Øystein Gjermundnes

Exploiting Arithmetic Built-In Self-Test Techniques for Path Delay Fault Testing

Thesis for the degree doktor ingeniør

Trondheim, November 2006

Norwegian University of Science and Technology
Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Electronics and Telecommunications

NTNU
Innovation and Creativity

NTNU
Norwegian University of Science and Technology

Thesis for the degree doktor ingeniør

Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Electronics and Telecommunications

ISBN 82-471-8257-2 (printed version)
ISBN 82-471-8256-4 (electronic version)
ISSN 1503-8181

Doctoral theses at NTNU, 2006:237

Printed by NTNU-trykk

Exploiting Arithmetic Built-In Self-Test Techniques for Path Delay Fault Testing

Øystein Gjermundnes

November 21, 2006

Abstract

This thesis describes the implementation of a system for analyzing circuits with respect to their path-delay fault testability. The system includes a path-delay fault simulator and an ATPG for path-delay faults combined into a test tool. This test tool can run standalone on a single machine, or as one of several clients that communicate through a central server. The test tool is used in this thesis in order to evaluate the performance of 14 different test vector generators that can be used in various built-in self-test arrangements.

The test generators exploit pseudo-random stimuli generation. We have used six different strategies for weighting of input signals, and performed comprehensive experiments to evaluate the efficiency of the strategies. Each of the experiments typically consists of three phases:

- In the first phase, the ATPG is used in order to find the K-longest non-robust testable path-delay faults. The corresponding path numbers are then saved together with the corresponding test vector for later use. The paths constitute the target fault list during simulation. Experiments that consider all possible faults skip this phase.
- In the second phase, weights are generated for the weighted pseudo-random generators. These weights are stored for later use. This phase is skipped for experiments where the generator is unweighted.
- In the third phase the actual simulation takes place. In all experiments 10M single-input-change test patterns were applied and repeated ten times for each generator and circuit in order to cover some statistical variations. Only non-robust faults (including robust faults) were considered.

Two groups of pseudo-random generators have been evaluated. The first group, GA, consists of accumulator based pseudo-random generators. The second group, GT, consists of Mersenne twister based pseudo-random generators.

The result has shown that the GT group of pseudo-random patterns give marginally better results than the GA group. Since GA generators are much less computationally intensive, GA generators are reccommended over GT generators in practical applications. Experiments have also been conducted in order to evaluate the benefit of weighted stimuli compared to unweighted stimuli. The results show that test time can be reduced with up to 15 times for the circuits in the ISCAS'85 benchmark suite.

Based upon comprehensive experiments with various weighting schemes on ISCAS benchmarks, one can conclude that the following three-phase approach works well: First, generate test patterns to detect the K(20000) longest paths. Subsequently, compute weights for each input based upon the gennerated patterns. Finally, employ an accumulator based BIST scheme with the weights on non-robust path-delay faults.

Preface

This doctoral thesis is submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree *Doktor Ingeniør*.

The work herein was performed at and funded by the Department of Electronics and Telecommunications, NTNU, under the supervision of Professor Einar J. Aas.

Acknowledgements

It is a beautiful Sunday evening in the middle of November, and I am sitting at the desk I have been sitting at so many times before. It is a good place to sit, of the six desks in room B312, I think I got the better. I don't have a window though. Who needs a window anyway, when Microsoft provides Windows. It's that old joke again floating through my head. Soon someone else will spend four years on a chair in my little corner. When I came to sit on this chair for the first time over four years ago, I was an infant 25 year old PhD student facing an interesting, but long and challenging, journey. My footing slipped more than just once through this period, and each time I was helped gently onto my feet again by my mentor, Professor Einar J. Aas. Interestingly enough, I now face an equally difficult challenge in how to, with few words, properly express my gratitude towards my supervisor and friend Professor Einar J. Aas. As with all other NP-complete problems, we don't have time to wait for the optimum solution, so therefore I have picked the following words: Thank you for helping me all those times, with technical questions, moral support, and writing!

I would also like to thank my friends and colleagues at the Department of Electronics and Telecommunications at NTNU, for stories, laughs, smiles and friendly faces. Each and every one of you contribute every day in making the Department a really good place to work.

Finally, I would like to thank my family. Mom and Dad, thank you for your unconditional love and support through 29 years. I would not have achieved this if it wasn't for you. Svanhild, Kristine and Bjarne, you are the best sisters and brother there are. A couple of years ago, this would have been the end of this preface, but then I met Torun (Thank you for bringing us together Tajeshwar, you enjoyed that, didn't you :)), and three weeks ago I became the proud father of the sweetest girl ever. Torun and Margareta, I love you!

Contents

List of Figures

List of Tables

List of Definitions

List of Algorithms

Chapter 1

Introduction to testing

1.1 A brief history of integration

1.1.1 The first integrated transistor

In 2000 the Nobel Prize in Physics was awarded to Jack S. Kilby for ”his part in the invention of the integrated circuit”. This great achievement took place in 1958 when Kilby was working at Texas Instruments with electronic component miniaturization (Figure 1.1). Independently and at the same time Robert Noyce, co-founder of Fairchild Semiconductors and Intel Corp, and later known as ”Mayor of Silicon Valley”, also realized that a whole circuit could be integrated on a single chip.

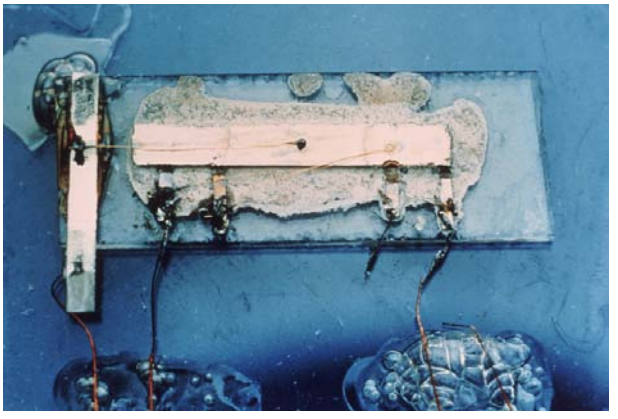


Figure 1.1: Kilbys integrated circuit. (Courtesy of Texas Instruments)

The first thing Kilby did with his integrated circuit was probably to test it.

1.1.2 Moore's law

Ever since that day more and more transistors have been successfully integrated, which brings us to Gordon E. Moore and his famous prediction [Moo65] from 1965. Moore stipulated, in what is later known as Moores law, that the number of transistors that can be integrated onto one chip will double every year. Moore's original sketch of his prediction is shown in Figure 1.2.

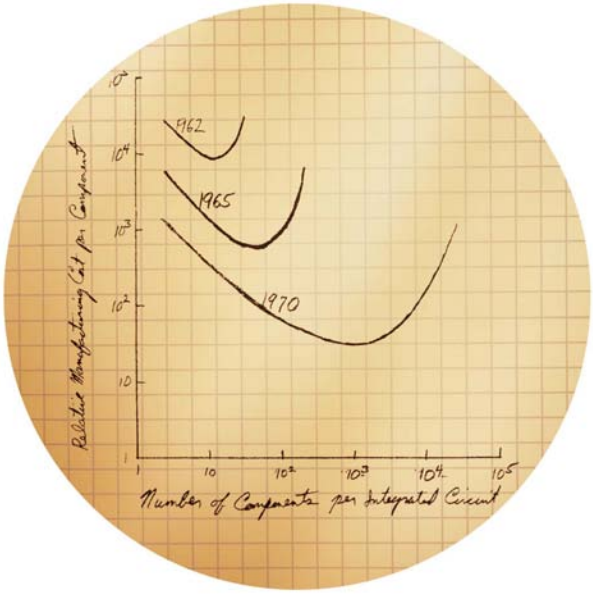


Figure 1.2: In 1965, Gordon Moore sketched out his prediction of the pace of silicon technology. Decades later, Moores Law remains true, driven largely by Intels unparalleled silicon expertise. (Copyright 2005 Intel Corporation.)

The industry has since then successfully lived up to the expectations of Moore's law, although the rate of growth has been reduced to about a doubling of the number of transistors pr 18 months.

1.1.3 International Technology Roadmap for Semiconductors

In 1977 the five innovators Robert Noyce (Intel), Wilfred Corrigan (LSI Logic), Charles Sporck (National Semiconductor), W.J. Sanders III (AMD), and John

Welty (Motorola) founded the Semiconductor Industry Association (SIA). This trade association was founded with the purpose of representing the U.S. semiconductor industry. Among the activities of the SIA is to maintain the International Technology Roadmap for Semiconductors (ITRS). According to SIA the ITRS is ”the fifteen-year assessment of the semiconductor industry’s future technology requirements”.

The ITRS maps the challenges in all areas of the semiconductor industry, including testing. According to the 2005 edition of the roadmap ”Changing circuit sensitivities are likely to make defects that were benign in the past become killer defects in the future. For example, shorter clock cycles mean defects that cause 10s or 100s of picoseconds of delay are more likely to cause circuit failures. Furthermore power-optimized and/or synthesized designs will result in fewer paths with significant timing margin, which implies that random delay-causing defects will be more likely to cause failure. Similarly increasing noise effects, such as crosstalk and power/ground (GND) bounce, decrease noise and timing margins and again increase circuit susceptibility to defects” [ITR05].

In such a world, testing for delay defects is a necessity.

ITRS 2005 also emphasis the need for BIST techniques. There are mainly two reasons for BIST: at-speed testing, and the problem of test data volume increase at the ATE (Automatic Test Equipment).

1.2 Main contribution

The main contributions of this thesis may be summarized as follows:

A comprehensive system for path delay fault detection has been developed. The system encompasses:

- An exact path delay fault simulator.
- An automatic test pattern generator for detecting the K longest path-delay faults.
- Stimuli generators employing various arithmetic BIST schemes have been investigated.
- A well known pseudorandom source, the Mersenne twister, has been exploited as a gauge for pseudorandom quality of other sources, notably different arithmetic BIST schemes.

- A distributed fault simulator, with encapsulation in screen savers, to fully utilise idle computing power in a distributed network.

Exploration of the power of the system on iscas'85 circuit benchmarks, with the following results:

- Arithmetic BIST schemes may have strong pseudorandom properties, approaching the computationally expensive Mersenne twister.
- The overall most efficient Arithmetic BIST scheme for path delay fault detection, in terms of quality and overhead, is a weighted pseudo-random generator. The weights are based on relating signal values applied to the inputs of the circuit under test with the detected number of faults for that vector.
- Due to the large variation of characteristics of digital circuitry, there exists no overall best strategy for BIST-based path delay fault detection.

1.3 Organization of this thesis

This thesis is organized in the following manner:

In Chapter 2 an overview of existing delay fault models is presented, including the *path-delay fault model* which is the fault model assumed in all experiments presented in this thesis.

Chapter 3 gives an overview of different test application schemes for delay faults. One of the aspects of delay testing discussed here is that tests for delay faults require transitions to be applied to the circuit under test (CUT), and how such tests can be applied to the CUT.

In Chapter 4 common ways of classifying path-delay faults are presented. The two most important fault classes mentioned are the class of *robust sensitizable* and the class of *non-robust sensitizable* path-delay faults. These fault classes are included in the path-delay simulator, and the ATPG which was used during the experiments that will be presented in this thesis.

