INCREMENTAL TEST PATTERN GENERATION FOR STRUCTURALLY SIMILAR CIRCUITS

by

Jerin Joe

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2023

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Irith Pomeranz, Chair

School of Electrical and Computer Engineering

Dr. Cheng-Kok Koh

School of Electrical and Computer Engineering

Dr. Milind Kulkarni

School of Electrical and Computer Engineering

Dr. T. N. Vijaykumar

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

To my parents and to Jom, for their unconditional love and support.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Irith Pomeranz, for her unwavering support, patience, encouragement, and her willingness to share her enormous expertise. Her prompt suggestions, meticulous scrutiny, and scientific approach have helped me to a great extent. This work would not have been accomplished without her guidance.

I would also like to thank Dr. Janusz Rajski and Dr. Nilanjan Mukherjee, my advisors at Siemens Digital Industries Software, for providing funding and suggesting ideas for this thesis. Their exceptional patience, guidance, and encouragement have helped me during my studies and research.

I would like to thank Dr. Cheng-Kok Koh, Dr. Milind Kulkarni, and Dr. T.N. Vijaykumar for serving on my advisory committee.

I would like to extend my gratitude to Dr. Chen Wang, and Dr. Yingdi Liu at Siemens Digital Industries Software for going above and beyond to help me whenever I needed one.

I am deeply grateful to my friend, Dr. Binod Kumar, for his insightful comments and unwavering support throughout my research.

Lastly, I would like to mention my sincere gratitude to Jom. Without his continuous support and motivation, this Ph.D. journey would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The advancement of semiconductor technology has resulted in the development of devices that are fast, cost-effective, low-power, and high-performance. To achieve this, many gates are integrated into smaller areas, resulting in increased complexity of digital circuits. Increased size and complexity result in a large number of faults, which increases the time taken to test the circuit. However, as the size of the digital designs increases, they also exhibit structural similarities. This thesis describes a test generation process that utilizes structural similarity to speed up the test generation process. The property of structural similarity can be seen in circuits that are subjected to engineering change order (ECO), circuits that are modified during place and route, circuits subjected to retiming, circuits with multiple cores such as central processing units (CPUs), graphics processing units (GPUs), and artificial intelligence (AI) chips. The goal of the thesis is to determine the testability of a circuit (circuit2) given a test set for a structurally similar circuit (circuit1). This is achieved by transforming a test set generated for circuit1 into a test set for circuit2 without repeating the entire test generation process. The process described in the thesis starts with a structural analysis of circuit1 and circuit2 that captures their structural properties using an integer-arithmetic based computation called signatures. The signatures are used to obtain a partial mapping between the inputs and outputs of the two circuits. The mapping is used for transforming test patterns for circuit1 into test patterns for circuit2. The first chapter looks into similar circuits obtained after modifying the gate-level netlist. In the next chapter, structurally similar circuits were obtained by modifying the RTL, and the gate-level was resynthesized. This chapter proposed a mapping methodology to accommodate the changes introduced during the resynthesis of a netlist. Lastly, the thesis described a test generation methodology where transition faults are considered, which required two-cycle tests to be detected.

1. INTRODUCTION

Advancing growth in semiconductor technology has led to the development of fast, cost-effective, low-power, and high-performance devices [1], [2]. Complex circuits consist of a large number of gates that are integrated into small-sized chips, which increases the complexity of integrated circuits. A decrease in size and an increase in complexity results in an increased number of faults to be tested [2]. With complex and dense ICs, the time taken to test, debug and verify the circuit becomes the bottleneck for chip design. [3].

The physical design flow [4] is iterative to fix errors, improve performance and solve power issues. In order to meet the circuit specifications, designers make changes in the circuit [5], [6]. The required changes are performed incrementally so that they would have less impact on the existing design. Such changes are known as Engineering Change Orders (ECO). Testability bottlenecks are addressed by performing test generation early in the design flow. The generated test sets remain valid as long as the modifications do not alter the gate-level description of the circuit. Every time a modification changes the gate-level description, the test patterns for the original design become invalid, and new test patterns need to be generated for the modified design. This visibly increases the overall test generation time and time-to-market for the chip.

The solution explored in this thesis is to exploit structural similarity prevalent in the current designs to speed up the test generation process. The thesis provides a solution for incremental test generation for similar circuits obtained by modifying gate-level netlists and the RTL when stuck-at and transition faults are considered.

This chapter presents the basics of design for test (DFT), its techniques, a few of the fault models involved in the study and review of the previous works.

1.1 VLSI Design Flow

The VLSI chip density is expanding exponentially as transistor feature sizes continue to shrink. As a result, the current and future VLSI technology are extremely complicated. In

order to meet the current demand, billions of transistors are integrated on a single chip. In addition, every manufactured chip must be reliable and should be thoroughly tested. A robust VLSI design and test flow which results in the production of reliable chips is reviewed in this section.

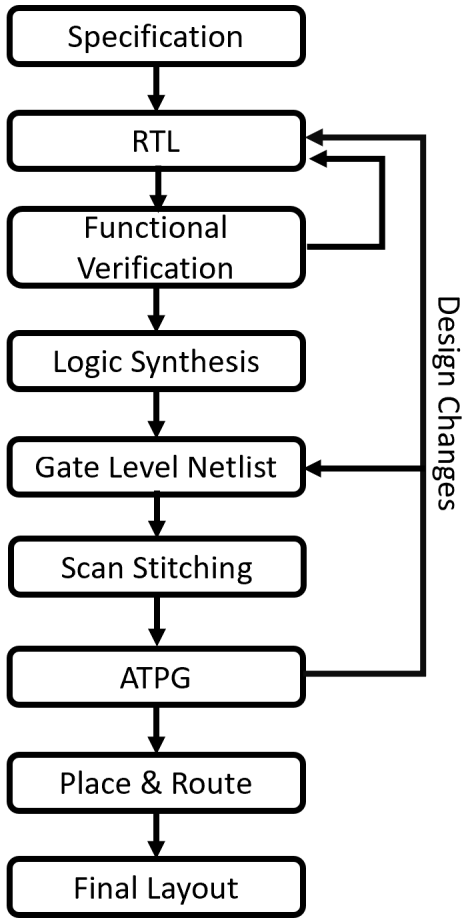


Figure 1.1. VLSI Design Flow [4]

Figure 1.1 shows a flow diagram for the VLSI design. The initial stage of the VLSI design flow is to defining the specifications of the circuit. The specifications define the necessary power, area, timing constraints that needs to be met for the design. the behavioral aspect of the circuit is defined using a hardware description language (HDL)[7], [8]. Every time, before moving on to the next level, a verification step is performed.

Functional verification [9] is performed before synthesizing the RTL description to ensure the design is done according to the specifications. After the verification, the RTL is converted

into a gate-level netlist using a synthesis tool. To ensure that the gate-level netlist meets all the circuit requirements, the design needs to be tested. Automatic Test Pattern Generation (ATPG) is performed on the circuit to determine the fault coverage achieved. Binary test patterns are generated and applied to the circuit under test (CUT) to excite the faults in the functional modules in the design[10]. This is shown in figure 1.2. The output is compared to the desired response of the circuit. If the response from the applied test pattern does not match the desired response, the test pattern has detected a fault. These test patterns are used to test the designs for any defects. The quality of the test patterns is determined by the number of faults detected during automatic test pattern generation (ATPG). If these patterns does not detect a reasonable number of faults, design needs to be incrementally modified in order to meet the circuit requirements.

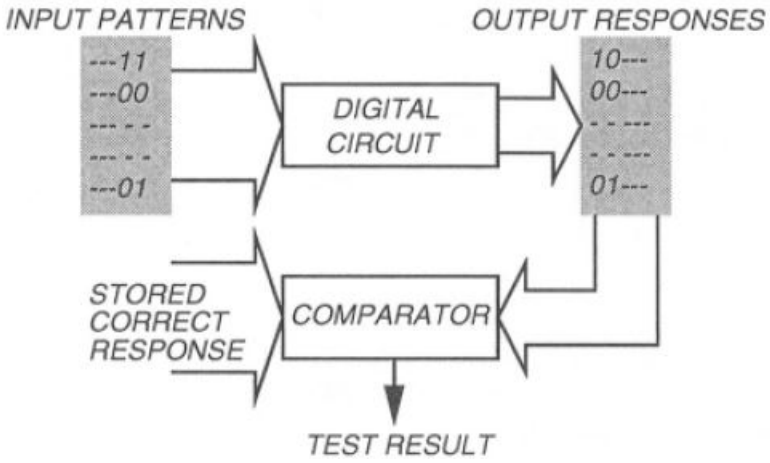


Figure 1.2. Basic Principle of Testing of Digital Circuits [10]

The final stage is the placement of the synthesized modules. The functional modules need to be placed in precise locations within a chip to get an optimized design. After the placement, the functional modules are connected using wires.

1.2 Design for Testability

A large, complex design consists of combinational and sequential logic. The circuit with sequential logic is difficult to test as test patterns need to be applied over multiple cycles to obtain the response at the output of the design. This increases the test volume and test

application time. With the increase in complexity of a design, digital circuits need to be designed by taking into account the time taken for testing. Design for testability (DFT) [11] techniques improve the testability of the circuit by adding more hardware in the CUT. Using the DFT techniques, the difficulty of testing sequential circuits is minimized. Some of the DFT techniques involved are scan design, Built-in-self-test (BIST), boundary scan.

1.2.1 Scan Design

Scan based [12] DFT technique is one of the widely used techniques. In this DFT technique, the flip flop in the design is modified by adding a test mode to the design, as shown in the figure. This makes the flip flop easier to test by making it controllable and observable.

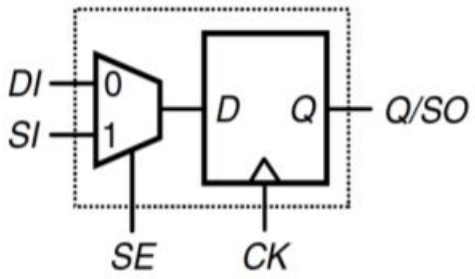


Figure 1.3. Scan Flip Flop [13]

The scan structure has an extra input known as scan-in (SI), which is fed to the flip flop through a two-input multiplexer. The multiplexer is controlled using a scan enable (SE) pin, which selects the data path (D input) or scan input path (SI input) as per the test mode.

One or more shift registers are formed by connecting the scan cells together, as shown in Figure 1.4. Each scan cell is made observable and controllable, thus gaining access to the internal modules in the circuit. The scan structure can be set to any desired value during the test mode by shifting the value in the shift registers. Combinational logic in the design can be tested by assigning values to the scan cells. The responses to the applied stimuli will be captured in the scan cells. By shifting out of the registers, the values in the scan structure can be observed.

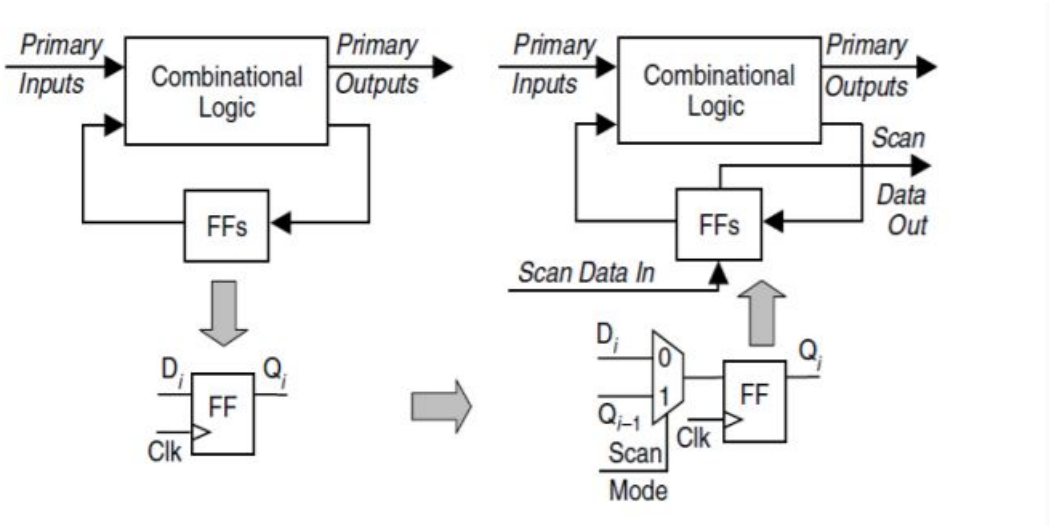


Figure 1.4. Scan Based Design [13]

1.2.2 Built-In-Self-Test (BIST)

BIST [14] was introduced to test the circuit by itself at the operating speed of the design. This DFT technique uses additional hardware such as an internal test pattern generator (TPG), an output response compacter, a comparator, and a ROM, as shown in Figure 1.5. A TPG is integrated within the design to generate the test patterns internally. Next, the output responses of the CUT are sent through a data compacter. Finally, the compacted value is compared with the reference value stored in the ROM to verify the correctness of the design.

1.2.3 Boundary Scan

A boundary scan [15] enhances the access to test the components that are embedded within the design. In this technique, a register with additional circuitry is placed at each of the input/output (I/O) pins. These cells at the periphery are serially connected to form a boundary-scan. The test stimuli can be serially shifted through the boundary scan during the test mode. The cells in the boundary scan can either force the data onto the CUT or

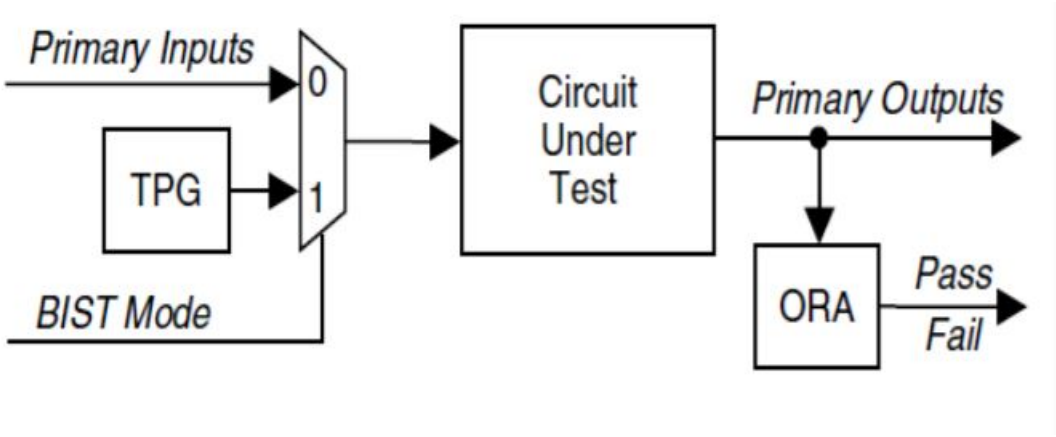


Figure 1.5. Built-In-Self-Test [13]

capture the response. With direct access to I/O pins, the interconnects in the circuit can also be tested separately, in addition to testing the logic of the circuit.

1.3 Fault Models

During the manufacturing, development, or operation of the design, a defect like short or open can occur within a design. With the increasing size and shrinking feature size of an integrated circuit, the number of physical defects can be enormous. A fault model is a way to depict the behavior of the defects in the design accurately. Some of the basic fault models are described in the following sub-section.

1.3.1 Stuck-At Fault

Stuck-At-fault [16] describes the faulty behavior of the signal line connecting the gates within the design. The line can be tied to logic 1, which is called stuck-at-1 (s-a-1), or tied to logic 0, which is called stuck-at-0 (s-a-0). Figure 1.6 shows an example of stuck-at-fault.

In Figure 1.6, the output line of gate G2 is stuck at logic value '0'. By applying the test vector '0011' on the primary inputs, it can be seen that the fault-free output of gate G2 should be 1, but the faulty circuit will produce a 0. This effect of the stuck-at fault at the output of gate G2 can be propagated to the primary output by setting the primary inputs

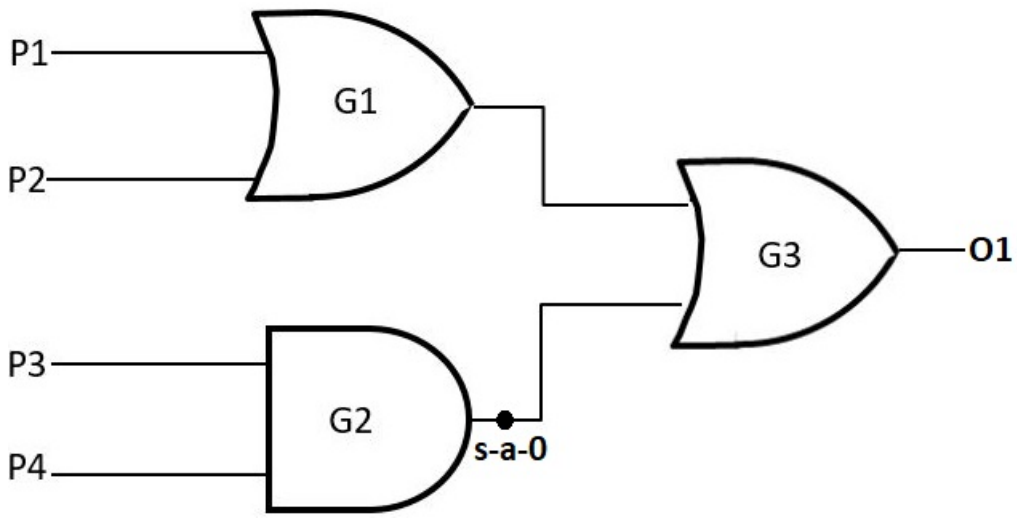


Figure 1.6. Stuck-At-Fault

P1 and P2 to 0. Thus, we get the test vector '0011', for which the output response of faulty and fault-free circuits differ at the primary output. Hence, the test vector '0011' detects the s-a-0 fault at the output of gate G2.

1.3.2 Transition Fault

The defects in the circuit can induce improper timing behaviors. These defects can cause the circuit to fail when operated at the normal speed of operation. Such defects are modeled by delay faults. The transition fault model [17], [18] is one of the most extensively used delay fault models. A transition fault increases the time taken for a signal on a line to change. Transition faults can be categorized into two: slow-to-rise and slow-to-fall faults. When a signal takes a long time to change from 0 to 1, we see a slow-to-rise fault. Similarly, a slow-to-fall fault can be observed when a signal takes a long time to transition from 1 to 0.

In order to detect the fault, we have to create a transition on the line. Two patterns are required to excite a transition fault and be observed at the output. The first pattern initializes the circuit to a stable state and the second pattern creates the necessary transition to detect the fault. For example, for a slow-to-fall(rise) fault, the line first needs to be set to

1 (0), which is done using the first pattern. Then using the second pattern, the signal on the line is transitioned to 0 (1). As the line is faulty, the signal will take more time to change to logic value 0 (1), and this effect can be propagated to the primary output or a scan cell. This is shown in the Figure. 1.7.

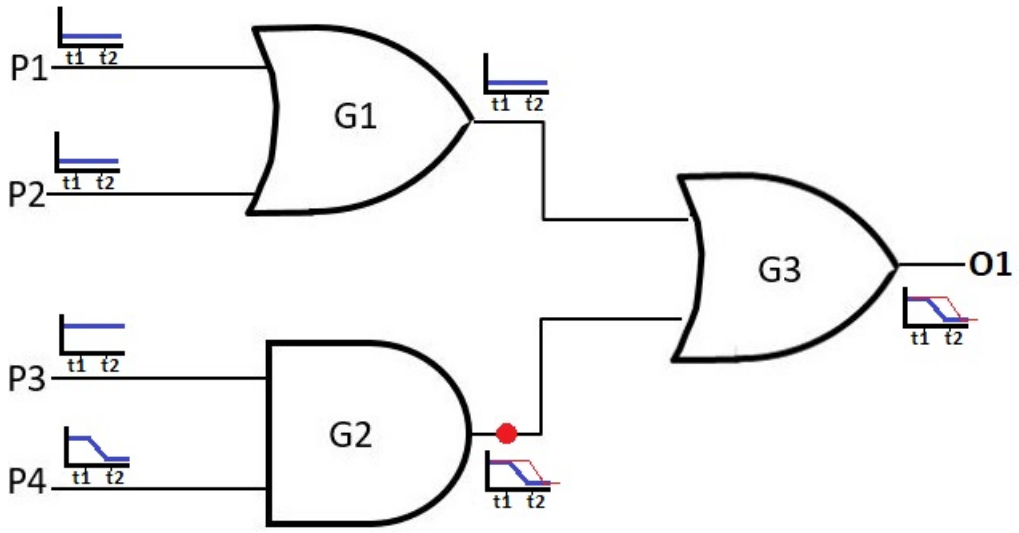


Figure 1.7. Slow-to-Fall

There is a slow-to-fall fault at the output line of gate G2. The first pattern is applied at time t1, which sets the output line of G2 to 1. Once the outputs are stable, the second pattern is applied at time t2, creating a transition that forces the output line of gate G2 to fall to logic 0 slower than the normal speed of operation. Due to the slow-to-fall fault, the output remains at logic 1 at time t2 instead of logic 0, and the fault gets detected.

1.3.3 Path Delay Fault

Path delay faults [19] occur when the cumulative delay of a combinational path exceeds the clock period due to the defects in the circuit. A combinational path consists of a primary input or a scan cell connected to a primary output or scan cell via few combinational gates. Similar to the transition faults, there are two types of faults associated with each path in the circuit: rising path delay and falling path delay. The number of path delay faults increases exponentially with the size of the circuit.

1.3.4 Bridging fault

Two interconnect wires can be placed close to one another during fabrication. Such wires can unintentionally get shorted to each other. Such faults are modelled by bridge faults [20], [21]. If two interconnect lines are bridged, then the fault on these lines can be excited when the two lines have different logic values. There are various types of bridging fault: wired-OR, wired-AND, and Dominant. When there is a wired-OR (AND) fault, a logic 1 (0) on any line will force the other line to logic 1 (0). A dominant bridging fault will force the value of the dominating line onto the other wire.

1.4 Previous Works

There are many different ways in the literature [22], [23], [24], [25], [26] where the test generation effort is being improved. These methods help in reducing the overall test generation time, which is usually performed before moving on to the next stage of the design flow. However, there is no framework to our knowledge that utilizes the test generated for a circuit and transforms it into a test for another structurally similar circuit.

The concept of incremental test generation was introduced in [26] for a different context that considered a single version of a design. This is reviewed in Section 1.4.1. The concept of incrementally modifying the flow has been explored in different fields like the synthesis of a design, place and route. In these works, the knowledge from a structurally similar design helps in reducing the effort for another structurally similar design. The following sections review incremental changes done in synthesis [27], [28],[29] (Section 1.4.2), and place and route [30] (Section 1.4.3).

