# Pseudo-Functional Testing: Bridging the Gap between Manufacturing Test and Functional Operation

**YUAN, Feng**

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
August 2009

Thesis/Assessment Committee

Professor Yu Liang Wu (Chair)
Professor Qiang Xu (Thesis Supervisor)
Professor Kin Hong Lee (Committee Member)
Professor Tim Cheng (External Examiner)

# 摘要

隨著半導體工業的不斷發展，晶片在正常工作狀態下和在結構性測試環境下表現出的差異性對生產測試造成日益嚴重的負面效應。僞功能測試能夠有效的解決這一問題，這一技術首先識別出那些正常工作狀態下無法出現的狀態（非法狀態），然後在測試向量生成的時候避免這些非法狀態的出現。然而由於各種因素的存在現有的僞功能測試方法只能提取很小一部分的非法狀態。另外，相比起傳統的結構性測試向量來說，僞功能測試通常需要在測試向量設置更多的比特，因此非確定性的比特所占的比例會相應減少。這一特點使得一些測試向量壓縮技術不利於運用在僞功能測試向量上。

在這篇論文裏，爲了解決上訴問題，我們首先說明了晶片裏存在非法狀態的主要原因是晶片的組合邏輯網路中存在多扇出結構的連接線，根據這一特點，我們又提出了一種用於提取非法狀態的高效演算法。除此之外，我們還提出了一種非常適用于測試向量壓縮技術的僞功能測試解決方案，這一方案能達到與傳統結構性測試向量基本相同故障覆蓋率，同時可以消除過度測試問題並且不損失測試向量壓縮率。綜上所述，通過縮小晶片的生產測試模式與正常工作模式的差距，本文所提出的各種技術極大地加強了生產測試的有效性。

# Abstract

The discrepancy between integrated circuits' activities in normal functional mode and that in structural test mode has an increasingly adverse impact on the effectiveness of manufacturing test. By identifying functionally-unreachable states in the circuit and avoiding them during the test generation process, pseudo-functional testing is an effective technique to address this problem. Existing methods, however, can only extract a small set of illegal states in the system due to various limitations. In addition, to avoid violating functional constraints, pseudo-functional patterns typically feature much fewer dont-care bits when compared to conventional structural patterns, making them less friendly to the de-facto test compression techniques widely-used in the industry.

To address the above issues, in this thesis, we first show that illegal states in a circuit are mainly caused by multi-fanout nets in it, and we develop efficient and effective heuristics to identify them. Then, we introduce our solutions to apply pseudo-functional tests in linear decompressor-based test compression environment, which is able to achieve similar fault coverage as conventional structural patterns, without incurring over-testing to the circuits neither sacrificing test compression ratio loss. The effectiveness of IC tests can be significantly enhanced with the proposed techniques that bridge the gap between manufacturing test and functional operation of the circuit under test.

# Acknowledgement

At the very beginning, I am deeply indebted to my supervisor, Professor Qiang Xu, who patiently motivated me to conceive and develop the main ideas in the thesis. I would like to express to him my sincere gratitude for his seasoned guidance from the very early stage of this research work as well as providing constructive advices throughout the entire study. In particular, I also would like to thank him and his wife for their concerns about my daily life.

My research partners Yubin Zhang, Lin Huang, Xiao Liu and Li Jiang in the CUhk REliable Computing Laboratory (CURE), thank you for your insightful comments on my research work. I am also grateful to all the colleagues in 506 EDA office, Linfu Xiao, Liang Li, Minqi Jiang, Zaichen Qian, Xiaoqing Yang, Zigang Xiao and Yan Jiang, it is you who bring me laugher and make my post-graduate study life colorful. Special thanks are also to my friends Qiang Ma and Lei Shi, who helped me to settle down when I just came to this campus.

Last but not the least, my father Guangfu Yuan, my mother Guangpin Gui, my girlfriend Rong Huang and all my family members, without your love and support, I cannot achieve anything. I would like to give my greatest appreciation to you.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Manufacturing Test

Integrated circuit (IC) fabrication is an extremely complex process, and it is inevitable that some manufactured chips are defective, due to various hard-to-controlled factors (e.g., impure material, temperature variation and quantum effect). Semiconductor industry thus relies on manufacturing test to identify those defect-free ICs and ship them to customers.

Manufacturing test is typically conducted with the help of automatic test equipment (ATE). When testing a circuit, both test patterns and the expected test responses are stored in the ATE. During the manufacturing test process, test patterns are transported from ATE to the circuit, and then the actual responses captured by the circuit are sent back to ATE to compare against the expected responses. Those circuits that have different responses from the expected ones are marked as defective products.

### 1.1.1 Functional Testing vs. Structural Testing

Functional testing was historically used to test IC products, wherein a large amount of test patterns are required to completely excise the circuit's functionalities. Generally speaking, the number of input patterns for functional testing will be $2^n$ for a circuit with $n$ inputs. Taking a 64-bit ripple-carry adder as example, $2^{129}$ patterns are needed to apply complete functional test, which would take $2.158 \times 10^{22}$ years to finish such test on a 1 $GHz$ ATE [4]. Due to such exhaustive nature of functional testing, it is impractical for any reasonable-sized circuits. In addition, due to the need of applying functional tests at speed, the functional tester is much more expensive. The semiconductor industry hence mainly resorts to structural testing for this duty, wherein test patterns are selected based on circuit structural information and a set of fault models. One of the greatest advantages of structural test is that it allows us to develop structural search algorithms to achieve efficient testing. For the same 64-bit ripple-carry adder, 1728 patterns are enough for structural testing based on stuck-at fault model (introduced later).

Defects in an electronic system is defined as the unintended differences between the implemented hardware and its intended design [4]. It is very hard to generate tests for every possible type of physical defects. Fault models, therefore, are proposed to abstract faulty behaviors induced by defects. To generate test patterns effectively, faults are always modeled at a certain level of design abstraction, such as behavioral level, logic/gate level or transistor level. Fault models at behavioral level usually have no clear correlation to manufacturing defects and hence are used more often in design verification rather than manufacturing test. Transistor level fault models are also known as technology-dependent faults and are mainly used in analog circuit testing. Fault models at logic level (i.e., circuit is modeled as an interconnection of boolean gates, called *netlist*) are technology-independent and over time have been proven to be quite efficient and effective for testing digital

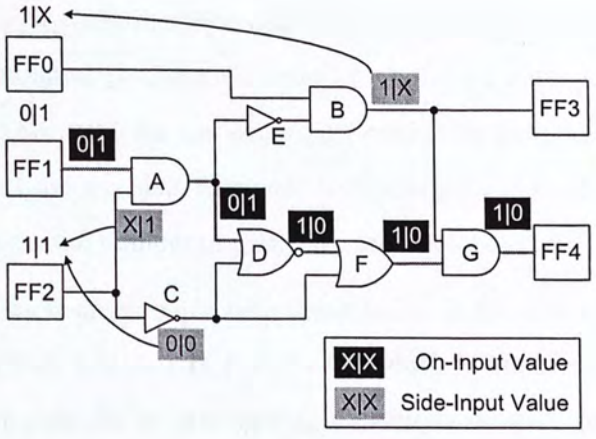


Figure 1.1: An Example Path Delay Fault

circuits [4]. Here, we introduce two main kinds of fault models that are widely-used in the industry.

### 1.1.2 Fault Model

- Stuck-at Fault Model

  Stack-at fault model is the fundamental fault model used in IC testing, which assumes a single line of the logic network to be stuck at a logic 0 (s-a-0) or logic 1 (s-a-1).

- Delay Fault Model

  Delay faults model those defects that cause the combinational delay of a circuit to exceed its clock period. Commonly-used delay fault models include the transition fault model (also called gross-delay fault model) and the path delay fault model. Transition fault model is based on the assumption that only a single gate delay is changed. It has the advantages of easy test pattern generation and comparably small test set size, but it is less accurate due to

its simplistic assumption. Path delay fault model, on the other hand, considers the cumulative propagation delay of a combinational path and hence is much more accurate, but test pattern generation for path delay faults is quite complicated and it is also associated with a large number of test patterns due to the exponential number of paths in a sequential circuit.

Fig. 1.1 depicts an example path delay fault. In this example, the targeted fault is on path $\{FF1, A, D, F, G, FF4\}$, which is manifested by the cumulated delay induced by propagating transitions on this path. To sensitize such transitions, two consecutive patterns $< 1,0,1;X,1,1 >$ need apply on $\{FF0, FF1, FF2\}$. The correct response for this circuit at output $FF4$ is $< 1;0 >$. Due to the fact that delay on the targeted path exceeds clock period, logic 0 cannot arrive at $FF4$ before the capture clock edge. The faulty response is therefore $< 1;1 >$ at $FF4$.

It is important to note, as technology scales, at-speed delay testing has become increasingly popular to ensure the quality of shipped products.

### 1.1.3 Automatic Test Pattern Generation

Given a fault model, the task of automatic test pattern generation (ATPG) tool is to find out vectors that can activate the targeted faults and propagate their faulty effects to observable outputs. In essence, most structural ATPG algorithms are based on branch-and-bound principle, which try to quickly find a solution to detect targeted faults and backtrack immediately when some pre-determined values at circuit nodes are found to be infeasible. According to the nature of the circuit in test mode, it can be categorized into combinational ATPG and sequential ATPG. Both have been proved to be NP-Complete problem [4].

- Combinational ATPG

# List of Tables

Combinational ATPG is the fundamental problem for IC testing and it has been subject to extensive research over the past decades. In the following we list a few milestone work in this area: (i) D-algorithm proposed by Roth [28], which established the calculus and algorithms for ATPG using D-cubes; (ii) Goel's PODEM algorithm [10] proposed to use path propagation constraints to efficiently limit the search space of the ATPG engine; (iii) Fujiwara and Shimono's FAN algorithm further improved the efficiency of ATPG and is widely used in the industry. The basic idea of this technique is to detect infeasible solutions as early as possible so that the processing time between backtracks can be significantly reduced.

- Sequential ATPG

  For circuits with state elements that cannot be directly controlled in test mode, sequential ATPG is required when generating test patterns. Generally speaking, sequential ATPG is much more complicated than combinational ATPG. This is because: (i). the test responses of the circuit are dependent not only on input test pattern, but also on the initial states of the circuit's sequential elements; (ii). activating a fault from primary outputs requires the circuit to be driven to a known state, which itself requires more than one pattern and essentially involves multiple combinational ATPG procedure; (iii). propagating faulty effects to primary outputs also takes multiple clock cycles and incurs high computational complexity.

It is worth to note that, test patterns generated from ATPG process typically feature a large percentage of "dont-care" bits (also known as *X-bits*) and they can be assigned to any logic value without affecting the fault coverage of the test set.

Fig. 1.2 presents the flowchart for a typical ATPG framework, containing two stages. In the first stage, random patterns are generated and fault simulation is conducted in the hopes that some easy-to-detect faults can be covered by such pat-

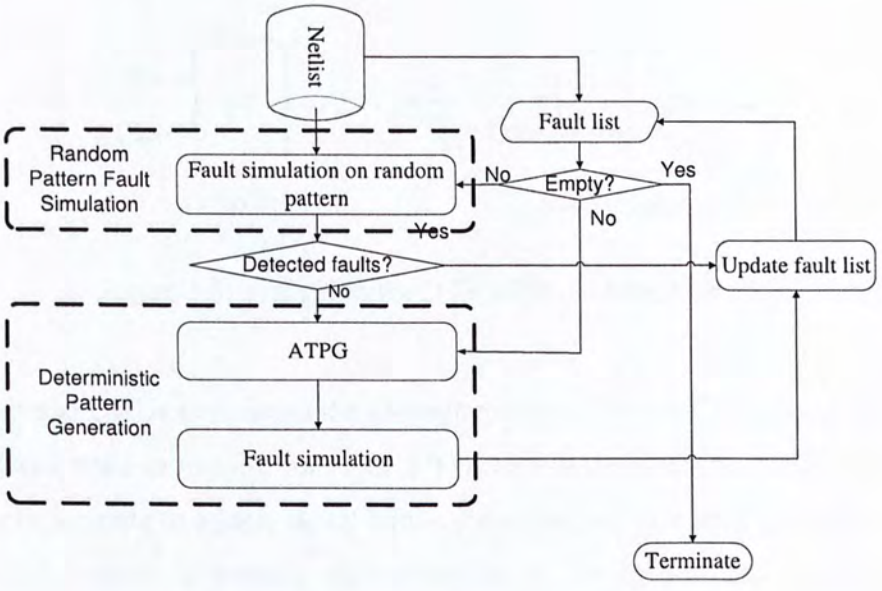


Figure 1.2: Flowchart of General ATPG Process

terns. This process continues until no fault can be detected by random patterns. Next, deterministic ATPG is used for those random-resistant faults. Since a test pattern generated by ATPG can potentially detect several other faults, fault simulation is again applied for every pattern. The whole flow terminates when all faults are detected or we have reached the runtime limit for the ATPG process.

### 1.1.4 Design for Testability

DfT is the design process which embeds special hardware for testing purpose only. This section presents the *de-facto* DfT techniques widely-used in the industry.

#### Scan-Based DfT

With the ever increasing transistor-to-pin ratio in IC products, sequential ATPG is no longer applicable on today's complex sequential circuits. The main purpose for

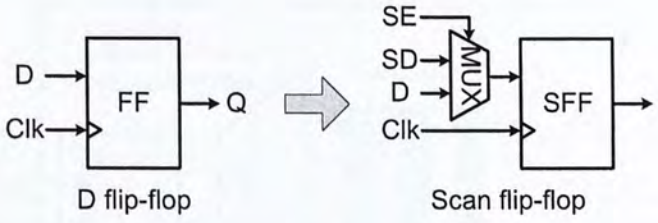


Figure 1.3: Transform the D Flip-Flop to Scan Flip-Flop

scan-based DfT is to increase the controllability (i.e., the ability to set a particular circuit node to logic '0' or logic '1') and the observability (i.e., the ability to observe the state of a logic signal within the circuit) of the circuit's internal node so that it is easier to generate test patterns for the circuit. In scan-based circuits, we substitute normal flip-flops (FFs) with scan FFs (SFFs), making them directly accessible in test mode. By doing so, from the test generation point of view, the circuit under test is a combinational circuit and hence the more tractable combinational ATPG can be used to generate test patterns.