Chapter 5 discusses how path-delay fault simulation can be implemented, and presents an overview of the path-delay simulator developed for use in the experiments.

The number of paths and path-delay faults in a circuit can be exponential in the number of gates in the circuit. For such circuits it is only feasible to test a fraction of all the path-delay faults. Chapter 6 presents an ATPG for path-delay faults. The purpose of the ATPG is to sort out the longest testable paths

in the circuits. In the experiments different pseudo-random test generators were applied to the CUT, and their ability to detect these longest path-delay faults evaluated.

Chapter 7 presents a set of test pattern generators. The generators presented will later be evaluated with respect to their ability to detect path-delay faults.

Chapter 8 presents various experiments carried out with the test generators from Chapter 7, using the path-delay simulator from Chapter 5, and the path-delay ATPG from Chapter 6.

Chapter 9 presents concluding remarks and discusses future work.

In Appendix A a software library for directed acyclic graphs is given. The library was used as a part of the path-delay fault simulator and the path-delay fault ATPG developed.

Chapter 2

Delay fault models

In this chapter an overview of existing delay fault models is given. A more elaborate overview is found in [KC98].

2.1 Introduction

A simple sequential circuit with its primary and secondary inputs and outputs is illustrated in Figure 2.1. Let us assume that all primary inputs are driven directly by the output of flip-flops. This illustration will be used as an aid in the description of the delay test problem.

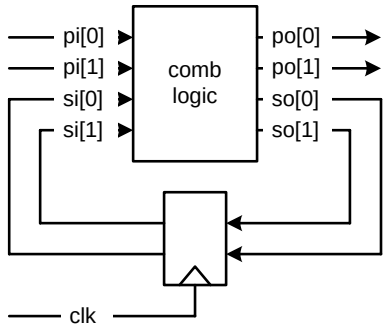


Figure 2.1: A sequential circuit.

During operation of the circuit, the combinational part of the circuit will receive stable input signals shortly after the flip-flops driving the input signals have settled. If the new input vector differs from the previous vector, a series

of signal change events will be generated through the combinational circuit towards the outputs. After a while the effects of the signal changes will start reaching the output signals, and may cause transitions. This is illustrated Figure 2.2. The boundaries of the transient region are determined by the delay of the shortest and longest combinational path present in the circuit. After all signals have settled, the next positive clock edge is applied and the process is repeated.

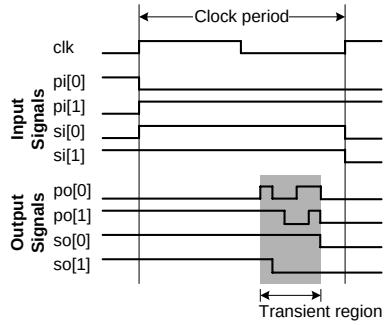


Figure 2.2: Signal transitions at the outputs of the combinational part of a sequential circuit.

In a properly manufactured and working chip the delay of the critical path is shorter than the clock period. Defects in the chip can however change the delay of the circuit so that it becomes longer than the clock period. If this happens the circuit is said to contain a delay fault.

Definition 1 (Delay fault) *A delay fault is a fault that cause the combinational delay of a circuit to exceed the clock period.*

Defects can be divided into two main groups, namely spot defects and distributed defects. Spot defects are local to a small area of the chip, and are e.g. caused by fine grained dust particles. Distributed defects may be caused by variations in the production process.

The delay fault models can also be divided into two main groups according to the type of defect that is captured most efficiently by the fault model. The transition fault model, the gate-delay fault model and the line-delay fault model are all examples of delay fault models that are used for representing defects located at single gates. The segment-delay fault and the path-delay fault, on the other hand, are used to model defects that are distributed over several gates. All these five fault models will be discussed in this chapter.

2.2 An ideal fault model

Chip manufacturers want to create chips that deliver the correct function and performance throughout the guarantee period of the product. In order to achieve this the chips have to be tested.

The purpose of testing is to separate the functionally faulty circuits from the good circuits. The pile of good circuits should not contain any functionally faulty (test escapes) circuits, and the pile of bad circuits should not contain any circuits without functional faults.

Study and modelling of physical defects and electrical problems (signal integrity, noise) in the circuits have proven to be the most successful way to isolate the functionally faulty circuits. Two observations are important in this context:

- A functionally faulty circuit *implies* that a problem of some sort exist in the circuit. The problem could be a defect in the circuit or signal integrity issues.

- A defect in the circuit does *not imply* that the functionality or performance of the circuit is *always* altered. However, a defect usually alters the functionality or degrades the performance of a circuit.

With these observations in mind, one way to sort the bad chips from the good is to look for defects. Throwing away all defective circuits may lead to discarding a properly working circuit (over testing). This is however not as big a problem as letting a defect circuit pass the test (test escape).

In order to design good tests that can isolate circuits with defects, it is necessary to find a good fault model. Ideally a fault model should capture the defects in the circuit in such a way that if a test for all possible faults is devised then the test will be able to sort all defective circuits from the others. Furthermore, the ideal fault model must be efficient enough to handle large designs. Efficient software, such as fault simulators and automatic test pattern generators, must also be possible to build around the fault model.

The following sections present some of the most successful fault models targeting delay faults. All models considered in the following sections model the physical defects at the logical level. Models at the electrical level is out of the scope of this thesis but might be necessary when dealing with for instance signal integrity problems.

2.3 Transition fault model

The *transition fault model* was first introduced in [WLRI87] and a definition as it appears in [BA02] is given in Definition 2.

Definition 2 (Transition delay fault model (from [BA02])) *It is assumed that in the fault-free circuit all gates have some nominal delay and that the delay of a single gate has changed. The gate-delay, usually an increase over the nominal value, is assumed to be large enough to prevent a passing transition from reaching any output within the clock period, even when the transition propagates through the shortest path. Possible transition faults of a gate are slow-to-rise and slow-to-fall types and hence the total number of transition faults is twice the number of gates. Transition faults model spot defects and are also called gross-delay faults."*

The main advantage of the transition fault model (compared with for instance the path-delay fault model) is that the number of faults is linear in the number of gates. Additionally, an ordinary ATPG for stuck-at faults can be used for creating transition fault tests. In order to detect a slow-to-rise or a slow-to-fall transition fault at a given gate, a transition must be propagated through the gate. This can only be achieved through a two pattern test $T = \{v_1, v_2\}$. One way to device a test for a slow-to-rise (slow-to-fall) transition fault is to find a test ,v_2, for the output of the gate stuck-at-0 (stuck-at-1). v_2 will cause the output of the gate to be 1(0) in a fault-free circuit. It also ensures that the transition fault is observable at one of the outputs. The first vector, v_1, can be any vector causing the output of the gate to be 0(1).

The transition fault model has some drawbacks as well. Figure 2.3 shows a small circuit with a slow-to-rise transition fault present at gate h. A valid test for the transition fault is $T = \{v_1 = 1010000, v_2 = 1110000\}$, which propagates a rising transition through the faulty gate to the output y. However, the path on which the transition propagates is very short and has a lot of slack. The large slack on the path may prevent the transition fault from being detected although the transition fault is present in the circuit and will cause erroneous behaviour in another situation. The assumption that the gate-delay is always "large enough to prevent a passing transition from reaching any output within the clock period", may not always be valid in practice due to large slack on some paths.

Another, somewhat unrealistic, assumption is that the delay fault only affects one gate in the circuit. A delay defect, even a spot defect, may affect more than one gate. Individually the delay faults may not cause any erroneous be-

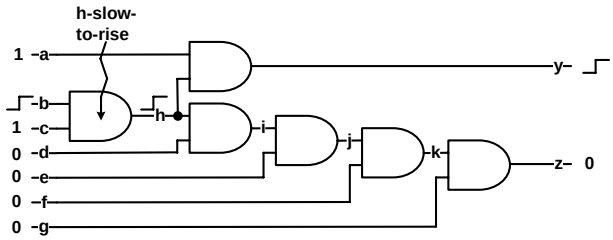


Figure 2.3: Test for a transition fault.

haviour, but together the increased delays in the gates might add up to a delay large enough to cause erroneous behaviour of the circuit.

2.4 Gate-delay fault model

The *gate-delay fault model* was developed by among others Iyengar [IRW90, IRS88a, IRS88b]. The gate-delay fault model has much in common with the transition fault model, but it tries to cope with its shortcomings by taking into account the size of the delay fault. A definition of the gate-delay fault model is given in Definition 3.

Definition 3 (Gate-delay fault model) *It is assumed that in the fault-free circuit all gates have some nominal delay and that the delay of a single gate has changed. The increase in this gate-delay is called the* size *of the gate-delay fault. Depending on the* size *of the gate-delay fault the gate-delay fault might cause performance degradation of long paths through the fault site and erroneous behaviour of the circuit.*

The main advantage of the gate-delay fault model is that the number of faults grow linear with the number of gates in the circuit. The main disadvantage is that the gate-delay fault model may fail to detect delay faults caused by several smaller defects.

2.5 Line-delay fault model

The *line-delay fault model* was proposed by Majhi *et al.* [MJPA96]. A definition of the line-delay fault model is given in Definition 4.

Definition 4 (Line-delay fault model) *It is assumed that in the fault-free circuit all lines (signals) have some nominal delay associated with them, and that the delay of a single line has changed. The line-delay, is assumed to be large enough to prevent a transition propagated along the longest sensitizable path through the fault site to reach the output within the clock period. Possible line-delay faults are* rising *and* falling *types and hence the total number of line-delay faults is twice the number of lines. Tests for line-delay faults detect spot defects and even some distributed defects.*

The main advantage of the line-delay fault model is that the number of faults is linear in the number of lines. Furthermore, it represents an improvement compared with the transition fault and gate-delay fault model in that it also detects some distributed defects. This can be illustrated by the following: A vector pair that creates and propagates a transition on the longest sensitizable path in the circuit will be a valid test for all lines on that path. A distributed defect present at more than one line might thus be detected even though the increased delay at the individual lines were too small to be detected by tests for the corresponding transition faults. However, since only one propagation path through each line is considered, it may fail to detect some distributed defects.

2.6 Segment-delay fault model

The *segment-delay fault model* was proposed by Heragu *et al.* [HPA96b, HPA96a]. A definition of the segment-delay fault model is given in Definition 5.

Definition 5 (Segment-delay fault model (from [BA02])) *A segment of length L is a chain of L combinational gates. Such a segment can be contained in one or more input to output paths. A segment-delay fault increases the delay of a segment such that all paths containing the segment will have a path-delay fault. If L is taken as the maximum combinational depth of the circuit, then segment-delay faults become the same as path-delay faults. For $L = 1$, segment-delay faults become identical to transition faults. Two faults, corresponding to two types (rising and falling) of transitions are modelled for each segment.*

The segment-delay fault model represents a trade-off between the transition fault model and the path-delay fault model. By choosing a sensible value of L, the number of segment-delay faults will be much smaller than the number of path-delay faults in the circuit. At the same time the set of segment-delay faults can detect distributed defects affecting the delay of the gates in the segments.

