Compression-Aware Pseudo-Functional Testing

Feng Yuan and Qiang Xu
CUhk REliable computing laboratory (CURE)
Department of Computer Science & Engineering
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
Email: {fyuan,qxu}@cse.cuhk.edu.hk

Abstract

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. 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. Pseudo-functional patterns, however, feature much less don't-care bits when compared to conventional structural patterns, making them less friendly to test compression techniques. In this paper, we propose novel solutions to address the above problem, which facilitate to apply pseudo-functional testing in linear decompressor-based test compression environment. Experimental results on ISCAS'89 benchmark circuits demonstrate the effectiveness of the proposed methodology.

1 Introduction

Scan is the most popular technique used to increase the testability of integrated circuits (ICs), which makes automatic test pattern generation (ATPG) viable for large ICs. In scan-based designs, however, the circuit states in test mode may do not exist in functional mode. For example, consider a finite state machine (FSM) encoded with one-hot code, the legal combinations of values in the circuit's storage elements are only those with a single logic '1' and all the others logic '0'. In scan test mode, however, it is possible to have patterns that contain multiple logic '1's without considering such functional constraints during the ATPG process.

While such discrepancy does not have high impact on stuck-at tests, recent design evaluations have revealed that at-speed scan patterns can be up to 20% slower than any functional pattern [24]. As at-speed delay testing is essential to ensure the quality of IC products fabricated with latest technology, over-testing due to such discrepancy may significantly increases IC manufacturing cost. That is, some good ICs that would work in application might fail at-speed delay tests and are thrown away, leading to unnecessary *test yield loss* (also known as *test overkill*) [16]. With today's tight profit margins for semiconductor products, just a small variation in yield percentage can translate to millions of dollars of revenue change. Therefore, how to reduce test yield loss has become a serious concern for the industry [4, 21, 23].

Recently, pseudo-functional testing has been proposed to resolve the above discrepancy problem in manufacturing test [13]. In this technique, *functionally-unreachable states* (also known as *illegal states* or *functional constraints*) in the circuit are extracted and fed to the ATPG engine to generate *functional-like* patterns. As any test pattern that is scanned in conforms "closely" to a functionally reachable state, the chip is expected to operate close to the functional mode during test application, thus reducing the possibility of test yield loss.

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 don't-care bits (i.e., *X-bits*) in pseudo-functional patterns can be quite low. This has a severe impact on the effectiveness of test data compression (TDC) 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* design-for-testability (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.

In this paper, 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

978-1-4244-4867-8/09/$25.00 ©2009 IEEE

benchmark circuits show the effectiveness of our proposed compression-aware pseudo-functional testing technique.

The remainder of this paper is organized as follows. Section 2 reviews related work and motivates this paper. In Section 3 and Section 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 Section 5 to show the effectiveness of the proposed solution. Finally, Section 6 concludes this paper.

2 Preliminaries and Motivation

2.1 Pseudo-Functional Testing

When testing delay faults in scan-based designs, it is possible to activate *functionally infeasible paths* during test application [28]. If a chip fails a particular at-speed test that exercises such paths, this chip may be able to work in application but is considered as a bad chip.

One way to reduce such test overkills is to identify functionally-untestable but structurally-testable (FU-ST) delay faults in the circuit and do not target them during test generation [3, 5, 6, 9]. However, the complexity of FU-ST delay fault identification problem is exponential to the circuit's size. In addition, even if we are able to generate test patterns for those functionally-testable faults only, it is still possible that they incidentally detect some FU-ST faults and hence lead to test overkills [15]. From another perspective, several power-aware test generation methodologies were proposed to reduce switching activities in scan capture mode to ensure timing safety in delay testing to avoid test overkills [7, 11, 22, 30]. However, if we over-restrict test power, under-testing might occur as some speed-related defects may not exhibit themselves, leading to *test escapes* [1]. Therefore, the real question is: *How can we exercise the worst-case timing of the circuits in their functional mode during manufacturing test?*

Pseudo-functional testing was proposed to tackle the above problem and has attracted lots of attention recently [12, 13, 18, 25, 33, 34]. In this technique, instead of identifying FU-ST delay faults in the CUT, functionally-unreachable states in the circuit are extracted and fed to a constrained ATPG tool, which backtracks immediately when illegal states are reached during test generation to obtain pseudo-functional patterns [15].