1.4.1 Incremental Test Pattern Generation

The state-of-the-art test generation tools are fault-oriented. First, these tools generate tests for a targeted fault. Once the targeted fault has been detected or considered undetectable, the tool selects another fault and generates tests for it. The expectation in [26] is that starting from a test generated for one fault, it is possible to modify the test into a test

for another fault more quickly than starting from an all-unspecified test. This is illustrated next.

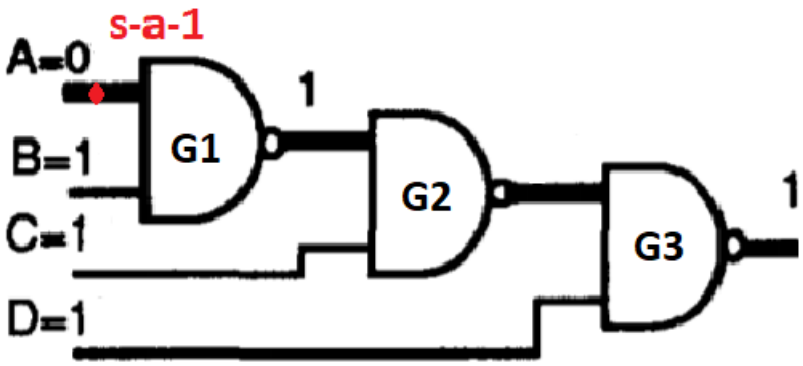


Figure 1.8. Test Pattern for Input A s-a-1 Fault

Figure 1.8 shows a circuit whose input A has an s-a-1 fault. To generate the test for input A s-a-1, the fault needs to be excited. Therefore, all inputs are unspecified, and input A is assigned logic value 0. In order to propagate this fault effect to the output, non-controlling values are assigned to the gates along the output path.

After the test vector detects the s-a-1 fault at input A, another undetected fault is picked, say s-a-1 at input B. In traditional tools, all the inputs are unspecified when a new fault is picked. Whereas [26] reuses the line justifications used for detecting the earlier fault. In order to excite the fault, input B is set to logic value 0, and input A is set to logic value 1, which is the non-controlling value of the AND gate. With the help of inherited values for the fault s-a-1 at input A, the injected faulty signal can be propagated to the output. This is shown in Figure 1.9.

In Figure 1.9, on the value shows that the value on the line was determined from the previous justification. Thus, in this example, for finding a test vector for the fault s-a-1 at input B, the earlier test vector helped to reduce the number of justifications by 2, which helps to speed up the test generation process.

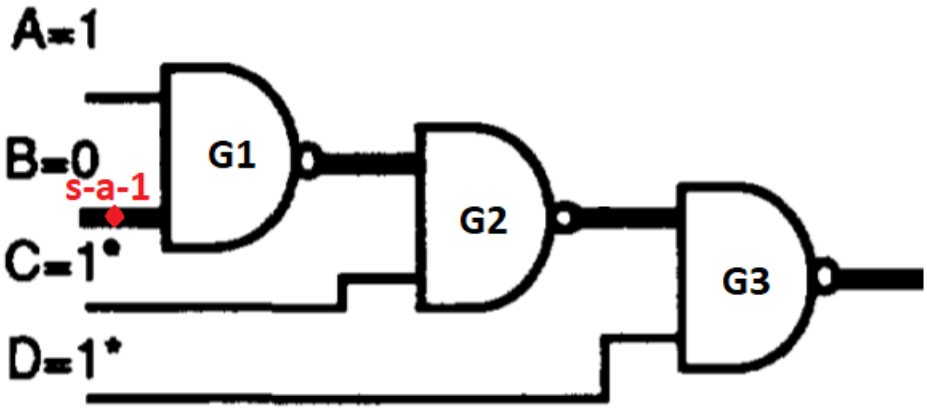


Figure 1.9. Test Pattern for Input B s-a-1 Fault Using Inherited Values

1.4.2 Incremental Logic synthesis using structural similarity

Design is modified when the circuit does not meet the design requirements. A minor modification can result in a different optimization when synthesizing the modified netlist during logic synthesis. The literature shows that the effort and time spent synthesizing the modified circuit can be reduced by incremental synthesis where gates are reused from the previous implementation.

In [27], a method is proposed to identify the correspondence between gates and pins of the baseline and the modified circuits. First, the correspondence between the circuits is computed by creating an identification index for every gate based on the name of the primary input/output (IO) lines, type of gate, number of fanins, and number of fanouts. Then, a gate matrix is created using this index that includes every pair of gates in the baseline and the modified circuits. The algorithm identifies gates that are structurally compatible in the baseline and the modified circuits based on the gate matrix. Using the compatible gates, the procedure of [27] generates a synthesized circuit by retaining a maximum number of gates from the synthesized baseline circuit. This method identifies the logic gate components that must be modified due to the functional changes in the design and thus reduces the effort required for synthesis from the beginning.

The procedure of [28] first checks for isomorphic cones in the circuit, as all gates in such cones are structurally equivalent. By identifying such isomorphic cones, the algorithm can retain the implementation of all these cones during incremental synthesis. Next, the algorithm evaluates every gate in both circuits to check the type of gate and the nets these gates are connected. Using these two steps, the algorithm determines the structurally similar gates. After the structural and functional equivalence is established between the baseline and the modified circuits, a mapping is established between the circuits, which is used to guide the synthesis process to generate a synthesized circuit with minimal change from the baseline circuit.

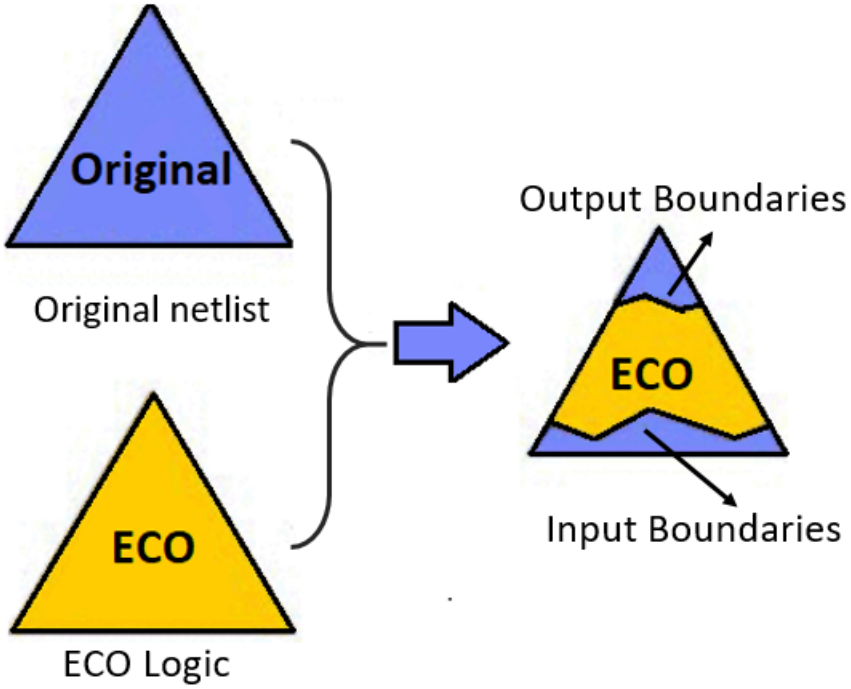


Figure 1.10. DeltaSyn Method From [29]

The paper [29] introduces a two-phase flow approach. The steps involve identifying the input and output boundaries of the modifications. The gates within these boundaries are the ones that need to be replaced due to ECO. First, identification of the input boundary is made using the structural and functional information to identify signals in the circuit that are functionally equivalent. Then, using these signals, the input side boundary is determined with the help of forward-sweeping algorithms. [31].

The output side boundaries are identified in the second step. This is done using a Boolean matching algorithm [32]. A recursive backward traversal is done from the primary outputs to obtain the logic equivalency. The output boundary is defined every time the circuit is matched from the output side. The synthesis tool can reuse all the gates that are outside the input and output boundary. This reduces the time taken for synthesis as the majority of the synthesized circuit has been reused.

1.4.3 Structural similarity during Place & Route

In [30], an ordering technique is proposed for all gate instances in the design. This technique is invariant to the changes in the circuit due to Engineering Change Order. For the ordering, a value is obtained for each gate instance by a linear combination of the type of the gate, number of fan-in gates, number of fanout gates, and terminal node name of the gate instances. In addition, the terminal node name of the connected fan-in and fanout gates are also used. This value is defined as a signature in [30]. After assigning signature values to all gate instances, a traversal is done in the input and output cones to obtain the updated signature values of the gate instances by linearly combining the signature of the fanout and fan-in gates in the cone, respectively. Thus, the ordering of gate instances based on the computed signature values guarantees a minimal change in the gate instance order even if the ECO process changes the ordering of the gate instances.

1.5 Contribution

The state-of-the-art test generation tools do not take advantage of the structural similarity in the circuits. Every time the gate-level is modified, the test generation must be done from the beginning discarding all the patterns generated earlier.

This thesis provides a novel test generation procedure whose overview is shown in Figure 1.11 and is described next.

In both original and modified designs, an integer arithmetic-based computation is performed on each gate. This computed value is called the signature. The signature sets from the original and modified designs are compared to find structural similarity between the ver-

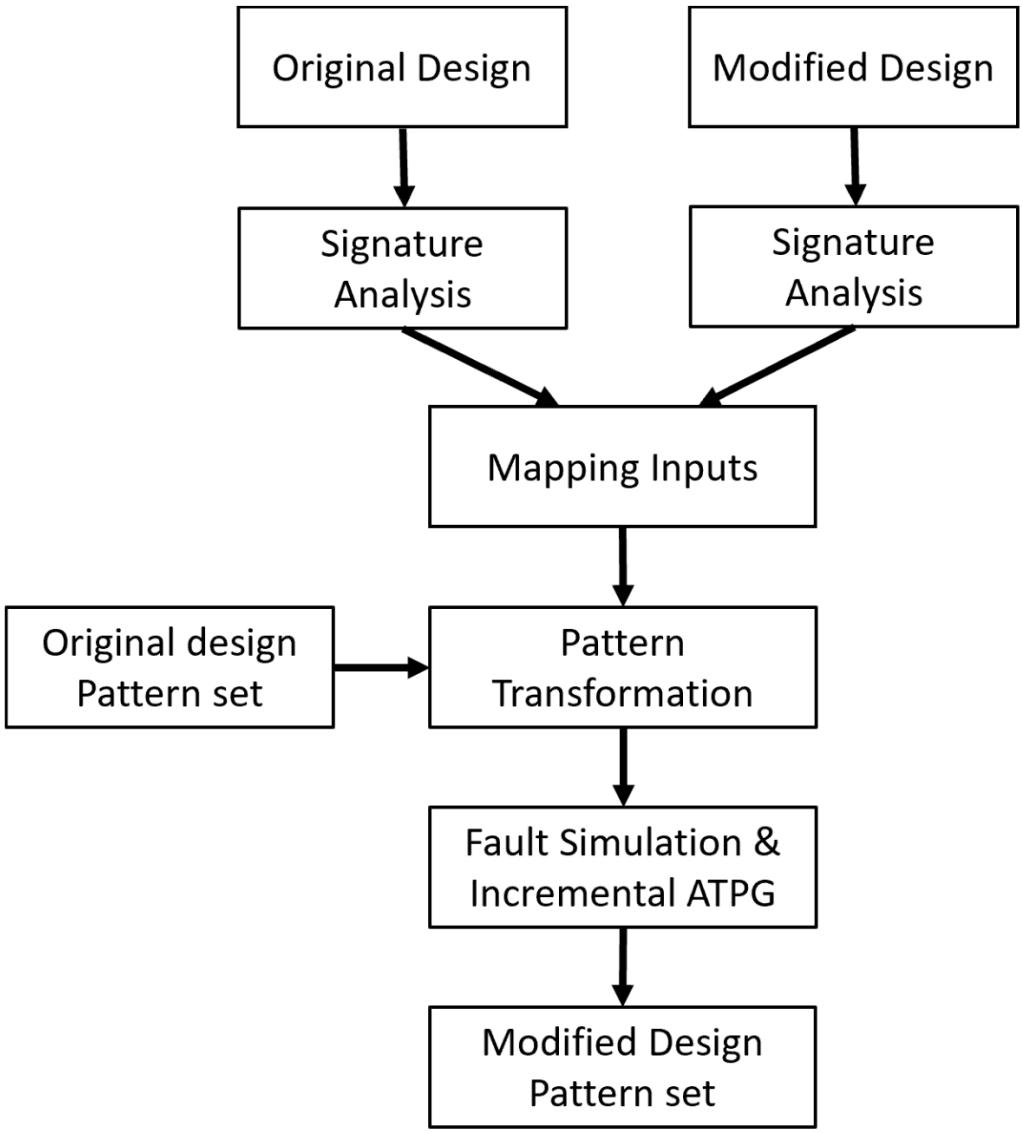


Figure 1.11. Overview of ATPG flow in the thesis

sions. This comparison is used to map the inputs and outputs between both designs. Using the mapping information, the generated test patterns for the original circuit are transformed into test patterns for the modified circuit. Transformed test patterns are fault simulated, and test patterns are generated incrementally for the remaining faults from the modified portion of the design. The overall procedure can be applied to any structurally similar design and

any fault model. Identifying structural similarity and reusing of existing patterns speeds up the test generation procedure compared to generating patterns from the beginning.

1.6 Thesis Overview

The thesis explores the aspect of accelerating the test generation for various structurally similar circuits. This is done by transforming the test generated for one circuit into a test pattern for another structurally similar circuit. The thesis is organized as follows.

Chapter 2 describes a fast test generation process for structurally similar circuits obtained by modifying the gate-level netlist. Chapter 3 describes an iterative flow when designs are modified at RTL. Chapter 4 explores a methodology when two-cycle tests are used to detect faults in a structurally similar circuits. Finally, chapter 5 gives the summary of the thesis.

2. FAST TEST GENERATION FOR STRUCTURALLY SIMILAR CIRUITS

This chapter describes a fast test generation process for digital circuits that exhibit extensive structural similarity. The property of structural similarity can be seen in circuits that are subjected to engineering change order (ECO), circuits that are modified during place and route, circuits subjected to retiming, and circuits with multiple similar cores. The goal of this chapter is to determine the testability of a circuit (circuit2) given a test set for a structurally similar circuit (circuit1). This is achieved by transforming a test set generated for circuit1 into a test set for circuit2 as efficiently as possible, without repeating the entire test generation process. The process described in this chapter starts with a structural analysis of circuit1 and circuit2 to obtain a mapping between their inputs and outputs. The mapping is used for transforming test patterns from circuit1 into test patterns for circuit2. The experiments conducted on industrial designs show an average of more than 10-fold reduction in runtime, compared with running the entire test generation process for circuit2.

2.1 Introduction

The digital revolution has led to a steep increase in the complexity and density of Integrated Circuits (IC) [33]. Multiple cores are integrated within a chip to increase its throughput. For complex and dense ICs, the time to test, debug and verify the circuit becomes the bottleneck for chip design [34], [35], [36].

State-of-the-art synthesis flows are iterative to accommodate the need to fix bugs, and address performance and power constraints [37], [38]. Test generation [39], [40], [41], [42] is performed early in the design flow to identify testability issues that cannot be identified using approximations such as testability measures [43],[44]. The test set obtained at an early

stage of the design process does not need to be optimized, and the speed of test generation is essential to the fast convergence of the design process. This chapter presents a novel approach for fast test generation by observing that structural similarity is prevalent with the current synthesis flow and state-of-the-art designs, as discussed next.

A large number of similar cores can be seen in conventional, gaming, and graphical processors. ICs like CPUs, GPUs, and AI chips show structural similarity within cores [45], [46], [47]. Moreover, a new design that reuses the same cores shows significant structural similarity to a previous design. Netlists before and after place and route, and circuits subjected to engineering change order (ECO) [38], [27], [48], [28], [49], or retiming [50], [51], also exhibit significant structural similarity.

The property of structural similarity between circuits is not utilized in the current test generation tools. As a result, test generation has to be run in its entirety after every change that may affect the testability of a circuit. This chapter aims to analyze the testability of a circuit (circuit2), which is structurally similar to another circuit (circuit1), by efficiently transforming the test set generated for circuit1 into a test set for circuit2 without repeating the entire test generation process. This is achieved through a process that uses incremental test generation.

The concept of incremental test generation was introduced in [52] for a different context that considered a single version of a design. The expectation in [52] is that starting from a test generated for one fault, it is possible to transform the current test into a test for another fault in the same design more quickly than starting from an all-unspecified test. In this chapter, the goal is to accommodate the fact that two circuits are structurally similar. Reusing an existing test set for circuit1 would significantly reduce the run time compared to running test-pattern generation from the beginning for circuit2.

Among the variations that may occur between similar circuits, circuit1 and circuit2, is a change in the order of the inputs or outputs. In this case, simply simulating an existing test set of circuit1 on circuit2 may not result in the detection of a significant number of faults. An incremental test generation solution needs to be independent of the order of the inputs and outputs. It also needs to accommodate the fact that state-of-the-art CPU, GPU, or AI chips have large numbers of identical logic blocks within them [46]. This creates more options by

which inputs and outputs of structurally similar circuits can be matched. An incremental test generation approach suitable for this context is developed in this chapter. As a part of the procedure described in this chapter, a structural correspondence is established between circuit1 and circuit2.

Structural correspondence is also considered in [27], [28] and [30]. In [27] and [28], structural similarity is used to determine the gates that can be re-used in synthesis after ECO. In [30], structural similarity of the circuit before and after ECO is exploited in the context of place and route.

In [30], an ordering technique is proposed for all gate instances. This technique is invariant to the changes in the circuit due to ECO. For the ordering, an integer value is calculated for each gate instance by a linear combination of the type of the gate, number of fan-in gates, number of fan-out gates, and terminal node name of the gate instance. In addition, the terminal node names of the connected fan-in and fan-out gates are also used. The value assigned to a gate instance is defined as its signature in [30]. After assigning signature values to all gate instances, a forward traversal is done from inputs to outputs to obtain updated signature values for the gate instances. This is achieved by linearly combining the signatures of the fan-in gates for each gate instance. The ordering of gate instances based on the computed signature values guarantees a minimal change in the gate instance order even if the ECO process changes the ordering of the gate instances.

An approach based on signatures is also followed in this chapter to determine the structural similarity between circuit1 and circuit2. The distinguishing feature in this chapter is that the signature values are computed based on properties of the gates and are not dependent on the predefined names of the gates. This is important because names may not be preserved between circuit1 and circuit2. In addition, the procedure proposed in [30] for signature computation traverses the circuit only once from inputs to outputs. In this chapter, a forward traversal is done from inputs to outputs to calculate output signature values. Using these values, a backward traversal is done from outputs to inputs to compute input signature values. In this way, the input signature captures the structural difference in the circuit even when the difference is not in the input cone of logic. This method of forward and then backward traversal is done to embed the properties of the output and input cones

into the input signature. The mapping between the inputs of the two circuits is established using these unique values on the input and output pins, and the input-output list of a logic cone.

This chapter focuses on the application where a circuit is subjected to ECO or layout changes that result in modifications in the combinational or sequential logic of the circuit. These modifications change the netlist by the addition/removal of modules or modifying the logic in the circuit. In these contexts, designers change the circuit incrementally to impact only a small part of the design [53], [54]. The effectiveness of the proposed algorithm depends on the two circuits being structurally similar, and on the accuracy of the mapping of the circuit inputs. Synthesis can affect both of these properties. Hence to verify the algorithm, structurally similar netlists were obtained by making changes to the gate-level netlist so that the circuit before the change (circuit1) is structurally similar to the circuit after the change (circuit2).

This chapter is organized as followsSection II details the test generation methodology. Section III presents results for industrial designs followed by conclusions in Section IV.

2.2 Test Generation Process

2.2.1 Overview

The methodology we propose has three main steps. In the first step, we establish a mapping between the inputs and outputs of circuit1 and circuit2 based on structural equivalence [27],[28],[30]. We use input and output signatures for this purpose. A signature is defined by an integer-arithmetic based computation done on each gate of a logic cone to produce a unique value that captures the structure of the cone. If the circuits are structurally equivalent, the input and output signatures of circuit1 and circuit2 will allow us to find a perfect match between them. Due to the variations between the circuits, the matching is not perfect and some of the inputs and outputs remain unmatched. The corresponding input and output cones are where circuit2 is expected to be structurally different from circuit1.

In the second step, we transform a test set T_1 generated for circuit1 into a test set T_2 for circuit2. We use the mapping identified in Step 1 to copy values of matching inputs from T_1

to T_2, leaving unmatched inputs unspecified. The unspecified inputs are assigned random values and then simulated to determine the faults detected using the test set T_2.

In the third step, we carry out incremental test generation to detect faults in circuit2 that have not been detected. These patterns are appended to T_2.

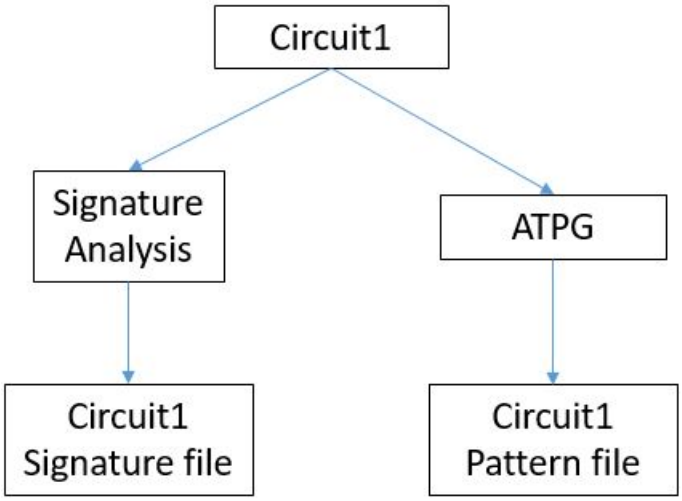


Figure 2.1. ATPG Flow of Circuit1

Figure 2.1 describes the computations performed for circuit1, including test generation to produce a test set for circuit1, and signature analysis to produce input and output signatures.

Figure 4.2 describes the computations performed for circuit2, including signature analysis to produce input and output signatures, input mapping based on the signatures computed for circuit1 and circuit2, computation of the test set T_2 and incremental test generation to extend T_2.

In the next sections, we describe each of these steps in more detail.