SFFs can be implemented in various manners, e.g., mux-based SFFs, double latched SFFs, level sensitive scan latches SFFs [1, 4]. Fig. 1.3 depicts the transformation from a normal FF into a mux-based SFF. In the mux-based SFF, a multiplexor is inserted before the input of the FF with two inputs *D* and *SD*, which represent the original data input and the scan data input, respectively. Scan enable (*SE*) signal is used to select which channel as input of FF. By replacing normal FFs with SFFs, these state elements can be connected serially to form one or more long shift registers (called scan chain) through *SD* input, and the first and the last SFF of each scan chain are connected with an input pin and an output pins of the circuit. All the SFFs can be set as arbitrary states by shifting logic values into the scan chains. Similarly, the states of these SFFs can be observed by shifting out the contents of the shift registers.

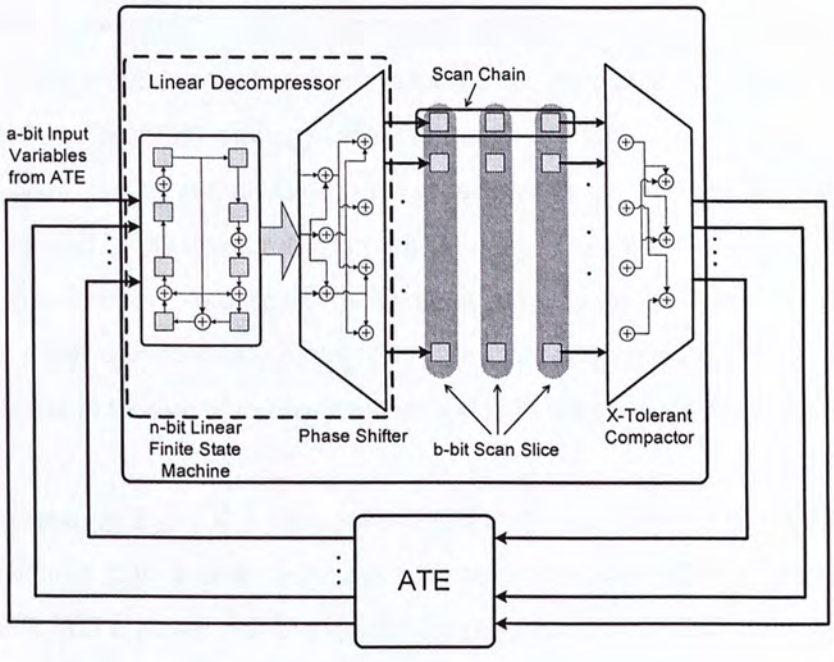


Figure 1.4: Linear Decompressor-Based Test Compression Infrastructure

The test procedure in scan-based testing can be divided into three phases.

- Scan in: *SE* signal is asserted to configure the circuit as scan mode. Test pattern is then shifted into scan chains for $N_{sc}$ clock cycles, where $N_{sc}$ is the length of longest scan chain;
- capture: *SE* signal is de-asserted, and the circuit applies the test pattern in functional mode and capture its responses into the same SFFs;
- Scan out: test responses are shifted out in the similar manner as the scan in process.

**Test Data Compression**

The rapidly-growing test data volume has becmoe a serious concern for the industry because it not only prolongs the ICs' testing time, but also raises memory

depth requirements for the ATE. Test data compression, consisting of test stimulus compression at the input side and test response compaction at the output side, has become the *de facto* test strategy for today's large circuits.

For test stimulus compression, various techniques (as surveyed in [34]) have been proposed in the literature, all of which exploit the large amount of X-bits in the given test cubes to reduce test data volume. Among the existing TDC methodologies, linear decompressor-based technique is the most popular one used in the industry due to its ease of implementation and high compression ratio (e.g., [2, 25, 35, 37]).

As shown in Fig. 1.4, a typical linear decompressor consists of a *n-bit* finite state machine that receives *a-bit* input variables from the ATE to generate test sequences and a phase shifter (typically implemented with XOR network) used to expand these sequences to a large amount of scan chains with reduced linear dependencies. In each clock cycle, *b-bit* ($b >> a$) values are shifted into scan slices. Typically, a two-pass ATPG flow is utilized to generate compressible test patterns. That is, after ATPG generates a test cube, a linear solver is invoked to compress it. If the solver cannot find a solution, a different test cube would be generated to target the fault.

Traditionally, for test response compaction, multiple input signature register (MISR) is used to generate a small signature. This simple compactor, however, suffers from fault coverage loss due to aliasing and unknown logic values in test responses (e.g., due to bus contention and multiple clock domains). To tackle the above problem, a number of X-tolerant compactors were proposed [8, 22, 23, 26, 33, 36], which are able to tolerate a small percentage of X-bits in test responses at the cost of higher DfT overhead and less compaction ratio. Generally speaking, with the growth of X-bits, the compaction ratio is decreased and the silicon area used to tolerate these X-bits increases. Therefore, the number of *X's* in test

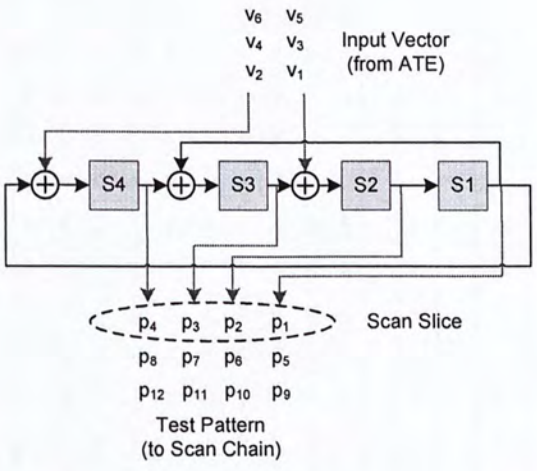


Figure 1.5: An Example Linear Decompressor

responses cannot be too high.

### Linear Algebra in Decompressor and Compactor

In linear decompressor-based TDC technique, we generate the large-sized deterministic test cubes by expanding small *input variables*. We use an example linear decompressor shown in Fig. 1.5 to demonstrate the test compression process. For the sake of simplicity, we omit phase shifter in this example.

The inputs supplied to the linear decompressor are comprised of the initial state of the linear FSM and the input variables coming from the ATE; while the output from the linear decompressor is the actual test pattern applied to the circuit. The structure of the linear decompressor can be represented by a *transformation matrix* and it determines the linear relationship between the input vector and the output vector $M \times V = P$, as shown in the following:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \\ p_9 \\ p_{10} \\ p_{11} \\ p_{12} \end{pmatrix} \tag{1.1}$$

It is important to note that, for each deterministic test cube, we only need to solve a subset of linear equations that correspond to the specified bits in $P$. Suppose we are given a test cube as $P = (1\ X\ 0\ X\ X\ X\ 1\ X\ X\ 1\ X\ X)^T$, by omitting those equations that correspond to $X$-bits, we have $M_s \times V = P_s$ as:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} \tag{1.2}$$

According to linear algebra theory, the above equation system is solvable if and only if the *coefficient matrix* $M_s$ has the same *rank* as that of the corresponding *augmented matrix* $(M_s|P_s)$. *Gaussian elimination* is a widely used algorithm

to find the rank of a matrix, which reduce a matrix to *row echelon form* by elementary row operations. After conducting it, we can obtain the following *reduced augmented matrix* $(M_s'|P_s')$:

$$\left(\begin{array}{cccccccccc|c} \mathbf{1} & \mathbf{0} & 0 & \mathbf{0} & 0 & 1 & 0 & 0 & \mathbf{0} & 0 & 0 \\ \mathbf{0} & \mathbf{1} & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 1 \\ \mathbf{0} & \mathbf{0} & 0 & \mathbf{1} & 0 & 1 & 0 & 0 & \mathbf{0} & 0 & 0 \\ \mathbf{0} & \mathbf{0} & 0 & \mathbf{0} & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \end{array}\right) \tag{1.3}$$

We name $M_s'$ and $P_s'$ as *reduced coefficient matrix* and *reduced result column*, respectively. The first non-zero entry in each row of $M_s'$ is called a *pivot*, and its corresponding column is a *pivot column* (in bold) which only includes one non-zero entry. Consequently, the test pattern is compressible if and only if the reduced result column $P_s'$ is not a pivot column.

For test response compaction, in addition to traditional compactors based on multiple input signature registers (MISR), a number of compactors have been proposed to handle fault coverage loss due to aliasing and unknown logic values in test responses (e.g., arise from bus contention and multiple clock domains). We briefly introduce the X-compact technique presented in [22] here as it is used in this thesis work. X-compactor in [22] is essentially a combinational XOR network that can be also represented as a transformation matrix. The authors proved that it is able to detect $k_1$ error bits in present of any $k_2$ *X-bits* within a single scan slice. Consequently, with the help of such circuit-independent compactor, it is not necessary to store the transformation matrix of compactor during the ATPG process. Instead, we only need to control the number of X-bits in every scan slice and the number of bits used to detect error in it. Obviously, with the increase of $k_1$ and $k_2$, the DfT area overhead for the compactor increases and the compaction ratio is reduced.

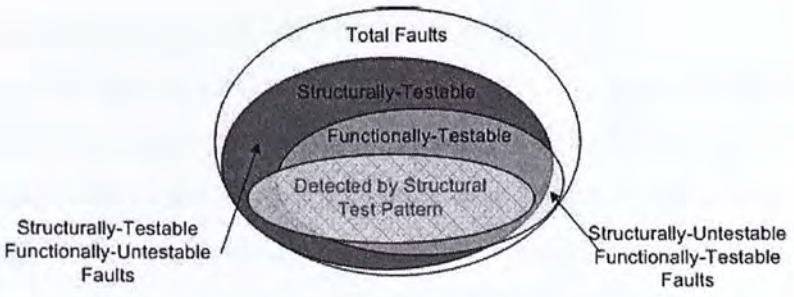


Figure 1.6: Fault Classification

## 1.2 Pseudo-Functional Manufacturing Test

The manufacturing test itself, unfortunately, is also not perfect. On one hand, as it is impossible to model all the possible manufacturing defects and generate test patterns for them, some bad chips may pass the test, leading to *test escapes*. On the other hand, since a circuit in test mode oftentimes operates differently from the one in functional mode, some good chips that would work in application may fail the test and are marked as defective, leading to *test overkills*.

We can classify manufacturing defects based on their detectabilities as shown in Fig. 1.6 [19]. As can be observed, there might be a small number of defects that are functionally-testable but structurally-untestable (FT-SU). For instance, inserting test logic between random logic and RAM blocks may cause some defects in the functional paths to become untestable. At the same time, there are many functionally-untestable while structurally-testable (FU-ST) defects. This is mainly due to the fact that the functional constraints existing in the circuit are possibly violated in structural test mode. Consider a finite state machine encoded with one-hot code, the legal combinations of values in these state elements are only those with a single logic 1 and all the others logic 0. Without taking this functional constraint into consideration, we may have structural patterns that contain multiple logic 1s and hence are illegal in functional mode. Those defects that are detectable only

with such patterns are FU-ST defects. It can be also seen from the figure that not all structurally-testable defects are covered by test patterns due to imperfect fault modeling and test cost considerations. Test escapes occur for those ICs containing functionally-testable defects that are not covered by test patterns; while the ICs that would work in application but are rejected in manufacturing test become test overkills (also known as false rejects). Test overkills can be attributed to FU-ST defects covered by test patterns and/or excessive noises in test mode.

Traditionally, the primary objective of manufacturing test is to obtain low test escapes in order to ensure the quality of the shipped products, and a limited number of false rejects are considered as acceptable loss. For ICs fabricated with latest technology, however, at-speed testing is essential to ensure the quality of the shipped IC products, rendering over-testing due to the discrepancy between circuits' activities in functional mode and that in test mode a serious concern for the industry [27, 29, 5]. Recent design evaluations have revealed that at-speed scan patterns were up to 20% slower than any functional pattern [30]. Consequently, some good ICs that would work in application might fail at-speed delay tests [21]. With today's tight profit margins, particularly for chips that go into consumer products, achieving a high manufacturing yield can mark the difference between success and failure, and just a small variation in yield percentage can translate to millions of dollars of revenue change. Therefore, how to reduce yield loss caused by test overkills has become a main concern for the industry.

One way to reduce test overkills is to identify FU-ST delay faults in the circuit and do not target them during test generation. A significant amount of research work has been conducted in this direction (e.g., [3, 6, 7, 11, 32]). However, FU-ST delay fault identification generally has the same complexity as that of sequential ATPG [19], which is exponential in terms of circuit's size. In addition, while test patterns are only generated for those functionally-testable faults in the

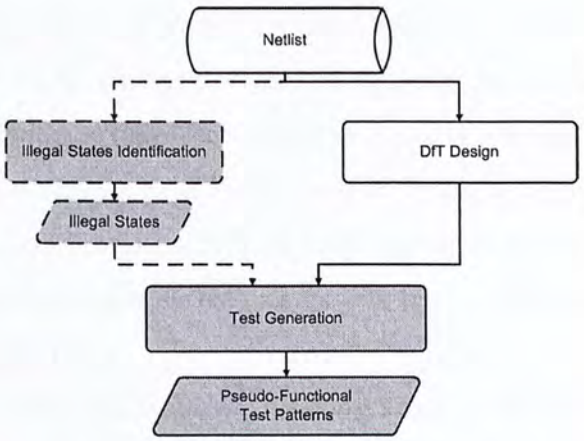


Figure 1.7: Pseudo-Functional Test Pattern Generation

above methods, it is still possible that they incidentally detect some FU-ST faults and hence lead to test overkills [19]. Moreover, even for patterns that detect only functionally-testable faults, as we typically let them target as many faults as possible to reduce testing time, they might lead to excessive noises that could not occur in functional mode, again, rendering possible test overkills.

Instead of identifying FU-ST delay faults in the circuits, Lin *et al.* [16] proposed the concept of pseudo-functional testing to tackle the above problem. The overall flow of pseudo-functional testing is illustrated in Fig.1.7, wherein functional-like patterns were generated for manufacturing test by identifying illegal states (i.e., *functionally-unreachable* states, we refer the *functional constraints* as the constraints that test pattern cannot include *functionally-unreachable* states) in functional mode and avoiding them during the test pattern generation process.