2.7 Path-delay fault model

The *path-delay fault model* was proposed by Smith [Smi85]. A definition of the path-delay fault model is given in Definition 6.

Definition 6 (Path-delay fault model (from [BA02])) *The delay defect in the circuit is assumed to cause the cumulative delay of a combinational path to exceed some specified duration. The combinational path begins at a primary input or a clocked flip-flop, contains a connected chain of gates, and ends at a primary output or a clocked flip-flop. The specified time duration can be the duration of the clock period (or phase), or the vector period. The propagation delay is the time that a signal event (transition) takes to traverse the path. Both switching delays of devices and transport delays of interconnects on the path contribute to the propagation delay.*

There are two path-delay faults associated with each physical path in the circuit: one slow-to-rise path-delay fault and one slow-to-fall path-delay fault. Thus, the total number of path-delay faults in a circuit is equal to twice the number of paths in the circuit.

The path-delay fault model has the ability to detect distributed defects caused by statistical process variations. A test for a path-delay fault will also detect eventual spot defects along the path. The delay fault model is therefore often considered to be closest to the ideal delay fault model [KC98].

The number of paths, and thus path-delay faults, can, due to reconvergent fan-out, be exponential in the number of gates in the circuit. An example of this is shown in Figure 2.4. Combinational multipliers are famous for a lot of reconvergent fan-outs and is another good example of a class of circuits with a huge number of paths.

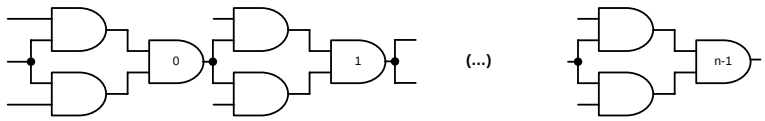


Figure 2.4: Circuit with an exponential number of path-delay faults.

The possible exponential number of path-delay faults makes it unfeasible to test all path-delay faults in circuits such as multipliers and other circuits with a lot of reconvergent fan-out. This is the main problem with the path-delay fault model.

Chapter 3

Test application schemes

In order to reveal any delay faults, it is necessary to bring the circuit under test, faulty or not, into a known state and then generate a transition that travels through the fault site. This requires both a test, and an algorithm or method for applying the delay test to the circuit under test. The latter is called a test application scheme, and in the following sections such schemes for both combinational and sequential circuits are presented.

Common to all schemes, except the rated clock tests in sections 3.5 and 3.6, is that initialization and generation of a transition is achieved through a two-pattern test. The first pattern initializes the circuit and the second generates the transition through the fault site.

The following assumption applies to all delay faults considered in this book: The maximum delay, T_s, of a faulty circuit will not exceed twice the maximum delay, T_r, of a fault free circuit. Two clocks can be derived from this assumption. The rated clock with period T_r, and the slow clock with period $2T_r$.

3.1 Slow-clock combinational test

Delay test of combinational circuits and sequential circuits with flip-flops only on inputs and outputs can be tested for delay faults by the test application scheme illustrated in Figure 3.1. In normal mode the flip-flops on the input and outputs are controlled by the same rated clock. However, in delay test mode, the input flip-flops and output flip-flops are controlled by two slow clocks, skewed relative to each other, but with the same frequency. In order to detect any delay faults a transition must be created and propagated through the circuit. This can be achieved by a two-pattern test $V = \{P1, P2\}$. The first test pattern initializes

the circuit to a known state independent of the presence of any delay faults. It is assumed that the period of the slow input clock is long enough to ensure that. The second test pattern is then applied in order to generate a transition and the response is sampled one rated clock period later by the output clock. A delay fault is detected if the content of the output registers is different from the expected result.

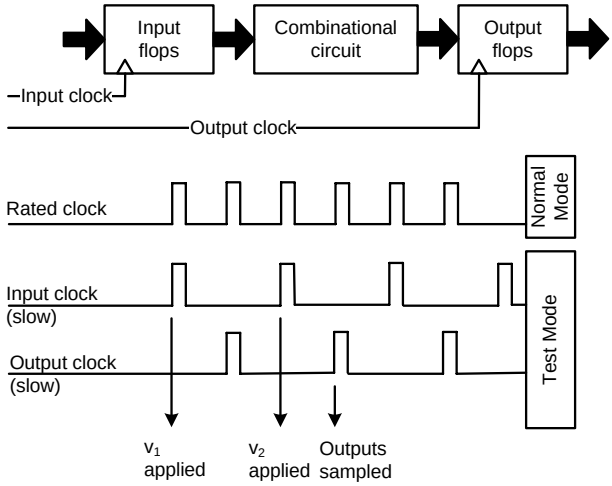


Figure 3.1: Slow-clock combinational test.

3.2 Enhanced-scan sequential test

This scheme [DS91] makes it possible to apply any two-pattern test to a sequential circuit provided that the circuit is equipped with enhanced-scan flip flops. Enhanced-scan flip-flops can store two bits instead of one. This makes it possible to scan in and store any two-pattern test in the enhanced-scan flip flops. During test the two pre-scanned test patterns are applied in two consecutive clock cycles. The disadvantages of this approach are very high area overhead and long test application time.

3.3 Standard scan testing

It is possible to test standard full-scan sequential circuits for delay faults as well, however not all two-pattern tests are possible to apply. This stems from the re-

strictions on the second test pattern in the two-pattern test. During test, the first test pattern is scanned into the flip-flops. The second test pattern on the other hand must be derived by either applying a one-bit shift to the scan flip-flops, or by using the captured response from the first test pattern. The first method is called *scan-shift delay test* [CDK93] or *skewed-load delay test* [Sav92, PS92], and the second method is called *functional justification* or *broad-side delay test* [SP94].

The advantage of this method is that if the circuit is already equipped with scan-cells it does not impose any additional costs except for the added area due to eventual reordering of scan-flip flops. One problem with this approach is that the fault coverage depends on the circuit under test. High coverage cannot be guaranteed due to the dependencies between the two test patterns in a test.

3.4 Slow-fast-slow sequential test

If the circuit has only partial scan or no scan at all, it is necessary to construct test programs. Each test program must go through three phases: *Fault initialization*, *fault activation* and *fault propagation*. The phases are shown in Figure 3.2. During the fault initialization phase the inputs and flip-flops in the circuit are set to a desired value through a sequence of input vectors. When the desired initial state is reached the fault is activated by applying a two-pattern test $T = \{v_1, v_2\}$. Both test patterns are justified functionally as described in section 3.3. Finally a test sequence is applied in order to propagate the result to the output.

Delay faults exist in all three phases of the test. In order to avoid manifestation of faults in the initialization and propagation phases a slow-fast-slow strategy is often applied where the CUT is running at the rated speed only during the fault activation phase. The use of the slow-fast-slow simplifies the analysis of the delay test problem and thus the test generation process. The test application process, on the other hand, is somewhat complicated due to the need for both a slow and a fast clock.

Unfortunately, there are one more challenge associated with this test application scheme. The illustration in Figure 3.2 suggests that the fault effect caused by a delay fault on the sensitized path is captured by the shaded flip-flop. However, other delay faults present in the circuit might also affect the signals and flip-flops and thus faulty values can be captured by more than the shaded flip-flop. This complicates fault simulation, since it is difficult to encertain what the next-state will be in the presence of multiple path-delay faults. Analysis of this problem has been carried out by Chackraborty *et al.* [CAB92].

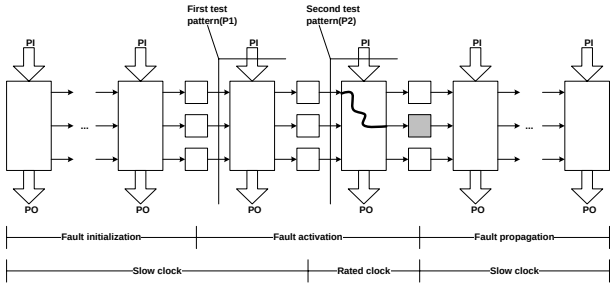


Figure 3.2: Slow-fast-slow sequential test

3.5 Rated-clock combinational test

Two-pattern test sequences are used in the test application schemes presented so far. These are valid under the assumption that the combinational logic reaches a steady state following the first test pattern before the second test pattern is applied. In order to satisfy this assumption a dual clock strategy is used with one slow and one rated clock.

Hsu and Gupta have stated [HG96] that *circuit manufacturers rely on the use of at-speed tests to detect delay faults. Typically, the coverage of delay faults is obtained by using built-in self-test where tests are applied at the normal circuit speed.* Ordinary two-pattern tests are invalidated when applied at-speed since it cannot be guaranteed that the combinational logic reaches a steady state following the first pattern before the second pattern is applied.

The research in [BAA98] shows that the path-delay fault coverage drops with as much as 90 % when two-pattern tests intended applied under a dual-clock scheme are applied at-speed. It is assumed that faulty circuit delays have an upper bound of two clock periods. Three consecutive patterns must thus be considered compared to the two necessary in a dual clock scheme. This assumption is pessimistic but necessary in order to ensure that tests are not invalidated. A 8-valued algebra suitable for at-speed simulation and a 41-valued algebra suitable for test generation of at-speed test patterns is presented in [BAA98].

3.6 Rated-clock sequential test

The at-speed or rated-clock test application scheme for sequential circuits is the easiest way to apply test patterns. However, the analysis and generation of test suitable for this test method is not trivial. The strategy assumes that the

fault is initialized, activated and propagated under a fast clock. A target delay fault may thus be activated in several time-frames. Analysis of this problem has been presented for the *path-delay fault model* in [BA95, BAA98], and for the *transition fault model* in [KT93].

3.7 Test application scheme assumed in this thesis

This thesis presents several accumulator based test generators for use in a built-in self-test environment. The analysis of these test generators is built around the following restrictions and assumptions:

- Only combinational circuits or sequential circuits that can be reduced to combinational circuits through enhanced-scan are considered.
- Only two-pattern tests are considered. It is assumed that the combinational logic reaches a steady state following the first test pattern before the second test pattern is applied.
- The maximum delay of a faulty circuit will not exceed twice the maximum delay of a fault free circuit.
- The path-delay fault model is the only fault-model considered.
- Any two-pattern test generated by a generator must be possible to apply to the circuit under test.

The only test application schemes that fulfils these requirements are the *slow-clock combinational test* and the *Enhanced-scan sequential test*. These are dual-clock strategies, however it is possible to apply the patterns correctly by using only the rated clock if the first test pattern is repeated [BAA98]. The repetition of the first test pattern will thus ensure that the signals reaches a steady state. The test programs presented in Chapter 7 may be applied using this test application strategy.

Chapter 4

Classification of path-delay faults

There are two common ways of classifying path-delay faults. The most common way of classifying path-delay faults [CC93] is based on the criteria used for *sensitizing* the target path-delay fault. Tests for path-delay faults have different *quality*. A high quality test for a path-delay fault guarantees the unconditional detection of the fault. However, high quality tests require very stringent conditions for path sensitization, and may thus be difficult to find for a given path-delay fault. If it is not possible to find a high quality test it might still be possible to find a lower quality test for the same path with less stringent path sensitization criteria. However, a lower quality test only guarantees the detection of the path-delay fault under certain assumptions, and the test may thus be invalidated if the assumptions are not met. It is common to partition the path-delay faults into the following classes based on sensitization criteria: *single-path sensitizable*, *robust, non-robust*, *functional sensitizable* and *functional unsensitizable*. These are found in the bottom row of Figure 4.1 [KC98]. This chapter gives an overview of these classes.