Illegal state identification is one of the fundamental problems in pseudo-functional testing. Several approaches were proposed in the literature to tackle this problem, including SAT-based method [13], implication-based strategy [34], mining-based technique [33], and a recent justification-based method [35]. In this work, we use [35] to generate the set of illegal states since it is able to obtain a large amount of illegal states with limited computational time. In this method, the authors studied the structural root cause for illegal states and showed that they are mainly caused by multi-fanout nets in the circuit. That is, illegal states would imply logic violations at different branches of the same multi-fanout net, explicitly in the same time frame or implicitly across multiple time frames. Based on this observation, this work defined the

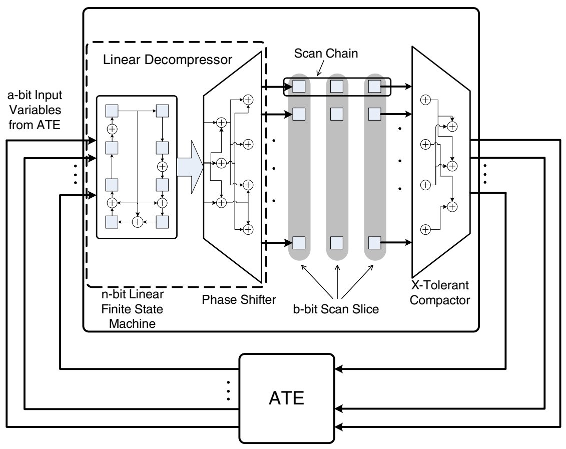

Figure 1. Linear Decompressor-Based Test Compression Infrastructure

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. Therefore, illegal states are identified by detecting combinations of justification schemes which can imply contradictory logic values on the same muli-fanout net explicitly or implicitly.

2.2 Test Data Compression

The rapidly-growing test data volume is a serious concern for the industry because it not only prolongs the ICs' testing time, but also raises memory depth requirements for the automatic test equipment (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 [27]) 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, 19, 29, 32]).

As shown in Fig. 1, 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

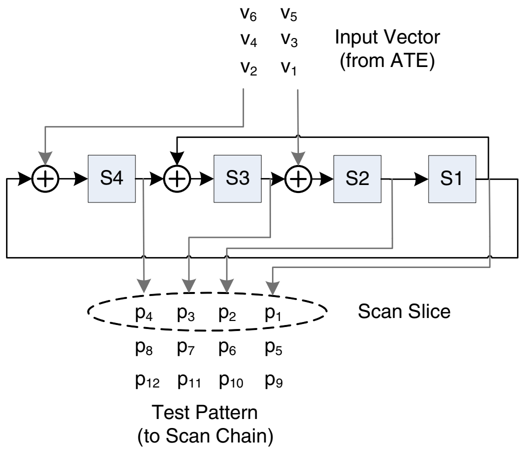

Figure 2. An Example Linear Decompressor

compactors were proposed [17, 20, 26, 31], 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 responses cannot be too high.

2.3 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. 2 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:

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

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:

$$\begin{pmatrix} 0&1&0&0&0&0&0&0&0&0\\ 1&0&0&1&0&0&0&0&0&0\\ 1&1&0&0&0&1&0&0&0&0\\ 1&1&0&0&0&1&0&0&1&0 \end{pmatrix} \times \begin{pmatrix} s_1\\ s_2\\ s_3\\ s_4\\ v_1\\ v_2\\ v_3\\ v_4\\ v_5\\ v_6 \end{pmatrix} = \begin{pmatrix} 1\\ 0\\ 1\\ 1 \end{pmatrix} \qquad (2)$$

