 June 1980
- [**Gouders91**] Gouders, N.; Kaibel, R.; , "Test generation techniques for sequential circuits," *VLSI Test Symposium, 1991. 'Chip-to-System Test Concerns for the 90's', Digest of Papers* , vol., no., pp.221-226, 15-17 Apr 1991
- [**Hamzaoglu98**] Hamzaoglu, I.; Patel, J. H.; , "New techniques for deterministic test pattern generation," *VLSI Test Symposium, 1998. Proceedings. 16th IEEE* , vol., no., pp.446-452, 26-30 Apr 1998
- [**Hamzaoglu00**] Hamzaoglu, I.; Patel, J.H.; , "Deterministic test pattern generation techniques for sequential circuits," *Computer Aided Design, 2000. ICCAD-2000. IEEE/ACM International Conference on* , vol., no., pp.538-543, 2000
- [**Hsiao96**] Hsiao, M.S.; Rudnick, E.M.; Patel, J.H.; , "Alternating strategies for sequential circuit ATPG," *European Design and Test Conference, 1996. ED&TC 96. Proceedings* , vol., no., pp.368-374, 11-14 Mar 1996
- [**Hsiao97**] Hsiao, M.S.; Rudnick, E.M.; Patel, J.H.; , "Sequential circuit test generation using dynamic state traversal ," *European Design and Test Conference, 1997. ED&TC 97. Proceedings* , vol., no., pp.22-28, 17-20 Mar 1997
- [**Ibarra75**] Ibarra, O.H.; Sahni, S.K.; , "Polynomially Complete Fault Detection Problems," *Computers, IEEE Transactions on* , vol.C-24, no.3, pp. 242- 249, March 1975

[**Kelsey89**] Kelsey, T.P.; Saluja, K.K.; , "Fast test generation for sequential circuits," *Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on* , vol., no., pp.345-347, 5-9 Nov 1989

[**Kelsey93**] Kelsey, T.P.; Saluja, K.K.; Lee, S.Y.; , "An efficient algorithm for sequential circuit test generation," *Computers, IEEE Transactions on* , vol.42, no.11, pp.1361-1371, Nov 1993

[**Kirkland87**] Kirkland, T.; Mercer, M.R.; , "A Topological Search Algorithm for ATPG," *Design Automation, 1987. 24th Conference on* , vol., no., pp. 502- 508, 28-1 June 1987

[**Kunz92**] Kunz, W.; Pradhan, D.K.; , "Recursive Learning: An attractive alternative to the decision tree for test generation in digital circuits," *Test Conference, 1992. Proceedings., International* , vol., no., pp.816, 20-24 Sep 1992

[**Lee93**] H.K. Lee and D.S. Ha, "Atalanta: an Efficient ATPG for Combinational Circuits," *Technical Report, 93-12*, Dep't of Electrical Eng., Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1993

[**Ma88**] Ma, H.-K.T.; Devadas, S.; Newton, A.R.; Sangiovanni-Vincentelli, A.; , "Test generation for sequential circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* , vol.7, no.10, pp.1081-1093, Oct 1988

[**Marchok95**] Marchok, T.E.; El-Maleh, A.; Maly, W.; Rajski, J.; , "Complexity of sequential ATPG," *European Design and Test Conference, 1995. ED&TC 1995, Proceedings.* , vol., no., pp.252-261, 6-9 Mar 1995

[**Marlett78**] Marlett, R.A.; , "EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits," *Design Automation, 1978. 15th Conference on* , vol., no., pp. 335- 339, 19-21 June 1978

[**Marlett86**] Marlett, R.; , "An Effective Test Generation System for Sequential Circuits," *Design Automation, 1986. 23rd Conference on* , vol., no., pp. 250- 256, 29-2 June 1986

[**Muth76**] Muth, P.; , "A Nine-Valued Circuit Model for Test Generation," *Computers, IEEE Transactions on* , vol.C-25, no.6, pp.630-636, June 1976

[**Niermann91**] Niermann, T.; Patel, J.H.; , "HITEC: a test generation package for sequential circuits," *Design Automation. EDAC., Proceedings of the European Conference on* , vol., no., pp.214-218, 25-28 Feb 1991

[**Prinetto94**] Prinetto, P.; Rebaudengo, M.; Sonza Reorda, M.; , "An automatic test pattern generator for large sequential circuits based on Genetic Algorithms," *Test Conference, 1994. Proceedings, International* , vol., no., pp.240-249, 2-6 Oct1994

[**Putzolu71**] Putzolu, G.R.; Roth, J.P.; , "A Heuristic Algorithm for the Testing of Asynchronous Circuits," *Computers, IEEE Transactions on* , vol.C-20, no.6, pp. 639- 647, June 1971

[**Roth66**] Roth, J. Paul; , "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development* , vol.10, no.4, pp.278-291, July 1966

[**Roth67**] Roth, J. Paul; Bouricius, Willard G.; Schneider, Peter R.; , "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *Electronic Computers, IEEE Transactions on* , vol.EC-16, no.5, pp.567-580, Oct. 1967

[**Rudnick95**] Elizabeth M. Rudnick, Janak H. Patel; , "Combining Deterministic and Genetic Approaches for Sequential Circuit Test Generation," *Design Automation, 1995. DAC '95. 32nd Conference on* , vol., no., pp.183-188, 1995

[**Rudnick97**] Rudnick, E.M.; Patel, J.H.; Greenstein, G.S.; Niermann, T.M.; , "A genetic algorithm framework for test generation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* , vol.16, no.9, pp.1034-1044, Sep 1997

[**Saab92**] Saab, D.G.; Saab, Y.G.; Abraham, J.A.; , "CRIS: A test cultivation program for sequential VLSI circuits," *Computer-Aided Design, 1992. ICCAD-92. Digest of Technical Papers., 1992 IEEE/ACM International Conference on* , vol., no., pp.216-219, 8-12 Nov, 1992

[**Saab94**] Saab, D.G.; Saab, Y.G.; Abraham, J.A.; , "Iterative simulation-based Genetics + Deterministic Techniques = Complete ATPG," *Computer-Aided Design, 1994., IEEE/ACM International Conference on* , vol., no., pp.40-43, 6-10 Nov 1994

[**Schulz88**] Schulz, M.H.; Trischler, E.; Sarfert, T.M.; , "SOCRATES: a highly efficient automatic test pattern generation system," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* , vol.7, no.1, pp.126-137, Jan 1988

[**Seshu62**] Seshu, S.; Freeman, D. N.; , "The Diagnosis of Asynchronous Sequential Switching Systems," *Electronic Computers, IRE Transactions on* , vol.EC-11, no.4, pp.459-465, Aug. 1962

[**Silva94**] Marques Silva, J.P.; Sakallah, K.A.; , "Dynamic Search-Space Pruning Techniques in Path Sensitization," *Design Automation, 1994. 31st Conference on* , vol., no., pp. 705- 711, 6-10 June 1994

[**Trischler84**] E. Trischler, "ATWIG, An Automatic Test Pattern Generator with Inherent Guidance," *Proc. Int'l Test Conf.*, Nov. 1984, pp. 80-87.

[**Xijiang98**] Xijiang Lin; Pomeranz, I.; Reddy, S.M.; , "MIX: a test generation system for synchronous sequential circuits ," *VLSI Design, 1998. Proceedings, 1998 Eleventh International Conference on* , vol., no., pp.456-463, 4-7 Jan 1998

[**Xijiang99**] Xijiang Lin; Pomeranz, I.; Reddy, S.M.; , "Techniques for improving the efficiency of sequential circuit test generation," *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on* , vol., no., pp.147-151, 1999