The number of path-delay faults may be exponential in the number of gates in the circuit, but not all paths have to be tested in order to guarantee the performance of the circuit. Division of paths into paths that have and don't have to be tested is another way of classifying path-delay faults. A lot of different methods of dividing path-delay faults in this way have been proposed in [KM94, GBA95, LSBSV95, SLCR95, KM95, CC96, KCC96, SS97]. By finding good ways of sorting out paths that do not have to be tested, it is possible to find a better estimate of the fault coverage. Methods for extraction of paths that

do not have to be tested will not be covered in this thesis. (An overview of some methods can be found in [KC98].) Hence, in this thesis the fault coverage will thus be measured as the number of detected faults relative to the total number of path-delay faults in the circuit.

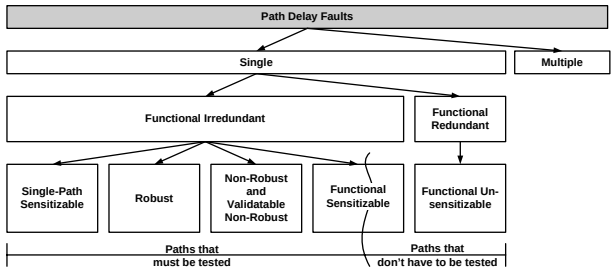


Figure 4.1: Classification of single path-delay faults.

First in this chapter, nomenclature used in the discussion is presented. The classification that will be presented is based on sensitizing criteria. These criteria are based on reasoning on transitions and thus a section is devoted to the discussion of transitions in Section . The sections 4.3 through 4.7.

4.1 Terminology

This section presents some terms and symbols used throughout the discussion in this chapter.

A combinational circuit, $C(G, F)$, consists of a set of gates G and a set of connections F between the gates.

Each physical path $P = \{g_0, f_0, g_1, f_1, ..., f_{k-1}, g_k\}$ in the combinational circuit $C(G, F)$, is identified by the list of alternating gates g and connections f the path passes through. The first gate g_0 is a primary input and the last gate in the path g_k is a primary output.

Two path-delay faults P_x, $x \in \{rising, falling\}$, are associated with each physical path one for each type of transition applied to the input of the physical path.

The connection f_i, $0 \leq i \leq k-1$, is a signal *on* the path P_x and connected to one input of gate g_{i+1}. Such signals are referred to as *on-inputs* to P_x. The gates on the path P_x also have other inputs which are not on the path. These are called *off-input* signals to P_x.

The input of a gate can either have a controlling (cv) or non-controlling

(*ncv*) value. A controlling value applied to the input of a gate determines the value at the output of the gate alone (i.e. independent of values at other inputs of the gate). The controlling value for *AND* and *NAND* gates is 0, and for *OR* and *NOR* gates the controlling value is 1. The non-controlling value for a gate is the complement of the controlling value.

Detection of path-delay faults requires a two-pattern test $T = \{v_1, v_2\}$ to be applied to the circuit under test. Further details may be found in Chapter 3.

4.2 The nature of transitions

4.2.1 Transition

The classification of path-delay faults is based on an analysis of the way *transitions* are propagated along the different paths in the circuit. It is therefore important to have a good understanding of how transitions behave on the way through the circuit. A transition should be understood as described in Definition 7.

Definition 7 (Signal transition) *A* transition *at a signal or a gate in a circuit is a single change in the logic level. A transition may be of one of two different types. A signal change from* 0 *to* 1 *is called a* rising *transition, and a signal change from* 1 *to* 0 *is called a* falling *transition.*

4.2.2 The maximum number of transitions and their position in time

In order to test a path-delay fault a transition must be applied to the input of the path. Although only one transition is applied to only one input, one might observe several transitions at the output of the path, and at other outputs as well. This effect is caused by fan-out present in the circuit. Figure 4.2 a illustrates how one transition at each of the inputs of a gate might result in two transitions at the output of the gate. Another example is shown in Figure 4.2 b.

The maximum number of transitions that can be observed at the output of a gate is equal to the sum of the maximum number of transitions observable at each input of that gate. One pass through a topologically sorted netlist using this rule will yield the maximum number of potential transitions observable at any gate in the circuit. This is illustrated in Figure 4.2 c where it is assumed that one transition is applied to each input. An interesting observation is that the maximum number of transition observable at the output of the circuit is

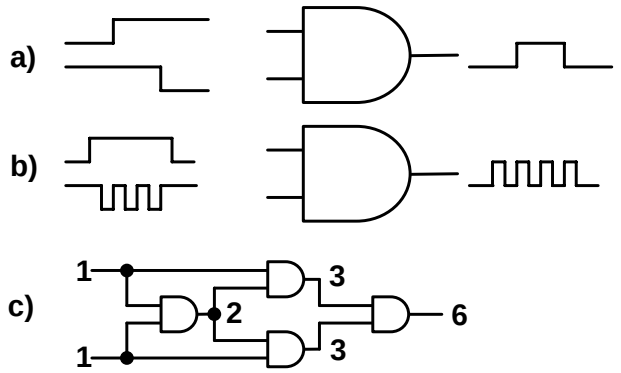


Figure 4.2: The maximum number of transitions at a gate.

equal to the number of paths in the circuit. However, in practice one will not experience the maximum number of transitions because transitions are often masked (see Figure 4.3) and prevented from further propagation through gates by controlling values on the other gate inputs. The actual number of observed transitions depends on the exact delay of the gates and nets in the circuit under test in addition to its structure and the applied test patterns.

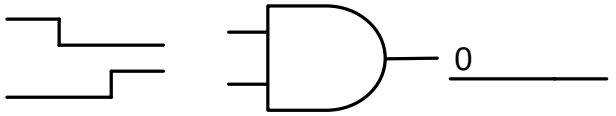


Figure 4.3: Masking of transitions.

A transition propagating through a circuit is delayed by an amount equal to the sum of the delays through the gates and nets on the path it is travelling through. An example, using a simple transport delay, is shown in Figure 4.4 a. The circuit contains two paths and a maximum of two transitions may thus be observed at the output o. The first transition stems from the transition propagated through the shortest path $i-k-o$ with a delay of 1 unit, and the second transition stems from the transition propagated through the longest path $i-j-o$ with a delay of 2 units. If the delay of a path is increased, the position of the corresponding transition observable at the output will be adjusted with the same amount. This is illustrated in Figure 4.4 b where the longest path is increased by 1 unit.

The number of paths in the circuit and the delay of the paths determine the

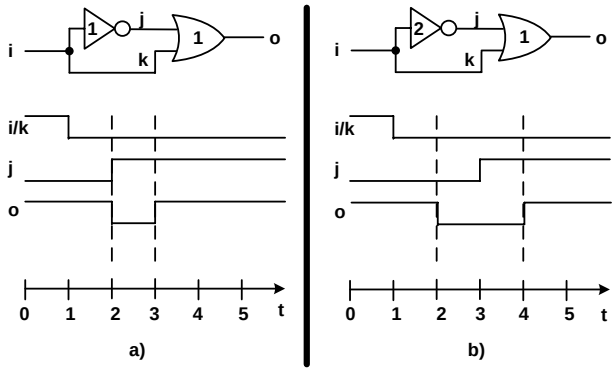


Figure 4.4: Propagation delay through paths.

points in time where transitions might occur at the output of an circuit. Whether or not an transition is observable at the output depends on, as mentioned earlier, both the structure of the circuit, the delay of the paths and the applied test patterns. Figure 4.5 a shows an imagined circuit with 6 paths from the inputs to the output. The delay of the paths is shown in Figure 4.5 b. From this information one can deduce that maximum 1 transition may be observed at the output 1 unit after a transition is applied to the input of the circuit, no transitions may occur after 2 units, a maximum of three transitions may occur after 3 units, and maximum 2 transitions may occur after a delay of 4 units. The observed waveforms at the output of the circuit might thus be similar to the ones shown in Figure 4.5 c for three different two pattern tests vectors.

4.2.3 Hazards

When more than one transition occurs during a short time interval, the transitions are referred to as a *hazard*. There are two types of hazards *static* and *dynamic*. *Static hazard* is the term used on a group of two rapid transitions from the initial value and back again. There are two types of static hazards *static one hazard* (also called *low going glitch*) and *static zero hazard* (also called *high going glitch*). The hazard is called a *static one hazard* when the initial signal value is 0 followed by a $0 \rightarrow 1$ transition and a $1 \rightarrow 0$ transition. The hazard is called a *static zero hazard* when the initial signal value is 1 followed by a $1 \rightarrow 0$ transition and a $0 \rightarrow 1$ transition. *Dynamic hazard* is the term used on a group of three rapid transitions from the initial value to the final value, back to the initial value, and back to the final value again.

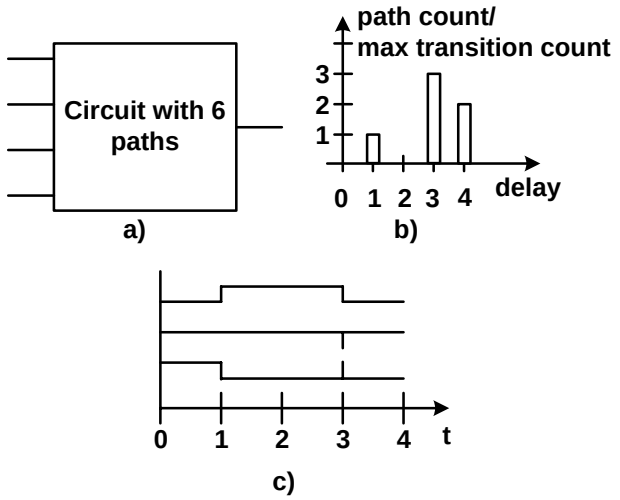


Figure 4.5: Possible transitions and their location in time.

4.3 Single-path sensitizable path-delay faults

A delay fault on a path that is single-path sensitizable is called a single-path sensitizable path-delay fault. The criterion for a path to be single-path sensitizable is that all off-inputs must be set to *ncv* for both test vectors. If a transition is propagated through a path with stable values on the off-inputs, only one transition will be observed at the output of the path. A single-path sensitizable path-delay fault is very rare but it is of highest quality and guarantees that the design under test will fail *if and only if* the path under test has excessive delay. Since only a fault on the target path will be detected, single-path sensitizable path-delay faults are suited for diagnostic purposes. However, only a very small number of paths are single-path sensitizable.

Figure 4.6 shows a typical waveform observed at the output of a single-path sensitized path for a good circuit, a circuit where the target path has excessive delay, and a circuit with several path-delay faults (including the target path). Since all off-inputs have stable non-controlling values under both vectors (see Figure 4.7), no other paths will be able to interfere with the propagation of the transition along the target path, and thus the detection of a path-delay fault on the target path will be guaranteed even if the circuit contains multiple path-delay faults. The arrival time t_p of the transition is a measure of the delay of the path. The target path contains a path-delay fault if and only if t_p is larger than the clock period T.