2.2.2 Signature Computation

This section describes the computation of the input and output signatures. We first compute output signatures and then input signatures. A dictionary is created, where for every output signature (referred to as a key), we store vectors of inputs that drive outputs with a signature equal to the key. The dictionary is used for mapping inputs from circuit1

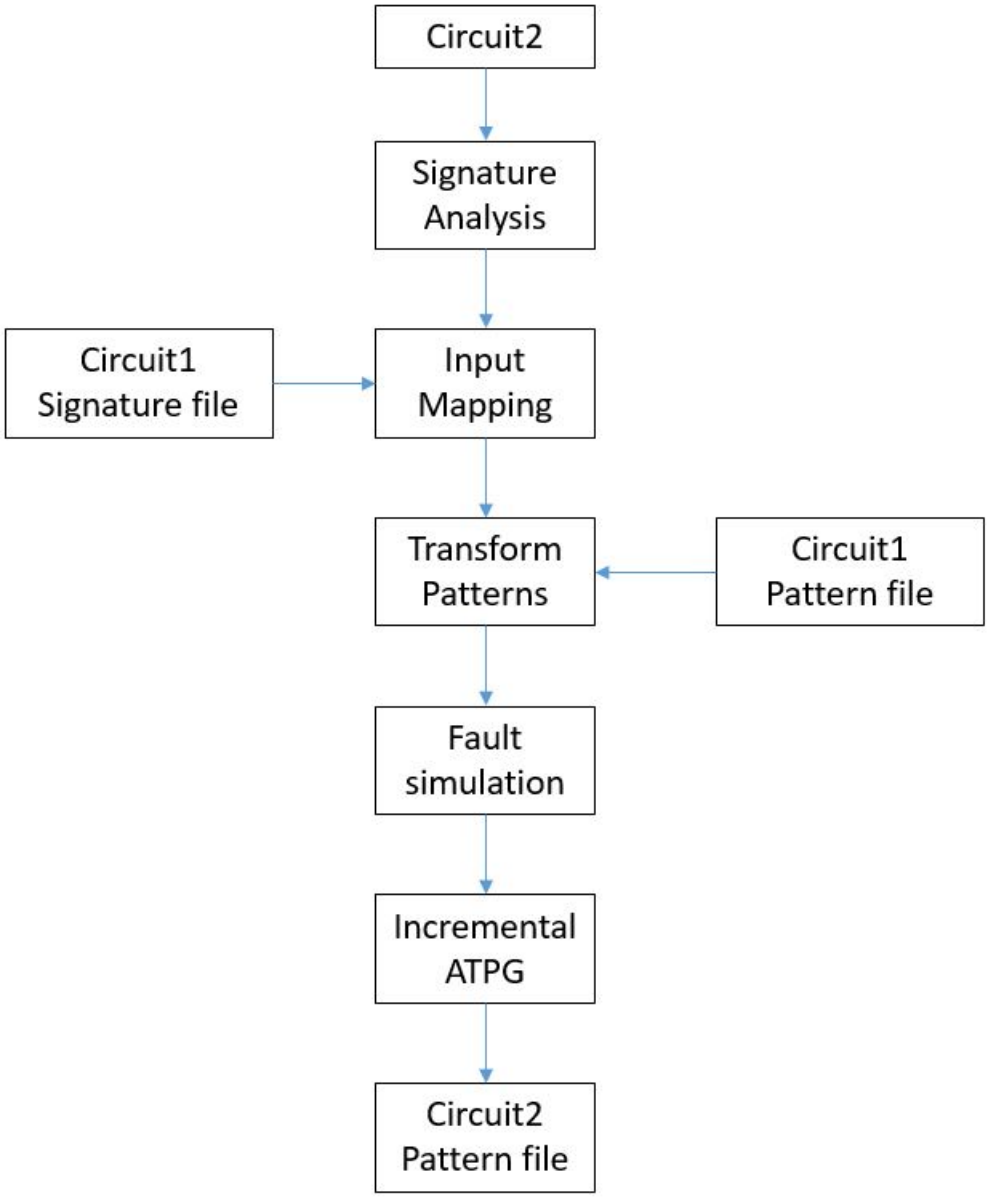


Figure 2.2. ATPG Flow of Circuit2

to circuit2. The input signatures and the dictionary are written to a file for later processing. We experimented with different approaches to computing the signatures and concluded that the one discussed next produces accurate results.

A signature needs to capture the structure of the design. To embed the gate types into the signature, each type of gate was assigned a unique prime number. A large number is

chosen to reduce the probability of the same signature being obtained for different structures (aliasing). Furthermore, the level of a gate is captured by rotating the computed value during the traversal of the circuit. Rotation multiplies a number by two, causing larger numbers to be obtained with more levels. Using this approach to signature computation, the aliasing of signatures was not seen in the experiments performed.

For signature computation, the gate types are grouped into five categories. Each category is assigned a unique prime number randomly selected in the range between 1 - 15 million. This is called the mask for the group. The grouping of the gates and an example of the masks used for each group are shown in Table 2.1.

Table 2.1. Categories of Gate Type and Prime Number Used as Mask

	Gate Types	**Prime Number (Mask)**
1	AND, NAND	1,540,681
2	WIRE, BUS	2,572,261
3	OR, NOR	4,980,727
4	XOR, XNOR	5,210,099
5	BUFFER, INVERTER	9,137,657

A variable called "Output_Invert" is used during the signature computation to distinguish gate types in each category as shown in Table 2.1. "Output_Invert" is set to false for OR, AND, BUFFER, XOR, and WIRE gate types, whereas it is set to true for NOR, NAND, INVERTER, XNOR, and BUS gate types. When the "Output_Invert" is true, the bitwise complement of the computed value is stored as the signature of the gate.

For output signature computation, the inputs are initialized to a prime number randomly chosen between 1-15 million. The signatures are computed by traversing the circuit from inputs to outputs. The output signature of a gate is computed by an equation that takes

into account the signatures of the gate inputs and the mask of the gate. The equation for gate types AND, WIRE, OR, XOR, and BUFFER is shown below in equation (2.1).

$$\begin{aligned}&\text{Output Signature of a Gate}_i = \\ &\quad \text{Rotate}\left(\sum_i (\text{Rotate (Output signature of fan\_in} \right. \\ &\quad \left. \text{of Gate}_i) + \text{Group Gate Mask of Gate}_i\right)\end{aligned} \tag{2.1}$$

The equation for gate types NAND, BUS, NOR, XNOR, and INVERTER is shown below in equation (2.2).

$$\begin{aligned}&\text{Output Signature of a Gate}_i = \\ &\quad \left(\text{Rotate}\left(\sum_i (\text{Rotate (Output signature of fan\_in} \right.\right. \\ &\quad \left.\left. \text{of Gate}_i) + \text{Group Gate Mask of Gate}_i\right)\right)^C\end{aligned} \tag{2.2}$$

The addition operations in equations (2.1) and (2.2) are done with carry. The C in equation (2.2) stands for the bitwise complement. Rotate refers to a rotate left operation. Output signature computation is illustrated in Fig. 2.3.

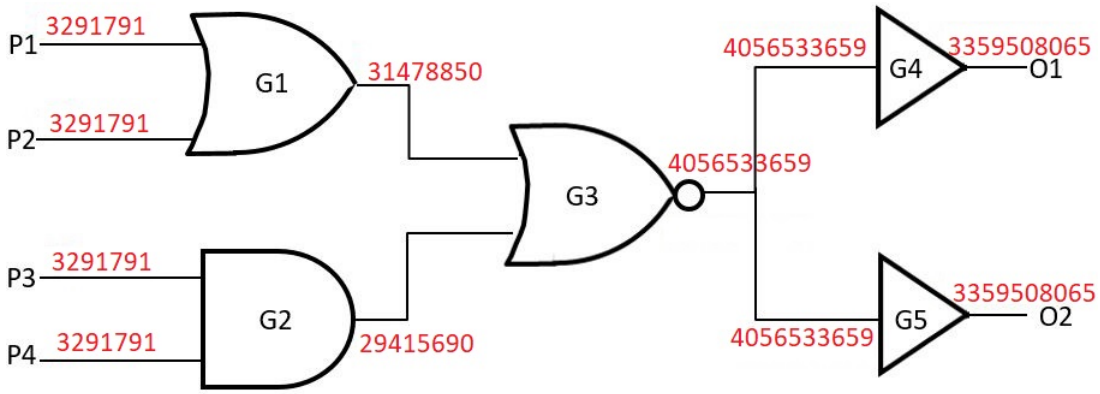


Figure 2.3. An Example of Output Signature Computation for a Logic Circuit

Output signature computation is illustrated in Figure 2.3. The circuit in Figure 2.3 has four inputs (P1, P2, P3, P4), two outputs (O1, O2), one AND gate (G2), one OR gate (G1), one NOR gate (G3), and two BUFFER gates (G4, G5). The output signature computation of gate G1 is discussed next.

Gate G1 is an OR gate and belongs to group 2. The inputs of G1 are P1 and P2, whose output signatures are initialized to 3,291,791. First, the output signatures of the inputs P1 and P2 are rotated to obtain 6,583,582. Next, the rotated output signatures of P1 and P2 are added along with the group gate mask of 2,572,261. This result is rotated, and the value obtained is 31,478,850. This value is stored as the output signature of G1.

The output signatures of gates G2, G4, and G5 are computed in the same way by equation (2.1), and for G3, the output signature is the bitwise complement of the value computed using equation (2.1). The value 4,056,533,659 is obtained assuming 32-bit integers. The output signatures of all the gates are shown in Figure 2.3.

For the input signature computation of the logic circuit in Fig. 2.3, the input signature of the outputs O1 and O2 are initialized to their output signatures. The algorithm traverses the circuit from outputs to inputs.

The equation for the computation of the input signature for gate types AND, WIRE, OR, XOR, and BUFFER is shown below in equation (2.3)

$$\begin{aligned}&\text{Input Signature of a Gate}_{\text{i}} = \\ &\text{Rotate}\Big(\sum_{\text{j}} \text{Rotate}\ \big(\text{Input signature of fan\_out}_{\text{j}} \\ &\text{of Gate}_{\text{i}}\big) + \text{Group Gate Mask of Gate}_{\text{i}}\Big)\end{aligned} \tag{2.3}$$

The equation for the computation of the input signature for gate types NAND, BUS, NOR, XNOR, and INVERTER is shown below in equation (2.4)

$$\text{Input Signature of a Gate}_i = \left(\text{Rotate}\left(\sum_j \text{Rotate}\ \left(\text{Input signature of fan\_out}_j \text{ of Gate}_i\right) + \text{Group Gate Mask of Gate}_i\right)\right)^C \tag{2.4}$$

The addition operations in equations (2.3) and (2.4) are done with carry. The input signatures of gates G2, G4, and G5 are computed as shown in equation (2.3), and G3 is computed as shown in equation (2.4).

Algorithm 1 Signature computation for a circuit

Input: Design

Output: Output signatures, Input signatures, Dictionary

1: *Initialization* :
2: For every gate, assign its output and input signatures to 0
3: Initialize all output signatures of primary inputs to prime-number1 randomly chosen between 1-15 million.
4: Initialize all output signatures of pseudo primary inputs to prime-number2 randomly chosen between 1-15 million.
5: Assign different prime numbers as masks for each type of gate
6: *Output Signature*
7: **for** each gate_i from inputs to outputs **do**
8: Rotate the output signature at the inputs of gate_i and add
9: Add mask of gate_i to the result in step 8
10: Rotate the output signature obtained in step 9
11: **end for**
12: **if** Output_Invert is true **then**
13: Complement the signature
14: **end if**
15: *Input Signature*

16: Initialize all input signatures of primary outputs and pseudo-primary outputs to their output signatures
17: **for** each $gate_i$ from outputs to inputs **do**
18: **for** each fanout of $gate_i$ **do**
19: Rotate the input signature
20: EXOR result in Step 19 with the mask of the $gate_i$
21: **end for**
22: Add signatures after Step 20
23: **end for**
24: **if** Output_Invert is true **then**
25: Complement the signature
26: **end if**
27: *Dictionary*
28: **for** $gate_i$ that is an observation point **do**
29: Store the inputs driving the output in a vector
30: For each output signature, insert a vector of inputs driving the output cone.
31: **end for**

Algorithm 1 outlines the signature analysis done for a given design. The signature computation is done on both circuit1 and circuit2 to compute all input and output signatures. This generates two dictionaries: $dict_1$ and $dict_2$ pertaining to circuit1 and circuit2, respectively. A dictionary contains output signatures as the key value. Each key is linked to a set of vectors that correspond to the inputs driving the output whose signature is equal to the key. All output cones with equivalent structures will have the same signature but possibly different vectors of inputs. Using the generated dictionaries, a set of input vectors from a cone in circuit1 is mapped to a set of input vectors of an identical cone in circuit2.

2.2.3 Mapping between circuit1 and circuit2

In this step, the goal is to establish a correspondence between circuit1 and circuit2 based on the computed signatures. This step reads the dictionaries $dict_1$ and $dict_2$. For every key

in $dict_1$ and $dict_2$, a mapping is done when the input signatures of the input vector in $dict_1$ match the input signatures of the input vector in $dict_2$.

Algorithm 2 Comparing signatures from circuit1 and circuit2

Input: Input signatures, $dict_1$, $dict_2$

Output: Mapping Information

```
1: Mark all vectors in dict1 and dict2 as unselected
2: for each key in dict1 do
3:    if key in dict2 then
4:       for every unselected vector1 in dict1(key) do
5:          for every unselected vector2 in dict2(key) do
6:             if input signatures of vector1 and vector2 are equal then
7:                Map the inputs in the vectors according to the input signatures
8:                Mark vector1 and vector2 as selected
9:             end if
10:         end for
11:      end for
12:   end if
13: end for
```

In Algorithm 2, input signatures, $dict_1$ and $dict_2$ are given as inputs. The keys of these dictionaries are the output signatures. The algorithm finds the output signatures common in $dict_1$ and $dict_2$ and iterates through the input vectors to find ones that have the same input signatures in $dict_1$ and $dict_2$. Once a vector in $dict_1$ is matched with a vector in $dict_2$, these vectors are not considered in further mapping. In this way, inputs are mapped from circuit1 to circuit2 in those parts of the circuit where input and output cones are structurally identical. In all the experiments done on industrial circuits, the mapping algorithm accurately identified cones that were identical in circuit1 and circuit2.

2.2.4 Transforming the Pattern File

In circuit1, ATPG produces the set of patterns T_1. Considering circuit2, the values of inputs in circuit2 that have a match in circuit1 are copied from T_1 to a new test set T_2. Random values are assigned in T_2 to inputs that do not have a match in circuit1. Circuit2 is fault simulated using T_2 as the set of test patterns. This transformation helps detect faults from the output cones of circuit2 which are structurally identical to the output cones of circuit1. A structurally identical cone will have the same input and output signatures in both circuits. All the faults from such input and output cones of circuit2 are detected by T_2.

The transformed test set may not detect faults from a cone that does not have a match in circuit1. In addition, it may not detect faults in a cone of circuit2 that shares inputs with a cone that is structurally different in circuit1. This is illustrated next.

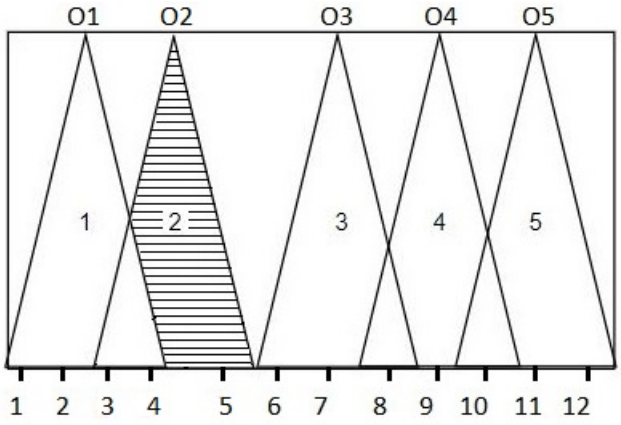


Figure 2.4. A Logic Circuit with Modification

Fig. 2.4 shows a logic circuit with five logic cones whose outputs are labeled as O1-O5 and inputs are labeled as 1-12. This circuit (circuit2) differs from another circuit (circuit1) at the shaded region in cone 2, such that cone 1, cone 3, cone 4, and cone 5 are structurally identical in both circuits. The output of cone 2 is O2 and its inputs are 3, 4, and 5. The structural difference causes the output and input signatures of cone 2 to be different in circuit1 and circuit2. In contrast, the output signatures of O1, O3, O4, and O5 would remain the same. Fig. 2.4 shows that some of the inputs of cone 2 are shared by cone 1 (inputs 3 and 4).

Hence, in cone 1, the signatures of inputs 3 and 4 would be different in circuit2 compared to circuit1. After mapping of test patterns and fault simulation, the simulated test patterns are guaranteed to detect all the faults from the structurally equivalent cones 3, 4, and 5 (all the input and output signatures of these cones remain the same), whereas only some of the faults are detected from cones 1 and 2. We accept this effect on cone 1 to ensure that the mapping between inputs is accurate. Without computing input signatures, it would not be possible to avoid matching inputs of cones that were structurally different in both circuits.

2.2.5 Incremental ATPG

The undetected faults of circuit2 are obtained after the fault simulation of T_2. Using the reduced fault set, test pattern generation is carried out to detect the remaining faults. The patterns generated are appended to T_2 to obtain the complete test pattern set for circuit2.

2.3 Experimental Setup and Results

Several experiments were performed to determine the accuracy of the proposed signature computation method, and to evaluate the test generation procedure. In Section A, three experiments are described to demonstrate the effectiveness of the signature computation method. Sections B and C evaluate the test generation procedure on 11 industrial circuits. These experiments consist of logical (Section B) and sequential (Section C) changes. Single stuck-at faults were used for the evaluation, and a compacted test pattern set was computed for circuit1 using a commercial ATPG tool.

2.3.1 Experimental Study on Signature Aliasing

The goal of the experiments reported in this section is to verify experimentally whether aliasing, which refers to obtaining the same signature for structurally different cones, occurs in the circuits. This is important since the mapping algorithm relies on identical signatures to identify identical cones. If two different cones produce the same signatures, the mapping algorithm may result in an incorrect mapping of their inputs. The first experiment considered

a 4-level AND-OR gate connection, as shown in Figure 2.5. The experiment considered the 16 possible gate combinations, as well as the addition of an input to each gate.

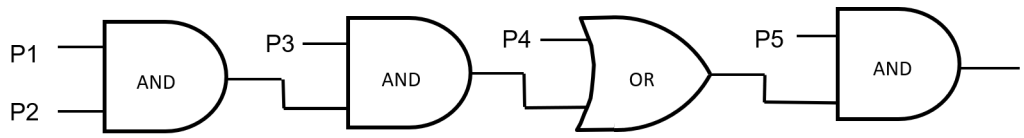


Figure 2.5. 4-level AND-OR gate connection

Table 2.2. Output signature for different configurations

	Level 1	Level 2	Level 3	Output
1	29415690	133911286	551893670	2223823206
2				2225886366
3			553956830	2232075846
4				2234139006
5		135974446	560146310	2256833766
6				2258896926
7			562209470	2265086406
8				2267149566
9	31478850	142163926	584904230	2355865446
10				2357928606
11			586967390	2364118086
12				2366181246
13		144227086	593156870	2388876006
14				2390939166
15			595220030	2397128646
16				2399191806
17	42582854	186579942	762568294	3066521702
18	29415690	147078450	604562326	2434497830
19	29415690	133911286	565060834	2276491862
20	29415690	133911286	551893670	2236990370

The first 16 rows of the Table 2.2 show the output signature of all 16 possible combinations of AND and OR gates, in the order AND-AND-AND-AND, AND-AND-AND-OR, . . . , OR-OR-OR-OR and the last four rows of Table 2.2 show the four cases of adding a single input to one of the gates of AND-AND-AND-AND configuration. All the configurations resulted in unique output signatures. The use of a different mask for each type of gate resulted in a distinct signature at each level. Furthermore, the rotate operation captures the level of

each gate during the gate traversal. This experiment also shows that replacing a 2-input AND gate with a 3-input AND gate results in distinct output signatures. For the next two experiments, 100 logic cones were chosen from 11 industrial designs. The first experiment conducted exhaustive single gate changes where a single gate type is changed into every other gate type. In the second experiment, the number of gates to be changed (between 1-8) in a single modification was randomly chosen. Then, the chosen gates were randomly changed to another gate type. In each case, the difference between the output signatures of the baseline and modified cones was found. Its absolute value is referred to as the deviation.

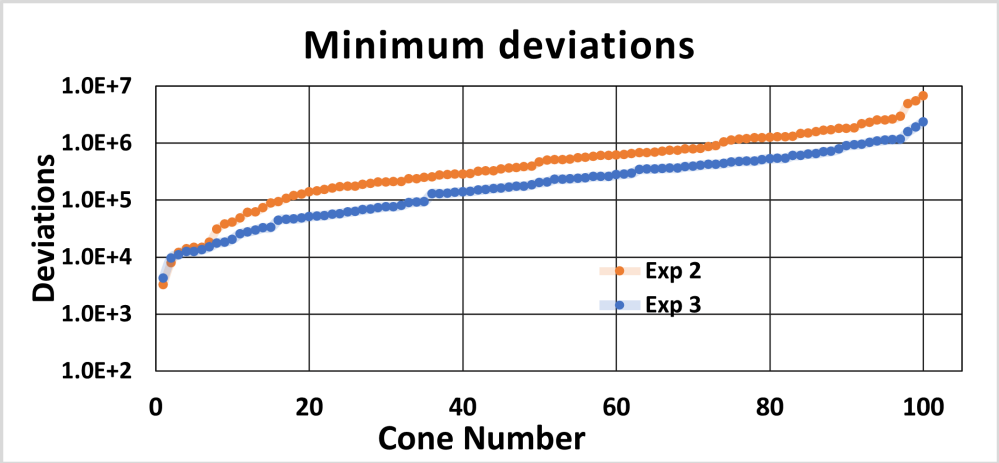


Figure 2.6. The minimum of deviation in logarithmic scale (base 10)

Figure 2.6 shows the minimum deviation for the two experiments. Both experiments together compared output signatures of 1.1 million modifications to the output signature of the baseline cone. The results showed that aliasing was not seen in these circuits, i.e., two different structures produced distinct signatures, and it is very unlikely for two different structures to have identical signatures.

2.3.2 Logic changes

A combinational logic change consists of changing the type of a gate, e.g., from AND to OR, and adding or removing inputs from a multi-input gate. Similar designs are obtained by introducing one change at a time, as well as a small number of changes simultaneously. Such modified versions of a circuit represent designs that are very similar to one another,

or minor changes introduced due to ECO or after place and route. We experimented with versions obtained by introducing a large number of changes and found that the test generation procedure is also effective in these cases.

The experiment discussed next introduces one gate type change at a time and performs test generation to study the effectiveness of the incremental test generation procedure. Then, for each circuit, ten different versions are generated by introducing a single change at a random location in each of the versions. The results for logic changes are tabulated in Table 3.3. The first part of Table 3.3 represents changing the gate type, and the second part represents the addition or removal of inputs from multi-input gates.