One of the key issues in pseudo-functional testing is to identify illegal states. In [16, 17], Lin *et al.* used a sequential boolean satisfiability (SAT) solver [20] to extract the functional constraints in the system. Wu and Hsiao [38] proposed a mining-based strategy for illegal state identification. In this scheme, the circuit

is expanded to multiple time-frames first and then simulated with a number of random patterns. By analyzing the obtained data, some suspicious functional constraints are extracted and they use SAT solver to verify whether they are actually functionally-illegal. Zhang *et al.* [39] proposed an implication-based technique for illegal state identification, which try to detect values combination on FFs to imply impossible logic scenario on a single gate (e.g. both input and output of a *NOT* gate are logic '1').

When illegal states are available, the constrained ATPG process can be conducted as shown in [17], in which illegal states are represented as clauses in conjunctive normal form (CNF), for example $\{A(0) \wedge B(0)\} \vee \{A(1) \wedge B(1) \wedge C(1)\}$. Whenever the ATPG engine tries to assign value on an FF, say $A = 0$, all the clauses including $A(0)$ are first extracted (e.g., $\{A(0), B(0)\}$). If $B$ is the only un-assigned FF in this clause, ATPG automatically assign $B$ to be logic '1' to satisfy this functional constraint. Because of this, pseudo-functional test patterns typically feature much fewer X-bits when compared to conventional structural test patterns without considering functional constraints.

## 1.3 Thesis Motivation and Organization

It is important to note that, pseudo-functional test patterns do not necessarily guarantee to be within the functionally-reachable space even after imposing functional constraints, because the set of extracted illegal states is usually incomplete and thus a subset of the complete illegal-state set (see Fig. 1.8). Therefore, whether we could minimize the discrepancy between functional mode and test mode highly relies on whether we could effectively identify as complete illegal states as possible. Although the SAT-based technique can ensure the completeness, such a brute-force method can only be applied to a small circuit (or a sub-circuit) at a time. Other existing methods cannot even answer the question "how far are they from the com-

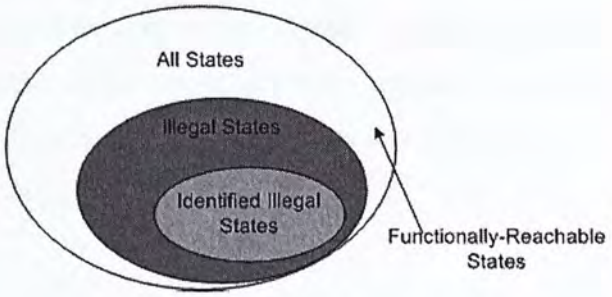


Figure 1.8: Illustration of Incomplete Identified Illegal State

pleteness?". Motivated by the above challenges, the first part of this thesis tries to investigate the fundamental problem of why there are illegal states in the system from the structural point of view. Such root cause analysis is important since the solution space for illegal state identification can be significantly reduced without sacrificing its completeness.

Recently, test compression has become the *de facto* DfT methodology to handle the growth of test data [25, 34]. As discussed earlier, to avoid a large amount of illegal states, the percentage of X-bits in pseudo-functional patterns can be quite low. This has a severe impact on the effectiveness of test data compression techniques, since they mainly exploit X-bits in test cubes to achieve significant test volume reduction without sacrificing fault coverage. Therefore, how to effectively apply pseudo-functional patterns in test compression environment is another relevant problem for the success of pseudo-functional testing and it is the focus of the second part of this thesis work.

The remainder of this thesis is organized as follows. Chapter 2 presents the theorem and proof for the structural root cause of illegal state. Based on this, effective and efficient algorithms are proposed to systematically identify illegal states in the circuit under test. Next, Chapter 3 describes our solutions to apply pseudo-functional tests in linear decompressor-based test compression environ-

ment, which is able to achieve similar fault coverage as conventional structural patterns, without incurring over-testing to the circuits and sacrificing test compression ratio loss. Finally, chapter 4 concludes this thesis and points out our future research directions.

---

□ **End of chapter.**

# Chapter 2

# On Systematic Illegal State Identification

## 2.1 Introduction

Pseudo-functional testing seems to have great potential for resolving the discrepancy problem between structural test mode and functional mode. However, whether we could realize this potential highly relies on whether we could effectively identify as many illegal states as possible. That is, if the extracted illegal states are far from complete, there is still a high possibility that the generated pseudo-functional test patterns are not within the circuit's functionally-reachable space, which invalidates the objective of pseudo-functional testing.

Several approaches were proposed for illegal state identification in the literature, including SAT-based methods [16], implication-based strategies [31, 39], and mining-based techniques [38]. None of the above techniques, however, answers the fundamental question why some states are functionally-unreachable from a structural point of view. They also have their own specific limitations and can only extract a small set of illegal states in the circuit (detailed in chapter 2.2). Conse-

quently, for the success of pseudo-functional testing, more effective techniques are required for illegal state identification.

In this chapter, we propose novel solutions to tackle the above problem. The contributions of our work include:

- we show that the illegal states in circuit are mainly caused by the multi-fanout nets in the circuit.
- we propose novel algorithms that are able to effectively identify much more functionally-unreachable states when compared to state-of-the-art techniques.

The remainder of this chapter is organized as follows. Chapter 2.2 reviews related prior work and motivates this chapter. In Chapter 2.3, we study the structural root cause for illegal states. The main flow and the key concept for the proposed illegal state identification scheme are then detailed in Chapter 2.4 and Chapter 2.5, respectively. Next, Chapter 2.6 presents our experimental results on several ISCAS'89 benchmark circuits. Finally, Chapter 2.7 concludes this chapter.

## 2.2 Preliminaries and Motivation

Illegal state identification has been studied in the context of sequential ATPG in several earlier works [12, 14], wherein they were used to prune the search space for sequential tests. In [12], known illegal states are used to generate larger candidate illegal spaces by eliminating one assignment in the illegal states at a time and trying to justify them. [14], on the other hand, identified invalid states by exploring all valid states through simulation from an unknown initial state. These techniques used in sequential ATPG are not practically viable for today's large ICs due to their extremely high computational complexity.

In [16], Lin *et al.* used a sequential boolean satisfiability (SAT) solver to extract the functional constraints in the system. While theoretically a SAT solver is able

to find almost all the unreachable states in a circuit, its computational complexity is extremely high and hence cannot be applied to a large circuit. Therefore, the authors in [16] proposed to divide the flip-flops (FFs) in the circuit into a number of much smaller groups based on topological analysis and the targeted fault information, each containing few FFs only (e.g., less than 10), and run a SAT solver within each group to identify illegal states. Apparently, it is possible that functional constraints exist among different groups and this method cannot identify such kind of illegal states. Moreover, the SAT solver might still abort the computation even within a small group of FFs.

Zhang *et al.* [39] proposed an implication-based technique for illegal state identification. The method starts from any gate, say gate $A$, and finds the implications when its output value is logic '1' and logic '0', respectively. Suppose $B$ and $C$ are internal flip-flops in the circuit, and there exist two implications: $[A(0) \rightarrow B(1)]$ (i.e., $A = 0$ implies $B = 1$) and $[A(1) \rightarrow C(0)]$, we then have $[B(0) \rightarrow A(1)]$ and $[C(1) \rightarrow A(0)]$ according to the contrapositive law. Consequently, we can conclude $\{B(0),C(1)\}$ is an illegal state cube. The approach proposed in [39] also considered the implications from impossible input-output combinations (e.g., for a 2-input AND gate, when an input is logic '0' while the output is logic '1'). To keep the computational complexity manageable, their approach implies values based on a single node in one time-frame only. This, however, significantly restricts the number of identified illegal states.

Syal *et al.* [31] considered multi-node functional constraints obtained through sequential implications, which cannot be identified within a single time-frame as in [39]. They also advocated to represent illegal states in the form of boolean expressions on arbitrary nets in the circuit instead of values on the state elements. Unlike what the authors claimed, however, using this representation may increase the storage overhead for the constraints significantly, because the number of arbi-

trary nets is much larger than the number of state elements in the circuit and lots of their obtained functional constrains are redundant in nature.

Recently, Wu and Hsiao [38] proposed a mining-based illegal state identification strategy. In this work, the circuit is expanded to multiple time-frames first and then simulated with a number of random patterns. By analyzing the obtained data, some suspicious functional constraints are extracted and they use a SAT solver to verify whether they are actually functionally-unreachable. While this dynamic learning method accelerates the search procedure, due to the large amount of data, it can only check pair-wise and three-node relations within small groups of state elements and hence cannot find many illegal states in the system.

To sum up, identifying illegal states using SAT-based [16] or mining-based techniques [38] can be classified as brute-force approaches, which can only be applied to one small sub-circuit at a time. Implication-based methods [31, 39] that consider circuit structural information to tackle this problem seem more promising. Unfortunately, none of them answers the fundamental question why there are illegal states in the system from the structural point of view. *Finding the root cause of illegal states is extremely important for this problem, as with this information the solution exploration space can be significantly reduced without sacrificing its completeness.* This facilitate pseudo-functional testing to be applicable to real industrial designs. It is the above observation that motivates this work.

## 2.3 What is the Root Cause of Illegal States?

Let us examine an example circuit as shown in Fig. 2.1 to demonstrate our structural root cause analysis for illegal states. This is the gate netlist of a finite state machine that contains six illegal state cubes (see Fig. 2.1). A closer observation of the circuit shows that except $\{FF2(0), FF3(0), FF4(0)\}$, all the other five illegal state cubes imply logic violation at a multi-fanout net. For example, let us try to

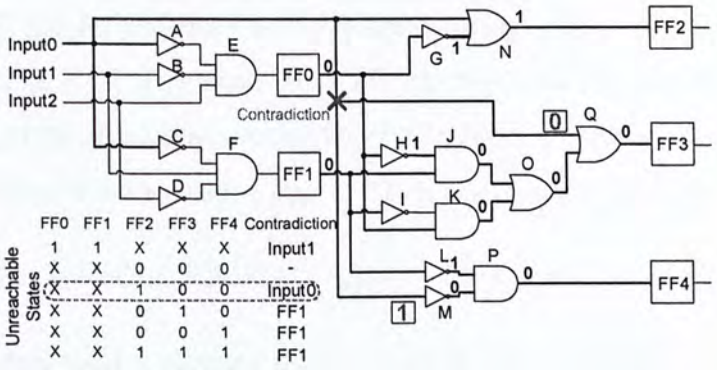


Figure 2.1: Unreachable State Analysis

justify illegal state $\{FF2(1), FF3(0), FF4(0)\}$. First, we can learn $Input0(0)$ and $O(0)$ through OR gate $Q$, and $G(1)$ and $FF0(0)$ through OR gate $N$. We can then derive $J(0)$ and $K(0)$ through OR gate $O$ and $H(1)$ through the inverter $H$. Next, $FF1(0)$ is justified through AND gate $J$. Finally, we can infer $Input0(1)$ through AND gate $P$ with $FF1(0)$ and $FF4(0)$, which, however, contradicts to the aforementioned $\{Input0(0)\}$ justified through another fanout going through gate $Q$. On the other hand, although the illegal state cube $\{FF2(0), FF3(0), FF4(0)\}$ do not imply any logic violation at a multi-fanout net directly, it actually infers another illegal state cube $\{FF0(1), FF1(1)\}$. In other words, this particular illegal state also causes logic violation on a multi-fanout net $Input1$ implicitly, which occurs at another time frame. This example motivates us to consider whether multi-fanout nets are the main root cause of illegal states.

**Definition 1** *Consider a circuit that does not contain any multi-fanout nets, denoted as circuit C. $FF_i$ is the $i^{\text{th}}$ flip-flop in circuit C. The flip-flop set $\mathcal{F}$ contains all flip-flops that do not belong to any sequential loop. The flip-flop set $\mathcal{L}_j$ contains all flip-flops belonging to the $j^{\text{th}}$ sequential loop, and $\mathcal{L}$ ($\mathcal{L} = \bigcup_j \mathcal{L}_j$) is the set of all flip-flops belonging to any sequential loops.*

**Definition 2** *$\mathcal{S}(FF_i)$ is defined as the **father-cone set** of $FF_i$, including all state elements (including both primary inputs and flip-flops) that directly determine the next state of $FF_i$. **Ancestor-cone set** $\mathcal{A}(FF_i)$ includes all state elements that directly or indirectly determine the state of $FF_i$ in one or more clock cycle(s).*

**Lemma 1** *In circuit C, $\mathcal{S}(FF_k) \bigcap \mathcal{S}(FF_n) = \emptyset$ when $k \neq n$.*

**proof 1** *If there exist a common state element within $\mathcal{S}(FF_k)$ and $\mathcal{S}(FF_n)$, there must be a multi-fanout net in the circuit, which is conflicted with the definition of circuit C.*

**Lemma 2** *The states of any elements in $\mathcal{L}_j$ cannot affect the states of elements outside of this set.*

**proof 2** *Suppose there is a flip-flop $FF_k$ belonging to the flip-flop set $\mathcal{L}$, whose father-cone set $\mathcal{S}(FF_k)$ includes at least one flip-flop $FF_n$ belonging to $\mathcal{L}_j$. Since all flip-flops in $\mathcal{L}_j$ are connected one by one to form a sequential loop, $FF_k$ also belongs to the father-cone set of another flip-flop $FF_q$ in $\mathcal{L}_j$. Therefore, $\mathcal{S}(FF_k)$ and $\mathcal{S}(FF_q)$ have a common state element, which contradicts Lemma 1.*

**Definition 3** *The* sequential level *of a flip-flop $FF_i$ in set $\mathcal{L}$ is set to be the maximal sequential level of all state elements in $\mathcal{S}(FF_i)$ plus one, assuming the sequential level of all primary inputs is 0.*

**Theorem 1** *In a circuit C, suppose the maximum sequential level of the elements in $\mathcal{L}$ is n, any state of $\mathcal{L}$ can be reachable within n clock cycles from the primary inputs.*

The above theorem proves that no functionally-unreachable states exists in the *NLF* of *CS*. However, it cannot guarantee the same conclusion holds true for sequential loop structure (i.e. some FFs are connected by combinational logic and

form a cycle). Consider the circuit shown in Fig. 2.2 as an example, we observe that if $FF0$ and $FF1$ are initialized as $\{FF0(0), FF1(0)\}$, it would never escape from this state even if we change the states of $FF2-FF5$, the other three states for $FF0$ and $FF1$, therefore, are not reachable. Such sequential loop-induced illegal states are very difficult, if not impossible, to be found by any automatic illegal state identification method. This is because: (i). a sequential loop expands their effects to different sequential levels in an infinite number of time frames; (ii). which states are reachable depends on the initial state of the involved flip-flops.