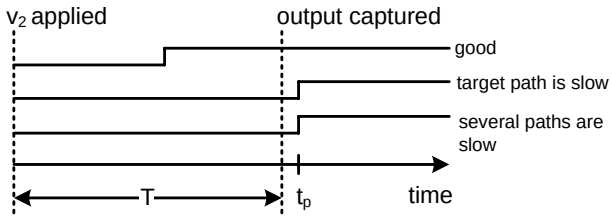


Figure 4.6: Test of single-path sensitizable path-delay fault.

Fault class /path example	Sensitization Criterion	
	on-input	off-input
Single-path sensitizable (SS)	cv→ncv ncv→cv	ncv→ncv ncv→ncv
Robust testable path (RT)	cv→ncv	cv→ncv
Non-robust testable path (NR)	ncv→cv	cv→ncv
Functional sensitizable path (FS)	ncv→cv	ncv→cv
Functional unsensitizable path (FU)	cv→ncv cv→ncv ncv→cv	cv→cv ncv→cv cv→cv
Unsensitizable path(US) (When no transitions on the on-input occur)	cv→cv ncv→ncv	* *

Figure 4.7: Fault classes and sensitization criteria.

Figure 4.7 summarizes the sensitiziation criteria for the fault classes presented in this chapter. The first column lists the different fault classes together with two examples of each fault class. The two last coloumns lists the particular requirements for $on-inputs$ and $off-inputs$. A path-delay fault can be tested

according to a particular sensitization criterion if one or more gates on the path (*shaded* in Figure 4.7) is sensitized according to the particular criterion, and the rest of the gates are sensitized according to a more restricted sensitization criterion (*white* in Figure 4.7).

4.4 Robust testable path-delay faults

In 1985 Smith wrote the paper *Model for Delay Faults Based upon Paths* [Smi85] where he introduced the path-delay fault model. Delay simulation using physical models is very computationally expensive. This motivated Smith to find a detection criterion for path-delay faults (see Definition 8) that was *independent* of the numerical delay values of the gates in the circuit. The detection criterion was formalized by Lin and Reddy [LR87], which also gave the name *robust path-delay faults* to the set of path-delay faults detected by this criterion. A short description of the detection criterion follows.

For a target path-delay fault to be robustly tested a transition must be applied to the input of the path and propagated through the path to the output. The test is robust if no transition is propagated to the output of any gate on the target path before a transition has occurred on the *on-input* of that gate. The first transition observed at the output of the path is thus known to be delayed with at least as much as the delay on the target path. If this transition is late at the output, the test has detected a robust path-delay fault. A robust sensitizable path-delay fault is of highest quality and guarantees that the fault can be observed independently of the delays on signals outside the target path.

The robust sensitization criterion can be described in terms of transitions produced on the paths on- and off-inputs as follows [CKC96, KC98]: The robust sensitization criterion is equal to the criterion for single-path sensitization except that it is also allowed for off-inputs to have $cv \rightarrow ncv$ transitions if the corresponding on-inputs of the gates have a $cv \rightarrow ncv$ transitions (see Figure 4.7).

Definition 8 (Robust path-delay test [BA02]) *A robust path-delay test guarantees to produce an incorrect value at the destination if the delay of the path under test exceeds a specified time interval (or clock period), irrespective of the delay distribution in the circuit.*

Figure 4.8 shows a typical waveform observed at the output of a robustly sensitized path for a good circuit, a circuit where the target path has excessive delay and a circuit with several path-delay faults (including the target path). It

is guaranteed, in all three cases, that the first transition observed at the output of the path is delayed with a time t_p that is equal or larger than the delay on the target path. This property holds even if the circuit contains multiple path-delay faults. Eventual transitions observed subsequently stems from paths with even larger delay.

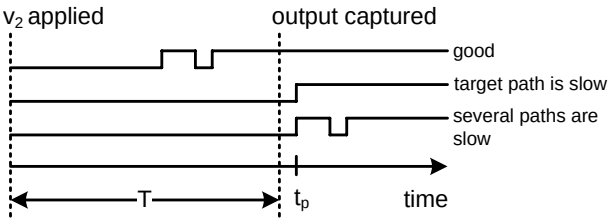


Figure 4.8: Test of robust path-delay fault.

4.5 Non-robust testable path-delay faults

A delay fault on a path that is non-robust sensitizable is called a non-robust testable path-delay fault (see Definition 9).

Definition 9 (Non-robust path-delay test [BA02]) *A test that guarantees to detect a path-delay fault, when no other path-delay fault is present, is called a non-robust test for that path. A path-delay fault for which a non-robust test exists is called a "singly testable path-delay fault [GBA95]."*

The non-robust sensitization criterion can be described in terms of transitions produced on the paths on- and off-inputs as follows [CKC96, KC98]: The criteria for a path to be non-robust sensitizable is equal to the criteria for robust sensitization except that it also allows for the off-inputs to have $cv \rightarrow ncv$ transition if the corresponding on-input of a gate has a $ncv \rightarrow cv$ transition (see Figure 4.7).

Figure 4.9 shows a typical waveform observed at the output of a non-robustly sensitized path for a good circuit, a circuit where the target path has excessive delay and a circuit with several path-delay faults (including the target path). It is guaranteed that excessive delay on the target path will be detected if the target path is the only faulty path in the circuit. However, if multiple faulty paths are present, the fault effect might be masked by the delays on other paths than the target path. In such cases the delay fault is said to be *invalidated*.

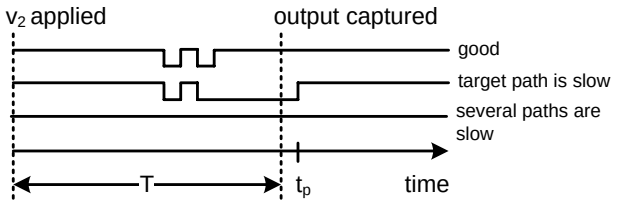


Figure 4.9: Test of non-robust path-delay fault.

A non-robust test can be invalidated if the $cv \rightarrow ncv$ transition on the off-input occurs arrives later than the on-input $ncv \rightarrow cv$ transition. The following example illustrates invalidation of a non-robust test. Figure 4.10 (top) shows the situation when the path leading to the off-input is fault free and arrives early. The transition on the on-input is thus successfully propagated to the output of the gate and the test is thus not invalidated. The opposite happens in Figure 4.10 (bottom). In this situation the path leading to the off-input is faulty and arrives later than the transition on the on-input. The transition on the on-input is thus masked and the test invalidated. Cheng *et. al* presents methods in [CKC96] for reducing the probability of invalidation and thus increasing the quality of non-robust tests.

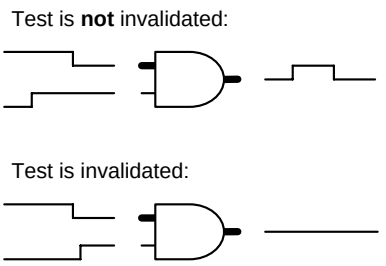


Figure 4.10: Invalidation of a non-robust test.

4.6 Functional sensitizable path-delay faults

A delay fault on a path that is functional sensitizable is called a functional sensitizable (FS) path-delay fault. The criteria for a path to be functional sensitizable is equal to the criteria for non-robust sensitization except that it is also allowed for the off-inputs to have $ncv \rightarrow cv$ transition if the corresponding on-input of a gate also has a $ncv \rightarrow cv$ transition (see Figure 4.7).

A functional sensitizable path-delay fault requires that there exists more than one faulty path in the circuit in order for the target fault to be detected. If at least one FS off-input is not late, the FS path cannot impact the performance.

For the other faults so far it has been enough to find one test per fault in order to detect the fault. FS paths on the other hand always works in groups. Thus, it is necessary to create a number of tests for a target FS path; one for each group it is a part of. A group of such collaborating FS faults are classified as a primitive fault [KM94, KM95, KC98].

Fortunately not all FS paths have to be tested. A functional sensitizable path-delay fault cannot affect performance unless some other fault exists in the circuit as well. If all these other faults are tested and pass the tests, the functional sensitizable path cannot affect performance and hence does not have to be tested. There are a lot of different ways to partition the set of paths that does not have to be tested. Every approach agrees on how to deal with functional unsensitizable faults: they don't have to be tested at all, but the big group of functional sensitizable faults can be divided in many different ways. The goal is of course to reduce the number of paths that needs to be tested down to a minimum. Some approaches can be found in [KM94, GBA95, LSBSV95, SLCR95, KM95, CC96, KCC96, SS97].

4.7 Functional unsensitizable path-delay faults

Functional unsensitizable (also called functional redundant) path-delay faults do not need to be tested at all. A definition is given in Definition 10 (see also Figure 4.7).

Definition 10 (Functional unsensitizable path-delay fault [KC98]) *Functional unsensitizable path-delay faults are defined as the faults for which under all possible input vector pairs, either (1) at least one off-input in the path has a controlling value under vector v_2 when the corresponding on-input has a non-controlling value, or (2) at least one off-input is assigned a stable controlling value. Such off-inputs are called functional unsensitizable off-inputs.*

Chapter 5

Path-delay fault simulation of combinational circuits

This chapter is devoted to the discussion of the path-delay fault simulator constructed in order to experiment with the stimuli generators found in Chapter 4.2. The path-delay fault simulator [GA05a, GA05c, GA05b], named *PDFSim*, is a part of the *GFault* test-tool suite. Successful construction of a path-delay simulator requires careful attention to the following:

- *Simulation algebra*. It is necessary to have a good understanding of the path-delay fault model and the different path-delay fault classes. This was discussed in Chapter 4. The sensitization rules discussed Chapter 4 can be implemented in a simulation algebra (alphabet and logic rules) which is a crucial part of any logic/fault simulator. *PDFSim* uses the algebra developed by Smith and presented in [Smi85].

- *Netlist representation*. A thought-through computer representation of the netlist simplifies the implementation of algorithms on top of the netlist representation (simulation, fault grading, etc.) as well as increasing the speed of the simulator. It also makes it easier to extend the simulator with other features.

- *Logic simulation*. There are two common ways of implementing the signal propagation procedure in a logic simulator: compiled-code simulation and event-driven simulation. *PDFSim* is an event-driven simulator.

- *Fault grading*. Fault grading, the process of finding the number of detected faults to the total number of faults, is an important task in any fault

simulator. However, fault grading is much more complicated in a path-delay fault simulator than in e.g. a stuck-at fault simulator because the circuit under test may contain an exponential number of path-delay faults in the number of gates. Relations between fault grading and estimation of real defect coverage is important to assess the overall test quality.

- *Stimuli generators*. The main purpose of designing the simulator was to experiment with different stimuli generators. It must therefore be easy to add new generators to the simulator.

- *Other aspects*. Result presentation, test cases, parallel processing.

A discussion of all these important aspects of designing path-delay simulators follows, using the features and choices made during the implementation of *PDFSim* as an example. The software architecture of *PDFSim* is presented in the end of this chapter.

5.1 Simulation algebra

This section presents simulation algebras for the two path-delay fault classes supported by the *PDFSim* path-delay fault simulator: robust and non-robust path-delay faults. The algebras presented is based on the algebra presented by Smith in [Smi85]. Other algebras can be found in e.g. [LR87, BAA98].