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* $(M_s'|P_s')$:

$$\left(\begin{array}{cccccccccc|c} \mathbf{1}&\mathbf{0}&0&\mathbf{0}&0&1&0&0&\mathbf{0}&0&0\\ \mathbf{0}&\mathbf{1}&0&\mathbf{0}&0&0&0&0&\mathbf{0}&0&1\\ \mathbf{0}&\mathbf{0}&0&\mathbf{1}&0&1&0&0&\mathbf{0}&0&0\\ \mathbf{0}&\mathbf{0}&0&\mathbf{0}&0&0&0&0&\mathbf{1}&0&0 \end{array}\right) \qquad (3)$$

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, we use the X-compactor proposed in [17], which is essentially a combinational XOR network that can be also represented as a transformation matrix. In this work, the authors proved that X-compactor 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. Hence, we select to use $k_1 = 2$ and $k_2 = 1$ in our design.

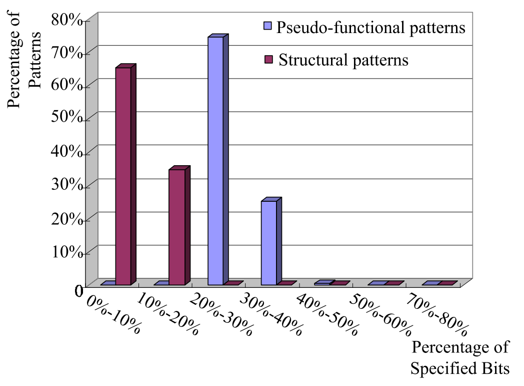

Figure 3. Specified Bits in Pseudo-Functional Patterns and Structural Patterns for s9234