Table 2.3. Experimental Result for Logic Changes

	Size	Tot Cone in Base Cir	Diff Cones in Mod	RunTime (seconds)			Test Patterns			Fault Coverage(%)	
				Base Cir	Mod Cir	Gain (Ratio)	Base Cir	Mod Cir	Incr (%)	Base Cir	Mod Cir
d1	8,611,749	429,708	1,262	7,020	506	13.87	2,334	2,842	21.76	97.06	97.06
d2	3,435,493	276,462	367	622	99	6.28	1,459	1,565	7.27	92.9	92.91
d3	4,833,052	334,108	4,218	1,568	123	12.74	1,408	1,509	7.17	92.08	92.08
d4	10,213,281	601,469	20,548	9,300	994	9.35	7,424	8,022	8.05	90.85	90.87
d5	15,102,044	1,215,337	5,461	13,740	1,177	11.67	4,544	5,067	11.5	94.46	94.47
d6	3,048,599	174,505	1,151	1,260	128	9.84	2,944	3,198	8.62	91.92	91.92
d7	3,519,135	219,280	3,043	5,103	405	12.6	5,155	5,780	12.12	97.02	97.03
d8	6,389,260	302,540	2,741	1,777	244	7.28	2,520	2,649	5.11	90.83	90.86
d9	12,167,308	440,540	12,063	6,446	843	7.64	7,168	7,297	1.79	91.65	91.70
d10	1,203,568	45,345	1,361	484	26	18.61	1,811	2,046	12.97	88.58	88.58
d11	1,699,509	89,908	5,328	10,168	459	22.15	2,163	2,891	33.65	97.62	97.71
d10	1,203,568	45,345	861	484	23	21.04	1,811	1,895	4.63	88.58	88.58
d11	1,699,509	89,908	4,828	10,168	521	19.51	2,163	2,952	36.47	97.62	97.65

In Table 3.3, each row corresponds to one of the 11 industrial circuits. The row shows the average of each one of several parameters for all the ten versions of the circuit. These results are shown under the sub-column Mod Cir. Sub-column Base Cir shows the results for the baseline circuit in the respective categories. The first column, "size" gives the total number of gates in each circuit. Column "Tot Cone in Base Cir" shows the total number of output cones in the baseline circuit. Column "Diff Cones in Mod" shows the number of output cones that changed in the modified circuit compared to the baseline circuit. Column Run Time shows the overall time taken in seconds for test pattern generation. For a modified circuit, the total runtime includes the time taken for signature computation, mapping, fault simulation, and incremental test generation. Sub-column "Gain" is the ratio of the runtime of the baseline circuit to the average of the runtime of the modified circuits. Column Test Patterns shows the total number of test patterns after test generation. The sub-column "Incr (%)" gives the percentage increase in the number of test patterns. The last column, Fault coverage, shows the fault coverage achieved in the baseline circuit and the average fault coverage seen in the different versions of the modified circuit.

From column "Diff Cones in Mod" in Table 3.3, it can be seen that introducing a single change can affect between 0.13% to 5.9% of the total number of output cones for the 11 industrial circuits. Within the 10 different versions of a circuit, an average of 11% variation is seen between the minimum and the maximum number of cones that are affected. The number of output cones different in the modified circuit compared to the baseline circuit directly affects the time taken for the incremental test generation. The lower the number of different cones, the higher the number of matched inputs, resulting in a larger number of faults being detected in the modified circuit after mapping of test patterns. By transforming the test set from T_1 to T_2 and then fault simulating, faults from cones that are structurally identical in the modified and baseline circuits are detected. The transformation of test patterns results in an improvement in the total runtime of the modified circuit. The improvement in runtime is observed because only a small subset of the total faults require test generation. This translates to an average of 13-fold gain in runtime to achieve the same fault coverage in the modified circuit compared to the baseline circuit for all the different industrial circuits. The

gain in runtime depends on the circuit and varies from 6 to 22-fold. This shows that the test generation time can be considerably reduced by utilizing the structural similarity between the circuits for various changes.

The algorithm proposed in this chapter focuses on determining the testability of a circuit given a test set for another structurally identical circuit and does not optimize the number of test patterns. As a result, there is an average of 13% increase in the number of test patterns in circuit2 compared to circuit1. This can be mitigated by rerunning test pattern generation after the design converges or when the number of patterns increases significantly.

Multiple changes within a circuit

Multiple changes in the range of 10 to 10,000 were introduced in a circuit at random locations to see how the test generation procedure performs in the presence of multiple changes. The procedure was able to determine the changed cones, and faults in the circuit were detected in less than 25% of the total runtime of the baseline circuit. The proposed method took more time to detect all the faults as the number of changes in the circuit increased. It should be noted that these numbers of changes are not realistic, and they were considered only for the purpose of verifying the performance of the test generation procedure.

2.3.3 Sequential changes

The second experiment considers sequential changes in a circuit. Basic steps of forward and backward retiming are done where registers are moved forward or backward in the combinational blocks. This changes the total number of scan cells in the modified circuit compared to the baseline circuit. The results for sequential changes are tabulated in Table 2.4.

Column “# of Mod SC" in Table 2.4 gives the average of the number of scan cells added or removed in 10 modified versions of each circuit. Column “Test Pat Inc (%)" gives the percentage increase in the number of test patterns. The change in the number of scan cells in the modified circuit varies from 3-11 compared to the baseline circuit. Table 2.4 shows an average of 11-fold gain in runtime to achieve the same fault coverage in the modified circuit

Table 2.4. Experimental Result for Sequential Changes

	Diff Cones in Mod	**# of Mod SC**	**RunTime (seconds)**			**Test Pat Inc**
			Base *Cir*	**Mod *Cir***	**Gain *(Ratio)***	***(%)***
d1	3,418	7	7,020	685	10.25	19.66
d2	721	6	622	79	7.87	2.6
d3	1,282	6	1,568	141	11.12	19.38
d4	2,335	7	9,300	821	11.33	2.35
d5	2,104	6	13,740	975	14.09	7.54
d6	920	5	1,260	130	9.7	3.34
d7	2,593	6	5,103	328	15.6	13.13
d8	2,785	7	484	51	9.49	32.21
d9	1,355	5	10,168	391	26.0	20.29
d10	981	6	1,777	278	6.39	5.9
d11	21,043	6	6,446	899	7.17	3.29

compared to the baseline circuit. The results from sequential changes are consistent with the observations drawn from Table 3.3. The time taken for incremental ATPG depends on the number of cones affected by the change introduced in the baseline circuit. As the procedure focuses on the testability of the circuit, the average increase of 12% seen in the number of test patterns of the modified circuit compared to the baseline circuit is acceptable. This can be mitigated as discussed earlier.

2.4 Conclusion

This chapter described a fast test generation procedure that can be used for analyzing the testability of structurally similar circuits. The procedure finds the similarities between the two circuits, maps the inputs, transforms the test patterns, fault simulates, and performs incremental test pattern generation for the remaining faults. Two sets of experiments were conducted to demonstrate the performance of the procedure. The experiments were performed on 11 industrial circuits, and a summary of both experiments showed an average of 13-fold gain and 11-fold gain, respectively, in runtime compared with running the complete ATPG process.

3. TEST GENERATION FOR AN ITERATIVE DESIGN FLOW WITH RTL CHANGES

A typical VLSI design flow is iterative, implying that performance, power, area and testability are improved iteratively. With the shift left paradigm, most of the changes made to a design, including to a large extent changes to address testability, occur at the RTL. Test generation is an exception with a gate level netlist being required by ATPG tools. Within an iterative flow, repeated ATPG to reevaluate the testability of a design after its RTL has been changed becomes a bottleneck. To address this bottleneck, the test generation process needs to transform a test set generated for an earlier version of the design into a test set for a new version without repeating the entire test generation process. To enable the transformation, it is necessary to find a mapping between the inputs and outputs of the earlier and new versions of the design. The main contribution of this chapter is to compute such a mapping after RTL changes and resynthesis produce a new gate level netlist, where signal names may have changed, new signals may have been introduced, and signals that existed earlier may have been removed. Experimental results for industrial circuits with changes made at the RTL show an average of 5-fold reduction in test generation time.

3.1 Introduction

The complexity of designing digital circuits has grown significantly with lower technology nodes, larger design sizes, and heterogeneous integration of multiple dies into a single package. It has been estimated that the total design effort needed to manufacture a design in lower technology nodes more than doubles with every new generation [33], [34], [55]. Therefore, there is an urgent need for EDA tools to keep up with the increasing complexity of such designs by constantly finding new ways to address the challenges associated with

performance, memory footprint, and cost of running the tools. Designing such a complex circuit consists of numerous steps [56],[57], [58], as shown in Figure 4.1.

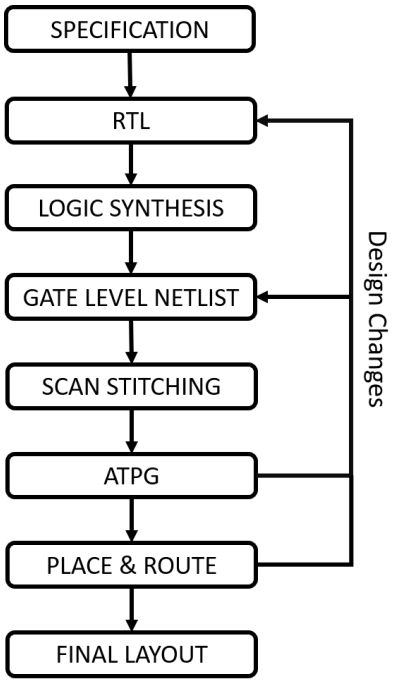


Figure 3.1. VLSI Design Flow [56],[57], [58]

Traditionally, Design-for-Test (DFT) [11] was restricted to the gate-level after the functional logic had been finalized and synthesized into gates. However, this has become a challenge for lower technology nodes, as logic insertion at the gate-level impacts area and timing optimization, affecting the overall design cycle time. Designers are often reluctant to add any logic at the gate-level post-synthesis. Consequently, there has been a significant increase in the adoption of a "Shift-Left" strategy across the industry, where significant effort is being placed to execute DFT-related tasks at RTL. Some of these tasks include running DFT-related DRCs, fixing the design to make it scan-friendly, DFT analysis, insertion of BIST **bist**, compression, boundary-scan, and in-system test execution engines in RTL. Moreover, because of the emergence of domain-specific accelerators, designers are creating slightly different versions of the same design and fine-tune per the requirements of the particular application.

Although the “Shift-Left" strategy allows moving most of the DFT analysis and insertion tasks to RTL, there are a few key aspects such as scan stitching and Automatic Test Pattern Generation (ATPG) [39], [40], [41], [42] that are still done at the gate-level. ATPG allows one to verify the fault coverage (assuming certain fault models) needed to meet the target quality goals. If the fault coverage does not meet the desired goals, one has to iteratively circle back-and-forth between RTL and gate-level to make design changes or add more DFT fixes to improve testability. Consequently, there is a push towards having the ability to quickly compute ATPG fault coverage during RTL development. DFT engineers can identify testability-related issues early on and get the designer's help to modify the design by considering testability, in addition to power, performance, and area constraints.

Typically, the ATPG tools require the test generation process to be performed from the beginning whenever the gate level netlist has changed. Although the designs obtained in consecutive iterations are similar, the present-day commercial tools cannot map two similar gate level circuits based on their structure accurately enough to make it effective to reuse the generated test patterns from earlier iterations. This can slow down the convergence of the design process considerably.

The methodology proposed in [59] reuses patterns generated for earlier versions of the design when there are changes made to the gate level netlist directly because of last-minute Engineering Change Orders (ECO) [60], [27], [54]. To enable the transformation, one of the steps of the procedure from [59] finds a mapping between the inputs and outputs of the earlier and new versions of the design. This step from [59] is not applicable when changes are made at the RTL, and resynthesis produces a new and substantially different gate level circuit, where signal names may have changed, new signals may have been introduced, and signals that existed earlier may have been removed.

The main contribution of the chapter is to compute a mapping between the inputs and outputs after RTL changes and resynthesis produce a new gate level netlist. The proposed methodology is primarily targeted for DFT engineers working in tandem with designers to evaluate the testability of the design early on and catch any design changes that have a negative impact on the overall testability. It reduces the expensive loop turn-around time between DFT and RTL engineers, which in turn affects the overall design cycle time.

The experimental results for industrial circuits showed a 5-fold reduction in test generation time when modifications on average affected 6.34% of the total cones. The gain remains meaningful even with large numbers of changes that, on average affected 22% of the total cones in the circuit. In general, the efficiency of the proposed methodology is expected to decrease when there are significant structural variations between two versions of the design. It is developed for the iterative design flow from Figure 4.1 where changes are expected to create structurally similar circuits.

The chapter is organized as follows. Section 4.2 discusses the motivation for this work, and provides background for the proposed methodology. Section 3.3 presents the proposed methodology for test generation. Section 3.4 presents results for industrial designs followed by conclusions in Section 3.5.

3.2 Motivation and Background

In this section, we discuss the motivation for this work, and provide background for the proposed methodology.

3.2.1 Review

The authors in [59] describe a test generation process based on transformation of patterns of an earlier version of a design into patterns for a new version when changes are made to the gate level netlist. Figure 3.2 shows an overview of the test pattern generation process from [59] which is discussed next.

A signature set is associated with the previous and new versions of the gate level circuit. The signature set contains two integers for every gate, referred to as its input and output signatures. The signatures capture the structure of the input and output cones of the gate. Output signatures are computed by traversing the circuit from inputs to outputs. Using these signature values at the outputs, a traversal from outputs to inputs is performed to compute the input signatures. Every additional logic level causes the signatures to be doubled in every traversal of the circuit. In addition, every gate type has a different contribution to the signatures. As a result, the signatures assigned to the inputs and outputs of the circuit

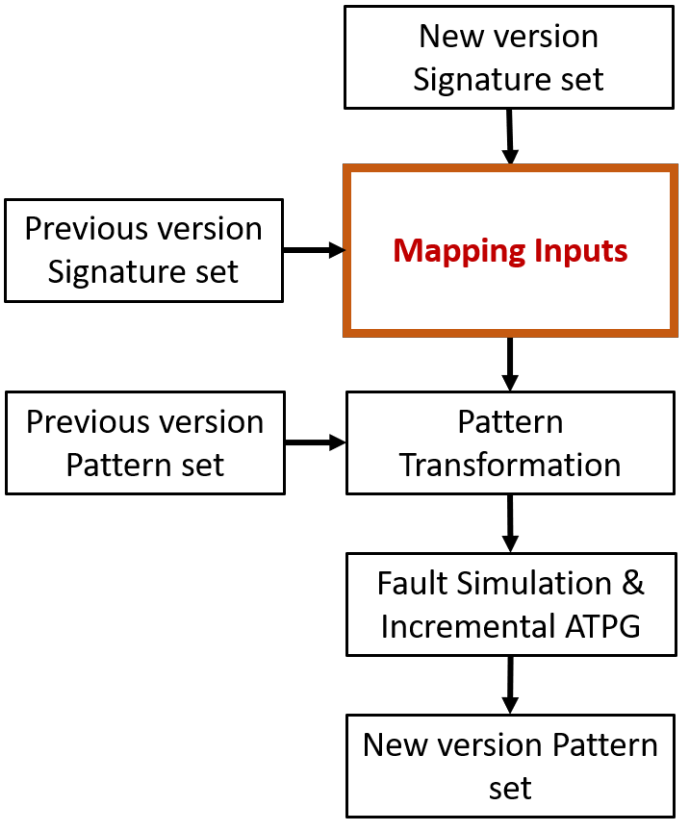


Figure 3.2. Overview of ATPG flow

capture its structure in a unique way. Signatures were found to be useful in other applications as described in [27], [61], [29], [60] and [30].

3.2.2 Mapping between two versions of a circuit

The input and output signatures of the inputs and outputs, respectively, are used for establishing a mapping between the pins of the previous and new versions of the circuit. The mapping algorithm in [59] compares vectors that consist of an output, and its output signature, the inputs of its logic cone, and the corresponding input signatures. A vector of the previous version is considered to match a vector of the new version if all the signatures match. The two logic cones represented by the vectors are identical in this case. Accordingly, the inputs with the same signatures in the two vectors are mapped. This simple approach

to mapping worked well for gate level changes, but it is unable to provide effective mappings after RTL changes, as discussed below.

3.2.3 Mapping between two versions of a circuit

The input and output signatures of the inputs and outputs, respectively, are used for establishing a mapping between the pins of the previous and new versions of the circuit. The mapping algorithm in [59] compares vectors that consist of an output, and its output signature, the inputs of its logic cone, and the corresponding input signatures. A vector of the previous version is considered to match a vector of the new version if all the signatures match. The two logic cones represented by the vectors are identical in this case. Accordingly, the inputs with the same signatures in the two vectors are mapped. This simple approach to mapping worked well for gate level changes, but it is unable to provide effective mappings after RTL changes, as discussed below.

3.2.4 Pattern Transformation and Fault Simulation

In the procedure from [59], the mapping between the inputs is used for transforming the test patterns generated for the previous version into test patterns for the new version of the circuit. Specifically, if an input A1 in the earlier version has a corresponding input B1 in the new version, the values of A1 are copied to B1. Inputs of the new version that do not have corresponding inputs in the earlier version are left unspecified and filled randomly.

Figure 3.3 shows the different steps involved in the transformation of a pattern from the previous version to the new version. Figure 3.3a shows how inputs of the previous version are mapped to the new version. An x in Figure 3.3a indicates that no mapping decision has been made. Figure 3.3b shows one of the patterns of the previous circuit. The steps involved in transforming the pattern in Figure 3.3b to a pattern for the new version are shown in Figure 3.3c.

Fault simulation is carried out using the transformed patterns. Among the undetected faults after fault simulation, test generation is carried out only for those faults that are left undetected in the modified logic cones. A fault may be present in the intersection of

New version	B1	B2	B3	B4	B5	B6	B7
Mapping	A1	x	A4	x	A5	A6	A2

(a) Mapping between two circuits

Circuit 1	A1	A2	A3	A4	A5	A6	A7
Pattern	1	0	0	1	1	1	0

(b) One pattern from previous version of pattern set

Circuit 2	B1	B2	B3	B4	B5	B6	B7
Transform	1	x	1	x	1	1	1
Random fill	1	1	1	0	1	1	1

(c) Transformation of pattern in Figure 3.3b

Figure 3.3. Transformation of a pattern

several cones. In this case, if any one of the cones has been modified, and the fault is not detected after fault simulation, it will be considered during incremental test generation. The combined set of test patterns is used to determine the testability of the new version of the circuit.

3.2.5 Examples of different types of RTL changes

The approach in [59] performs well for changes done at the gate level. However, when changes are made at the RTL, more regions of the circuit are affected. This is because designers use different techniques like clock gating and area minimization during synthesis

to meet the power and area constraints. In addition, timing and boundary optimizations are performed during synthesis for performance improvement. Such changes modify the gate level netlist.

These changes can result in addition or removal of inputs and outputs, and alter their order and names in a modified gate level netlist. With such changes, the mapping algorithm of [59] fails to map the inputs accurately. This reduces the efficiency of the test generation process. It necessitates a different mapping approach that can efficiently handle the changes observed in a circuit when modifications are done at the RTL, and the circuit is resynthesized.

```
50 : sig2 = A1.A2;
 ....
61 : if (A4)
62 :      sig3 = A3;
63 :  else
64 :      sig3 = A2;
 ....
83 : A9 = sig1 + sig2 + sig3 ;
 ....
```

(a) RTL code snippet for the previous version

```
50 : sig2 = A1.A2;
 ....
61 : if (A5)
62 :      sig3 = A4;
63 : else
64 :      sig3 = A3;
 ....
83 : A9 = sig1 + sig2 + sig3 ;
 ....
```

(b) RTL code snippet for the new version

Figure 3.4. RTL code snippet for two versions of the circuit

The following example illustrates how changes to the RTL affect the gate level netlist after synthesis.

Figure 3.4a shows an RTL code snippet in Verilog for a previous version of a design. Figure 3.4b shows an RTL code snippet for a new version of the same design. In Figure 3.4, the RTL code snippet of the new version differs from the previous version, as highlighted in red. Figure 3.5 shows the synthesized circuits for the RTL code snippets shown in Figure 3.4. The inputs A3 and A4 in the previous version, and the inputs with the same names in the new version, do not have identical output cones. The input A5 in the new version replaces input A4 in the previous version as the select input of the multiplexer, and it is

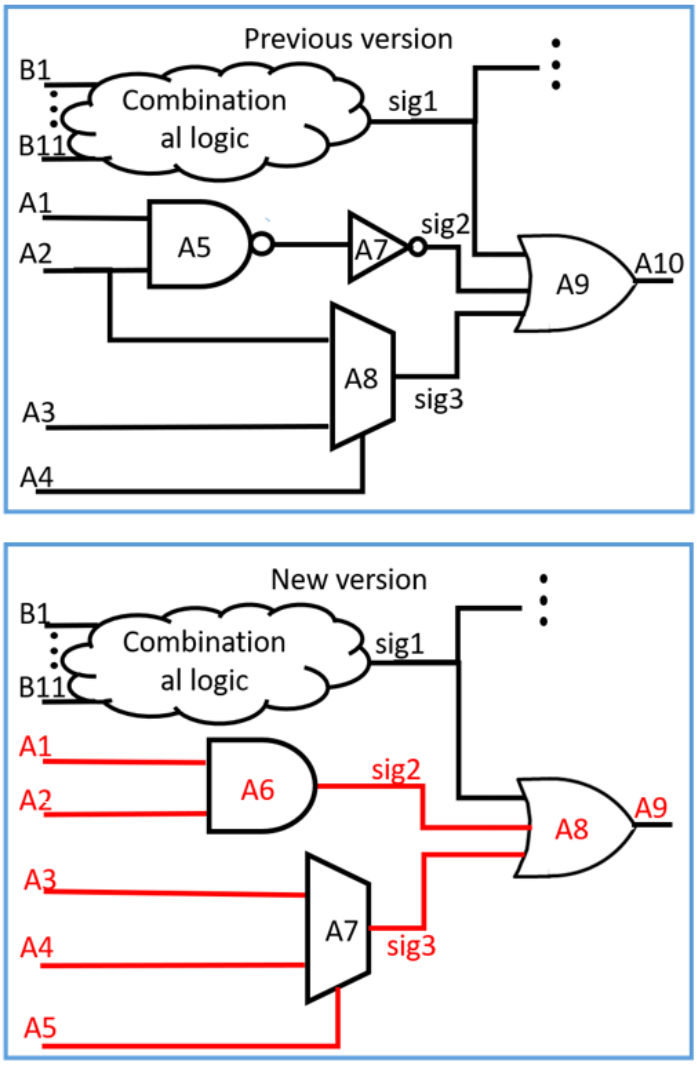


Figure 3.5. An example of a portion of a synthesized circuit before and after a modification

added following A4. A simple mapping of test patterns based on the input names will not be effective in detecting faults in the new version because of these changes.