5.1.1 Algebra for robust propagation of path-delay faults

Smith presented a six value alphabet for simulation of *robust* path-delay faults in [Smi85]. The signal values during the two pattern test is encoded into these symbols. Each symbol consist of two ordered pairs: $S0$, $S1$, $P0$, $P1$, -0, -1. The second element in each symbol is a boolean 0 or 1 indicating the final value after the second test pattern has settled. The first element can be one of S (steady), P (path), $-$ (neither S nor P).

The value S is reserved for gates with stable, hazard free, values during both test vectors. $S0$ is assigned to stable low signals and $S1$ is assigned to stable high signals.

The value P is reserved for gates g_i that satisfy the following conditions:

- At least one of the gate-inputs must be P. This ensures that at least one path exists, on which all gates have a P-value, from the inputs of the circuit to the gate g_i.

- The initial and final signal values at g_i must be different. $P0$ is given to gates with a *low* final value. $P1$ is given to gates with a *high* final value.
- Any odd number of transitions may take place at the output of g_i, but the output must be guaranteed not to change before the first transition on all the inputs of g_i with a P-value has arrived.

Signals that do not meet the criteria for S and P are given the value $-$. A gate with the value $-$ assigned to it, may have none, one or many transitions. Thus the initial and the final value of such a signal may and may not differ. The final value is the only value that can be determined. -0 is assigned to signals with a *low* final value. -1 is assigned to signals with a *high* final value.

Illustrations of the symbols in Smith's alphabet is given in Figure 5.1.

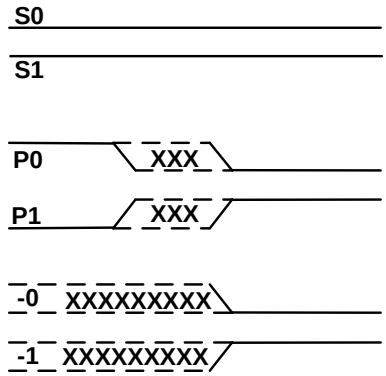


Figure 5.1: Symbols in Smith's alphabet.

A test will be a robust test for a path if all the gates on that path have been assigned either the $P0$ or the $P1$ value, otherwise the test is not a robust test for that path. Many paths can satisfy this condition at the same time, thus a two pattern test may detect more than one robust path-delay fault.

5.1.2 Implication tables for robust propagation

By using the rules for robust propagation of path-delay faults explained in Section 5.3, it is straightforward (details can be found in [Smi85]) to create truth tables for common gate types. Truth tables for the two-input *and-gate*, *or-gate* and the *inverter* are presented in Figure 5.2.

Truth tables for other gates such as *NAND*, *nor*, *XOR* and *XNOR* are found easily by creating functionally equivalent circuits out of *and-gates*, *or-gates*,

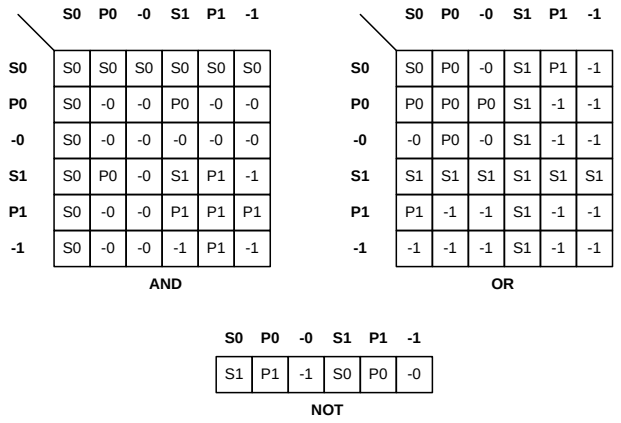


Figure 5.2: Implication tables for robust propagation of and-gates, or-gates and inverters (not).

and *inverters*. Correct output value of gates with more than two inputs may be found in a similar manner. Truth tables may also be generated and stored for larger boolean functions as well. The size of a truth table for a function with n inputs is 6^n, so it is only feasible to store truth tables for functions with a small number of inputs. Only truth tables for *inverter* and two-input *and*, *NAND*, *or*, *nor*, *XOR*, and *XNOR* gates are stored in *PDFSim*.

A logic simulator must be able to efficiently compute the correct output value of a gate given the values on the inputs of the gate. One way to do this is by enumerating the symbols in the simulation alphabet with integers from 0 to $|A| - 1$ ($|A|$ is the number of symbols in the alphabet). The truth table for a gate with n inputs can thus be stored as an n-dimensional array indexed by the integer value of the simulation value present on the inputs of the gate. This lookup table approach is used in *PDFSim*.

Other methods, that avoids lookup-tables, exist as well. One method is to encode each symbol in the alphabet in such a way that the boolean instructions in the computer, such as *and/or* instructions, can be used directly to compute the correct output of the corresponding gate. This approach is not more efficient than the lookup-table approach, but by packing k symbol into each computer word it is possible to simulate k vectors at a time. *Word-parallel event driven simulation* will however not give k times speedup, since an *word-parallel* simulator would generate more events than a simulator simulating only one test vector at a time.

5.1.3 Algebra for non-robust propagation of path-delay faults

It is possible to modify the algebra for robust propagation presented in Section 5.1.1 so that it performs non-robust propagation instead. This can be achieved by consulting the sensitization criteria for non-robust propagation of path-delay faults is described in Section 4.5. The criteria for a path to be non-robust sensitizable is equal to the criteria for robust sensitization except that it is also allow for the off-inputs to have $cv \rightarrow ncv$ transition if the corresponding on-input of a gate has a $ncv \rightarrow cv$ transition (see Figure 4.7). In other words, in order for a path to be non-robust testable, all off-inputs have to settle to the non-controlling value under v_2.

If the on-input of an *AND* gate has a $ncv \rightarrow cv$ transition, the on-input will be assigned the value $P0$. If the path is non-robustly sensitized it is allowed for the off-input to have a $cv \rightarrow ncv$ transition (in addition to the values allowed by the robust sensitization criteria). This corresponds to $-1/P1$ assigned to the off-input. The output of the *AND* gate should thus be assigned the value $P0$ if one input has the value $P0$ and the other $-1/P1$. These modifications are implemented in the implication table presented in Figure 5.3.

If the on-input of an *OR* gate has a $ncv \rightarrow cv$ transition, the on-input will be assigned the value $P1$. If the path is non-robustly sensitized it is allowed for the off-input to have a $cv \rightarrow ncv$ transition (in addition to the values allowed by the robust sensitization criteria). This corresponds to $-0/P0$ assigned to the off-input. The output of the *OR* gate should thus be assigned the value $P1$ if one input has the value $P1$ and the other $-0/P0$. These modifications are implemented in the implication table presented in Figure 5.3.

5.1.4 Implication tables for non-robust propagation

Implication tables [SFF89] for non-robust propagation is presented in Figure 5.3.

5.2 Netlist representation

The fundament on to which every other part of the simulator is built is the netlist representation. In this context the representation of the netlist involves the *gates*, the *connections* between the gates, and *attributes* associated which each gate or connection in the graph. Since the netlist usually pervades the whole simulator it is important to carefully think through the implications of the netlist architecture, or one can easily end up with a slow simulator that is difficult to maintain and extend with new features.

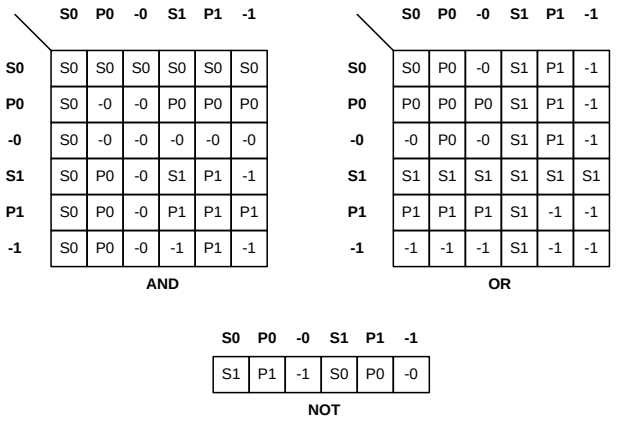


Figure 5.3: Implication tables for non-robust propagation of and-gates, or-gates and inverters (not).

Experience with four early versions of the path-delay fault simulator have lead to a netlist representation that is built around a library of graph manipulation subroutines for directed acyclic graphs. This unlocks all the power of graph-theoretic terminology, concepts and algorithms developed through the years[1]. This makes it very easy to maintain, extend and modify the source code. A presentation of the graph library is found in Appendix A. The netlist architecture is illustrated in Figure 5.4 and contains the following components:

- Two instances of a directed acyclic graph class is used in order to represent the *structure* of the netlist. One of the DAGs stores the fan-outs associated with each gate and the other stores the fan-ins associated with each gate in the netlist. This makes it very easy to perform forward and backward propagation through the circuit.

- The netlist also contains two instances of a event-list class. One event-list associated with each DAG. The event-list is used both during signal propagation and fault grading. More information about the event list is found in Section 5.3.

- A container for storing attributes associated with each gate and net in the netlist. This includes the gate type associated with each gate, the delay associated with each gate or net, and also the driver and readers of a

[1]Leonhard Euler's solution to the *Seven Bridges of Königsberg* problem in 1736 is considered the first theorem of graph theory

particular net. It also contains structural information extracted from the DAG instances such as lists of *sources* and *sinks* in the graph.

- It is possible to reduce the size of the netlist by extracting fan-out free logic cones. These cones can be thought of as super-gates containing other gates. Common for all gates in a cone/super-gate is that all gates have a fan-out of one, except the gate driving the output of the cone. If a netlist is built by such super-gates, it will result in a smaller netlist measured in the number of gates, but it will nevertheless retain the fan-out structure of the original netlist. This can be used in order to speed up e.g. the fault-grading algorithm in the path-delay fault simulator. The netlist representation contains two such cone-extracted DAGs corresponding to the two un-reduced DAGs also representing the netlist. Cone extraction is described in detail in Section A.16.
- Two event-list associated with the two cone extracted DAGs.

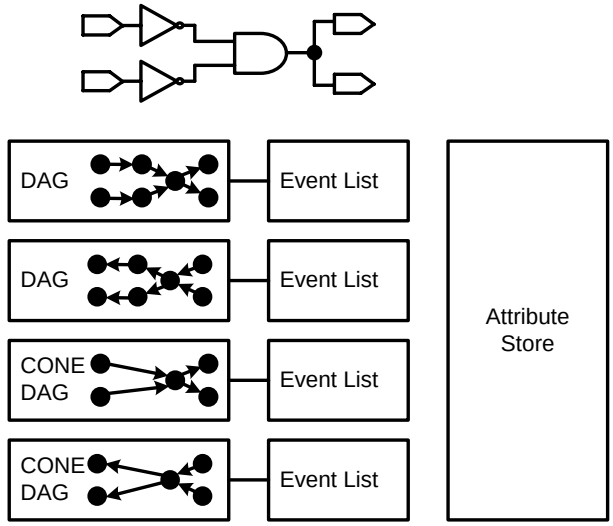


Figure 5.4: Netlist architecture.