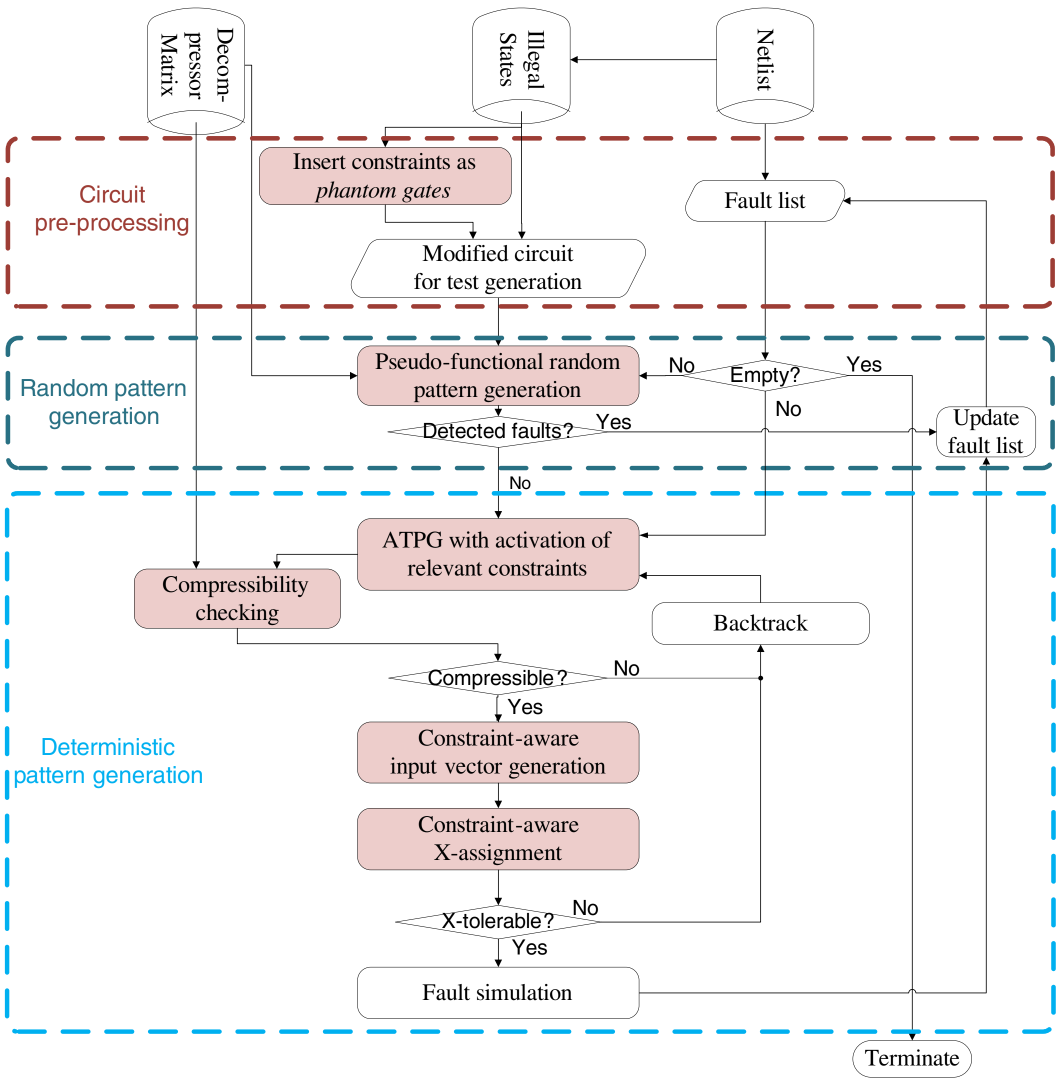

Figure 4. Pattern Generation Framework in Compression-Aware Pseudo-Functional Testing

2.4 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 $s9234$ as an example (see Fig. 3), after applying functional constraints extracted from [34], more than 70 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 [18] 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-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 paper, as detailed in the following section.

3 Proposed Methodology

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

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

That is, *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 less 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 [14]. 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. 4. 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 section.

4 Pattern Generation in Compression-Aware Pseudo-Functional Testing

As shown in Fig. 4, our proposed pattern generation framework takes the circuit netlist, the transformation matrix for the linear decompressor and the functional constraints (i.e.,

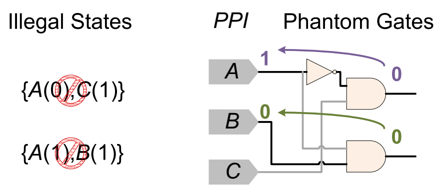

Figure 5. Insertion and Activation of Functional Constraints as Phantom Gates

illegal state cubes) in the circuit extracted using [34] 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.

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 [15], we insert *phantom logic AND gates* into the circuit to represent them, as shown in the example in Fig. 5. 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 *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 extremely important for our proposed methodology, as shown in Section 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 using [34]. 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.

```
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($i = 0; i \le Num\_Random\_Cycles; i$++) {
6.          Simulate $SP\{32\}$, output $OP\{32\}$;
7.          $SP\{32\} = OP\{32\}$;
8.      }
9.      $Num\_Valid\_Pattern$=Constraint_Check($OP\{32\}$);
10. }
11. $Num\_Detected\_Faults$=fault_simulation($OP\{32\}$);
12. if($Num\_Detected\_Faults$!=0){
13.     update_faultlist();
14.     goto line 1;
15. }
16. else
17.     terminate;
```

Figure 6. Algorithm for Pseudo-Functional Random Test Pattern Generation.

Based on this observation, we propose to generate pseudo-functional random patterns with the algorithm shown in Fig. 6.

In line 1, we initialize a variable $Num\_Valid\_Pattern$ to be zero. Then, in each iteration (lines 2-10), we generate 32 random input vectors $IP\{32\}$ 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 $OP\{32\}$ 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 sections.

4.3 Compressible Test Pattern Generation for Pseudo-Functional Testing

We implement a constrained ATPG engine based on *FAN* algorithm [8] for deterministic pattern generation. In such 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.

4.3.1 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.