In addition, RTL designers use complex Verilog constructs, such as System Verilog Interfaces, multi-dimensional array instances, or generate for-loops to instantiate modules that help represent design intent in the RTL. This can result in significant changes in the naming of the signals at the gate level. Designers often use synthesis tool options to incorporate global name changes for the entire design or a subset of Verilog modules, which further

complicates identifying the similarities between two revisions of an RTL design based on their names. For example, experimentation with the name-changing options during synthesis showed up to 70% change in the name of the sequential cells in the synthesized netlist. This shows that the use of names of the pins to map between different versions is not a robust method. A new mapping approach to transform patterns generated from earlier iterations is required, which is described next.

3.3 Proposed Methodology

The proposed methodology follows the process illustrated in Figure 3.2. The key difference is that the mapping of the inputs accommodates the complexities introduced when changes are made at the RTL and the gate level netlist is resynthesized. The use of the signatures is also modified to compute independent input and output signatures, and utilize both input and output signatures for every input and output.

The input to the mapping algorithm is the signatures of the previous and new versions of the gate level netlist. The signatures are given in the form of a list. This list contains a vector for every output. The vector includes the output, the inputs of its logic cone, and both their input and output signatures. Each vector describes a logic cone. Each entry is associated with a pair of signatures, an input and an output signature. The list for the earlier version is denoted by CONES1, and the list for the new version is denoted by CONES2.

3.3.1 Overview

An overview of the proposed mapping algorithm is shown in Figure **??**. The algorithm has three main steps. The first step finds a mapping between the circuits based on the outputs with unique input and output signatures. An input-output signature pair is considered unique when the signature pair appears only once among the signature pairs of all the outputs in the circuit. A unique input-output signature pair at the output of a logic cone shows that the structure of the logic cone is distinct among all the logic cones in the circuit. If the same structure appears only once in the new version, then a mapping between the logic cones is performed.

In the second step, the algorithm finds vectors that have identical input-output signature pairs in both versions of the circuit. It extends the input mapping based on such vectors.

In the third step, the algorithm uses the unmapped outputs with the same input-output signature pairs in both versions. It maps only those inputs whose signature pairs match in both versions, leaving the remaining inputs of the set unmapped. This is called partial mapping of inputs.

The mapping information is stored in an array, denoted by MAP. The array is updated on every iteration of the algorithm. Next, we illustrate the use of unique input-output signature pairs, and describe the three steps of the procedure.

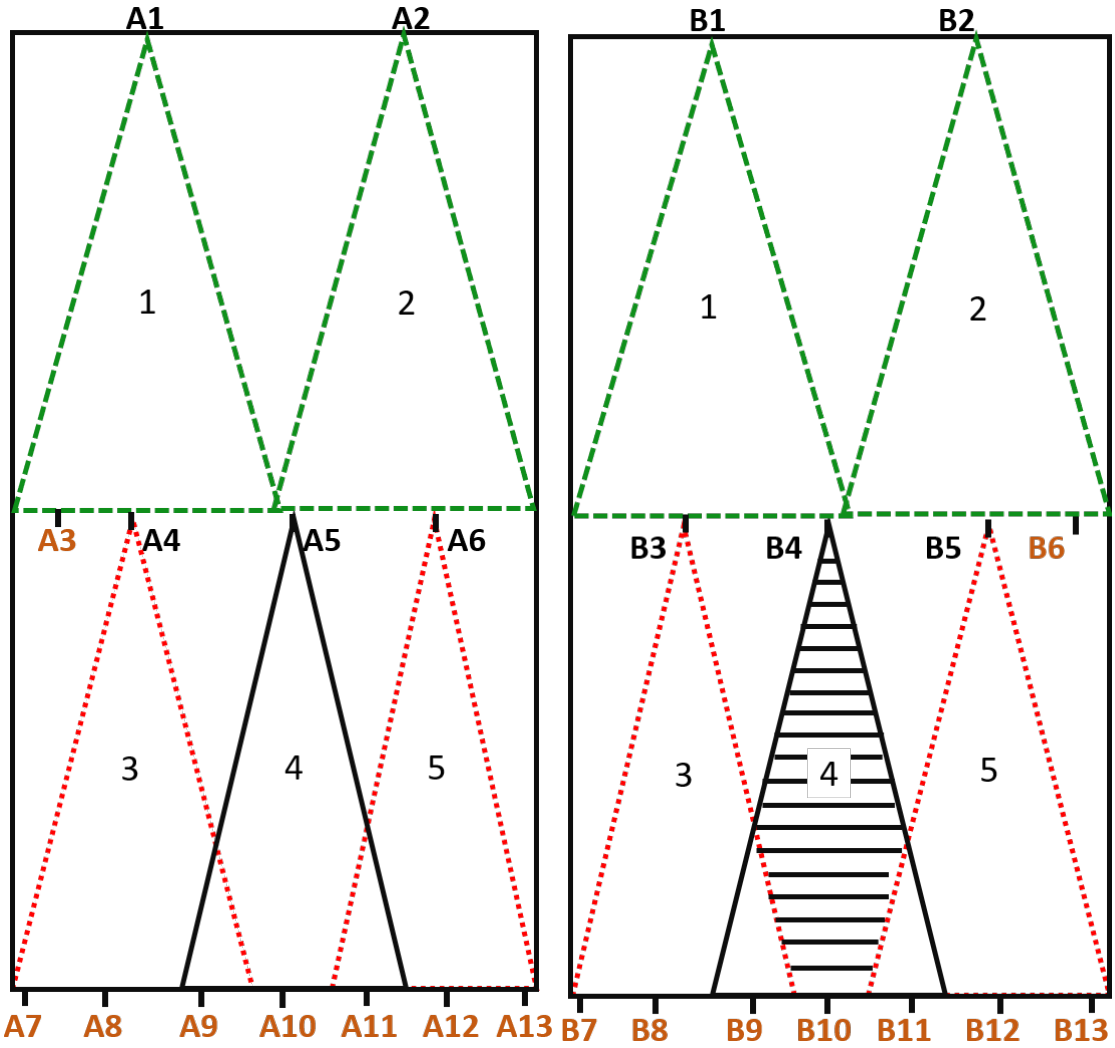
3.3.2 Unique Signature Mapping Illustration

Table 3.1. Input and Output Signatures for Figure 3.6

Previous Version		
Pins	Input signature	Output signature
A1	2293139	2084757
A2	2293139	2084757
A3	6126959	3291791
A4	7158539	4301876
A5	7158539	5182193
A6	6126959	4301876
A7	5857759	3291791
A8	4839217	3291791
A9	8196213	3291791
A10	9402730	3291791
A11	8196213	3291791
A12	4839217	3291791
A13	5857759	3291791

New Version		
Pins	Input signature	Output signature
B1	2293139	2084757
B2	2293139	2084757
B3	6126959	4301876
B4	7158539	3102078
B5	7158539	4301876
B6	6126959	3291791
B7	5857759	3291791
B8	4839217	3291791
B9	8196213	3291791
B10	1723395	3291791
B11	8196213	3291791
B12	4839217	3291791
B13	5857759	3291791

Figure 3.6 shows two versions of the same circuit. The circuit consists of five logic cones and 13 pins in both versions. A1 and A2 of the previous version, and B1 and B2 of the new version, are primary outputs. A3, A7-A13 of the previous version, and B6-B13 of the new version, are primary inputs. A4, A5, and A6 of the previous version, and B3, B4, and B5



(a) Previous version of the circuit (b) New version of the circuit

Figure 3.6. An example to illustrate unique signature mapping

of the new version, are scan cells, and act as both inputs and outputs of logic cones. Table 3.1 shows pairs of input-output signatures for all the inputs and outputs in both versions of the circuit. The primary inputs all have the same output signature, 3291791, and primary outputs have the same input signature, 2293139 in Figure 3.6.

In this example, the modification made to obtain the new version has affected the shaded portion in cone 4, such that the inputs shared by other cones do not see the effect of the modification in their input cones. For the new version, the affected signatures are shown

in red in Table 3.1. Among the five logic cones, A4, A5, and A6 have unique input-output signature pairs in the previous version. Similarly, B3, B4, and B5 have unique input-output signature pairs in the new version.

The first iteration finds that A4 with signature pair (7158539, 4301876) has the same signature pair as B5 in the new version. The corresponding vectors in CONES1 and CONES2 are A7, A8, A9, and B11, B12, B13 with their input-output signature pairs. All the entries of the two vectors have the same input-output signature pairs and can be mapped according to their signatures. In the same way, A6 and B3 have the same signature pairs in their corresponding vectors in CONES1 and CONES2 and can be mapped. Figure 3.7 shows the progress of MAP with every iteration for the first step of unique signature mapping. The first row shows the initial MAP, where an x indicates that no mapping decision has been made. The second row shows the mapping after the outputs A4 and B5 are considered. The third row shows the mapping after considering the outputs A6 and B3. In this way, the algorithm distinctly identifies logic cones with unique structures in two versions of the circuit.

New version	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13
Initial	x	x	x	x	x	x	x	x	x	x	x	x	x
Iteration1 B5-A4	x	x	x	x	**A4**	x	x	x	x	x	**A9**	**A8**	**A7**
Iteration2 B3-A6	x	x	**A6**	x	**A4**	x	**A13**	**A12**	**A11**	x	**A9**	**A8**	**A7**

Figure 3.7. Mapping information for different iterations in MAP

3.3.3 Mapping Based on Unique Signature Pairs

The first step of the mapping algorithm establishes a mapping between inputs and outputs of logic cones that have distinct input-output signature pairs. Every pair of outputs with unique and equal signature pairs is added to the queues Q1 and Q2. An iteration considers the first outputs in the queues Q1 and Q2. It compares the input-output signature pairs of all the inputs in the respective vectors. If all the signatures are equal, the outputs and inputs

of these logic cones are mapped. They are marked as matched and are added to the queues for further mapping of their respective vectors. In this way, the algorithm identifies logic cones with distinct structures in both circuits and aids in the first step towards accurate mapping between the circuits.

Algorithm 1.1 MapUniqueSignature

Input: CONES1, CONES2

Output: Unique Mapping Information

1: Mark all inputs and outputs as unmatched
2: Empty sets U1 and U2
3: **for** every output of previous version **do**
4: **if** input-output signature pair is unique **then**
5: Store the corresponding output position from CONES1 in U1
6: **end if**
7: **end for**
8: **for** every output of new version **do**
9: **if** input-output signature pair is unique **then**
10: Store the corresponding output position from CONES2 in U2
11: **end if**
12: **end for**
13: Empty queues Q1 and Q2
14: **for** every output in U1 **do**
15: **if** U2 contains an output with the same input-output signature pair **then**
16: Insert outputs from U1 and U2 into Q1 and Q2, respectively.
17: **end if**
18: **while** Q1 and Q2 are not empty **do**
19: Pop Q1 to pos1 and pop Q2 to pos2
20: **if** input-output signature pairs of every element in the vectors corresponding to the outputs at pos1 and pos2 are equal **then**

21: Map the unmatched elements in vectors corresponding to pos1 and pos2 according to their input-output signature pairs
22: Mark the elements in vectors corresponding to pos1 and pos2 as matched
23: Add matching inputs from vectors of pos1 and pos2 to Q1 and Q2, respectively
24: **end if**
25: **end while**
26: **end for**
27: **End**

3.3.4 Mapping Common Signature Pairs

The second step of the mapping algorithm establishes a mapping between the unmapped outputs with structurally identical output cones in both versions of the circuit. A mapping is done when the input-output signature pairs of two vectors in both versions of the circuit match, but they are not unique. In this case, the pairs of outputs are selected arbitrarily. The use of unique signature pairs reduces the number of inputs that are matched by Algorithm 1.2, reducing the effects of arbitrary selection.

Algorithm 1.2 MapCommonSignature

Input: CONES1, CONES2

Output: Mapping Information

1: **for** every unmapped output O1 of previous version **do**
2: **for** every unmapped output O2 of new version **do**
3: **if** Input-output signature pairs in the vector for O1 are equal to those of output O2 **then**
4: Insert the output position of O1 and O2 in Q1 and Q2, respectively
5: **end if**
6: **end if**
7: **while** Q1 and Q2 are not empty **do**
8: Pop Q1 to pos1 and pop Q2 to pos2

9: **if** input-output signature pairs of every element in the vectors corresponding to output at pos1 and pos2 are equal **then**
10: Map the unmatched elements in vectors corresponding to pos1 and pos2 according to their input-output signature pairs
11: Mark every element in vectors corresponding to pos1 and pos2 as matched
12: Add matching inputs from vectors of pos1 and pos2 to Q1 and Q2, respectively
13: **end if**
14: **end if**
15: **end while**
16: **end for**
17: **end for**
18: **End**

3.3.5 Mapping for the Remaining Unmapped Outputs

The third step establishes a partial mapping of the inputs of the logic cones that have not been matched yet. A complete matching of inputs is not possible due to the structural variations in the gate level netlist from the modifications at the RTL.

Algorithm 1.3 PartialMap

Input: CONES1, CONES2

Output: Mapping Information

1: **for** every unmapped output O1 of previous version **do**
2: **for** every unmapped output O2 of new version **do**
3: **if** input-output signature pair of O1 is equal to input-output signature pair of O2 **then**
4: **for** every unmatched $input_i$ in vector of O1 **do**
5: **for** every unmatched $input_j$ in vector of O2
6: **if** input-output signature pairs of $input_i$ and $input_j$ are equal **then**
7: Map $input_i$ to $input_j$

```
8:                Mark input_i to input_j as matched
9:              end if
10:             end if
11:           end for
12:         end for
13:       end if
14:     end for
15: end for
```

Algorithms 1.1, 1.2, and 1.3 form the mapping algorithm that finds a correspondence between the inputs and outputs of two versions of the design.

3.4 Experiment and Results

In the experiment described next, ten industrial designs are used to evaluate the performance of the proposed algorithm for test generation. The evaluation was performed using single stuck-at faults [62]. A compacted test pattern set was obtained using a commercial ATPG tool for each design. Different versions of the same design were obtained by introducing changes into the RTL and resynthesizing the design. Changes were introduced by a commercial tool at random locations in the design. Modifications resulted in the addition or removal of logic gates and changed the number of sequential elements. These changes emulate modifications done at the RTL due to the iterative nature of the design flow.

For each design considered as the previous version, ten new versions were obtained. A commercial synthesis tool optimizes the circuit differently every time a modification is done at the RTL. The optimization can result in an increase or decrease in the total number of gates in the new version compared to the previous version.

Table 3.2 shows the average change in combinational and sequential logic in a circuit when modifications are done at the RTL. The results shown in Table 3.2 use the absolute value of change in the number of gates in the new version compared to a previous version. Column "Size" of Table 3.2 gives the total number of gates present in the design, which is considered as the previous version of the circuit. Columns "Number of Gates" and "Number of Scan

Table 3.2. Average Combinational and Sequential Changes

	Previous Version	Average Change in New Version	
	Size	Number of Gates	Number of Scan Cells
d1	1,521,890	4,276	6
d2	968,281	2,366	5
d3	5,266,903	1,585	5
d4	2,369,026	1,604	6
d5	6,411,557	1,474	6
d6	1,121,605	1,230	4
d7	1,089,511	10,527	4
d8	1,492,011	2,159	5
d9	3,836,692	9,935	5
d10	735,616	747	6

Cells" show the average number of gates and scan cells, respectively, added or removed in a gate level netlist of the new version after modifying the RTL. An average of 5 sequential elements and thousands of gates are added in all the versions of the circuit. The results show that a modification at the RTL changes the logical and sequential portion of the circuit in the resynthesized gate level netlist.

Table 3.3. Experimental Result for Circuits Modified at RTL

	Tot Cone in Prev Version	**Diff Cones in New Version**	**RunTime (seconds)**			**Test Patterns**			**Fault Coverage(%)**	
			Previous Version	**New Version**	**Gain (Ratio)**	**Previous Version**	**New Version**	**Increase (%)**	**Previous Version**	**New Version**
d1	113,172	6,756	512	113	4.53	8,751	11,290	29.01	94.76	94.76
d2	48,162	5,309	399	51	7.82	4,656	6167	32.45	93.41	93.41
d3	255,025	3,818	1,892	433	4.37	5,682	6,758	18.94	95.63	95.63
d4	119,069	13,146	481	76	6.33	2,874	3,642	26.72	96.61	96.69
d5	294,863	3,847	1,198	245	4.89	5,320	5896	10.82	94.12	94.13
d6	52,931	3,296	321	56	5.73	4,519	4,767	5.49	94.76	94.76
d7	44,116	3,706	389	95	4.09	2,824	3,405	20.46	92.31	92.31
d8	86,854	4,920	312	62	5.03	2,461	3,251	32.10	93.72	93.71
d9	224,977	8,998	2,597	494	5.25	9,240	11,741	27.06	94.51	94.51
d10	37,337	3,129	386	104	3.71	1,856	2,217	19.45	98.55	98.55
avg					5.17			22.25		

Each row in Table 3.3 corresponds to one of the 10 industrial circuits. The average of several parameters for all ten versions of the circuit is shown in each row. These results are shown under the sub-column New version. Sub-column Previous Version shows the results for the previous version (the original design) in the respective categories. Column “Total Cone in Prev Version” shows the total number of logic cones present in the previous version of the design. Column “Diff Cones in New Version" shows the number of output cones affected compared to the previous version when the RTL is modified, and the circuit is resynthesized. Column Run Time shows the total time taken in seconds for test pattern generation. For the previous version, test generation is run for all the faults starting from an empty test pattern set. The overall runtime for a new version includes the runtime for computing the signatures, mapping between two versions, fault simulation, and running the ATPG to detect the remaining faults. The ratio of the runtime of the previous version to the average of the runtime of the new version is shown in sub-column “Gain”. The total number of test patterns required to achieve the fault coverage is shown in column Test Patterns. The percentage increase in the number of test patterns in the new version compared to the previous version is shown in sub-column “Increase (%)". Column Fault coverage shows the fault coverage of the previous version and the average fault coverage of the ten different versions of the circuit after modification at RTL.

From column “Diff Cones in New Version" in Table 3.3, it can be seen that modifying the RTL and resynthesizing the gate level affects 1.4% to 11.02% of the total number of logic cones in the ten industrial circuits. On average, for all ten designs, the modifications result in 6.34% of the total cones of the new version that are not similar to the previous version. Between the ten versions of a design, the number of logic cones structurally different in the new version compared to the previous version of the circuit directly affects the time taken for detecting faults in the new version. A lower number of structurally different logic cones results in more inputs matching between two versions of a design. With more inputs accurately mapped, more faults will be detected in the new version after transforming the test patterns from a previous version.

The total time taken to detect the faults of a new version is reduced by using the transformed test patterns generated for the previous version. The reduction in runtime occurs

because test pattern generation is performed on a small set of faults. An average of 5-fold reduction in runtime is observed in the new version compared to the previous version for all the industrial circuits considered to obtain the same fault coverage. In addition, with higher structural similarity, the algorithm maps the inputs and outputs between two versions more efficiently, which results in a higher reduction in overall runtime.

There is an average of 22.25% increase in number of test patterns as the algorithm does not attempt to optimize the number of patterns required. Its goal is to assess the testability of a circuit by checking the fault coverage. The increase in number of test patterns can be mitigated by performing ATPG again starting from an empty test set after the design has converged.

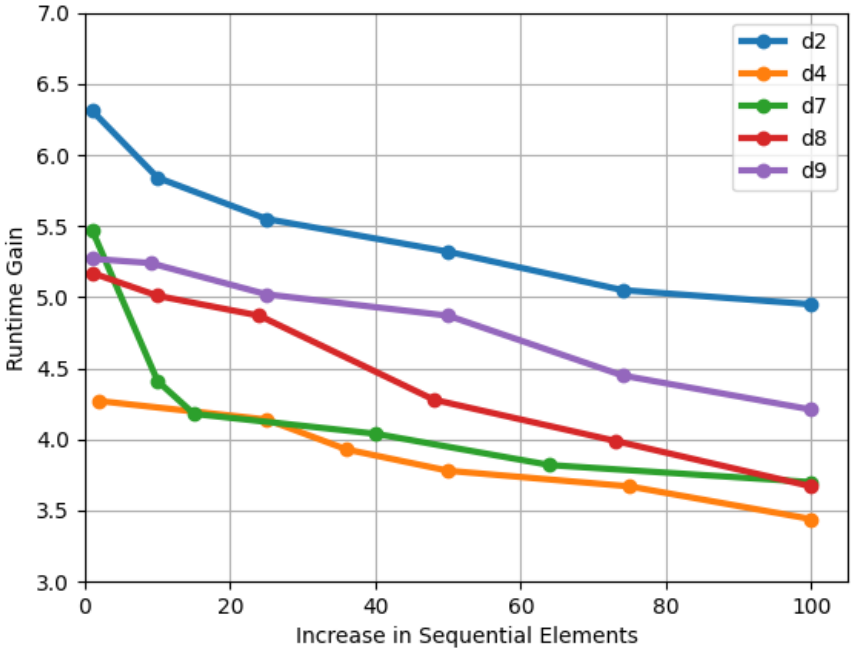


Figure 3.8. Runtime gain as a function of the number of changes

We also experimented with larger numbers of changes introduced into every version of the circuit, considering five of the designs. The larger number of changes resulted in larger numbers of sequential elements as well as larger numbers of gate changes. The results are described by Figure 3.8 that shows the runtime gain as a function of the increase in the

number of sequential elements. Figure 3.8 demonstrates that the efficiency of the proposed methodology decreases monotonically with an increase in the number of changes, but the improvement in runtime remains high even when a large number of changes are made.

3.5 Conclusion

This chapter described a mapping methodology to reuse the patterns generated in an earlier iteration of the design process when a circuit is resynthesized due to changes at the RTL. The proposed mapping procedure addresses the complexities introduced when changes are made at the RTL. This is achieved by first identifying and mapping logic cones that have distinct structures in both versions of the circuit. This is followed by mapping the inputs and outputs of all other structurally identical and structurally similar cones. The mapping obtained is used to transform the patterns, simulate faults, and perform test pattern generation for the remaining undetected faults. An experiment was performed on ten industrial designs where different versions of a design were obtained by changing the RTL. The experiments showed an average of 5-fold reduction in runtime compared with running the test pattern generation from the beginning.

4. GENERATION OF TWO-CYCLE TESTS FOR STRUCTURALLY SIMILAR CIRCUITS

VLSI design flows improve design parameters (performance, power, area, and testability) iteratively. Whereas the "shift left" trend implies that changes at the RTL are preferred for improving the design, it is sometimes necessary to make gate-level changes, e.g., because of layout changes or ECO. In an iterative design flow, repeated ATPG to evaluate the testability of a design after design changes have been made creates a bottleneck. The goal of this chapter is to address this bottleneck considering two-cycle tests for transition faults. The test generation procedure described in the chapter transforms a LOC test set generated for an earlier version of the design into a LOC test set for a new version without repeating the entire test generation process. To enable the transformation, it is necessary to find a mapping between the inputs and outputs of the earlier and new versions of the design, taking into consideration that RTL resynthesis may produce a new gate-level netlist, with new signal names and different input and output orders. To address two-cycle tests, the mapping is performed over two time-frames of the design. Experimental results for industrial circuits with changes made at the RTL as well as gate-level demonstrate significant runtime gains with the test generation procedure described in this chapter.