5.3 Logic simulation

Each gate in the CUT has a signal value associated with it. *Logic simulation* is the process of computing the correct simulation value of all gates in the circuit.

There are two common logic simulation algorithms: compiled-code simulation and event-driven simulation.

In a *compiled-code simulator*, the netlist is levelized (topologically sorted), represented in some programming language and compiled. During simulation, the correct signal value of each gate is computed one at a time in topologically sorted order. The levelization assures that the signal value of a gate is not computed before the signal values of the gates driving the inputs of that gate is computed. All gates in the netlist are visited once.

When a new test vector is applied during simulation, it can cause one or more signal changes at the inputs of the circuit. These signal changes are called events. If an event reaches one of the inputs of a gate, it might cause the output of that gate to change as well. The result is a chains of events that ends at primary outputs or internal gates that block the event propagation. *Event-driven simulators* traces such event chains in order to identify the gates that need recomputation of the signal value, and thus avoid unnecessary signal value recomputation of gates that cannot change signal value.

The rightmost graph in Figure 5.5 illustrates how a signal-change at one input generates a chain of events. In an event-driven simulator, only the simulation value of the gates on the chain need to be recomputed. The leftmost graph illustrates how all simulation values are updated in a compiled-code simulator. The *PDFSim* path-delay-fault simulator is implemented as an event-driven simulator in order to avoid updating unnecessary signals. This is especially important since the test generators generate single-input-change test patterns.

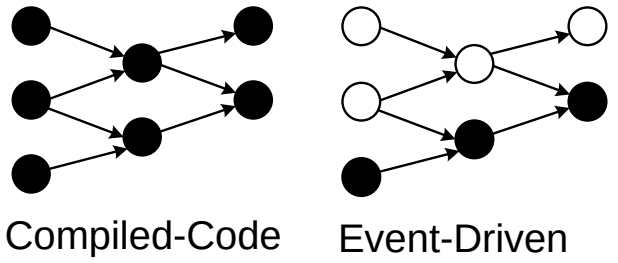


Figure 5.5: Compiled-code and event-driven simulation.

Crucial to any event-driven simulator is the *event-list* datastructure. The purpose of the event-list is to sort the events according to the logic level of which they occur in order to provide evaluation of the gates in levelized order. (Events can also be scheduled in *time* in order to facilitate simulation of arbitrary delays. This makes it possible to simulate switching activity as a function

of time. This is, however, not implemented in *PDFSim*.)

Figure 5.6 illustrates the architecture of the event-list found in *PDFSim*. The event-list is actually made up from several small lists, one associated with each level in the netlist. The size of these list is equal to the number of gates in the corresponding level. Each entry in these lists identifies a gate that must recompute its simulation value due to a change in its input values (an event). Glitches are filtered out by allowing each gate to occur only one time in the event-list. An array of visited-flags are used in order to keep track of whether a gate exists in the event-list or not.

Imagine that a new test vector is applied to the little netlist in Figure 5.6 (top) that changes the state of both input ports 0 and 1. The following initialization steps are then taken by the simulator for this particular case: The input-ports 0 and 1 are added to the event list associated with Level 0. The event-list pointer is updated to 1 to indicate the position of the last entry in the list. The visited-flags for input-ports 0 and 1 are changed from false to true.

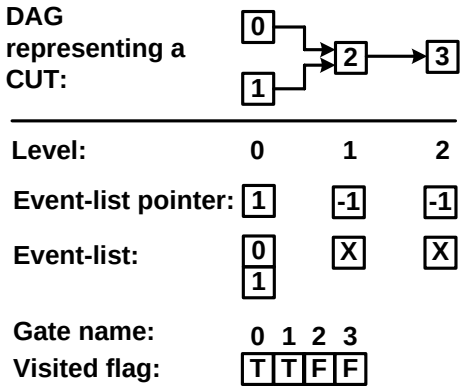


Figure 5.6: Event list architecture.

When the initialization steps are finished, the simulator starts recomputing the signal values of the gates according to Algorithm 1. If the event-list is not empty the event-list removes a gate g from the non-empty list with the lowest level (see Figure 5.6) and clears the corresponding visited-flag. A new signal value is computed for gate g, and if it differs from the old signal value, the fan-out gates of g are inserted into the event-list in the appropriate position if they are not already in the event-list. The simulation finishes when the event-list is empty.

Algorithm 1 (Event-based simulation)

```
EventSimulate(E, G, S){
  /*INPUT   : E, an event-list                          */
  /*INPUT   : G, a directed acyclic graph               */
  /*INPUT   : S, an array of signal values              */
  while(E is not empty){
    g = E.Remove();
    OldSignalValue  = S[g];
    NewSignalValue= UpdateSignalValue(g);
    S[g] = NewSignalValue;

    if (NewSingalValue != OldSignalValue){
      E.InsertOutgoingEdges(g);
    }
  }
}
```

5.4 Fault grading

When the logic simulation algorithm presented in Algorithm 1 is finished with computing the signal values, it is time for the fault grading algorithm to *count the exact number of detected path-delay faults*. This problem is very hard for one reason: The possibly *exponential* number of path-delay faults in the number of gates that a circuit can contain. One good example is the ISCAS'85 benchmark C6288 that contains $1.98 * 10^{20}$ path-delay faults.

There are two different types of fault-grading algorithms: enumerative and non-enumerative. *Enumerative* algorithms assigns a unique number [BAA93] to each path-delay fault and when a fault is detected the corresponding number is stored in a list. The enumerative version of the problem is intractable by definition since an exponential number of path-delay faults are counted one by one. *Non-enumerative* algorithms avoid enumeration and handles the detected faults implicit instead. Unfortunately, the non-enumerative version of the problem was shown in [KT02] to be intractable as well.

5.4.1 The simplest fault grading algorithm

A path-delay fault is detected, as described in Section 5.1.1, if the simulation value of all gates on the path is either $P0$ or $P1$. The simplest way of extracting the path-delay faults detected by a particular test vector, is to carry out a depth-first search along the sensitized paths in the graph starting at primary outputs with simulation value equal to either $P0$ or $P1$. (The search can be conducted in the other direction as well, but that is more inefficient since a partial path might not be sensitized all the way to an output. This is however guaranteed

when starting the search at an output.) Each time the search visits a primary input, it indicates that a new path-delay fault has been detected by the current test vector. If the path-delay fault has not already been detected by some other test vector, it is added to the list of detected faults as a list of vertices together with the transition applied to the input of the path.

5.4.2 The enumerative algorithm implemented in *PDFSim*

This section is devoted to the discussion of the fault grading algorithm implemented in *PDFSim*. The presentation is divided into three parts. The method used for assigning a unique number to each structural path is explained first. Then the method for extracting the detected path-delay faults detected by one test vector is described. Finally it is explained how the detected faults are stored and checked against previously detected faults.

Path enumeration

The fault grading algorithm is an enumerative algorithm. Such algorithms assigns a unique number to each path. The method used is similar to the one in [BAA93], and is easiest explained by an example. Figure 5.7 shows a DAG that represents a simple combinational circuit. This particular circuit is fanout-free in order to make the method easier to understand. Note that the method works just as well, in the general case, with reconvergent fanouts. The black vertices correspond to gates in the combinational circuit. The total number of structural paths from each vertex to any sink, as computed by Algorithm 6, is printed inside each vertex. The outgoing edges are ordered and labelled by an index.

The circuit in Figure 5.7 obviously contains nine structural paths. And since this circuit happens to be fan-out free, there is exactly one path from the source in the graph to each of the sinks. It is thus very easy to come up with the intuitively enumeration shown to the left in the Figure 5.7, next to the corresponding sink. Fortunately, this particular enumeration can be found more systematically as well.

Look at the highlighted path through the gates $G1$, $G2$ and $G3$. This path defines three subtrees, one subtree rooted at each of the three gates. A proper enumeration of the paths in the subcircuit defined by the subtree rooted at $G1$ is easy, the only path must be assigned the number 1. The subtree rooted at $G2$ combines three smaller threes into one larger tree. One way to enumerate the paths is to reserve the first path number to the subtree accessed by the edge 0. The second path number can be reserved to the subtree accessed by edge 1, and

the last path number can be reserved for the subtree accessed by edge 2. The highlighted path will thus get path number 2 in the graph rooted at $G2$.

The subtree rooted at $G3$ combines three smaller threes into one larger tree. One way to enumerate the paths is to reserve the first three path numbers to the subtree accessed by the edge 0. The next three path number can be reserved to the subtree accessed by edge 1, and the last three path number can be reserved for the subtree accessed by edge 2. The highlighted path will thus get path number 8 in the graph rooted at $G3$. Since $G3$ is the only source in the graph, the computation is finished and the highlighted path is assigned the path number 8. If the circuit contains more than one output, the corresponding DAG will contain an equal number of sources. In such cases it is necessary to add a help-vertex that combines the sources into one single source.

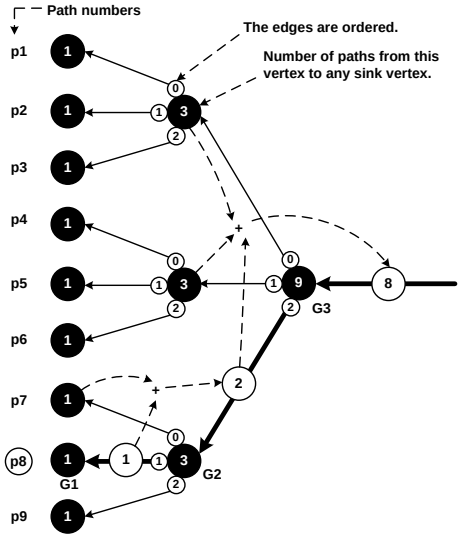


Figure 5.7: Path enumeration example.

Fault detection

A path-delay fault is detected, as described in Section 5.1.1, if the simulation value of all gates on the path is either $P0$ or $P1$. This is the case for the highlighted paths in the circuit in Figure 5.8. The detected path-delay faults are found by traversing the netlist breadth-first along the edges with signal values $P0$ and $P1$, while computing the path numbers as described in Section 5.4.2.

Since more than one path-delay fault may be detected, a list of faults is propagated along the highlighted paths in Figure 5.8. Each path-delay fault is identified by its structural path number together with the direction of the transition applied at the input of the path. This is solved in *PDFSim* by propagating two set of lists, one for each transition.

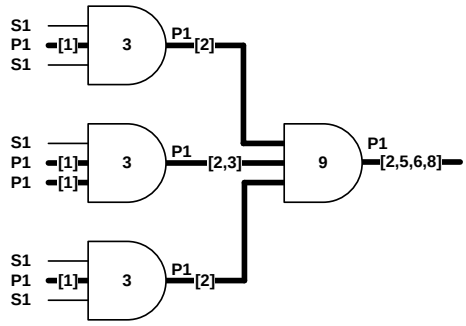


Figure 5.8: Fault detection example.

At this point two lists have arrived at the output of the netlist. One list contains the structural path number of detected path-delay faults with *rising* transition applied at the input of the path, and the other contains the structural path numbers for path-delay faults with a *falling* transition applied to the input of the path. (Only the list with rising path-delay faults is shown in Figure 5.8).