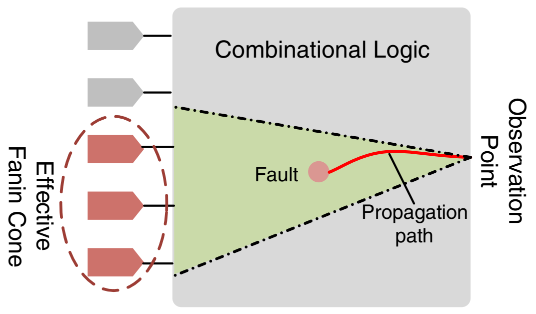

Figure 7. Effective Fan-in Cone for a Fault.

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 their corresponding phantom gates), 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. 7). 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.

4.3.2 Constraint-Aware Input Vector Generation

Whenever we generate a deterministic test pattern P, we need to check whether this pattern is compressible (see Section 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 section to achieve the 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. 3 as an example (see Section 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

```
1.  $V$=getinitialV();
2.  $Gain$=IFINITE;
3.  **while**($Gain == 0$) {
4.      $MGain = Gain = 0, Mbit = -1$;
5.      $P$=$M \times V$;
6.      $Num\_Vio$=violation_checking($P$);
        /* select a bit which reduce the most constraints*/
7.      **for**($i = 0; i \leq Num\_Vbit; i++$) {
            /*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.         $P = M \times V$;
12.         $Gain$=$Num\_Vio$-violation_checking($P$);
13.         flip($V[i]$);
14.         **if**($Gain > MGain$) {
15.             $MGain = Gain$;
16.             $Mbit = i$;
17.         }
18.         $Gain = MGain$;
19.     }
20.     **if**($Gain > 0$)
21.         flip($V[Mbit]$);
22. }
```

Figure 8. Algorithm for Constraint-Aware Input Vector Generation.

determined and they are equal to the scalar multiplication between P'_s and 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 $M'_s(3,4)$). 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. 8 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.

4.3.3 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. 9.

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 apply X-assignment intelligently via two loops, which tackle this problem from two different angles, considering the number of X-response bits within each scan slice cannot exceed 1 required by our X-tolerant compactor.

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 2-11). 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 result in the largest number of reduced violated constraints, 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 12-25) is

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

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	74.869	55.556	0.194	**7.236**
s1238	95.957	22.609	0.233	88.562	59.268	0.683	**7.395**
s5378	87.915	8.032	8.467	80.567	48.237	264.517	**7.348**
s9234	88.806	8.834	69.717	80.286	35.296	98.633	**8.52**
s13207	89.395	2.493	23.333	83.822	45.932	1827.683	**5.573**
s15850	86.099	3.634	65.9	82.276	22.674	331.55	**3.823**
s38417	98.463	2.069	194.017	92.759	22.104	10292.783	**5.704**
s38584	93.961	1.997	226.283	86.285	40.86	8759.26	**7.676**

Table 1. Conventional Structural ATPG vs. Pseudo-Functional ATPG for Transition Faults.

```
1.   initialization();
     /*break more constraints with less $X$-assignment*/
2.   while(TRUE)
3.       $index$ = getmaxviolation();
4.       if($index \neq$-1){
5.           Assign $PI[index]$ as $X$;
6.           update_constraint($Violated\_Constraints$);
7.           update_effect_ocone($Scan\_Count$);
8.       }
9.       else
10.          break;
11.  }
     /*for a particular constraint, select to assign $X$ to the $PPI$
     that makes potential $X$-response distributed more evenly*/
12.  while($Violated\_Constraints$ is not empty){
13.      $Constraint$ = Next($Violated\_Constraints$);
14.      $MinCost$=INFINITE, $Cost$=0, $MPPI$=-1;
         /*select a $PPI$ from illegal cube making
         potential $X$ response distributing evenly*/
15.      for($i$=0;$i < Constraint.Num$;$i$++)
             /*cost is the standard variation of $Scan\_Count$*/
16.          $Cost$=getcost($Constraint$, $i$, $Scan\_Count$);
17.          if($Cost < MinCost$){
18.              $MinCost = Cost$;
19.              $MPPI = i$;
20.          }
21.      }
22.      Assign $Constraint.PI[MPPI]$ as $X$;
23.      update_constraint($Violated\_Constraints$);
24.      update_effect_ocone($Scan\_Count$);
25.  }
```

Figure 9. Algorithm for Constraint-Aware X-Assignment.

that we wish the potential X-response position to be evenly distributed on different scan slices. Because an X value on a *PPI* can only be propagated to its fanout logic cone, we use the distribution of its driven scan cells to evaluate the impact of X-assignment for a *PPI*. Since the *Scan_Count* records the distribution of X-response, we use the standard variation of *Scan_Count* as our cost function when evaluating the impact of different X-assignment. *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 that some constraints cannot be broken and then we have to abandon this test pattern to avoid test overkills and backtrack to another solution.