4.1 Introduction

Developments in silicon fabrication processes support higher transistor densities while maintaining chip size [33], [34], [55]. Many steps are involved in designing a complex and dense circuit [56],[57], [58]. As shown in Fig. 4.1, the physical design flow is iterative. This allows incremental changes to be made to the design to fix errors and meet the necessary design requirements related to performance, power, area, and testability. The testability of

a circuit is evaluated by running automatic test generation (ATPG)[39], [40], [41], [42], [63] during the iterative design process to compute the fault coverage. If the fault coverage is not sufficiently high, incremental changes are made to improve the testability of the circuit.

Whereas many of the physical design steps are performed at the RTL [64], ATPG is run at the gate-level. Incremental changes to improve the design can also be made at the RTL or at the gate-level [54]. Making changes at the RTL is consistent with the industry "shift-left" trend [64] to make most of the design decisions and adjustments at the RTL. The gate-level circuit may go through ECO [60], [27], [65],[66], or layout changes that modify the combinational or sequential logic [67]. These modifications may add or remove a module or modify the logic within a module. Changes to the gate-level description of the design can be made such that they would affect a single or a small number of logic cones without otherwise disturbing the structure of the circuit. Changes made at the RTL require resynthesis to produce a modified gate-level description. After resynthesis, in addition to changes to the logic, the design may have more or fewer inputs and outputs, and the names of the inputs and outputs, as well as their order, may change.

Since the effects of incremental changes on the gate-level description are unpredictable, current solutions run the complete ATPG process every time the design is modified during the incremental design process and its testability needs to be evaluated. The simple solution of performing ATPG once and using the same test set throughout the design process is not sufficient since the test set achieves a low fault coverage, especially after the number and order of the inputs and outputs have changed.

A methodology for speeding up ATPG for testability evaluation that fits with the design flow of Fig. 4.1 was described in [59] and [68]. These works rely on the fact that the circuits before and after design changes are structurally similar. The ATPG process described in [59] addresses the simpler scenario where design changes are made at the gate-level. The ATPG process described in [68] is suitable when design changes are made at the RTL and the gate-level circuit is resynthesized.

An overview of the methodology described in [59] and [68] is shown in Fig. 4.2. The first step analyzes the structural similarity between the earlier and new versions of the design. The second step finds a mapping between the inputs and outputs of the two circuits. Two

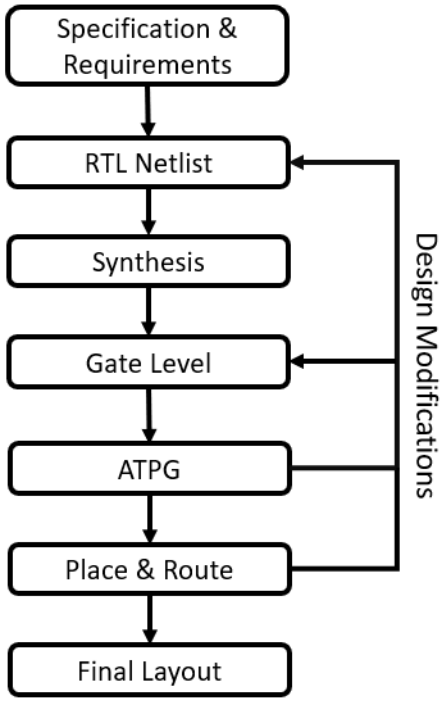


Figure 4.1. VLSI Design Flow [56],[57], [58]

inputs (outputs) are mapped when their logic cones are structurally similar. The pattern transformation step transforms the patterns generated for the earlier version into patterns for the new version using the mapping between their inputs. Fault simulation is carried out next for the new version, followed by incremental ATPG to detect faults in the new version that are not detected by the transformed patterns. Since the transformed patterns detect all the faults in the logic cones of the new version that have not changed, the overall runtime is reduced significantly compared with running the entire ATPG process for the new version.

To perform the mapping between the inputs and outputs of the earlier and new versions of the design, the procedures described in [59] and [68] capture the structure of a logic cone by performing two traversals of the circuit, from inputs to outputs, and from outputs to inputs. Integer arithmetic performed during each traversal associates two integers with every line. The integer computed during the forward traversal is referred to as the output signature, and the integer computed during the backward traversal is referred to as the input signature. During the traversals, the integers capture the types of the gates encountered, their fanin or

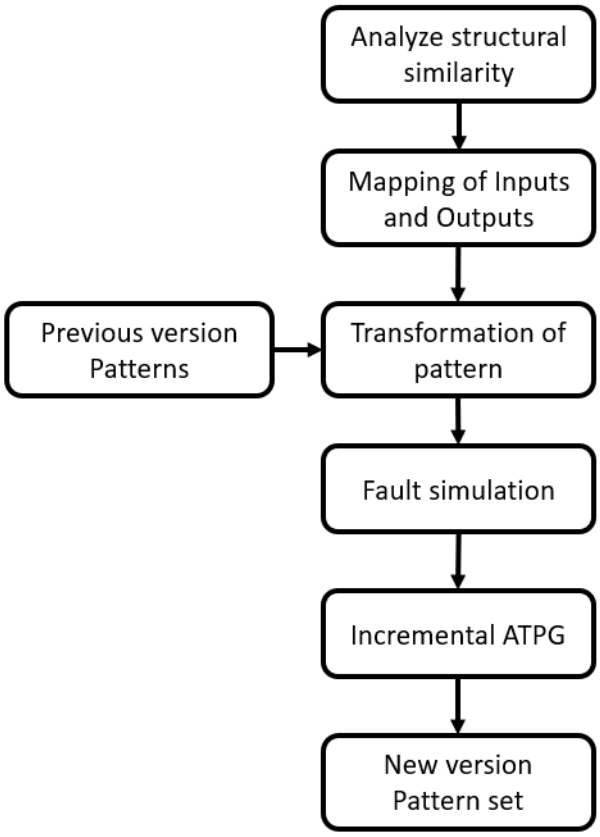


Figure 4.2. Overview of the ATPG Flow [59],[68]

fanout, and the number of levels traversed. As a result, logic cones with the same structure will have the same signatures on their inputs and outputs. The mapping procedure uses this fact to match the inputs and outputs of the earlier and new versions of the design.

Experimental results presented in [59] and [68] demonstrate the importance of an accurate mapping between the inputs and outputs of the earlier and new versions of the design. With an accurate mapping, the runtime gains reported in [59] and [68] are 11x and 5x on average for industrial designs. The difference between [59] and [68] is in the mapping algorithm. The algorithm in [68] accommodates unique signatures and partial mapping to address the more extensive changes introduced by the resynthesis of the RTL description. When the mapping is not sufficiently accurate, the transformed patterns of the earlier version achieve a lower

fault coverage for the new version, and the runtime for incremental ATPG is higher. The runtime gain in this case is lower.

The overall process in Fig. 4.2 is suitable for any fault model. Single stuck-at faults are considered in [59] and [68] under single-cycle tests. Experimental results using transition faults and two-cycle LOC tests show that the mapping from [59] or [68] is not sufficiently accurate when ATPG generates two-cycle tests targeting transition faults [69], [70], [71]. An analysis of the results shows that the source of the problem is the signatures. The signatures in [59] and [68] consider a single time-frame of the circuit. An accurate mapping of two-cycle tests for transition faults requires signatures that capture two time-frames of the circuit. Such signatures are introduced in this chapter.

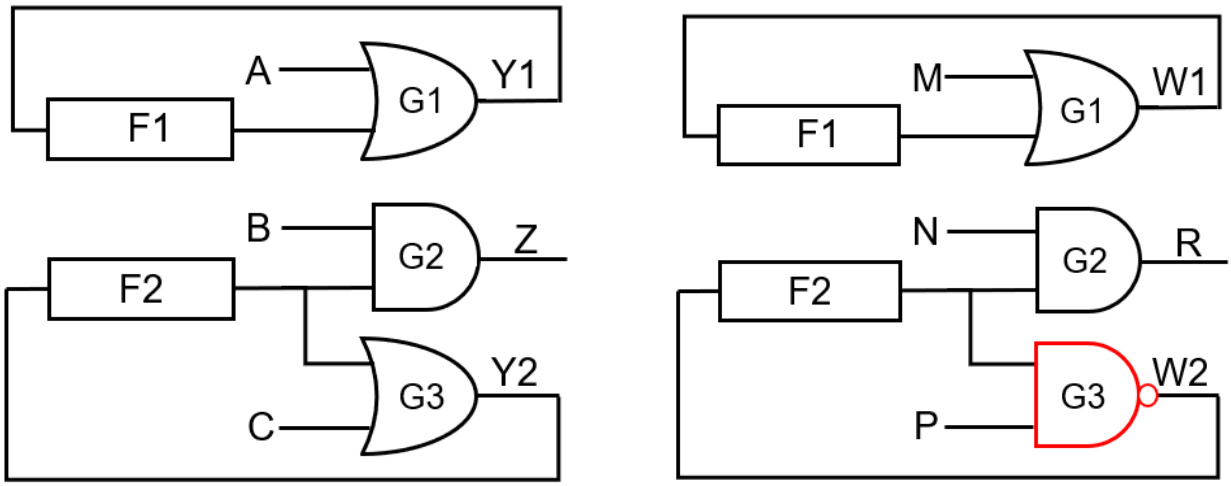
The methodologies proposed for both single-cycle tests and two-cycle tests are for incremental changes that affect a smaller percentage of the logic cones in the design.

The chapter is organized as follows. Section 4.2 explains the need for signatures computed over two time-frames referred to as two-cycle signatures. Section III describes the computation of the two-cycle signatures. Section IV provides additional details of the test generation process from Fig. 4.2 in the context of two-cycle tests. Section V presents experimental results for industrial designs. Section VI concludes the chapter.

4.2 Motivation for two-cycle signatures

This section discusses the need for two-cycle signatures using an example.

Fig. 4.3a shows a circuit with three primary inputs, A, B, and C, three gates G1, G2, and G3, a primary output Z, and two scan elements, F1 and F2. The present-state variable associated with scan element F1 (F2) is y1(y2), and the next-state variable is Y1 (Y2). The combinational logic of the circuit has five inputs, y1, y2, A, B, and C, and three outputs, Y1, Y2, and Z. The example will focus on the input cones of these outputs (the logic cones driving them) and their output signatures. The same discussion applies to the inputs of the combinational logic, the logic cones they drive (their output cones), and their input signatures.



(a) Earlier version of the circuit

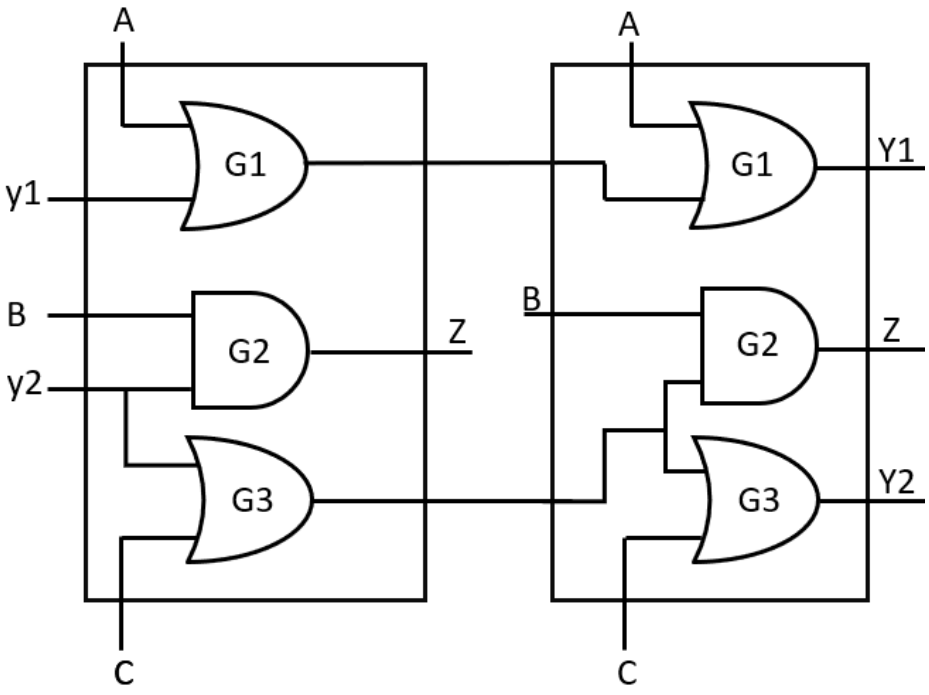
(b) New version of the circuit

Figure 4.3. Two versions of a circuit

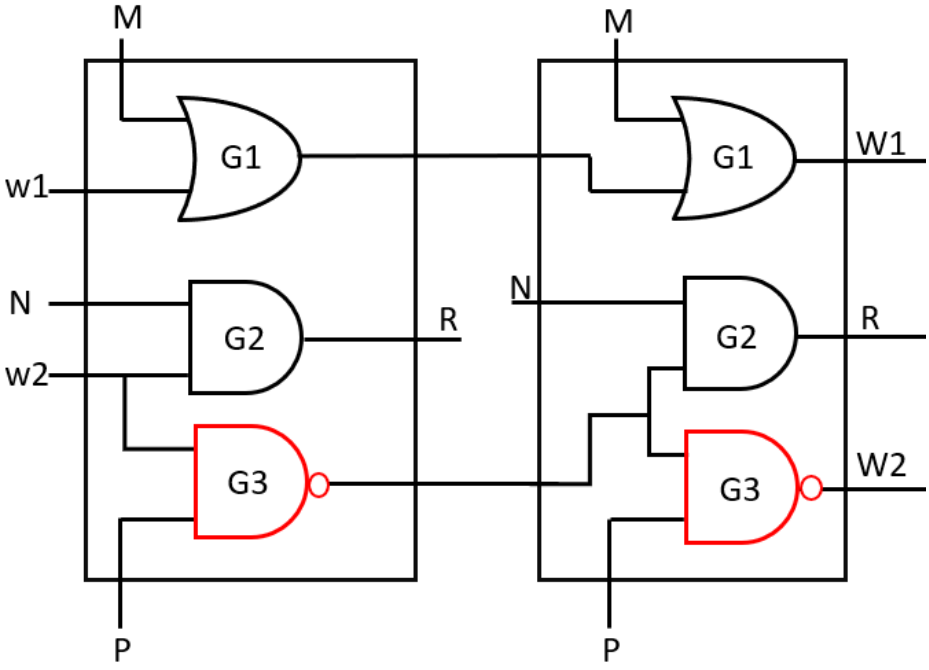
Fig. 4.3b shows the circuit after a design change where the type of G3 was changed from OR to NAND, and the names of the inputs and outputs were changed during resynthesis. An analysis of a single time-frame of the circuit shows that the input cones of Y1 and W1 are the same, and the input cones of Z and R are the same. The input cones of Y2 and W2 are different because of the change in the type of G3. The single-cycle signatures from [59] and [68] will produce the same signatures for Y1 and W1, the same signatures for Z and R, and different signatures for Y2 and W2. As a result, the mapping will associate W1 with Y1, and R with Z.

With two-cycle tests applied over two time-frames, it is important to analyze the circuit over two time-frames. The two time-frame expansion of the circuit from Fig. 4.3 is shown in Fig. 4.4. Considering the circuit that consists of two time-frames, the logic cone of Z includes G3 in the first time-frame, and G2 in the second time-frame. The same applies to R. In this case, the input cones of Z and R are different because of the change in the type of gate G3.

When the single-cycle signatures cause the test generation procedure to assume that the input cones of R and Z are the same, the mapping between the circuits of Fig. 4.3a and



(a) Earlier version



(b) New version

Figure 4.4. Two time-frame expansion

4.3b will be inaccurate. In addition, faults in the input cone of R will be dropped from consideration, assuming that the faults are already detected by the test set for the circuit from Fig. 4.3a. The inaccurate mapping will increase the run time for incremental test generation, and the removal of target faults from consideration may result in a loss of fault coverage.

For example, considering the design referred to in Section V as c5, test generation for transition faults yields a transition fault coverage of 88.34%. With one of the changes made to the design c5 at the RTL, the single-cycle signatures indicate that 2,432 input cones are affected by the change, and the remaining 110,747 input cones are not affected. After dropping from consideration the faults in the input cones that appear to be unaffected, incremental test generation for the remaining faults yields a reduced transition fault coverage of 85.12%. The two-cycle signatures reveal that 3,953 input cones are affected. The transition fault coverage after incremental test generation is 88.34%.

Two points are important to note. (1) Dropping of unaffected faults from consideration is important for the efficiency of the procedure. (2) Although fewer faults are dropped from consideration under incremental ATPG when more cones are found to be affected, the runtime gain is still high with higher fault coverage.

The next section describes the computation of the two-cycle signatures that will allow faults in unaffected input cones to be dropped from consideration.

4.3 Two-cycle Signature Computation

This section describes the computation of the two-cycle input and output signatures. The computation yields input and output signatures for the inputs of the combinational logic in the first time-frame, and the outputs of the combinational logic in the second time-frame. These signatures will be used for mapping of the inputs and outputs. The computation uses randomly selected prime numbers in the range between 1 and 15 million.

Output signatures are computed during a traversal of the circuit from the inputs of the first time-frame to the outputs of the second time frame. For initialization, all the primary inputs in the first and second time-frame are assigned a random prime number R1, and all

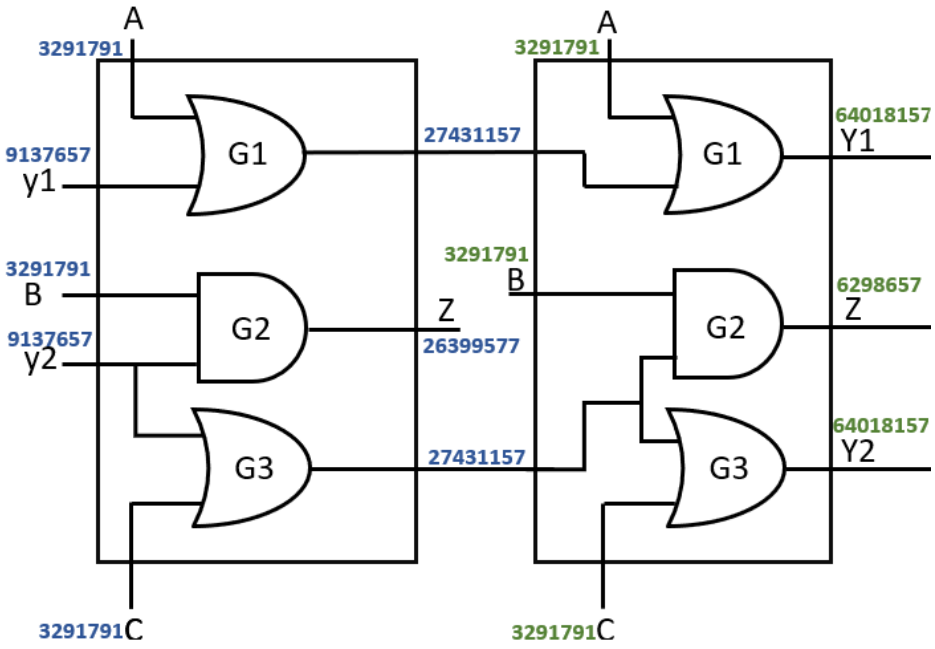
the present-state variables in the first time-frame are assigned a random prime number R2. After assigning a value to a next-state variable Yi in the first time-frame, the value is copied to the present-state variable yi in the second time-frame.

The input to output traversal captures the structure of the input cones as follows. To capture the gate types in an input cone, each gate type is associated with a random prime number R_{TYPE}, where TYPE is the gate type. To capture the number of levels, signatures are rotated left twice at every level. This causes the signature values to approximately quadruple with every additional level. The computation of the signatures is given by algorithm 1.

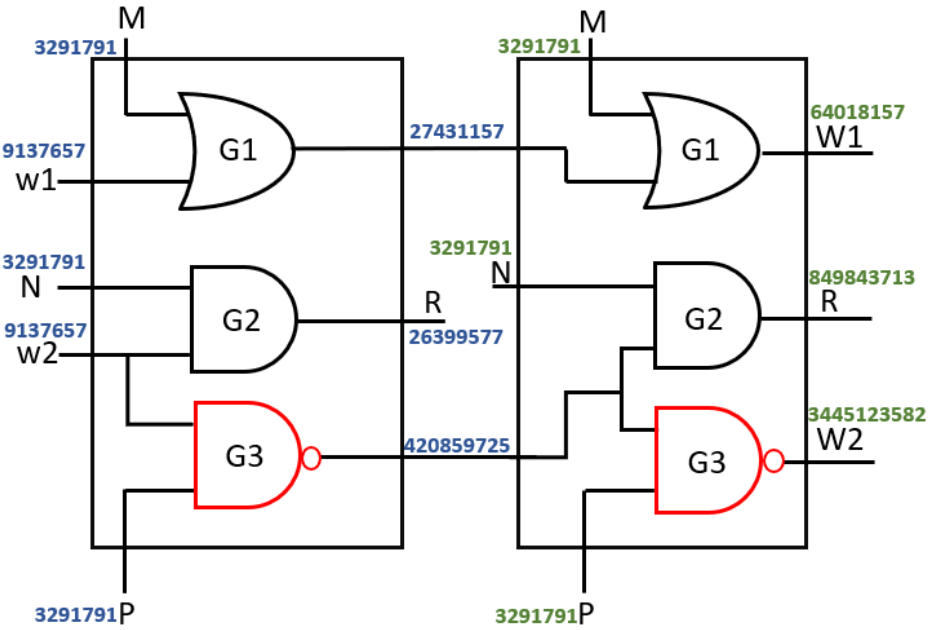
Algorithm 1 Output Signatures

1: **for** time-frame = 1 and 2
2: Initialize the primary inputs to a random prime number R1.
3: **if** time-frame = 1
4: initialize the present-state variables →a random prime number R2.
5: **end if**
6: **else for** every present-state variable yi
7: Copy the output signature of Yi in time-frame 1 to the output signature of yi in time-frame 2.
8: **for** every gate G_i considering the gates from inputs to outputs
9: Initialize the output signature of G_i to 0.
10: **for** every fan-in $H_{i,j}$ of G_i
11: Rotate the output signature of $H_{i,j}$ left and add it to the output signature of G_i.
12: **end for**
13: **for** G_i of type TYPE, add R_{TYPE} to the output signature of G_i.
14: Rotate the output signature of G_i
15: **if** G_i is an inverting gate
16: Complement its output signature
17: **end if**
18: **end for**
19: **end for**

END



(a) Earlier version



(b) New version

Figure 4.5. Example of output signature computation over two time-frames

Fig. 4.5 shows the output signature computation over two time-frames for the earlier and new version of the design from Fig.4.3. The new version is considered next. In time-frame 1, the output signature of primary inputs M, N and P are initialized to 3291791, and the output signatures of the present-state variables w1 and w2 are initialized to 9137657. The output signatures are computed according to algorithm 1. The computed output signature of W1 in time-frame 1 is 27431157, for W2 it is 420859725 and for R it is 26399577. For time-frame 2, the output signatures of M, N and P are again initialized to 3291791. The output signatures of W1 and W2 in time-frame 1 are copied to the output signatures of w1 and w2 in time-frame 2, respectively. Using these initial values in time-frame 2, the output signatures are computed for W1, W2, and R.