Fortunately, the illegal states caused by sequential loop are rare cases, because the conditions to form sequential loop-induced unreachable states are quite stringent: (i). between any connected flip-flops on the loop, there must be a controlling path as shown in Fig. 2.2, which is a logical path where a controlling value can be directly propagated from the beginning to the end; (ii). all controlling paths should have transitivity, i.e., the output controlling value of a path should be the input controlling value of the next path. If any one of the above two conditions is not true, any states in the sequential loop is functionally-reachable. For example, in Fig. 2.2, suppose we just change the NOR gate which connects with $FF0$ to an OR gate, the controlling path from $FF1$ to $FF0$ is broken. The value of $FF0$ is not solely dependent on its previous on-loop flip-flop $FF1$. Starting from this flip-flop, we first set its value by assigning $FF4 = 1$ and $FF5 = 1$ such that $FF0$ can be flipped to logic 1 and then controlling path between $FF0$ and $FF1$ will be also broken. Therefore, the state of the sequential loop in next clock cycle can be determined by $FF2-FF5$, thus can reach any state. Moreover, even for these rare sequential loop-induced unreachable states, their impact on testing is rather limited because the involved state elements typically span a number of clock cycles while we target defects in combinational logic between adjacent cycles in testing.

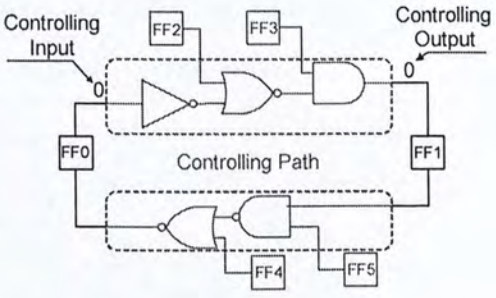


Figure 2.2: Sequential Loop-Induced Unreachable States

Because of the above reasons, we can simply ignore the unreachable states caused by sequential loops without damaging the effectiveness of pseudo-functional testing much.

## 2.4 Illegal State Identification Flow

As discussed earlier, illegal states would imply logic violations at different branches of the same multi-fanout net, explicitly or implicitly. A first thought to generate illegal states is then to propagate contradictive logic values at different branches of multi-fanouts concurrently, and find out which state cubes can justify this combination. These state cubes can then be deemed as illegal. We initially tried out for this intuitive method and found out it is not a good solution. This is because: (i). we need to avoid the case that two fanout branches with contradictive values propagate through the same logic elements (as we cannot determine its value under such circumstances), which results in incomplete identified illegal states; (ii). as we need to propagate contradictive logic values at the branches of each multi-fanout net pairwisely, for those nets that contain a high number of branches, we need to propagate along each branch multiple times and it is a waste of computational efforts. To resolve the above problems, instead of propagating contradictive values at

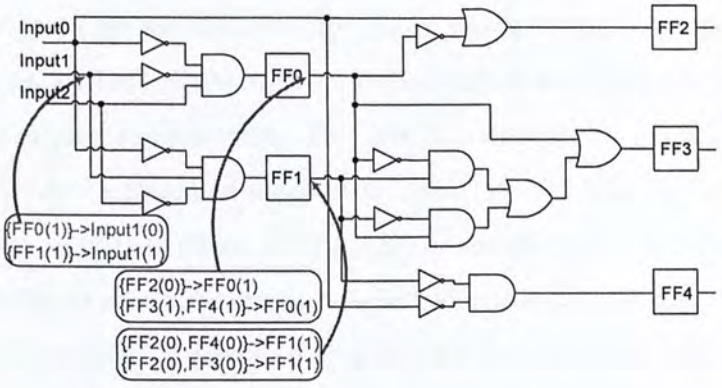


Figure 2.3: An Example Circuit for Illegal State Identification

multi-fanout nets, we determine which state cubes can justify logic '1'('0') at the multi-fanout nets independently and use this information to obtain illegal states. We use the an exemplar circuit shown in Fig. 2.3 to explain the main flow of the proposed illegal state identification scheme, as depicted in Fig. 2.4.

Let us define the so-called *justification scheme* at every circuit node in the format of $Cube0 \rightarrow 0$ and $Cube1 \rightarrow 1$, denoting that a state cube $Cube0/Cube1$ justifies logic '0/1' on this node. For example, a justification scheme $FF0(1) \rightarrow Input1(0)$ in Fig. 2.3 means that $FF0 = 1$ can justify logic '0' at circuit node $Input1$. All such justification schemes are systematically built for each net before unreachable state extraction (detailed in Chapter 2.5). According to previous discussion, we always start from a multi-fanout net and try to derive illegal states using contradictory justification schemes. In this example, for the multi-fanout at $Input1$, we have $\{FF0(1)\} \rightarrow Input1(0)$ and $\{FF1(1)\} \rightarrow Input1(1)$. We can therefore conclude that the state cube $\{FF0(1), FF1(1)\}$ is illegal as they justify contradictory values in this fanout. The above procedure needs to be conducted for every multi-fanout nets.

As the illegal state cubes obtained from different paths may contain redundant

information (e.g., the set of illegal state cubes shown in Fig. 2.1 is not the most compact one), we need to remove such redundancy so that all the cubes are minimized and disjoint to each other. This step is conducted by gradually building up a hypergraph for identified illegal state cubes. That is, each flip-flop ($FFx$) is split into two vertices ($FFx(0)$ and $FFx(1)$) to denote its two possible logic values, and an illegal state cube can be represented as a hyperedge that connects the corresponding vertices. We define the following relationships between hyperedges $A$ and $B$:

- if $A$ connects to only a subset of vertices of $B$, we denote this relationship as *$A$ dominates $B$*. Apparently, the corresponding illegal state cube for hyperedge $A$ contains that of $B$;
- if both $A$ and $B$ connect to $n$ vertices, and among them $n-1$ vertices are the same and the remaining one corresponds to the same flip-flop with different logic values, we denote this relationship as *$A$ and $B$ complement each other*. These two hyperedges can be replaced by a single hyperedge that connects to the $n-1$ common vertices according to our definition.

Every time we add a new hyperedge into the graph, we check the above relationships for its related edges and finally we get a compact set of illegal state cubes without redundancy. For example, for the illegal state cubes shown in Fig. 1, they can be finally compacted into five illegal state cubes: $\{FF0(1), FF1(1)\}$, $\{FF3(0), FF4(0)\}$, $\{FF2(0), FF3(0)\}$, $\{FF2(0), FF3(1), FF4(0)\}$, and $\{FF2(1), FF3(1), FF4(1)\}$, using the above method.

Next, we expand the current illegal state cubes to the next sequential level, again, using the justification scheme information. For the example circuit in Fig. 2.3, according to structural analysis, two justification schemes at $FF0$ and $FF1$ are detected. We explore all possible combinations of expanded unreachable cubes, and

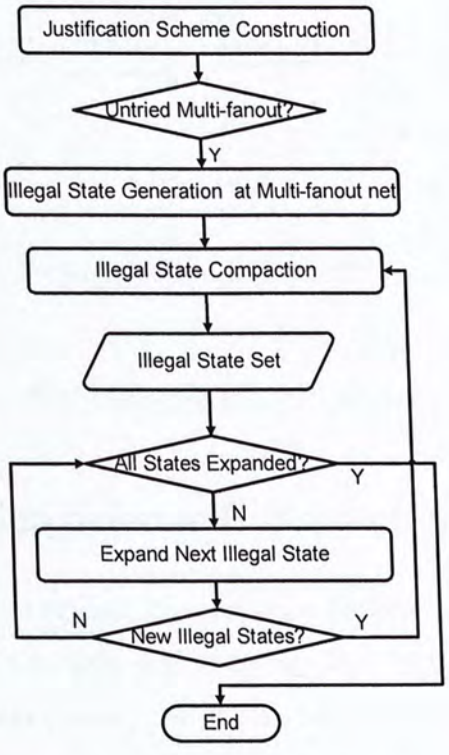


Figure 2.4: Flowchart for the Proposed Illegal State Identification Scheme

obtain new cubes, e.g., $\{FF2(0), FF4(0)\}$ and $\{FF2(0), FF3(0)\}$. As shown in our flowchart in Fig. 2.4, this step is conducted in a stack-like manner. That is, whenever we generate new illegal state cubes through expansion, they will be pushed into the illegal state set for later expansion as well. Compared to prior work that explicitly expand the circuit into a few time frames for illegal state justification, the above procedure is able to implicitly walk through an unlimited number of time frames of the circuit efficiently. Finally, the whole procedure for our illegal state justification terminates when there is no unexpanded illegal state.

So far we have discussed how to conduct illegal state extraction, expansion, and compaction, with available justification schemes at every circuit node. In the rest of the chapter, the key issue in our algorithm, how to construct these justification

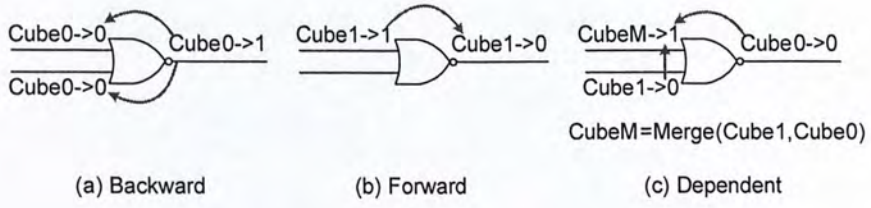


Figure 2.5: Propagation Rules for Justification Schemes

schemes in an efficient and effective manner, is discussed in detail.

## 2.5 Justification Scheme Construction

We start our justification scheme construction at the input of each flip-flop, which can be obtained directly with the state of the flip-flop. We then propagate these initial justification schemes to every circuit node based on the following propagation rules:

**Rule 1: Backward Propagation**

A non-controlled value at the output of a logic gate (e.g., logic '1' for a NOR gate) will imply non-controlling value for its inputs of this logic gate. Therefore, a justification scheme that justifies a non-controlled value at the output of a logic gate can be propagated to all inputs of this logic gate to justify the non-controlling value there, referred as *backward propagation*. The backward propagation of justification schemes over a two-input NOR gate is shown in Fig. 2.5(a) as an example.

**Rule 2: Forward Propagation**

A justification scheme that justifies the controlling value at any input of a logic gate also justifies the controlled value at its output (e.g., logic '1' for an OR gate). If this scenario occurs, we can propagate the justification scheme from the input to the output directly, referred as *forward propagation*. Similarly, an example for two-input NOR gate is shown in Fig. 2.5(b).

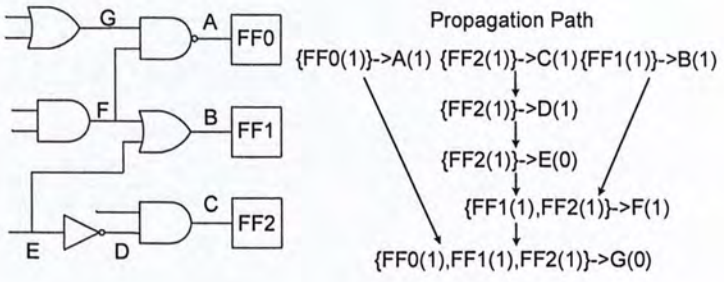


Figure 2.6: Generation of Sophisticated Justification Schemes

**Rule 3: Dependent Propagation**

Generally speaking, to justify a controlling value at a certain input of a logic gate, we need to justify its output to be the controlled value as well as all other inputs to be the non-controlling value. Therefore, dependency exist when propagating such justification schemes and we need to merge state cubes to obtain such justification schemes, referred as *dependent propagation*. Again, an example for NOR gate is shown in Fig. 2.5(c). Please note that, no justification schemes would be generated if there is any conflict when merging state cubes.

Let us use an example circuit (see Fig. 2.6) to show how we can build sophisticated justification schemes based on the above propagation rules. After initialization, three schemes $FF0(1) \rightarrow A(1)$, $FF2(1) \rightarrow C(1)$ and $FF1(1) \rightarrow B(1)$ are built. Then, justification scheme $FF2(1) \rightarrow C(1)$ is propagated along path $C-D-E$ and we can obtain justification scheme $FF2(1) \rightarrow E(0)$ at $E$. Together with $FF1(1) \rightarrow B(1)$, we have $\{FF1(1), FF2(1)\} \rightarrow F(1)$. According to the above and $FF0(1) \rightarrow A(1)$, we can finally obtain $\{FF0(1), FF1(1), FF2(1)\} \rightarrow G(0)$. The last two schemes which is generated based on dependent rule cannot be extracted by implication based technique since such method can only generate justification schemes between single flip-flop and single node.