It should be noted that, as our X-compactor can tolerate one X-bit in a scan slice provided that no greater than two error bits exist in the scan slice 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.

5 Experimental Results

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* [10], which originally targets stuck-at faults using FAN algorithm [8]. We have also implemented the implication-based illegal state identification technique presented in [34] to extract the functional constraints used in our framework. 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 (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 smaller 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 specified bits in pseudo-functional patterns is much larger than that of conventional structural ATPG.

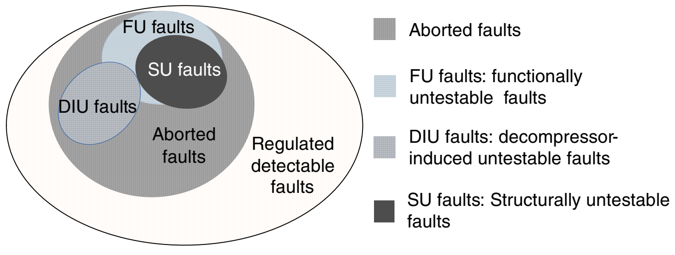
FU faults
SU faults
DIU faults
Aborted faults
Regulated detectable faults
Aborted faults
FU faults: functionally untestable faults
DIU faults: decompressor-induced untestable faults
SU faults: Structurally untestable faults

Figure 10. Regulated Detectable Faults

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	71.384	59	0.385	38.423	30	55.759	56	1.841	70.173	77	0.15	**-1.211**	**14.414**	**-30.51**
s1238	77.181	326	1.15	39.628	155	41.915	162	5.45	76.596	329	1.267	**-0.585**	**34.681**	**0.92**
s5378	79.611	715	3.717	45.507	195	45.507	195	120.183	78.725	660	31.267	**-0.886**	**33.218**	**-7.69**
s9234	71.995	902	46.717	36.667	413	36.944	415	203.95	70.003	898	83.85	**-1.992**	**33.059**	**-0.44**
s13207	83.222	1243	23.083	36.289	326	36.289	326	4654	80.972	1373	750.25	**-2.25**	**44.683**	**-10.46**
s15850	81.842	1164	25.75	54.481	563	56.097	570	1011.983	80.799	1188	54.183	**-1.043**	**24.702**	**2.06**
s38417	89.052	2784	222.1	60.697	1278	60.697	1278	11747	87.839	3118	5156.883	**-1.213**	**27.142**	**11.99**
s38584	82.728	2326	485.572	56.683	1008	61.351	1102	11154.338	81.988	2254	5983.125	**-0.74**	**20.637**	**-3.09**
Average												**-1.24**	**29.067**	**5.59**

Table 2. Results with 2-Input Decompressor.

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	72.446	65	0.235	43.224	43	56.325	62	0.792	69.354	63	0.152	**-3.092**	**13.029**	**-3.07**
s1238	82.606	381	1.133	48.298	203	50.851	231	4.617	82.447	387	0.2	**-0.159**	**31.596**	**-1.57**
s5378	79.996	645	7.017	49.569	192	49.569	192	108.833	79.33	619	43	**-0.666**	**29.761**	**-4.03**
s9234	78.447	1115	37.833	37.942	221	38.561	225	307.55	76.765	944	72.8	**-1.682**	**38.204**	**-15.34**
s13207	83.487	1214	23.1	36.564	399	38.623	403	4443.5	81.731	1393	547.717	**-1.756**	**43.108**	**14.74**
s15850	82.093	1117	48.1	52.532	433	55.325	442	1125.517	81.15	1240	77.583	**-0.943**	**25.825**	**11.01**
s38417	90.42	2898	361.76	58.829	810	58.829	810	14991.167	89.562	3188	7385.717	**-0.858**	**30.733**	**10.00**
s38584	83.215	2433	789.251	58.968	852	63.582	863	12389.73	82.797	2284	6553.581	**-0.418**	**19.215**	**-6.12**
Average												**-1.19**	**28.934**	**1.10**

Table 3. Results with 4-Input Decompressor.