For the earlier version of the design, the output signatures obtained are 64018157 for Y1 and Y2, and 6298657 for Z. Comparing with the output signatures obtained for W1, W2 and R in the new version of the design, the mapping algorithm will be able to identify that the input cones of Y1 and W1 are the same, but the input cones of Z, Y2, R and W2 are different, and they will not be mapped.

Algorithm 2 computes the input signatures for two time-frames.

Algorithm 2 Input Signatures

1: **for** time-frame = 2 and 1
2: Initialize the primary outputs to a random prime number R3.
3: **if** time-frame = 2
4: Initialize the next-state variables to a random prime number R4.
5: **end if**
6: **else for** every next-state variable Yi
7: Copy the input signature of yi in time-frame 2 to the input signature of Yi in time-frame 1.
8: **for** every gate G_i considering the gates from outputs to inputs
9: Initialize the input signature of G_i to 0.
10: **for** every fan-out $H_{i,j}$ of G_i
11: Rotate the input signature of $H_{i,j}$ left and add it to the input signature of G_i.

12: **end for**

13: **for** G_i of type TYPE, add R_{TYPE} to the input signature of G_i.

14: Rotate the input signature of G_i

15: **if** G_i is an inverting gate

16: Complement its input signature

17: **end if**

18: **end for**

19: **end for**

END

Single-cycle signatures are obtained by considering only time-frame=1 in Algorithm 1 and time-frame=2 in Algorithm 2. Experimental results presented in [59] and [68] demonstrate the accuracy of the single-cycle signatures in capturing variations in the structures of the logic cones that require new single-cycle tests to be computed.

4.4 Test Generation Procedure

The test generation procedure outlined in Fig. 4.2 is described in this section.

4.4.1 Mapping of Inputs and Outputs

For the purpose of mapping the inputs and outputs of the earlier and new versions of the design, each output of the second time-frame of each design is associated with a vector describing its input cone. The input cone spans the two time-frames of the circuit. The vector includes the inputs of the first time-frame, as well as primary inputs from the second time-frame, that drive the output.

For example, for R in Fig. 4.4, the vector includes primary input N from the second time-frame, and P and w2 from the first time-frame. No distinction is drawn between primary inputs from the first time-frame, and primary inputs from the second time-frame. This is consistent with the use of two-cycle tests with primary input vectors that do not change between the first and second clock cycles.

Considering a vector for an input cone, each one of the inputs, as well as the output, is associated with an input signature from the first time-frame and an output signature from the second time-frame. Algorithms 1 and 2 yield the signatures shown in Table 4.1 for all the inputs and outputs of both versions of the design in Fig. 4.4. Table 4.1 lists all the outputs of the earlier and new versions of the design and the vectors describing their input cones. In Table 4.1, some signatures are in bold in the new version to signify that these signatures are different from the ones obtained for the earlier version.

Table 4.1. Input and Output Signatures for Figure 4.4

Version	Output	(Input, Output) Signature pair	Input vector	(Input,Output) signature vector
Earlier	Z	(2293139, 6298657)	B, C, y2	(9418750, 3291791),(26705936, 3291791), (81350657, 64018157)
	Y1	(36000937, 64018157)	A, y1	(30155071, 3291791), (36000937, 64018157)
	Y2	(81350657,64018157)	C, y2	(26705936, 3291791), (81350657,64018157)
New	R	(2293139, **849843713**)	N, P, w2	(9418750, 3291791), (**429098929**, 3291791), (**51125109, 3445123582**)
	W1	(36000937,64018157)	M, w1	(30155071, 3291791), (36000937, 64018157)
	W2	(**51125109,3445123582**)	P, w2	(**429098929**, 3291791), (**51125109, 3445123582**)

Mapping consists of three steps. The first step uses unique vectors of input and output signatures. To call a vector unique, it should appear only once for the circuit it belongs to (the earlier or new version of the design). In Table 4.1, all the vectors are unique. If a unique vector of the earlier version and a unique vector of the new version are equal, except for the order of inputs that may be different, all the inputs and outputs they contain are matched. In Table 4.1, the signatures for the input cone of W1 in the new version match completely with the signatures for the input cone of Y1 in the earlier version. This allows us to match input M of the new version with A of the earlier version, and w1 of the new version with y1 of the earlier version. The same matching would occur if the order of M and w1 (or A and y1) had been reversed.

The second mapping step considers pairs of vectors of the earlier and new versions that contain identical signatures but are not unique. The procedure selects pairs of identical vectors arbitrarily. It then maps their inputs and outputs.

Finally, the third mapping step considers logic cones whose outputs have the same input-output signature pair in both versions of the design, but the signatures of the input vector do not match completely.

Among the five inputs in Fig. 4.4, inputs N, P and w2 remain unmapped after the three steps of mapping. In the new version, inputs N, P, and w2 drive a logic cone that was modified.

4.4.2 Transformation of Patterns

Once a mapping of the inputs is available, the set of patterns of the earlier design is transformed into a set of patterns for the new design as follows. For every input I of the new design that is matched with an input J of the earlier design, the values assigned to J are copied to I pattern by pattern. Inputs of the new design that are unmatched are left unspecified.

To demonstrate the transformation in the case of LOC tests, a LOC test is denoted by $\langle s_1,v_1,v_2\rangle$ where s_1 is the scan-in state, and v_1 and v_2 are primary input vectors. After s_1 is scanned in, the circuit is clocked in functional mode for two clock cycles. The primary input

Table 4.2. Pattern Transformation for LOC Tests

FIRST CYCLE PATTERN							
	Scan Elements s_1				Primary Inputs v_1=v_2		
Earlier Version	y1	y2	y3	y4	A1	A2	A3
s_1v_1	1	0	1	0	1	0	1

(a) One of the LOC Patterns for the Earlier Version

MAPPING							
	Scan Elements				Primary Inputs		
New version	w1	w2	w3	w4	M1	M2	M3
Earlier version	x	y1	y3	x	x	A1	A2

(b) Mapping Information

PATTERN TRANSFORMATION							
	Scan Elements				Primary Inputs		
New version	w1	w2	w3	w4	M1	M2	M3
Copy Values	x	1	1	x	x	1	0
Random Fill	0	1	1	1	1	1	0

(c) Pattern Transformation

vectors v_1 and v_2 are applied to the primary inputs during the functional clock cycles. For the tests considered in this article, v_1=v_2.

Table 4.2(a) shows an example of a LOC test for a circuit whose earlier version has present-state variables y1, y2, y3, and y4, and primary inputs A1, A2, and A3. The test ⟨1010,101,101⟩ is shown in Table 4.2(a). Table 4.2(b) shows the mapping obtained from the earlier version to a new version that has present-state variables w1, w2, w3, and w4, and primary inputs M1, M2, and M3. The transformed pattern is shown in Table 4.2(c). For example, the value of y1 is copied to w2 based on the mapping. When an input does not have a mapping, its value is unspecified (x). Unspecified values are filled randomly, as shown in Table 4.2(c).

4.4.3 Fault Simulation and Incremental ATPG

With the transformed set of patterns of the new design, fault simulation is carried out for the new design. Test pattern generation is performed only for faults that remain undetected in the logic cones modified by the change. The logic cone of an output X is considered modified if at least one of the inputs of the logic cone of X has not been mapped. The undetected faults from such modified logic cones are retained, and incremental test generation is carried out to obtain a complete test set for the new design.

For example, considering the circuit from Fig. 4.4, outputs R and W2 have inputs in their input cones that have not been mapped. The faults in these input cones will be retained. For output W1, all the inputs have been mapped. Therefore, the faults in its input cone that are detected by the test set of the earlier version are guaranteed to be detected by the transformed patterns, and they are not considered for incremental test pattern generation. This is important for avoiding incremental ATPG targeting undetectable faults.

4.5 Experimental Results

The test generation procedure was evaluated using several experiments. Transition faults are used for the evaluation. A compacted LOC test set generated using a commercial ATPG tool is used for detecting transition faults in the earlier version of a design. In Section 4.5.1,

different versions of the same design are obtained by making changes at the RTL and resynthesizing it. In section 4.5.2, different versions are obtained by changing the combinational and sequential logic at the gate-level. Section 4.5.3 discusses the difference between the results of the RTL and gate-level changes.

4.5.1 RTL changes

Ten different industrial designs are used for demonstrating the effects of RTL changes, as described next. A new version of a design is obtained by altering the RTL of the earlier version and resynthesizing it. Modifications at random places are made to the earlier version of the design using a commercial tool. Each modification results in changing some of the combinational and sequential logic of the design. For each of the ten designs, ten different modifications are done at the RTL.

Table 4.3. Experimental Result for Circuits modified at RTL

	Inp Cones in Earlier Version	**Aff Cones in New Version**	**% of Aff Cones**	**RunTime (seconds)**			**Fault Coverage(%)**			**Test Patterns**		
				Earl Ver	**New Ver**	**Gain (Ratio)**	**Earl Ver**	**New Ver**		**Earl Ver**	**New Ver**	**Incr (%)**
								Map	**Final**			
c1	86,854	9,408	10.83	1,645	545	3.02	90.07	83.38	90.08	12,531	15,882	26.74
c2	294,863	4,529	1.54	9,847	3,634	2.71	92.07	88.89	92.07	20,036	23,118	15.38
c3	52,931	6,122	11.57	1,281	394	3.25	94.76	91.12	94.76	10,816	13,449	24.34
c4	119,069	4,982	4.18	3,575	605	5.91	94.27	92.91	94.27	10,455	13,714	26.79
c5	113,179	3,953	3.49	4,227	1,142	3.70	88.34	84.76	88.34	21,820	25712	17.83
c6	48,162	16,607	34.48	17,692	4,702	3.76	90.78	80.40	90.77	6,778	8,410	24.07
c7	37,342	5,875	15.73	8,352	2,081	4.01	98.46	92.52	98.46	11,377	17,762	56.12
c8	107,501	23,022	21.42	48,131	15,100	3.19	95.08	88.08	95.08	6,876	9,844	43.16
c9	44,116	4,396	9.96	1,938	636	3.05	90.04	85.43	90.04	7,121	9,305	30.66
c10	224,977	18,206	8.09	71,531	14,117	5.07	93.08	90.10	93.08	84,244	90,221	7.09
avg			12.13			3.77						27.74

The results are shown in Table 4.3. Each row shows the various parameters for one of the ten industrial designs. The average of several parameters for all ten versions of the circuit is shown in each row. These results are shown under the sub-column New ver. Sub-column Earl Ver shows the results for the previous version (the original design) in the respective categories. Column “Inp Cones in Earlier Version" shows the total number of input cones in the earlier version. Column “Aff Cones in New Version” shows the average number of input cones affected in the ten modified versions of the design in comparison to the earlier version. Column “% of Aff Cones” shows the percentage of modified input cones between the new and earlier versions of the design. Column “Runtime” shows the runtime for test generation in seconds, considering the earlier and new versions of the design. For the new version, the runtime includes the time for signature analysis, mapping between two versions, pattern transformation, fault simulation, and incremental test pattern generation. Sub-column “Gain” shows the ratio where the runtime for the earlier version is divided by the average runtime for the new version. This is the improvement in test generation time. Column “Fault Coverage(%)” shows the fault coverage for the earlier and new versions. For the new design, sub-column “Map” shows the fault coverage achieved by fault simulation of the transformed patterns, and sub-column “Final” shows the fault coverage achieved by the proposed methodology. Column “Test Patterns” shows the number of test patterns. Sub-column “Incr (%)” shows the percentage increase in the number of test patterns between the new and earlier versions of the design.

Overall, the gain in test generation time is 3.77x on the average across all the circuits. The gain is higher when a lower percentage of cones are affected. For example, the highest gain of 5.91x is obtained for circuit c4 when 4.18% of the cones are affected. Exceptions occur because of the presence of hard to detect faults. For example, the lowest gain of 2.71x is obtained in circuit c2 when only 1.54% of the cones are affected, and fault simulation of the transformed patterns achieves a fault coverage of 88.89% out of 92.07%.

There is an average 27.74% increase in the number of test patterns. This is because the proposed methodology does not try to optimize the number of patterns needed. Instead, the objective is to evaluate the testability of a circuit as fast as possible, given test patterns

for another structurally similar circuit. ATPG can be run from the beginning on the new version when there is a significant increase in pattern count after the design has converged.

4.5.2 Gate-level changes

Two experiments that modify the combinational and sequential logic, respectively, are conducted to evaluate the proposed test generation procedure when changes are made at the gate-level. For each design in both experiments, ten different modifications are introduced at random locations.

The first experiment changes the combinational logic. The type of a gate is changed, e.g., from OR to NAND. The second experiment performs sequential modifications, which are introduced into the gate-level netlist by following the basic steps of forward and backward retiming. Table 4.4 tabulates results for combinational changes, and Table 4.5 for sequential changes. Column "# of mod SCs" in Table 4.5 shows the average increase or decrease in the number of scan cells in ten new versions of a design due to retiming steps.

Table 4.4. Experimental Result for Gate-level Combinational Changes

	Inp Cones in Earlier Version	**Aff Cones in New Version**	**% of Aff Cones**	**RunTime (seconds)**			**Fault Coverage(%)**			**Test Patterns**		
				Earl Ver	**New Ver**	**Gain (Ratio)**	**Earl Ver**	**New Ver**		**Earl Ver**	**New Ver**	**Incr %**
								Map	**Final**			
d1	417,626	982	0.24	17,740	1,440	12.32	97.08	97.07	97.12	3,456	3,965	14.72
d2	270,428	2,699	0.99	2,517	318	7.92	91.77	90.61	91.77	6,281	8,962	42.68
d3	325,885	772	0.24	5,472	766	7.14	89.08	88.21	89.08	5,745	6,196	7.85
d4	45,104	451	0.99	814	93	8.75	93.41	93.38	93.41	8,047	8,074	0.34
d5	300,542	1,187	0.39	4,836	512	9.45	93.19	93.15	93.19	8,066	8,219	1.90
d6	88,065	591	0.67	13,107	988	13.27	95.87	95.23	95.87	6,260	7,311	16.79
d7	601,469	5,611	0.93	10,216	724	14.11	91.23	90.65	91.23	29,122	31,393	7.80
d8	595,496	10,219	1.71	66,361	6,282	10.56	90.53	89.27	90.55	27,295	28,318	3.75
d9	173,840	885	0.50	3,729	412	9.05	91.34	90.56	91.34	11,367	12,622	11.04
d10	212,256	1174	0.55	69,045	4684	14.74	96.00	95.97	96.45	73,112	75,252	2.93
avg			0.72			10.73						10.98

Table 4.5. Experimental Result for Gate-level Sequential Changes

	Inp Cones in Earlier Version	**Aff Cones in New Version**	**% of Aff Cones**	**# of mod SCs**	**RunTime (seconds)**			**Fault Coverage(%)**			**Test Patterns**		
					Earl Ver	**New Ver**	**Gain (Ratio)**	**Earl Ver**	**New Ver**		**Earlier Ver**	**New Ver**	**Incr %**
									Map	**Final**			
d1	417,626	997	0.23	7	17,740	1,421	12.48	97.08	97.07	97.10	3,456	3,669	6.16
d2	270,428	2,319	0.86	6	2,517	297	8.47	91.77	90.95	91.77	6,281	7,628	21.44
d3	325,885	1,106	0.34	6	5,472	602	9.09	89.08	88.56	89.08	5,745	6,005	4.52
d4	45,104	231	0.51	6	814	87	9.36	93.41	93.40	93.41	8,047	8,165	1.46
d5	300,542	1,488	0.49	7	4,836	532	9.09	93.19	93.16	93.19	8,066	8,211	1.79
d6	88,065	438	0.50	6	13,107	804	16.30	95.87	95.23	95.87	6,260	6,514	4.05
d7	601,469	3,239	0.54	7	10,216	701	14.57	91.23	90.81	91.23	29,122	30,285	3.99
d8	595,496	9,811	1.64	5	66,361	5,911	11.23	90.53	89.77	90.54	27,295	28,161	3.17
d9	173,840	671	0.39	5	3,729	412	9.37	91.34	90.87	91.34	11,367	11.981	5.40
d10	212,256	876	0.41	7	69,045	4469	15.45	96.00	95.99	96.14	73,112	74,217	1.51
avg			0.59	6			11.54						5.35

An average of 10-fold improvement in runtime when modifications change the combinational logic, and an average of 11-fold improvement in runtime when modifications change the sequential logic are observed. The increase in the number of test patterns of the new version of the circuit compared to the earlier version can be mitigated, as discussed earlier.

4.5.3 Discussion

Modifications in the gate-level netlist result in an average of 0.72% of the logic cones being affected compared to an average of 12.13% of affected logic cones in the RTL. This is a result of the fact that RTL designs with changes require the gate-level circuit to be resynthesized, which affects a larger portion of the design due to re-optimization; whereas netlist modifications at the gate-level are localized and impact smaller parts of the design.

The smaller percentage of changes for gate-level modifications translates to a lower percentage of faults that need to be targeted in the new version after the transformation of patterns and fault simulation. A lower percentage of faults for incremental test pattern generation directly translates to a higher gain in runtime. Similarly, with a smaller percentage of faults to detect after fault simulation of the transformed patterns, a smaller increase in the number of patterns is required to detect those faults during incremental test pattern generation.

4.6 Conclusion

This chapter proposed a methodology to reuse test patterns generated for transition faults from one structurally similar circuit to another. An analysis of transition faults, that require two-cycle tests, showed that it is important to analyze a design over two time-frames to identify structural similarities. The proposed methodology found structural similarities between the two versions of a design over two time-frames by computing signatures. It used the signatures to map the inputs and outputs in both versions of the design. The patterns were transformed from the earlier to the new version according to the mapping information and fault simulated. Incremental test pattern generation was run on the remaining undetected faults from the modified logic cones. Three experiments were conducted to evaluate

the proposed test generation methodology for transition faults where changes were made at the RTL and gate-level. Experiments on two sets of ten industrial designs showed an average improvement of 3.77-fold in runtime when changes were made at the RTL, and more than 10-fold when changes were done at the gate-level compared to the case where the entire test generation process is run for the new version.

5. CONCLUSION

This thesis proposed a fast methodology to analyze the testability of a design, given the test patterns generated for another structurally similar design. The test generation time is reduced by exploiting the structural similarity prevalent between the designs. The test generation methodology described finds the similarities between the two circuits by performing the structural analysis and uses this information to map the inputs between the two circuits. The patterns generated for one design is transformed into test patterns for another design using the mapping information. The design is fault simulated using the transformed test patterns, and an incremental test pattern generation is performed for the remaining undetected faults.

The first chapter of the thesis describes a test generation methodology where structurally similar circuits were obtained by modifying the gate-level netlists. The second chapter proposed a mapping methodology where structurally similar designs were obtained by modifying the RTL, and the gate-level was resynthesized after every modification. The suggested mapping method handles the complexities that arise from the RTL modifications. The last chapter of the thesis described a test generation methodology that performed the signature analysis of the design over two-time frames to accommodate two-cycle patterns used to detect transition faults.

Experimental analysis using the proposed test generation methodology in different scenarios showed improvement in runtime compared to generating test patterns from the beginning. When the gate-level netlists were modified, more than an 11-fold improvement in runtime was observed. For designs where the RTl was modified and gate-level was resynthesized, the reduction in runtime was an average of 5-fold compared to running test generation from the beginning. For transition faults that required two-cycle test patterns, the improvement in runtime was 3.77x for designs where RTL was modified and the gate-level was resynthesized. An average of 10x improvement was observed when structurally similar designs were obtained by modifying the gate-level netlists.

REFERENCES

[1] R. Aitken, "Nanometer technology effects on fault models for ic testing," *Computer*, vol. 32, no. 11, pp. 46–51, 1999. DOI: 10.1109/2.803640.

[2] S. K. Jena, S. Biswas, and J. K. Deka, "Approximate testing of digital vlsi circuits using error significance based fault analysis," in *2020 24th International Symposium on VLSI Design and Test (VDAT)*, 2020, pp. 1–6. DOI: 10.1109/VDAT50263.2020.9190571.

[3] K.-T. Cheng and A. Krstic, "Current directions in automatic test-pattern generation," *Computer*, vol. 32, no. 11, pp. 58–64, 1999. DOI: 10.1109/2.803642.

[4] S. H. Gerez, *Algorithms for VLSI Design Automation*, 1st. USA: John Wiley Sons, Inc., 1999, ISBN: 0471984892.

[5] L. Trevillyan, D. Kung, R. Puri, L. Reddy, and M. Kazda, "An integrated environment for technology closure of deep-submicron ic designs," *IEEE Design Test of Computers*, vol. 21, no. 1, pp. 14–22, 2004. DOI: 10.1109/MDT.2004.1261846.

[6] H. Ren, R. Puri, L. Reddy, *et al.*, "Intuitive eco synthesis for high performance circuits," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 1002–1007. DOI: 10.7873/DATE.2013.209.

[7] Z. Navabi, *VHDL: Analysis and Modeling of Digital System*. McGraw-Hill, 1997.

[8] Z. Navabi, *Verilog Digital System Design (Professional Engineering)*. McGraw-Hill, 1999.

[9] A. Sagahyroon, G. Lakkaraju, and M. Karunaratne, "A functional verification environment," in *48th Midwest Symposium on Circuits and Systems, 2005.*, 2005, 108–111 Vol. 1. DOI: 10.1109/MWSCAS.2005.1594051.

[10] M. L. B. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer, Boston, MA, 2002.

[11] J. Aerts and E. Marinissen, "Scan chain design for test time reduction in core-based ics," in *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, 1998, pp. 448–457. DOI: 10.1109/TEST.1998.743185.

[12] E. B. Eichelberger and T. W. Williams, "A logic design structure for lsi testability," in *Proceedings of the 14th Design Automation Conference*, ser. DAC '77, IEEE Press, 1977, pp. 462–468.

[13] L.-T. W. C.-W. W. X. Wen, *VLSI Test Principles and Architectures*. Morgan Kaufmann.

[14] E. J. McCluskey, "Built-in self-test techniques," *IEEE Design Test of Computers*, vol. 2, no. 2, pp. 21–28, 1985. DOI: 10.1109/MDT.1985.294856.