The above propagation rules, however, are not enough for us to build complete

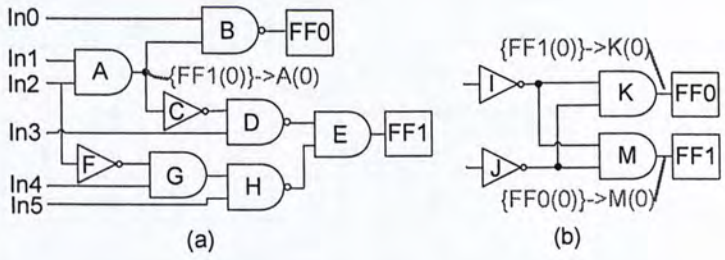


Figure 2.7: The Impact of Reconvergent Nodes

justification schemes. For example, for the example circuit shown in Fig. 2.7(a), if we would like to propagate justification scheme $FF1(0) \rightarrow E(0)$ backwardly, as logic '0' is a controlled value for AND gate $E$, based on dependent propagation rule, we need to merge $FF1(0)$ with the cube that justifies $H(1)$ (i.e., $FF1(1)$) to obtain the cube that justifies $D(0)$. Apparently, this merging is not possible and hence we have to stop propagation through gate $D$. Consequently, we cannot obtain justification scheme $FF1(0) \rightarrow A(0)$ based on our earlier propagation rules. This justification scheme, however, does exist, because $E(0)$ implies $D(0)$ and/or $H(0)$ while both $D(0)$ and $H(0)$ imply $A(0)$. Similarly, the justification schemes shown in Fig. 2.7(b) cannot be obtained with the above propagation rules. A closer examination of the circuit shown in Fig. 2.7(a) reveals that the existence of the justification scheme $FF1(0) \rightarrow A(0)$ is due to the fact that gate $E$ is a reconvergent node of multi-fanout net $In2$. That is, both inputs of gate $E$ are affected by $In2$ and this essentially leads to the implication of $A(1) \rightarrow In2(1) \rightarrow E(1)$ and hence $FF1(0) \rightarrow E(0) \rightarrow A(0)$. For the circuit in Fig. 2.7(b), even though there is no reconvergent node in the traditional sense, considering logic cubes are propagated both forwardly and backwardly, gates $K$ and $M$ can be also deemed as *virtual* reconvergent nodes.

As the occurrence of the above missing justification schemes can be attributed to reconvergent nodes, to resolve this problem, we conduct implication at every

| Benchmark | [39] | | | | Proposed | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total # | # of 2-bit | # of 3-bit | # of 4-bit | Total # | # of 2-bit | # of 3-bit | # of 4-bit | # of 5-bit | # of Large | # of Expanded Cubes |
| s208 | 16 | 16 | 0 | 0 | 23 | 23 | 0 | 0 | 0 | 0 | 0 |
| s444 | 16 | 12 | 4 | 0 | 49 | 14 | 20 | 15 | 0 | 0 | 0 |
| s953 | 116 | 101 | 15 | 0 | 365 | 153 | 133 | 78 | 1 | 0 | 0 |
| s1196 | 35 | 10 | 25 | 0 | 138 | 10 | 43 | 42 | 20 | 23 | 0 |
| s5378 | 1006 | 293 | 711 | 2 | 8516 | 399 | 2584 | 266 | 2383 | 2884 | 681 |
| s9324 | 195 | 47 | 148 | 0 | 1109 | 47 | 168 | 108 | 204 | 582 | 17 |
| s13207 | 2809 | 1222 | 1330 | 257 | 82291 | 1445 | 4156 | 14002 | 27760 | 34928 | 60140 |
| s15850 | 507 | 112 | 392 | 3 | 5568 | 239 | 399 | 777 | 261 | 3892 | 65 |
| s38417 | 988 | 483 | 496 | 9 | 90983 | 864 | 3094 | 16544 | 34641 | 35840 | 42559 |
| s38584 | 10609 | 1591 | 9018 | 0 | 63558 | 2163 | 11947 | 235 | 10783 | 28430 | 39213 |

Table 2.1: Experimental Results for Illegal State Identification.

circuit node and we record those implications that lead to the same non-controlling value of a logic gate (if not, there is no reconverging effects). For example, by doing so, we can learn $A(1)$ implies the non-controlling logic '1' at both inputs of logic gate $E$. Hence we have $A(1) \rightarrow E(1)$ and we would record $E(0) \rightarrow A(0)$, which can be used during propagation through gate $E$ to obtain $FF1(0) \rightarrow A(0)$. Based on the same principle, we would record $K(0) \rightarrow M(0)$ and $M(0) \rightarrow K(0)$ for the circuit shown in 2.7(b), which can then be used to build the justification schemes shown in the figure. It is important to note that, those implications that do not lead to the same non-controlling value of a logic gate would be discarded, as they can be obtained based on our propagation rules.

By conducting the above implication procedure before the justification scheme propagation process, we are able to build complete justification schemes for the circuit.

## 2.6 Experimental Results

To evaluate the effectiveness of the proposed solution, we perform experiments on several ISCAS'89 benchmark circuits, comparing against the implication-based algorithm in [39], which is shown to be able to obtain much more illegal states than the SAT-based method in [16]. We do not compare against [31] and [38] because they do not store functional constraints in the format of illegal states (in arbitrary nets instead). Our experiments are conducted on a 2GHz PC with 1GB memory.

In [39], the authors reported the total number of illegal state cubes only. Let us first compare this value obtained using the two algorithms, as shown in Table 3.2 (see Columns 2 and 7). Apparently, we can find much more illegal states when compared to [39]. It should be also highlighted that our algorithm is generally more effective for those large circuits.

As one 2-bit illegal state cube can be represented as two 3-bit illegal state cubes, this makes comparing the total number of illegal state cubes not quite dependable. As a result, for fair comparison, we re-implement the algorithm in [39] and all illegal state cubes detected by the two algorithms are compacted in the same manner to remove redundancy[1]. In Table 1, we classify all the illegal state cubes according to the number of specified bits. We can observe that most illegal state cubes detected by [39] are 2-bit or 3-bit cubes (there are no cubes with size more than 4). This is because their rather simple implication scheme can only obtain relatively small-sized functional constraints. The proposed method, on the other hand, is able to generate a great amount of large-sized illegal state cubes (i.e., cube size larger than 3). More importantly, our method also obtains much more 2-bit and 3-bit cubes than [39], which is able to restrict the functional space more

[1]Due to the different implementations, our results for [39] are different from that reported in [39]. We have sent the source code for our implementation to the authors of [39].

effectively. It can be also observed from the table that our illegal state expansion technique is quite effective. For certain benchmark circuits (e.g., s13207), the illegal cubes generated by expansion can reach nearly 80% of the total identified illegal states.

Finally, we carefully examine the illegal state cubes generated from the two algorithms, and we find out that all the illegal states obtained in [39] are covered in the illegal state cubes obtained using our proposed method. This further proves that multi-fanout nets are the main structural root cause for functionally-unreachable states, and gives us more confidence to realize the potential of pseudo-functional testing using the proposed methodology.

## 2.7 Conclusion

The discrepancy between ICs' activities in normal functional mode and that in structural test mode has an increasingly adverse impact on the effectiveness of manufacturing test with technology advancement, as reported in several industrial studies (e.g., [5, 29, 30]). Pseudo-functional testing seems to have great potential to resolve this problem, but whether we could realize this potential highly relies on whether we could effectively identify as many illegal states as possible. In this chapter, we show that the main structural root cause for illegal states is the multi-fanout nets in the circuit. Based on this observation, we develop efficient and effective algorithms for illegal state identification. Experimental results on ISCAS'89 benchmark circuits demonstrate that the proposed technique is much more effective when compared to state-of-the-art solutions.

---

**□ End of chapter.**

# Chapter 3

# Compression-Aware Pseudo-Functional Testing

## 3.1 Introduction

Since pseudo-functional tests need to avoid a large amount of identified illegal sates, the number of specified bits in pseudo-functional patterns are usually much larger than that of structural patterns. In other words, the percentage of X-bits in pseudo-functional patterns can be quite low. This has a severe impact on the effectiveness of test data compression techniques, since they mainly exploit X-bits in test cubes to achieve significant test volume reduction without sacrificing fault coverage. Since on-chip test compression has become a *de facto* DfT technique used in the industry, how to effectively apply pseudo-functional patterns in test compression environment is a key issue for the success of pseudo-functional testing.

Method [15] targeted the same problem as ours, by adding decompression logic into CUT and activating illegal states as constraints, they can generate the compressible and pseudo-functional test patterns directly. However, as number

of illegal states identified by our method is much larger than that used in [15], their method is not suitable for current situation. In this chapter, we propose novel compression-aware pseudo-functional testing techniques to address the above problem. Firstly, we insert functional constraints as *phantom gates* into the circuit so that the ATPG engine could take them into account automatically. Then, instead of activating all the functional constraints during ATPG, which inevitably leads to a large amount of specified bits in obtained patterns, we only activate the relevant ones for the targeted fault to generate compression-friendly patterns. Obviously, it is possible that the decompressed test patterns violate certain functional constraints as they are not considered. This issue is addressed by: (i). we propose novel heuristics to fill the free X-bits in test cubes so that the number of violated constraints is as small as possible; (ii). for the violated constraints (if any) that might lead to incidental test overkills, we take advantage of the available X-tolerant compactor in TDC architecture to mask the error effects from those faults that are detectable only because of the illegal states. Moreover, we also show how to generate compressible random patterns that do not violate functional constraints, by applying multiple launch cycles before the actual capture cycle. Experimental results on ISCAS'89 benchmark circuits show the effectiveness of our proposed compression-aware pseudo-functional testing technique.

The remainder of this chapter is organized as follows. Chapter 3.2 reviews related work and motivates this chapter. In Chapter 3.3 and Chapter 3.4, we detail our proposed methodology for compression-aware pseudo-functional testing and our test pattern generation process, respectively. Experimental results on several large ISCAS'89 benchmark circuits are then presented in Chapter 3.5 to show the effectiveness of the proposed solution. Finally, Chapter 3.6 concludes this chapter.

## 3.2 Motivation

The functionally-unreachable space for an integrated circuit can be quite large. Consequently, when we generate only functionally-reachable patterns during ATPG, the number of specified bits in pseudo-functional patterns would be much higher than conventional structural patterns that do not consider such functional constraints. Let us use benchmark circuit *s*9234 as an example (see Fig. 3.1), after applying functional constraints, more than 80 percent of the pseudo-functional patterns have specified bits in the range of $20\% - 30\%$. In contrast, only less than two percent of structural patterns have more than 20% specified bits.

Since TDC techniques rely on the large percentage of X-bits in test cubes for efficient test data volume reduction, if we directly apply pseudo-functional patterns in test compression environment, the compression ratio is reduced dramatically when compared to applying traditional structural patterns. Polian and Fujiwara [24] studied this issue using a code-based TDC technique and showed the compression ratio loss due to functional constraints.

As on-chip test compression techniques, in particular, linear decompressor-

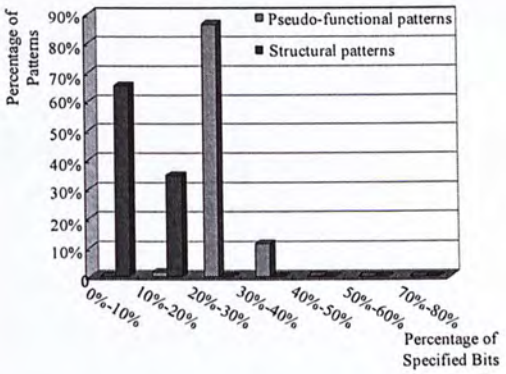


Figure 3.1: Specified Bits in Pseudo-Functional Patterns and Structural Patterns for s9234

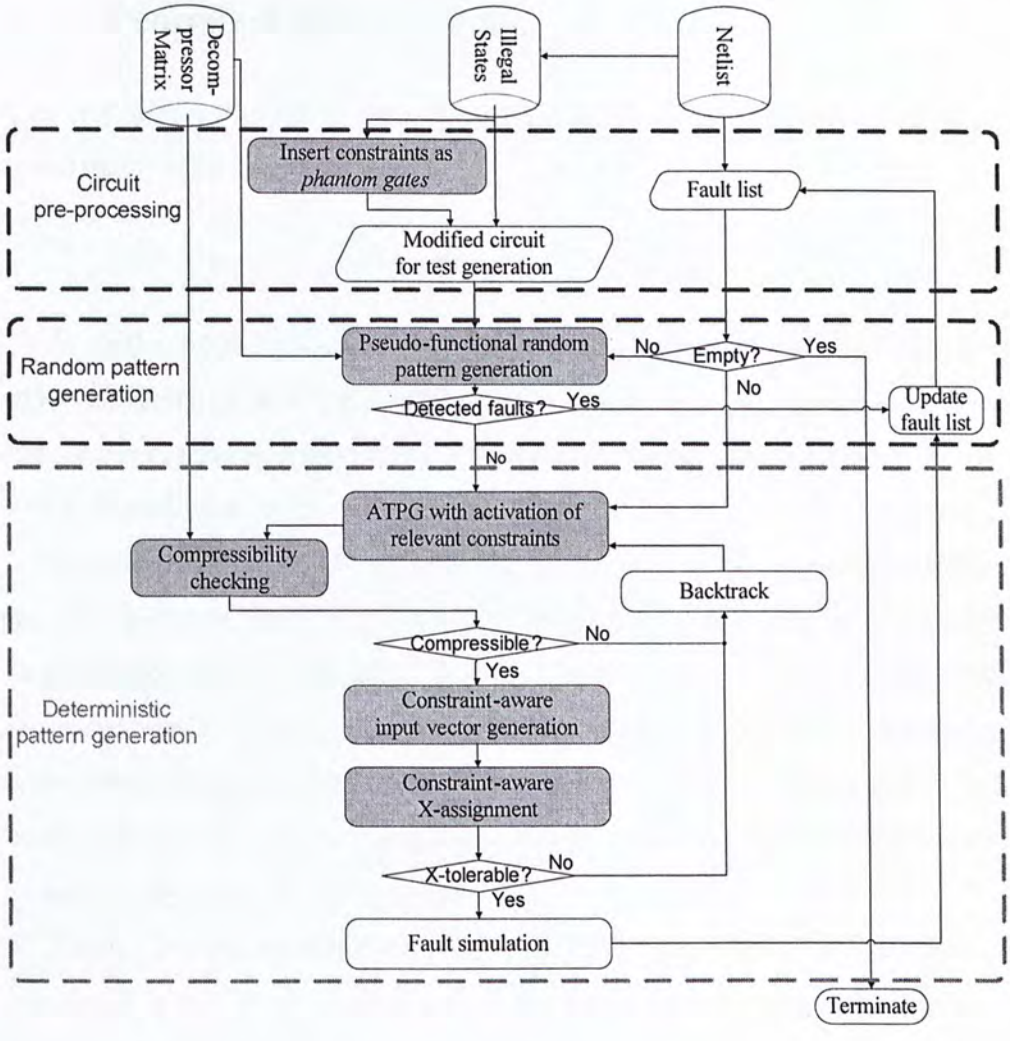


Figure 3.2: Pattern Generation Framework in Compression-Aware Pseudo-Functional Testing

based TDC has become the *de facto* DfT methodology widely used in the industry. How can we apply pseudo-functional tests in test compression environment effectively is an important and challenging problem. The above has motivated the *compression-aware pseudo-functional testing* methodology investigated in this chapter, as detailed in the rest of this chapter.