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. 10, 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 first conduct pseudo-functional ATPG 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.

5.2 Results and Discussion

Table 2 and Table 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. Note that, the fault coverage difference for the regulated ATPG between the two tables are mainly due to the different number of inputs to the decompressor.

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 4 deterministic patterns are generated to cover extra faults. This is expected, since when all the functional constraints 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. 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 generates 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 2.25% and on average 1.24% (Column 13) for the case with 2-input decompressor. The fault coverage loss with the 4-input decompressor case are even smaller, on average 1.19%. When compared to the ATPG method that activates all the functional constraints, the fault coverage increases by around 29% for both decompressor cases (Column 14). As can be seen from the last column in both tables, our proposed compression-aware ATPG has slightly higher test pattern count when compared to the regulated ATPG case (around 5% and 1% on average respectively). 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, our proposed method has longer runtime than regulated ATPG, because it needs to spend more time on X-assignment for those patterns containing illegal states.

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 paper, 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.

7 Acknowledgements

This work was supported in part by the General Research Fund CUHK417406, CUHK417807, and CUHK418708 from Hong Kong SAR Research Grants Council, and in part by a grant N_CUHK417/08 from the NSFC/RGC Joint Research Scheme.

References

[1] Moderator: K. Butler, Organizer: N. Mukherjee. Power-Aware DFT - Do We Really Need it? Panel, International Test Conference, 2008.

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

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

[6] 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.

[7] D. Czysz, M. Kassab, X. Lin, G. Mrugalski, J. Rajski, and J. Tyszer. Low Power Scan Shift and Capture in the EDT Environment. In *Proc. IEEE International Test Conference (ITC)*, paper 13.2, 2008.

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

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

[10] 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.

[11] J. Li, Q. Xu, Y. Hu, and X. Li. iFill: An Impact-Oriented X-Filling Method for Shift- and Capture-Power Reduction in At-Speed Scan-Based Testing. In *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, pp. 1184–1189, 2008.

[12] Y.-C. Lin and K.-T. Cheng. A Unified Approach to Test Generation and Test Data Volume Reduction. In *Proc. IEEE International Test Conference (ITC)*, page paper 18.2, 2006.

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

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

[15] 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.

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

[17] 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.

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

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

[20] 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.

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

[22] S. Remersaro, X. Lin, S. M. Reddy, I. Pomeranz, and J. Rajski. Scan-Based Tests with Low Switching Activity. *IEEE Design & Test of Computers*, 24(3):268–275, May-June 2007.

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

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

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

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

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

[28] L.-T. Wang, C. E. Stroud, and N. A. Touba. *System-on-Chip Test Architectures: Nanometer Design for Testability*. Morgan Kaufmann Pub., 2007.

[29] 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.

[30] X. Wen, K. Miyase, T. Suzuki, S. Kajihara, Y. Ohsumi, and K. K. Saluja. Critical-Path-Aware X-Filling for Effective IR-Drop Reduction in At-Speed Scan Testing. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pp. 527–532, June 2007.

[31] 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 *Proc. IEEE International Test Conference (ITC)*, pp. 727–736, Oct. 2003.

[32] 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 *Proc. IEEE International Test Conference (ITC)*, page paper 36.1, Oct. 2005.

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

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

[35] F. Yuan and Q. Xu. On Systematic Illegal State Identification for Pseudo-Functional Testing. *to appear in Proc. ACM/IEEE Design Automation Conference (DAC)*, July, 2009.