[15] A. Hassan, J. Rajski, and V. Agarwal, "Testing and diagnosis of interconnects using boundary scan architecture," in *International Test Conference 1988 Proceeding@m_NewFrontiersinT* 1988, pp. 126–137. DOI: 10.1109/TEST.1988.207790.

[16] J. Patel, "Stuck-at fault: A fault model for the next millennium," in *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, 1998, pp. 1166–. DOI: 10.1109/TEST.1998.743358.

[17] K.-T. Cheng, "Transition fault testing for sequential circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 12, pp. 1971–1983, 1993. DOI: 10.1109/43.251160.

[18] H. Cox and J. Rajski, "Stuck-open and transition fault testing in cmos complex gates," in *International Test Conference 1988 Proceedings m_New Frontiers in Testing*, 1988, pp. 688–694. DOI: 10.1109/TEST.1988.207853.

[19] K. Fuchs, F. Fink, and M. Schulz, "Dynamite: An efficient automatic test pattern generation system for path delay faults," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 10, pp. 1323–1335, 1991. DOI: 10.1109/43.88928.

[20] T. Storey and W. Maly, "Cmos bridging fault detection," in *Proceedings. International Test Conference 1990*, 1990, pp. 842–851. DOI: 10.1109/TEST.1990.114102.

[21] S. Ma, I. Shaik, and R. Fetherston, "A comparison of bridging fault simulation methods," in *International Test Conference 1999. Proceedings (IEEE Cat. No.99CH37034)*, 1999, pp. 587–595. DOI: 10.1109/TEST.1999.805783.

[22] M. Schulz, E. Trischler, and T. Sarfert, "Socrates: A highly efficient automatic test pattern generation system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 1, pp. 126–137, 1988. DOI: 10.1109/43.3140.

[23] D. Pradhan and J. Saxena, "A design for testability scheme to reduce test application time in full scan," in *Digest of Papers. 1992 IEEE VLSI Test Symposium*, 1992, pp. 55–60. DOI: 10.1109/VTEST.1992.232724.

[24] A. Pandey and J. Patel, "An incremental algorithm for test generation in illinois scan architecture based designs," in *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, 2002, pp. 368–375. DOI: 10.1109/DATE.2002.998300.

[25] K. Yang, K.-T. Cheng, and L.-C. Wang, "Trangen: A sat-based atpg for path-oriented transition faults," in *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No.04EX753)*, 2004, pp. 92–97. DOI: 10.1109/ASPDAC.2004.1337546.

[26] S.-H. Song and L. Kinney, "Incremental test pattern generation," in *Digest of Papers Eleventh Annual 1993 IEEE VLSI Test Symposium*, 1993, pp. 244–250. DOI: 10.1109/VTEST.1993.313353.

[27] T. Shinsha, T. Kubo, Y. Sakataya, J. Koshishita, and K. Ishihara, "Incremental logic synthesis through gate logic structure identification," in *23rd ACM/IEEE Design Automation Conference*, 1986, pp. 391–397. DOI: 10.1109/DAC.1986.1586119.

[28] D. Brand, "Incremental synthesis," in *IEEE/ACM International Conference on Computer-Aided Design*, 1994, pp. 14–18. DOI: 10.1109/ICCAD.1994.629736.

[29] S. Krishnaswamy, H. Ren, N. Modi, and R. Puri, "Deltasyn: An efficient logic difference optimizer for eco synthesis," in *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*, 2009, pp. 789–796. DOI: 10.1145/1687399.1687546.

[30] A. Dutta, N. Tuttle, and K. Anandh, "Canonical ordering of instances to immunize the fpga place and route flow from eco-induced variance," in *International Symposium on Quality Electronic Design (ISQED)*, 2013, pp. 359–363. DOI: 10.1109/ISQED.2013.6523635.

[31] Q. Zhu, N. B. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "Sat sweeping with local observability don't-cares," in *Advanced Techniques in Logic Synthesis, Optimizations and Applications*, K. Gulati, Ed. New York, NY: Springer New York, 2011, pp. 129–148, ISBN: 978-1-4419-7518-8. DOI: 10.1007/978-1-4419-7518-8_8. [Online]. Available: https://doi.org/10.1007/978-1-4419-7518-8_8.

[32] C.-F. Lai, J.-H. Jiang, and K.-H. Wang, "Boom: A decision procedure for boolean matching with abstraction and dynamic learning," in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 499–504. DOI: 10.1145/1837274.1837398.

[33] W. M. Holt, "1.1 moore's law: A path going forward," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, 2016, pp. 8–13. DOI: 10.1109/ISSCC.2016.7417888.

[34] S. Borkar, "Design perspectives on 22nm cmos and beyond," in *2009 46th ACM/IEEE Design Automation Conference*, 2009, pp. 93–94.

[35] K.-T. Cheng and A. Krstic, "Current directions in automatic test-pattern generation," *Computer*, vol. 32, no. 11, pp. 58–64, 1999. DOI: 10.1109/2.803642.

[36] S. Borkar, "Gpu accelerated vlsi design verification," in *2010 10th IEEE International Conference on Computer and Information Technology*, 2010, pp. 1213–1218.

[37] L. Trevillyan, D. Kung, R. Puri, L. Reddy, and M. Kazda, "An integrated environment for technology closure of deep-submicron ic designs," *IEEE Design Test of Computers*, vol. 21, no. 1, pp. 14–22, 2004. DOI: 10.1109/MDT.2004.1261846.

[38] H. Ren, R. Puri, L. Reddy, *et al.*, "Intuitive eco synthesis for high performance circuits," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 1002–1007. DOI: 10.7873/DATE.2013.209.

[39] X. Lin, R. Press, J. Rajski, *et al.*, "High-frequency, at-speed scan testing," *IEEE Design Test of Computers*, vol. 20, no. 5, pp. 17–25, 2003. DOI: 10.1109/MDT.2003.1232252.

[40] X. Lin and S. M. Reddy, "On gate function based tests for scan designs," in *2016 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2016, pp. 1–4. DOI: 10.1109/VLSI-DAT.2016.7482582.

[41] A. Kumar, J. Rajski, S. M. Reddy, and T. Rinderknecht, "On the generation of compact deterministic test sets for bist ready designs," in *2013 22nd Asian Test Symposium*, 2013, pp. 201–206. DOI: 10.1109/ATS.2013.45.

[42] H. Ma, R. Guo, Q. Jing, *et al.*, "A case study of testing strategy for ai soc," in *2019 IEEE International Test Conference in Asia (ITC-Asia)*, 2019, pp. 61–66. DOI: 10.1109/ITC-Asia.2019.00024.

[43] L. Goldstein, "Controllability/observability analysis of digital circuits," *IEEE Transactions on Circuits and Systems*, vol. 26, no. 9, pp. 685–693, 1979. DOI: 10.1109/TCS.1979.1084687.

[44] K.-H. Tsai, "Testability-driven fault sampling for deterministic test coverage estimation of large designs," in *2014 IEEE 23rd Asian Test Symposium*, 2014, pp. 119–124. DOI: 10.1109/ATS.2014.32.

[45] A. Kamran and Z. Navabi, "Homogeneous many-core processor system test distribution and execution mechanism," in *2014 19th IEEE European Test Symposium (ETS)*, 2014, pp. 1–2. DOI: 10.1109/ETS.2014.6847839.

[46] I. Ma, H. K. Lau, J. Reynick, and Y. Huang, "Innovative practices on dft for ai chips," in *2019 IEEE 37th VLSI Test Symposium (VTS)*, 2019, pp. 1–1. DOI: 10.1109/VTS.2019.8758655.

[47] M. Sharma, A. Dutta, W.-T. Cheng, B. Benware, and M. Kassab, "A novel test access mechanism for failure diagnosis of multiple isolated identical cores," in *2011 IEEE International Test Conference*, 2011, pp. 1–9. DOI: 10.1109/TEST.2011.6139171.

[48] Y. Watanabe and R. Brayton, "Incremental synthesis for engineering changes," in *[1991 Proceedings] IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1991, pp. 40–43. DOI: 10.1109/ICCD.1991.139840.

[49] C.-C. Lin, K.-C. Chen, and M. Marek-Sadowska, "Logic synthesis for engineering change," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 3, pp. 282–292, 1999. DOI: 10.1109/43.748158.

[50] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming (preliminary version)," in *Third Caltech Conference on Very Large Scale Integration*, R. Bryant, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 87–116.

[51] C. E. Leiserson and J. B. Saxe, *Retiming synchronous circuitry*, 1991.

[52] S.-H. Song and L. Kinney, "Incremental test pattern generation," in *Digest of Papers Eleventh Annual 1993 IEEE VLSI Test Symposium*, 1993, pp. 244–250. DOI: 10.1109/VTEST.1993.313353.

[53] Y. Watanabe and R. Brayton, "The maximum set of permissible behaviors for fsm networks," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, 1993, pp. 316–320. DOI: 10.1109/ICCAD.1993.580075.

[54] J.-H. R. Jiang, V. N. Kravets, and N.-Z. Lee, "Engineering change order for combinational and sequential design rectification," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 726–731. DOI: 10.23919/DATE48585.2020.9116504.

[55] J. McPherson, "Reliability trends with advanced cmos scaling and the implications for design," in *IEEE Custom Integrated Circuits Conference*, 2007, pp. 405–412.

[56] H. Reyserhove and W. Dehaene, *Efficient Design of Variation-Resilient Ultra-Low Energy Digital Processors*. Springer, 2019.

[57] L. Lavagno, A. Kondratyev, Y. Watanabe, *et al.*, "Incremental high-level synthesis," in *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010, pp. 701–706. DOI: 10.1109/ASPDAC.2010.5419798.

[58] K. Chakravadhanula, V. Chickermane, P. Cunningham, *et al.*, "Advancing test compression to the physical dimension," in *2017 IEEE International Test Conference (ITC)*, 2017, pp. 1–10. DOI: 10.1109/TEST.2017.8242035.

[59] J. Joe, N. Mukherjee, I. Pomeranz, and J. Rajski, "Fast test generation for structurally similar circuits," in *2022 IEEE 40th VLSI Test Symposium (VTS)*, 2022, pp. 1–7. DOI: 10.1109/VTS52500.2021.9794232.

[60] H. Ren, R. Puri, L. Reddy, *et al.*, "Intuitive eco synthesis for high performance circuits," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 1002–1007. DOI: 10.7873/DATE.2013.209.

[61] A. Stempkovskiy, D. Telpukhov, and R. Soloviev, "Fast and accurate resource-aware functional eco patch generation tool," in *2018 Moscow Workshop on Electronic and Networking Technologies (MWENT)*, 2018, pp. 1–6. DOI: 10.1109/MWENT.2018.8337192.

[62] K. Mei, "Bridging and stuck-at faults," *IEEE Transactions on Computers*, vol. C-23, no. 7, pp. 720–727, 1974. DOI: 10.1109/T-C.1974.224020.

[63] S. EggersglüSS, R. Krenz-Bååth, A. Glowatz, F. Hapke, and R. Drechsler, "A new sat-based atpg for generating highly compacted test sets," in *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2012, pp. 230–235. DOI: 10.1109/DDECS.2012.6219063.

[64] V. Bhardwaj, "Shift left trends for design convergence in soc: An eda perspective," in *International Journal of Computer Applications*, 2021, pp. 22–27.

[65] J.-H. R. Jiang, V. N. Kravets, and N.-Z. Lee, "Engineering change order for combinational and sequential design rectification," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 726–731. DOI: 10.23919/DATE48585.2020.9116504.

[66] I. H.-R. Jiang and H.-Y. Chang, "Ecos: Stable matching based metal-only eco synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 3, pp. 485–497, 2012. DOI: 10.1109/TVLSI.2011.2104377.

[67] N.-Z. Lee, V. N. Kravets, and J.-H. R. Jiang, "Sequential engineering change order under retiming and resynthesis," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 109–116. DOI: 10.1109/ICCAD.2017.8203767.

[68] J. Joe, N. Mukherjee, I. Pomeranz, and J. Rajski, "Test generation for an iterative design flow with rtl changes," in *2022 IEEE International Test Conference (ITC)*, 2022, pp. 1–9.

[69] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, "Transition fault simulation," *IEEE Design Test of Computers*, vol. 4, no. 2, pp. 32–38, 1987. DOI: 10.1109/MDT.1987.295104.

[70] J. Savir, "Skewed-load transition test: Part i, calculus," in *Proceedings International Test Conference 1992*, 1992, pp. 705–. DOI: 10.1109/TEST.1992.527892.

[71] J. Savir and S. Patil, "Scan-based transition test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 8, pp. 1232–1241, 1993. DOI: 10.1109/43.238615.

A. ADDITIONAL EXPERIMENTS FOR CHAPTER-1

A.1 Single Changes

The first experiment introduces one change at a time. For each circuit, ten different versions are generated by introducing a single change at a random location in each of the versions. The modifications involve changing a gate type, for example, AND to OR, AND to NAND, etc., in the gate-level netlist.

In Table A.1, each row corresponds to one of the 11 circuits. The row shows the average, minimum, and maximum of each one of different parameters for all the ten versions of the circuit. These results are shown under the sub-columns "avg", "min", and "max". Sub-column "Circuit1" shows the results for circuit1 in the respective categories. The first column, "size," gives the total number of gates present in each circuit. Column "Output cones" gives the total number of output cones in the circuit. The next column, "Different Output Cones", shows the number of output cones that changed in circuit 2 compared to circuit1. Column "Incremental ATPG" gives the time taken in seconds to run incremental ATPG on circuit2. Column "Run Time" shows the overall time taken in seconds for test pattern generation. For circuit2, the total runtime includes the time taken for signature computation, mapping, and fault simulation, as well as for incremental ATPG. Column "Test Patterns" shows the total number of test patterns in both the circuits after ATPG. The last column, "Fault coverage", shows the fault coverage achieved in circuit1 and the average fault coverage seen in the different versions of circuit2.

Table A.1. Experimental Result for Single Changes

	Size	Output	Different Output Cones			Incremental ATPG(s)			RunTime (seconds)			
		Cones	avg	min	max	avg	min	max	Circuitl	avg	min	max
1	8,611,749	429,708	1,562	16	10,146	266	18	1,040	7,020	508	266	1,271
2	3,435,493	276,462	377	129	41,081	30	9	35	622	99	77	104
3	4,833,052	334,108	9,486	3	69,257	31	0	139	1,568	125	93	233
4	10,213,281	601,469	30,380	647	140,369	340	59	1,105	9,300	1,345	1,045	2,105
5	15,102,044	1,215,337	8,380	340	30,146	216	85	485	13,740	1,207	1,074	1,472
6	3,048,599	174,505	3,751	214	21,687	20	5	42	1,260	133	115	152
7	3,519,135	219,280	7,003	53	37,669	190	9	421	5,103	414	228	643
8	1,203,568	45,345	2,361	21	8,240	15	4	35	484	28	17	48
9	1,699,509	89,908	11,097	2,436	22,824	362	5	1,181	10,168	469	90	1,317
10	6,389,260	302,540	7,317	57	33,756	28	10	78	1,777	252	202	281
11	12,167,308	440,540	24,063	405	161,267	181	20	629	6,446	882	712	1,329

	Test Patterns				Fault Coverage(%)	
	Circuit1	avg	min	max	Circuit1	avg
1	2,334	2,861	2,342	4,006	97.06	97.06
2	1,459	1,565	1,459	1,665	92.90	92.91
3	1,408	1,515	1,408	1,737	92.08	92.08
4	7,424	8,378	7,444	12,480	90.85	90.88
5	4,544	5,271	4,547	7,692	94.46	94.47
6	2,944	3,210	2,985	3,569	91.92	91.92
7	5,155	5,900	5,155	7,499	97.02	97.03
8	1,811	2,091	1,804	3,017	88.58	88.58
9	2,163	3,017	2,168	4,358	97.62	97.87
10	2,520	2,718	2,520	3,284	90.83	90.86
11	7,168	7,355	7,102	8,733	91.65	91.90

From the column "avg" under "different Output Cones" in Table A.1, it can be seen that introducing a single change can affect between 0.1% to 12.3% of total number of output cones for the 11 circuits. An average of 16% variation is seen between the minimum and the maximum number of different cones within ten versions of each circuit. The number of output cones different in circuit 2 compared to circuit1 has an affect on time taken for the incremental ATPG. The higher the number of different cones, the higher the number of unmatched inputs, which results in fewer faults being identified after mapping. This will increase the time taken for incremental ATPG which translates to increase in the total runtime. From Table A.1, it can be seen that there is an average of 11-fold gain in runtime to achieve the same fault coverage in circuit2 compared to circuit1 for all the circuits.

Table A.2. Results of Runtime Gain and Test Pattern Increase for Table A.1

	Runtime	Test Patterns
	Gain	Increase(%)
1	13.82	22.58
2	6.30	7.27
3	12.54	7.60
4	6.91	12.85
5	11.38	16.00

	Runtime	Test Patterns
	Gain	Increase (%)
6	9.47	9.04
7	12.33	14.45
8	17.29	15.46
9	21.68	39.48
10	7.05	7.86
11	7.31	2.61

For a more detailed analysis of the results, each one of the 10 versions of circuit 7 in Table A.1 is described in Table A.3.

In Table A.3, each row shows a different version of circuit 7. Column "Different cones" shows the number of output cones that changed in circuit2 compared to circuit1. Column "Inc ATPG" gives the time taken in seconds for running incremental ATPG on circuit2. Column "Fault Coverage" is further subdivided into "Map" and "ATPG" .Sub-column "Map" gives the fault coverage after mapping the patterns read from circuit1, while sub-column "ATPG" gives the total fault coverage after the whole process. Column "Test Patterns" shows the total number of test patterns in circuit2, and column "Run Time" shows the overall time taken in seconds for test pattern generation.

Table A.1 shows that before any change was done on circuit 7, it had a fault coverage of 97.02%. This was achieved with a total number of 5,155 test patterns in 5,103 seconds. From Table A.3, it can be seen that a single change can affect 54 to 37,669 different cones for this circuit. For version 4, with 37,669 different cones, the final fault coverage was achieved with 6,881 patterns. On the other hand, for version 10 with 10,703 different cones, the final fault coverage was achieved with 7,499 patterns. This difference in the number of test patterns generated during incremental ATPG is explained by the total number of hard to detect faults remaining after mapping patterns.

Table A.3. Results for Individual Runs for 10 Versions of Circuit 7 from Table A.1

	Different Cones	Inc ATPG	Fault Coverage		Test Pattern	Run-Time (secs)
			Map %	ATPG %		
1	59	20	97.02	97.03	5,157	241.5
2	1,734	218	97.02	97.15	5,922	440.5
3	15,144	376	96.97	97.15	6,134	596.7
4	37,669	359	96.94	97.28	6,881	597.5
5	1,662	224	97.02	97.15	5,923	448.0
6	281	12	97.02	97.03	5,173	238.0
7	988	38	97.02	97.03	5,222	260.0
8	54	9	97.02	97.04	5,155	228.4
9	1,724	223	97	97.15	5,937	443.4
10	10,703	421	96.55	97.03	7,499	643.3

Table A.2 shows the runtime gain and percentage increase in the number of test patterns based on the data in Table A.1. From Tables A.1, A.3 and A.2, it can be seen that with a small change, a large number of output cones can be affected. However, by utilizing the structural similarity between the circuits, the test generation time can be considerably reduced. The algorithm proposed in this paper focuses on determining the testability of a circuit given a test set for another structurally identical circuit and does not optimize the pattern inflation. There is an average of 14% increase in the number of test patterns in circuit2 compared to circuit1. This can be mitigated by rerunning test pattern generation after the design converges or occasionally when the number of patterns increases significantly.

A.2 Cumulative changes

For the second experiment, ten different versions for each circuit are created. The modifications are carried forward to the next versions, i.e., the first version has one change, the second version has one change on top of the first version, and so on. The signature and pattern file read for the analysis in circuit 2 comes from the previous version.

Table A.4 is tabulated in the same way as Table A.1. Here, the results for the cumulative changes in the modified circuit are shown. Table A.5 shows the summary of runtime gain and percentage increase in number of test patterns in similar way it is reported in Table A.3. From Table A.4, it can be seen that on average there is a 11x gain in runtime when the changes in the circuit is cumulative. There is an average of 58% increase in number of test patterns compared to the baseline circuit. This, again, can be mitigated by running test pattern generation from the beginning.

Table A.4. Experimental Result for Cumulative Changes

	Different Output Cones			RunTime				Test Patterns				Fault Coverage	
	avg	min	max	Baseline	avg	min	max	Baseline	avg	min	max	Baseline	avg
1	1,562	16	10,146	7,020	593	242	1,431	2,334	3,938	2,342	5,880	97.06	97.07
2	7,336	129	41,081	622	77	68	84	1,459	1,794	1,471	2,018	92.90	92.96
3	9,486	3	69,257	1,568	132	93	244	1,408	1,870	1,499	2,396	92.08	92.10
4	33,469	647	140,369	9,300	1,451	988	2,350	7,424	10,896	7,473	16,097	90.85	90.92
5	8,380	340	30,146	13,740	1,160	971	1,416	4,544	7,383	4,608	9,321	94.46	94.49
6	3,751	214	21,687	1,260	144	120	158	2,944	4,560	3,044	5,239	91.92	91.98
7	7,003	53	37,669	5,103	426	218	725	5,155	7,821	5,157	10,659	97.02	97.23
8	2,361	21	8,240	484	30	18	43	1,811	3,291	1,905	4,145	88.58	88.58
9	11,097	2,436	22,824	10,168	403	111	1,317	2,163	5,414	3,498	6,592	97.62	97.89
10	7,317	57	33,756	1,777	255	218	281	2,520	4,141	3,284	4,547	90.83	90.87
11	24,063	405	161,267	6,446	805	652	1,198	7,168	7,914	7,139	9,256	91.65	91.75

From Tables V and VI, it can be seen that there is an average of 11-fold gain in runtime to achieve the same fault coverage in circuit2 compared to circuit1. The results from cumulative changes remain consistent with the observations drawn from Tables A.1, A.3 and A.2. The time taken for incremental ATPG and total test generation process depends on the number of cones affected by the change introduced in the previous versions of a given circuit. For cumulative changes, the pattern file of the previous version is considered as T_1 which gets transformed to T_2. This results in accumulation of test patterns during TABLE A.4:

Table A.5. Results of Runtime Gain and Test Pattern Increase for Table A.4

	Runtime	Test Patterns
	Gain	% increase
1	11.84	68.72
2	8.08	22.96
3	11.88	32.81
4	6.41	46.77
5	11.84	62.48

	Runtime	Test Patterns
	Gain	% increase
6	8.75	54.89
7	11.98	51.72
8	16.13	81.72
9	25.23	150.30
10	6.97	64.34
11	8.01	10.41