## 3.3 Proposed Methodology

Our proposed methodology for compression-aware pseudo-functional testing is based on the following observation:

That is, Non-functional patterns do *NOT* always lead to over-testing.

In another word, *only if a non-functional pattern detects delay faults on functionally-infeasible paths*, good circuit may fail this test pattern. Therefore, instead of applying pseudo-functional patterns only, we allow non-functional patterns to be applied to the circuit in our proposed technique.

By doing so, we do not need to activate all the functional constraints during ATPG. Instead, we propose to only activate those relevant constraints for the targeted fault, which facilitates to generate compression-friendly test cubes with fewer specified bits. Obviously, it is possible that the decompressed test patterns violate some functional constraints as they are not considered. The question becomes how can we avoid test yield loss induced by these violated functional constraints (if any), and we tackle it as follows.

Firstly, for a compressible test cube, typically we still have many X-bits left in it after solving the linear equations corresponding to its specified bits. We propose novel heuristics to fill them so that the number of violated functional constraints is as small as possible.

Then, for the remaining violated constraints (if any), we take advantage of the available X-tolerant compactor in on-chip test compression architecture to mask the error effects from those faults that are detectable only because of the illegal states. To be specific, for every illegal state existing in the decompressed test pattern, we break the corresponding violation by setting one of its involved state elements to be 'X' during fault simulation. By doing so, if a delay fault is detectable by a decompressed test pattern only due to the existence of the illegal states in

the pattern, its error effect in test response will be also 'X'. Then, since we mask them in the X-tolerant compactor, such an unexpected fault becomes untestable with this pattern and hence the applied non-functional pattern would not result in test overkill.

In conventional broad-side delay testing, we typically apply two-pattern tests, i.e., one cycle for *launch* and one cycle for *capture*. If we are able to apply multiple launch cycles before the capture cycle, however, the chance to have non-functional patterns during capture is significantly reduced [18]. The reason is simple: instead of being scanned in with any possible values, the applied patterns have gone through the functional logic for several cycles and are largely functionally-constrained (not guaranteed though, due to the initial illegal launch pattern). While introducing multiple launch cycles in the deterministic test pattern generation process is usually prohibited due to the associated huge computational complexity, we propose to apply multi-launch cycles for random patterns, which is able to avoid over-testing without incurring high ATPG effort.

Based on the above, the overall framework for our compression-aware pseudo-functional testing methodology is presented in Fig. 3.2. It is worth noting that, while this framework is generic enough to be applicable for detecting any kinds of faults in linear decompressor-based test compression environment when over-testing is of a concern, we focus on transition faults in this work and we assume broad-side testing is applied to detect them. The pattern generation procedure is detailed in the following subchapter.

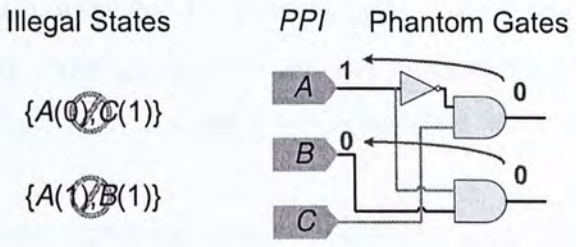


Figure 3.3: Insertion and Activation of Functional Constraints as Phantom Gates

## 3.4 Pattern Generation in Compression-Aware Pseudo-Functional Testing

As shown in Fig. 3.2, our proposed pattern generation framework takes the circuit netlist, the transformation matrix for the linear decompressor and the functional constraints (i.e., illegal state cubes) in the circuit extracted using the tool proposed in chapter 2 as inputs and output compression-aware input vectors to be stored in the ATE. It is comprised of three main phases: circuit pre-processing, pseudo-functional random pattern generation and compression-aware deterministic pattern generation for pseudo-functional testing.

### 3.4.1 Circuit Pre-Processing

In the circuit pre-processing phase, we first expand the circuit into two time frames, by changing the internal flip-flops to be pseudo-primary inputs (PPIs) and pseudo-primary outputs (PPOs).

Then, for the functional constraints that are given as illegal state cubes (e.g., $\{A(0), C(1)\}$), different from prior work that represents such constraints as independent formulas in conjunctive normal form (CNF) during the ATPG process [19], we insert *phantom logic AND gates* into the circuit to represent them,

as shown in the example in Fig. 3.3. Each phantom gate corresponds to an illegal state by linking its corresponding PPIs (directly or through an inverter), and its output would be logic '1' if and only if a fully specified test pattern contains this illegal state.

The above representation has several advantages: (i). the ATPG engine does not need to maintain a great number of independent CNF formulas; (ii). by integrating functional constraints into the circuit, it is more convenient to generate pseudo-functional patterns since we only need to set the outputs of the phantom *AND* gates as logic '0' and label them as an *unjustified* value. The ATPG engine will automatically take such functional constraints into account. (iii). we have the flexibility to activate a subset of the functional constraints in each ATPG run, which is important for our proposed methodology, as shown in Chapter 3.3.

### 3.4.2 Pseudo-Functional Random Pattern Generation with Multi-Launch Cycles

The functionally-unreachable space for an integrated circuit can be quite large. Take the ISCAS'89 benchmark circuit s9234 as an example, we can obtain forty seven 2-bit illegal state cubes.

For each of the 2-bit illegal state, a random pattern has $3/4$ probability to avoid it, but the probability will decrease to $(3/4)^n$ if there are $n$ independent 2-bit illegal states. Directly applying random test patterns, therefore, is almost certain to violate one or more functional constraints.

Fortunately, as discussed earlier, if we apply multiple launch cycles before the capture cycle, the chance for a test pattern to be a functional pattern is significantly increased. Based on this observation, we propose to generate pseudo-functional random patterns with the algorithm shown in Fig. 3.4.

In line 1, we initialize a variable *Num_Valid_Pattern* to be zero. Then, in each

```
1.  Num_Valid_Pattern=0;
2.  while(Num_Valid_Pattern < 16) {
3.      Generate 32 random input vectors IP{32};
4.      Obtain 32 patterns SP{32} in the first launch cycle;
5.      for(i = 0;i ≤ Num_Random_Cycles;i++) {
6.          Simulate SP{32}, output OP{32};
7.          SP{32} = OP{32};}
8.      Num_Valid_Pattern=Constraint_Check(OP{32});}
9.  Num_Detected_Faults=fault_simulation(OP{32});
10. if(Num_Detected_Faults!=0){
11.     update_faultlist();
12.     goto line 1;}
13. else
14.     terminate;
```

Figure 3.4: Algorithm for Pseudo-Functional Random Test Pattern Generation.

iteration (lines 2-10), we generate 32 random input vectors $V\{32\}$ supplied to the linear decompressor (by taking advantage of the parallel fault simulator). By doing so, the test vectors applied in the first launch cycle $SP\{32\}$ are guaranteed to be compressible. Next, we conduct functional simulation for a consecutive of *Num_Random_Cycle* cycles, which is a pre-defined value for the launch cycles and it is set as four in our experiment, to obtain the actual test patterns applied in the capture cycle $OP\{32\}$.

Next, we check how many patterns in $OP\{32\}$ do not violate any functional constraints, and record it in *Num_Valid_Pattern*. If *Num_Valid_Pattern* is equal or larger than 16, we conduct fault simulation for this batch of test patterns. Otherwise, they are abandoned. The idea behind this is that fault simulation takes longer time than logic simulation and we do not want to waste time to simulate only few functionally-constrained patterns.

Note that, we only conduct fault simulation with those patterns in $OP\{32\}$

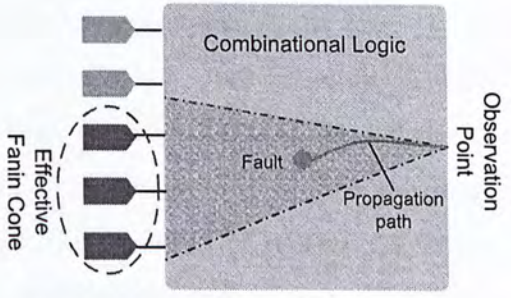


Figure 3.5: Effective Fan-in Cone for a Fault.

that do not violate any functional constraints and the number of detected faults is stored in *Num_Detected_Faults*. If we able to detect any new transition faults with these patterns, the procedure goes back to line 1 to generate another batch of random patterns. Otherwise, we abort random pattern generation and resort to deterministic patterns to cover the remaining faults, as detailed in the following subchapters.

### 3.4.3 Compressible Test Pattern Generation for Pseudo-Functional Testing

We implement a constrained ATPG engine based on the *FAN* algorithm [9] for deterministic pattern generation. In such an ATPG engine, we can put the values for internal gates as to-be-justified values and we are able to backtrack to try another solution whenever a logic conflict occurs.

#### Dynamic Activation of Functional Constraints

As discussed earlier, the illegal states in large ICs are enormous and if we activate all of them in test compression environment, dramatic fault coverage loss would be incurred because we are not able to generate many compressible patterns that

satisfy such large number of constraints.

In our proposed methodology, when generating a deterministic pattern for a particular transition fault, we only activate its *relevant functional constraints* (simply by assigning logic '0' at the output of its corresponding phantom gate), which is obtained as follows. During the ATPG process, the targeted fault propagates its faulty value to POs or PPOs (referred as *observation points*). For each observation point, we define its fan-in logic cone as an *effective fan-in cone (E-cone)* for the targeted fault (see Fig. 3.5). Since, if we want to detect the fault with a particular observation point, only those functional constraints existing in its E-cone can affect the detection of this fault, they are defined as the relevant functional constraints and therefore need to be activated during test generation.

Note that, there may exist more than one propagation path for a fault, and they correspond to different observation points and hence different E-cones for this fault. Our constrained ATPG tool selects one of them in each run and backtracks to try another one if conflicts occur (e.g., a functional constraint within its E-cone is violated or the pattern is not compressible). Whenever backtracking occurs, we dynamically activate a new set of functional constraints and at the same time de-activate the previous functional constraints.

### Constraint-Aware Input Vector Generation

Whenever we generate a deterministic test pattern $P$, we need to check whether this pattern is compressible (see Chapter 3.2.3). If not, we try to generate a different pattern. Otherwise, we need to solve the linear equations corresponding to the specified bits and get the input vector $V$ for this pattern. Since a compressible test cube may have more than one solution, for the ease of pseudo-functional testing, we would like to have a solution that violates the least number of functional constraints (if any). A greedy heuristic is proposed in this subchapter to achieve the

```
1.  V=getinitialV();
2.  Gain=IFINITE;
3.  while(Gain == 0) {
4.      MGain = Gain = 0,Mbit = −1;
5.      P=M × V;
6.      Num_Vio=violation_checking(P);
        /* select a bit which reduce the most constraints*/
7.      for(i = 0;i ≤ Num_Vbit;i++) {
            /*skip if this bit pivot input bit or it has been flipped*/
8.          if(is_pivotbit(V[i]) || has_flipped(V[i]))
9.              continue;
10.         flip(V[i]);
11.         P = M × V;
12.         Gain=Num_Vio-violation_checking(P);
13.         flip(V[i]);
14.         if(Gain > MGain) {
15.             MGain = Gain;
16.             Mbit = i;}
17.         Gain = MGain;}
18.     if(Gain > 0)
19.         flip(V[Mbit]);}
```

Figure 3.6: Algorithm for Constraint-Aware Input Vector Generation.

above objective.

Before introducing our algorithm, we first divide all the bits in an input vector into three categories: *pivot-bits*, *free-bits* and *stack-bits*, which correspond to *pivot columns*, *all zero columns* and the rest columns in the reduced coefficient matrix $M'_s$. Take the example reduced augmented matrix shown in Eq. 1.3 as an example (see Chapter 3.2.3), the pivot-bits are $s_1$, $s_2$, $s_4$ and $v_5$; the free-bits are $s_3$, $v_1$, $v_3$ and $v_6$; while $v_2$ is a stack-bit.

When solving the equations, the value for the pivot-bits are determined and

they are equal to the scalar multiplication between $P'_s$ and the corresponding pivot column (e.g., $s_1 = (1\ 0\ 0\ 0) \cdot (0\ 1\ 0\ 0)^T = 0$), stack-bits can be set as 0, free-bits can be freely assigned with $0/1$. Therefore, for this example, we can get an initial solution as $V = (\mathbf{0},\ \mathbf{1},\ 1,\ \mathbf{0},\ 0,\ \mathbf{0},\ 1,\ 0,\ \mathbf{0},\ 1)^T$, where the bold values represent those bits that are fixed.

Starting from a given input vector, we define a $flip(i,V)$ operation that transforms $V$ from current solution to another valid solution by flipping the $i^{th}$ bit in $V$ provided it is not a pivot-bit. Apparently, for free-bits, they can be freely flipped. However, the flipping of a stack-bit involves flipping some other bits to guarantee the generated vector is valid. Let us demonstrate how this is done by flipping stack-bit $v_2$. We first scan in its corresponding column in $M'_s$ (i.e., $(1,\ 0,\ 1,\ 0)^T$), and then we find all the non-zero entries (the first entry and the third one) and locate the pivots in the same rows (i.e., $M'_s(1,1)$ and $M'_s(4,3)$). Pivot bits of input vector which correspond to the first and forth columns also need to be flipped, and we can obtain a new valid input vector $V$=($\mathbf{1}$, 1, 1, $\mathbf{1}$, 0, $\mathbf{1}$, 1, 0, 0, 1).

Based on the above, Fig. 3.6 presents the pseudo-code of our proposed constraint-aware input vector generation algorithm. $V$, $P$ and $M$ represent the input vector, the test pattern and the transformation matrix, respectively. Procedure $violation\_checking(P)$ returns the number of violated functional constraints in pattern $P$ and $Num\_Vio$ is used to record this value. *Gain* represents the reduced number of violated functional constraints benefited from a $flip()$ operation. *MGain* denotes the maximum gain that we can achieve by flipping one bit in current input vector and *Mbit* is the index to that bit. Our algorithm starts from an initial input vector, and then enters a while loop to iteratively reduce the number of violated functional constraints. In each iteration, we try to flip one free-bit or stack-bit in $V$ and evaluate the *Gain*, and we select the bit with the maximum *Gain* to flip. Finally, the algorithm terminates itself when there is no *Gain* any more.

```
1.  initialization();
    /*break more constraints with less X-assignment*/
2.  while(TRUE){
3.      index = getmaxviolation();
4.      if(index ≠-1){
5.          Assign PI[index] as X;
6.          update_constraint(Violated_Constraints);
7.          update_effect_ocone(Scan_Count); }
8.      else
9.          break;}
    /*for a particular constraint, select to assign X to the PPI
    that makes potential X-response distributed more evenly*/
10. while(Violated_Constraints is not empty){
11.     Constraint = Next(Violated_Constraints);
12.     MinCost=INFINITE, Cost=0, MPPI=-1;
        /*select a PPI from illegal cube making
        potential X response distributing evenly*/
13.     for(i=0;i < Constraint.Num;i++)
            /*cost is the standard variation of Scan_Count*/
14.         Cost=getcost(Constraint, i, Scan_Count);
15.         if(Cost < MinCost){
16.             MinCost = Cost;
17.             MPPI = i;}}
18.     Assign Constraint.PI[MPPI] as X;
19.     update_constraint(Violated_Constraints);
20.     update_effect_ocone(Scan_Count);}
```

Figure 3.7: Algorithm for Constraint-Aware X-Assignment.

### Constraint-Aware X-Assignment

After the above input vector generation process, we have the fully-specified decompressed patterns and they may contain some illegal states since we only activate a few relevant functional constraints in our constrained ATPG tool.

Suppose we have a test pattern $P$ containing an illegal state ($p_1 = 0$, $p_3 = 0$), as discussed earlier, we can break it by assigning either $p_1 = X$ or $p_3 = X$[1] and mask the error effects in our X-tolerant compactor. The two choices, however, may lead to multiple $X$-bits in the test response, denoted as *X-response* hereafter. Recall that the X-compactor used in our design is able to tolerate only one $X$-bit in each scan slice, a bad choice may lead to significantly long runtime as we need to backtrack to try other choices.

While we can obtain the detailed information for *X-response* of each choice by conducting simulation, it is not wise to try out for all the options due to the associated high computational complexity, especially considering the decompressed pattern may violate many functional constraints. Therefore, we propose a novel heuristic to guide our X-assignment process, based on the impact of the *PPIs* that are involved in violated functional constraints and structural analysis for the circuit, as shown in Fig. 3.7.

Since $X$ value on a *PPI* can only be propagated to its fanout cone, we use the distribution of fanout cone to evaluate the impact of a *PPI*. For a circuit under test containing $n$ *PPI*s (i.e., scan cells) and $m$ scan slices, for each $PPI[i]$ that is involved in an illegal state, we count, in advance, how many *PPO*s in its fan-out cone are located on each scan slice, and this information is stored in a 2-dimension matrix $PP\_Count[n \times m]$. The entry $PP\_Count[i, j]$ record the number of *PPO*s in the fan-out cone of $PPI[i]$ that are located in the $j^{th}$ scan slice. Vector *Scan_Count* is the summation of all the row $PP\_Count[i]$ where the corresponding $PPI[i]$ has been assigned as an unknown value (i.e., $X$). *Violated_Constraints* denotes all violated constraints.

In the beginning of our algorithm, function *initialization*() is applied to construct the matrix *PP_Count* and to reset the vector *Scan_Count*. Afterwards, we

[1]Note, $p_1$ or $p_2$ is treated as an unknown value $X$, their values are still $p_1 = 0$ and $p_3 = 0$ in the actual applied pattern.

apply $X$-assignment intelligently via two loops, which tackle this problem from two different angles.

In our X-compactor, the number of $X$-response bits within each scan slice cannot exceed 1. Based on the observation that less $X$-bits in *PPI*s induce less $X$-response bits, an intuitive method is to use a few $X$-bits to break as many violated functional constraints as possible, which motivates the procedure conducted in the first **while** loop (lines 2-9). Function getmaxviolation() returns the *index* of the *PPI* that is involved with the largest number of violated functional constrains. This procedure returns -1 if no *PPI* is involved in more than one violated constraints. $X$ is always assigned to those *PPI*s that gives us the most benefit referring to maximum violated constraints reduction, and then we update $Violated\_Constraints$ and $Scan\_Count$. Finally, we jump out this loop when there is no overlapping of *PPI*s between violated functional constraints or all of them have been broken.

The idea behind the second **while** loop (lines 10-20) is that we wish the potential $X$-response position to be evenly distributed on different scan slices. Since the $Scan\_Count$ records the distribution of $X$-response, we use the standard variation of $Scan\_Count$ as cost function. $Constraint$ represents the functional constraint that is currently targeted, which has two members: $Constraint.Num$ is the number of *PPI*s in its illegal cube, and $Constraint.PPI$ is an array pointed to its corresponding *PPI*s. Variable $MPPI$ is the index pointed to the most beneficial *PPI*.

Focusing on one violated $Constraint$ in each iteration, the method tries to find pseudo-primary input $Constraint.PPI[i]$ within its illegal state cubes with minimal cost and assign it as $X$. Function *getcost* adds the row corresponding to $Constraint.PPI[i]$ in matrix $PP\_Count$ with $Scan\_Count$, and returns the standard variation. After assigning $X$ to $Constraint.PPI[MPPI]$, we update the values of $Violated\_Constraints$ and $Scan\_Count$. The procedure for X-Assignment terminates when either all the functional constraints have been broken or we find out

| Benchmark | Conventional Structural ATPG | | | Pseudo-Functional ATPG | | | $FC_A - FC_B$ |
|---|---|---|---|---|---|---|---|
| | $FC_A$ (%) | *ave_spe* (%) | CPU time (s) | $FC_B$ (%) | *ave_spe* (%) | CPU time (s) | (%) |
| s382 | 82.105 | 30.037 | 0.033 | 78.663 | 48.186 | 0.017 | **3.442** |
| s1238 | 95.957 | 22.609 | 0.233 | 90.352 | 56.522 | 0.173 | **5.605** |
| s5378 | 87.915 | 8.032 | 8.467 | 83.413 | 44.578 | 12.917 | **4.502** |
| s9234 | 88.806 | 8.834 | 69.717 | 83.396 | 26.148 | 33.517 | **5.410** |
| s13207 | 89.395 | 2.493 | 23.333 | 86.368 | 33.228 | 131.983 | **3.027** |
| s15850 | 86.099 | 3.634 | 65.900 | 84.524 | 17.878 | 77.083 | **1.575** |
| s38417 | 98.463 | 2.069 | 194.017 | 95.695 | 12.648 | 545.32 | **2.768** |
| s38584 | 93.961 | 1.997 | 226.283 | 89.133 | 39.86 | 705.62 | **4.828** |

Table 3.1: Conventional Structural ATPG vs. Pseudo-Functional ATPG for Transition Faults.

that some constraints cannot be broken and then we need to abandon this test pattern.

It should be noted that, as our X-compactor can tolerate one X-bit in a scan slice provided that no greater than two bits are used to detect faults, if during fault simulation, a particular fault results in more than two error bits for a scan slice with an X-bit, this fault is deemed to be undetected with this pattern.

## 3.5 Experimental Results

### 3.5.1 Experimental Setup

We implement our proposed compression-aware pseudo-functional pattern generation framework targeting transition faults on top of an academic ATPG tool *Atalanta* [13], which originally targets stuck-at faults using the FAN algorithm [9]. We extract the functional constraints with the tool proposed in chapter 2. Experiments are conducted on various ISCAS'89 benchmark circuits, and the linear decompressors that we use in our experiments consist of an 8-bit ring generator

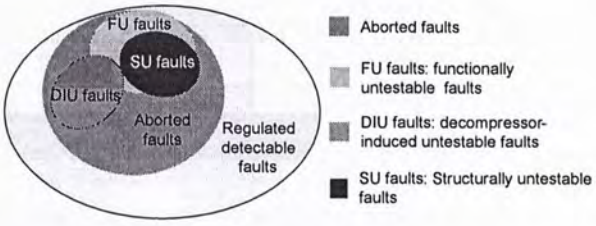


Figure 3.8: Regulated Detectable Faults

(with either 2-input vector or 4-input vector) and an 8-to-20 phase shifter.

Table 1 presents the test pattern generation results for transition faults using conventional structural ATPG and pseudo-functional ATPG, in which $FC$ denotes fault coverage while $ave\_spe$ represents the average percentage of specified bits in test patterns. It can be easily observed that, by taking the functional constraints into consideration, the fault coverage with only pseudo-functional patterns is lower than that of conventional structural ATPG, mainly due to the structurally testable while functionally untestable faults existing in the circuit. At the same time, we can observe that the number of specified bits in pseudo-functional patterns is much larger than that of conventional structural ATPG.

Since there are many reasons for ATPG tool to abort detecting a fault, before demonstrating our experimental results, we first define a concept namely *regulated detectable faults* to make fair comparison among different approaches. As can be seen in Fig. 3.8, we have structurally-untestable faults (SU faults), functionally-untestable faults (FU faults), and decompressor-induced untestable faults (DIU faults), and also some hard-to-detect faults that are aborted. Apparently, for compressible patterns targeting functionally-testable faults with a given ATPG tool, the faults we are after are only the rest of the faults, defined as *regulated detectable faults*.

To generate test patterns for the above regulated faults for fair comparison, we

| Benchmark | Regulated | | | Decompressor with All Cons | | | | | Our Proposed | | | Comparison | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $FC_R$ (%) | # *Pattern* | Runtime (s) | $FC_{RS}$ (%) | # *Random Pattern* | $FC_{TO}$ (%) | # *Total Pattern* | Runtime (s) | $FC_O$ (%) | # *Pattern* | Runtime (s) | $FC_O - FC_R$ | $FC_O - FC_{TO}$ | *Pattern Increase(%)* |
| s382 | 74.869 | 77 | 0.1 | 43.512 | 42 | 55.759 | 56 | 0.133 | 73.298 | 65 | 0.15 | **-1.571** | **17.539** | **-15.58** |
| s1238 | 77.872 | 332 | 1.5 | 50.387 | 208 | 53.617 | 219 | 3.000 | 76.277 | 340 | 1.583 | **-1.595** | **22.66** | **2.41** |
| s5378 | 79.586 | 614 | 7.883 | 49.756 | 257 | 49.756 | 257 | 144.217 | 79.103 | 656 | 29.783 | **-0.483** | **29.347** | **6.84** |
| s9234 | 74.811 | 931 | 55.383 | 43.687 | 437 | 43.687 | 437 | 159.45 | 73.447 | 981 | 85.883 | **-1.364** | **29.76** | **5.37** |
| s13207 | 84.011 | 968 | 83.205 | 42.953 | 365 | 42.953 | 365 | 1314.383 | 83.036 | 1192 | 353.95 | **-0.975** | **40.083** | **23.14** |
| s15850 | 82.259 | 1184 | 64.9 | 62.086 | 562 | 62.086 | 562 | 511.817 | 81.493 | 1174 | 90.3 | **-0.766** | **19.407** | **-0.84** |
| s38417 | 95.571 | 3102 | 218.883 | 74.698 | 985 | 75.89 | 996 | 3260 | 95.37 | 3057 | 1925.583 | **-0.201** | **19.48** | **-1.45** |
| s38584 | 88.142 | 2254 | 382.157 | 71.608 | 931 | 72.873 | 941 | 5338.233 | 87.525 | 2462 | 2851.817 | **-0.617** | **14.652** | 9.23 |
| Average | | | | | | | | | | | | **-0.946** | **24.116** | **3.064** |

Table 3.2: Results with 2-Input Decompressor.

first conduct pseudo-functional APTG and take its detectable fault list as the input to a compression-aware ATPG to obtain the regulated detectable fault list and their corresponding test patterns. It should be noted the runtime reported for the above *regulated ATPG* process is only the compression-aware ATPG runtime.

### 3.5.2 Results and Discussion

Table 3.2 and Table 3.3 present detailed comparison for the test pattern generation results from three approaches: regulated ATPG, ATPG in test compression environment with all functional constraints enabled, and our proposed compression-aware pseudo-functional ATPG, with 2-input linear decompressor and 4-input linear decompressor, respectively.

From these tables, we can observe that, if all the functional constraints are enabled in test compression environment, the fault coverage ($FC_{TO}$ in Column 7) suffers from great loss when compared to regulated ATPG ($FC_R$ in Column 2). As a matter of fact, it can be observed that most of the detected faults are covered by our multi-launch pseudo-functional random patterns (see $FC_{RS}$ in Column 5). For example, for s9234 with 4-input decompressor, only 2 deterministic patterns are generated to cover extra faults. This is expected, since when all the functional con-

| Benchmark | Regulated | | | Decompressor with All Cons | | | | | Our Proposed | | | Comparison | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $FC_R$ (%) | # *Pattern* | Runtime (s) | $FC_{RS}$ (%) | # *Random Pattern* | $FC_{TO}$ (%) | # *total Pattern* | Runtime (s) | $FC_O$ (%) | # *Pattern* | Runtime (s) | $FC_O - FC_R$ | $FC_O - FC_{TO}$ | *Pattern Increase*(%) |
| s382 | 75.916 | 67 | 0.167 | 38.743 | 26 | 59.948 | 46 | 0.167 | 74.822 | 76 | 0.083 | **-1.094** | **14.874** | **13.43** |
| s1238 | 85.851 | 394 | 1.533 | 45.053 | 221 | 55.372 | 247 | 2.783 | 85.372 | 392 | 0.733 | **-0.479** | **30.00** | **-0.51** |
| s5378 | 83.92 | 624 | 7.633 | 54.185 | 299 | 54.185 | 299 | 95.183 | 83.44 | 643 | 25.883 | **-0.48** | **29.255** | **3.04** |
| s9234 | 81.881 | 1134 | 71.983 | 42.412 | 566 | 43.588 | 568 | 184.633 | 81.692 | 1165 | 62.75 | **-0.189** | **38.104** | **2.73** |
| s13207 | 85.842 | 1302 | 92.686 | 46.314 | 415 | 48.526 | 418 | 1172 | 84.81 | 1268 | 494.317 | **-1.032** | **36.284** | **-2.61** |
| s15850 | 85.132 | 1252 | 49.683 | 60.283 | 585 | 62.311 | 591 | 466.4 | 84.908 | 1248 | 97.4 | **-0.224** | **22.597** | **-0.32** |
| s38417 | 95.623 | 3006 | 460.633 | 73.265 | 1220 | 76.289 | 1246 | 4596.433 | 95.399 | 3125 | 1911.383 | **-0.224** | **19.11** | **3.95** |
| s38584 | 89.002 | 2158 | 769.158 | 69.744 | 1061 | 73.329 | 1085 | 5412.567 | 88.939 | 2278 | 2797.917 | **-0.063** | **15.61** | **5.56** |
| Average | | | | | | | | | | | | **-0.473** | **25.729** | **3.16** |

Table 3.3: Results with 4-Input Decompressor.

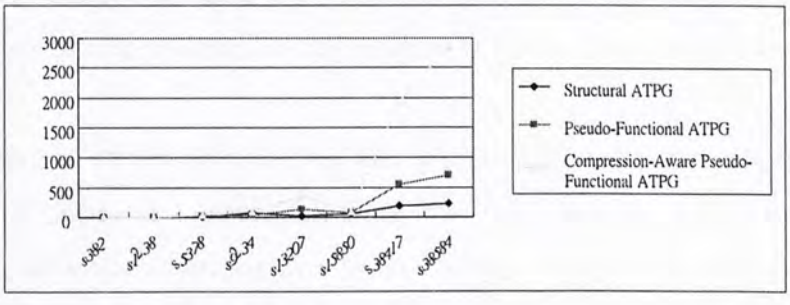


Figure 3.9: Runtime Comparison.

straints are activated, the patterns generated from the ATPG engine contains a large number of specified bits and it is very unlikely that such patterns are compressible. Consequently, in most cases, the ATPG engine backtracks multiple times to try different patterns and eventually aborts to detect the targeted faults. This also explains why the runtime for this kind of ATPG is the longest (Column 9), even though it only generate limited or even zero deterministic patterns.

With our proposed method that only activates the relevant functional constraints for the targeted fault, the fault coverage loss when compared to the regulated ATPG case is quite small, with at most 1.6% and on average 0.946% (Column 14) for the case with 2-input decompressor. The fault coverage loss with the 4-input decompressor case are even smaller, on average 0.473%. When compared

to the ATPG method that activates all the functional constraints, the fault coverage increases by around 25% for both decompressor cases (Column 15). As can be seen from the last column in both tables, our proposed compression-aware APTG has a slightly higher test pattern count when compared to the regulated ATPG case (around 3% on average). This is because, for a particular pattern that violate certain functional constraints, since we need to mask the error effects from those faults that are detectable due to the existence of illegal states in the pattern, these faults are deemed as untestable with this pattern, requiring other patterns to cover it. There are also some cases that our ATPG tool leads to fewer test patterns, and we attribute this phenomenon to the uncertainty for the faults covered by random patterns.

In addition, we compare runtime of our proposed method with traditional structural ATPG and normal pseudo-functional ATPG. It is shown in fig.3.9, runtime of our proposed method increases faster as the scaling of circuit size, because it needs to spend more time on $X$-assignment for those patterns containing illegal states.

## 3.6 Conclusion

With technology scaling, the discrepancy between integrated circuits' activities in normal functional mode and that in structural test mode has an increasing adverse impact on the effectiveness of manufacturing test. Pseudo-functional testing has been proposed to resolve this issue, but the generated patterns typically feature much less *X-bits* when compared to conventional structural patterns. Directly applying such patterns in test compression environment hence may lead to significant fault coverage loss. In this chapter, we propose novel compression-aware pseudo-functional testing techniques to tackle the above problem. Experimental results on ISCAS'89 benchmark circuits show that our proposed solution is able to achieve similar fault coverage as conventional structural patterns, without incurring over-

testing to the circuits.

---

□ **End of chapter.**

# Chapter 4

# Conclusion and Future Work

The discrepancy between ICs' activities in normal functional mode and that in structural test mode has an increasingly adverse impact on the effectiveness of manufacturing test with technology advancement, as reported in several industrial studies (e.g., [5, 29, 30]).

Pseudo-functional testing has great potential to resolve the above problem, but whether we could realize this potential highly relies on whether we could effectively identify as many illegal states as possible. In this dissertation, we show that the main structural root cause for illegal states is the multi-fanout nets in the circuit. Based on this, we develop efficient and effective illegal state identification algorithms. Experimental results on ISCAS'89 benchmark circuits demonstrate that the proposed technique is much more effective when compared to state-of-the-art solutions.

As pseudo-functional test patterns typically feature much less *X-bits* when compared to conventional structural patterns, directly applying such patterns in test compression environment may lead to significant fault coverage loss. In this dissertation, we propose novel compression-aware pseudo-functional testing techniques to tackle the above problem. Experimental results on ISCAS'89 benchmark

circuits show that our proposed solution is able to achieve similar fault coverage as conventional structural patterns, without incurring over-testing to the circuits.

There are several important topics yet to explore for future work. Firstly, while this thesis work facilitate to avoid over-testing by generating patterns that are more function-like, under-testing may happen since the current solution cannot guarantee to sensitize the worst case delay for the circuit in at-speed delay testing. The fundamental problem in delay testing is, *How can we exercise the worst-case timing of the circuits under test in their functional mode during manufacturing test?* We plan to take our extracted functional constraints into ATPG and try to generate patterns that activate the maximum electrical noises on targeted delay faults in our future work. Secondly, effective removal of false paths from static timing analysis (STA) is a critical task, because STA is used in the inner loop of many circuit optimization tools to resolve timing issues and the effectiveness of such optimization processes is deteriorated with the presence of false paths, leading to sub-optimal solution or even failure to achieve timing closure. Existing techniques, however, regard a path as a true path as long as a vector pair can be found to sensitize it. This is rather pessimistic since such a path might be activated only with illegal states in the circuit and hence it is actually functionally-unsensitizable. Hence, with the help of this thesis work, it is possible to identify more false paths in the circuit, thus dramatically improving the efficiency of STA tools.

---

□ **End of chapter.**

# Bibliography


[1] M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, 1990.

[2] C. Barnhart et al. Extending OPMISR Beyond 10x Scan Test Efficiency. *IEEE Design & Test of Computers*, 19(5):65–73, 2002.

[3] D. Brand and V. S. Iyengar. Identification of Redundant Delay Faults. *IEEE Transactions on Computer-Aided Design*, 13(5):553–565, May 1994.

[4] M. Bushnell and V. Agrawal. *Essentials of Electronic Testing*. Kluwer Academic Publishers, 2000.

[5] C. Shi and R. Kapur. How Power Aware Test Improves Reliability and Yield. EE Times, Sept. 15, 2004.

[6] G. Chen, S. M. Reddy, and I. Pomeranz. Procedures for Identifying Untestable and Redundant Transition Faults in Synchronous Sequential Circuits. In *Proceedings International Conference on Computer Design (ICCD)*, pages 36–41, 2003.

[7] K.-T. Cheng and H.-C. Chen. Classification and Identification of Nonrobust Untestable Path Delay Faults. *IEEE Transactions on Computer-Aided Design*, 15(8):845–853, August 1996.

[8] V. Chickermane, B. Foutz, and B. Keller. Channel Masking Synthesis for Efficient On-Chip Test Compression. In *Proceedings IEEE International Test Conference (ITC)*, pages 452–461, 2004.

[9] H. Fujiwara and T. Shimono. On the Accelaration of Test Generation Algorithms. C-32(12):1137–1144, Dec. 1983.

[10] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. C-30(3):215–222, March 1981.

[11] K. Heragu, J. H. Patel, and V. D. Agrawal. Fast Identification of Untestable Delay Faults Using Implications. In *Proceedings International Conference on Computer-Aided Design (ICCAD)*, pages 642–647, 1997.

[12] M. Konijnenburg, J. van der Linden, and A. van de Goor. Illegal State Space Identification for Sequential Circuit Test Generation. In *Proceedings IEEE/ACM Design, Automation, and Test in Europe (DATE)*, pages 741–746, 1999.

[13] H. K. Lee and D. S. Ha. On the Generation of Test Patterns for Combinational Circuits. Technical Report 12-93, Dept. of Electrical Eng., Virginia Polytechnic Institute and State University, 1993.

[14] H.-C. Liang, C. L. Lee, and J. E. Chen. Identifying Invalid States for Sequential Circuit Test Generation. *IEEE Transactions on Computer-Aided Design*, 16(9):1025–1033, Sept. 1997.

[15] Y. C. Lin and K. T. Cheng. A Unified Approach to Test Generation and Test Data Volume Reduction. In *Proceedings IEEE International Test Conference (ITC)*, pages 1–10, 2006.

[16] Y.-C. Lin, F. Lu, and K. Cheng. Pseudofunctional Testing. *IEEE Transactions on Computer-Aided Design*, 25(8):1535–1546, August 2006.

[17] Y. C. Lin, F. Lu, K. Yang, and K. T. Cheng. Constraints Extraction for Pseudo-Functional Scan-based Delay Testing. In *Proceedings IEEE Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 166–171, 2005.

[18] H. Liu, H. Li, Y. Hu, and X. Li. A Scan-Based Delay Test Method for Reduction of Overtesting. In *Proceedings International Symposium on Electronic Design, Test and Applications (DELTA)*, pages 521–526, 2008.

[19] X. Liu and M. S. Hsiao. A Novel Transition Fault ATPG that Reduces Yield Loss. *IEEE Design & Test of Computers*, 22(6):576–584, Nov.-Dec. 2005.

[20] F. Lu, M. K. Iyer, G. Parthasarathy, and K. T. Cheng. An Efficient Sequential SAT Solver With Improved Search Strategies. In *Proceedings IEEE/ACM Design, Automation, and Test in Europe (DATE)*, pages 1102–1107, 2005.

[21] P. Maxwell, I. Hartanto, and L. Bentz. Comparing Functional and Structural Tests. In *Proceedings IEEE International Test Conference (ITC)*, pages 400–407, 2000.

[22] S. Mitra and K. S. Kim. X-Compact: An Efficient Response Compaction Technique. *IEEE Transactions on Computer-Aided Design*, 23(3):421–432, March 2004.

[23] J. H. Patel, S. S. Lumetta, and S. M. Reddy. Application of Saluja-Karpovsky Compactors to Test Responses with Many Unknowns. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 107–112, 2003.

[24] I. Polian and H. Fujiwara. Functional Constraints vs. Test Compression in Scan-Based Delay Testing. In *Proceedings IEEE/ACM Design, Automation, and Test in Europe (DATE)*, pages 1039–1044, 2006.

[25] J. Rajski, J. Tyszer, M. Kassab, and N. Mukherjee. Embedded Deterministic Test. *IEEE Transactions on Computer-Aided Design*, 23(5):776–792, May 2004.

[26] J. Rajski, J. Tyszer, C. Wang, and S. M. Reddy. Finite Memory Test Response Compactors for Embedded Test Applications. *IEEE Transactions on Computer-Aided Design*, 24(4):622–634, April 2005.

[27] J. Rearick. Too Much Delay Fault Coverage Is A Bad Thing. In *Proceedings IEEE International Test Conference (ITC)*, pages 624–633, Nov. 2001.

[28] J. Roth. Diagnosis of Automata Failures: A calculus and a Method. *IBM Journal of Research and Development*, 10(4):278–291, July 1967.

[29] J. Saxena, K. Butler, V. Jayaram, and S. Kundu. A Case Study of IR-Drop in Structured At-Speed Testing. In *Proceedings IEEE International Test Conference (ITC)*, 2003.

[30] S. Sde-Paz and E. Salomon. Frequency and Power Correlation between At-Speed Scan and Functional Tests. In *Proceedings IEEE International Test Conference (ITC)*, paper 13.3, 2005.

[31] M. Syal, K. Chandrasekar, V. Vimjam, M. S. Hsiao, Y.-S. Chang, and S. Chakravarty. A Study of Implication Based Pseudo Functional Testing. In *Proceedings IEEE International Test Conference (ITC)*, page paper 24.3, 2006.

[32] M. Syal and M. S. Hsiao. New Techniques for Untestable Fault Identification in Sequential Circuits. *IEEE Transactions on Computer-Aided Design*, 25(6):1117–1131, 2006.

[33] N. Touba. X-canceling MISR - An X-Tolerant Methodology for Compacting Output Responses with Unknowns using a MISR. In *Proceedings IEEE International Test Conference (ITC)*, paper 6.2, 2007.

[34] N. A. Touba. Survey of Test Vector Compression Techniques. *IEEE Design & Test of Computers*, 23(4):294–303, April 2006.

[35] L.-T. Wang, X. Wen, S. Wu, Z. Wang, Z. Jiang, B. Sheu, and X. Gu. VirtualScan: Test Compression Technology Using Combinational Logic and One-Pass ATPG. *IEEE Design & Test of Computers*, 25(2):122–130, March-April 2008.

[36] P. Wohl, J. A. Waicukauski, S. Patel, and M. B. Amin. X-Tolerant Compression and Application of Scan-ATPG Patterns in a BIST Architecture. In *Proceedings IEEE International Test Conference (ITC)*, pages 727–736, Oct. 2003.

[37] P. Wohl, J. A. Waicukauski, S. Patel, F. DaSilva, T. W. Williams, and R. Kapur. Efficient Compression of Deterministic Patterns into Multiple PRPG Seeds. In *Proceedings IEEE International Test Conference (ITC)*, page paper 36.1, Oct. 2005.

[38] W. Wu and M. S. Hsiao. Mining Sequential Constraints for Pseudo-Functional Testing. In *Proceedings IEEE Asian Test Symposium (ATS)*, pages 19–24, 2007.

[39] Z. Zhang, S. Reddy, and I. Pomeranz. On Generate Pseudo-Functional Delay Fault Tests for Scan Designs. In *Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 215–226, 2005.