The procedure for collecting detected path-delay faults is equal for both robust and non-robust faults, with one important exception. In both cases paths with $P0/P1$ on all signals are traced, but in the case of non-robust faults not all such paths represent detected faults. When robust detection is considered, if all inputs of a gate have ether the $P0$ or the $P1$ value, each input represents one or more robustly sensitized subpath. Each input must thus be an on-input of the corresponding set of subpaths. When non-robust detection is considered on the other hand, there are one situation [SFF89] for each gate (AND/OR) that will block further non-robust sensitization for some subpaths. An AND gate with $P0$ on one input and $P1$ on the other will result in $P0$ on the output of the gate, and will thus only sensitize subpaths rooted at the input with the $S0$ value, the other input is an off-input for this path. A similar situation exist for OR gates.

Storing detected faults

Lists of detected faults are produced for each test vector applied to the CUT. When this is done the detected faults are stored in a datastructure together with

previously detected faults. Since the number of faults can be huge, it is important to use an efficient method to store the faults. Kapoor developed a method [Kap95] for storing detected path-delay faults based on intervals of consecutive path numbers. According to this method, the path numbers: 1, 2, 3, 4, 7, 8, 11, 12, 13, 14, 15, 20 would be represented by the intervals : [1,4], [7,8], [11,15], [20, 20]. A 2,3-three based datastructure is used for storing and manipulating the intervals.

The fault grading algorithm implemented in *PDFSim* is based on the fault grading method [Kap95] by Kapoor. But instead of using the 2,3-based datastructure that Kapoor used, skip-lists [Pug90] have been used instead. The skip-list is a probabilistic alternative to balanced trees. Skip-lists was chosen because the manipulation algorithms are easy to implement, and the *expected* running time is at least as fast as for the original 2,3-three based datastructure (the worst-case running time is worse because skip-lists can degenerate to ordinary lists).

When a path-delay fault is detected, a search in the skip-list is performed in order to find the proper position to store the fault. If the position is empty the new path-delay fault is stored, otherwise the fault is already detected. Two different situations can arise when a new path-delay fault is inserted into the skip list. If the path number is not adjacent to any other path number it is stored as a new interval. However, if the interval is adjacent with one or two other intervals then the path-delay fault is combined with the existing interval(s) into a larger interval.

Enhancements in *PDFSim* over the method by Kapoor

The fault grading algorithm used in *PDFSim* is based on the algorithm by [Kap95], but is enhanced on several points compared to the original algorithm.

When the list of detected faults are compiled in [Kap95], this is done by a breadth-first traversal of the netlist from the inputs to the outputs as shown in Figure 5.8. This is however not optimal since this some of the partial paths starting at the inputs of the netlist will not be sensitized all the way through the circuit under test. This will cause unfruitful propagation of some path-numbers through the netlist until the fault grading algorithm realizes that the path is blocked for further sensitization. An example of paths that is partially sensitized, but blocked before they reaches any outputs is shown in Figure 5.9. Fortunately, there is a method that can be used in order to avoid tracing partially sensitized paths, and that is to start the search from the outputs instead of from the inputs. Since a $P0/P1$ value at an output guarantees that there exist at least one sensitized path from the inputs to that output, there will be no unnecessary

propagation of path numbers. This method would for instance immediately conclude that no path-delay faults was detected since no outputs has a $P0/P1$ value.

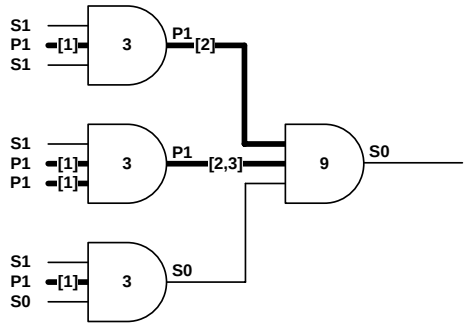


Figure 5.9: Partially sensitized path-delay fault.

Another enhancement over the method described in *PDFSim*, is the use of fan-out free cones (Algorithm 10). Cone-extraction reduces the number of vertices in the DAG representing the netlist while the number of the paths in the new DAG is the same as in the original DAG. Since the number of vertices is reduced, the running time of the fault grading algorithm is reduced as well.

5.4.3 State of the art non-enumerative fault grading algorithms

The fault grading algorithm implemented in *PDFSim* is fast and efficient for circuits with a small number of path-delay faults. However, when the number of path-delay faults are very high it is necessary with non-enumerative algorithms. They are usually more complex to implement and slower than simpler algorithms for circuits that contains few paths, but faster and much more efficient when the circuit under test contains a large number of paths.

The best fault grading algorithm today for circuits with a large number of path-delay faults has been developed by Padmanaban Michael and Tragoudas [PMT03]. The method uses a special type of binary decision diagrams called zero-suppressed binary decision diagrams [Min93] (ZBDD). ZBDDs is a canonical data structure that can be used to represent sets very efficiently. In this case sets of path-delay faults. This work has resulted in many good papers about fault simulation, fault grading and identification of detectable and undetectable path-delay faults [PMT03, PT04b, PT04a, PT05, KGT04, KG05].

Other good fault grading algorithms are the color counting based algorithm by Deodhar and Tragoudas [DT01] and the algorithm based on a datastructure

called the Path-Status Graph (PSG) [GBA96, PABS98] by Parodi, Agrawal, Bushnell, Gharaybeh and Wu.

5.4.4 Lower bound on the number of untestable faults and the computation of fault coverage

Fault coverage refers to the percentage of path-delay faults detected, under some sensitization criteria, during the test of an integrated circuit. Ideally the fault coverage should be computed relative to the total number of path-delay faults sensitizable under the given sensitization criteria (e.g. non-robust sensitization).

For the same reasons as counting the number of detected faults is intractable, the problem of determining the *exact* number of testable path-delay faults under a given sensitization criteria is also intractable. One possible solution would be to feed an ATPG with all possible path-delay fault and use the ATPG to determine if the path is testable or not. This procedure is only feasible for circuits with a few number of paths. The best solution to this problem to this date was presented by Padmanaban and Tragoudas in [PT05]. The method identifies a subset of untestable PDFs using static implication techniques. The remaining faults are *potentially testable* and targeted by a special ATPG that avoids path-enumeration. The testable paths are stored non-enumeratively in a ZBDD.

If speed is crucial, it is possible to find a lower bound [HPA97, LMB97, SRKP01] on the number of testable-path-delay faults instead. On obviously very pessimistic estimate would be twice the total number of structural paths in the circuit. Another method would be to identify a set of untestable path-faults and then regard the rest of the path-delay faults as potentially testable path-delay faults.

Fault coverage can also be computed relative to the number of faults in a fault-list. An algorithm for creation of fault lists containing the K-longest testable paths is the topic of Chapter 6.

5.5 Stimuli generators

Path delay simulation is initiated by applying a test vector to the inputs of the CUT. This task is carried out by the stimuli generator. The purpose of designing *PDFSim* was to evaluate different stimuli-generators with respect to their ability to detect path-delay faults. Several test generators have been tested, and it was important that the process of adding and modifying test generators was

straightforward, without sacrificing too much performance.

The way this is solved in *PDFSim* is by adding a python [Pyt06] script interface to the simulator, so that the test generators can be programmed in the python. Two aspects of this architectural decision makes it easy to add and modify test generators: 1) Since the generators are implemented in an interpreted language, no recompilation is needed each time a new generator is added. 2) Python offers a much cleaner syntax and a much more compact representation than would have been possible by using C++ (the simulator core is written in C++).

5.6 Other aspects

Other aspects of importance when designing a path-delay simulator are for instance the user interface and presentation of simulation results.

5.7 *PDFSim* software architecture

The core of *PDFSim* is shown in Figure 5.10. The path-delay simulator is built around the representation of the *netlist* which the other components interacts with during simulations. The *stimuli generator* applies test vectors to the inputs of the netlist, the *logic simulator* propagates the stimuli vector through the netlist, and the *fault grader* finally counts the number of detected faults. All of these tasks are controlled by the *controller*.

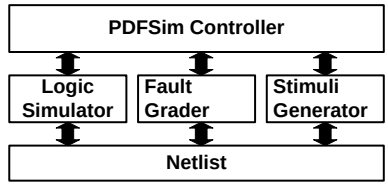


Figure 5.10: The PDFSim path-delay fault simulator.

One of the choices that must be made when creating a computer program is to choose a programming language. Different languages have different properties, and the choice of the wrong programming language can have severe impact on the end product. One way to categorize programming languages is to distinguish between interpreted languages and languages that can be compiled into

native machine code. Programs implemented in a interpreted language is usually much slower than programs implemented in programs compiled into native machine code. However, programs implemented in scripting languages tend to be more compact and much faster to create and modify than similar programs implemented in languages compiled down to machine code.

It was desired that the path-delay simulator and the ATPG presented in Chapter 6 should be both fast and at the same time flexible. This was solved by using the best of the two worlds. The simulator and ATPG cores are both written in C++ for maximum performance, but in order to make it easy to change generators and run repeated simulations with different parameters, the cores have been implemented as python extension modules. This allows the modules to be accessed from within a python program.

The building blocks of the python program that integrates the two cores (*PDFSim* and *PDFAtpg*) is illustrated in Figure 5.11. In addition to the two algorithm cores is a collection of *Test generator scripts*. New stimuli generators are easily added by just adding a new or modifying an existing script. No recompilation is needed after a new script is added. The program also contains a *Command-line interpreter*. The purpose of this module is to interpret the commands given to the *PDFTestTool* program by the user. One of the options that can be chosen during startup is the mode the program will be running in. Two modes are possible: *local* or *remote*. In the *local* mode the *PDFTestTool* program runs the job specified on the command line and dumps the result to file. In the *remote* mode on the other hand, the program connects to a remote server through the *Remote communicator*, and downloads an open job. The *PDFTestToolController* controls the overall behaviour of the program.

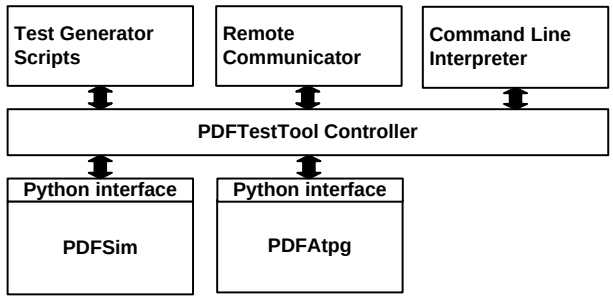


Figure 5.11: PDFSim and PDFAtpg are two modules in the scriptable PDFTestTool program.

There are two ways of improving the speed of a computer program. The

best way is to improve the algorithm or come up with an algorithm with better asymptotic running time. When time or creativity does not allow for further improvements in the algorithm, its time to add more processing power. This is exactly what have been done by the *PDFTestTool* program and the reason behind the mysterious *remote* mode. When *PDFTestTool* runs in *remote* mode (see Figure 5.12), it downloads an unfinished job from a *Job Server* that typically contains a long list of simulation jobs. These jobs have been uploaded to the job server from one or more *Command centers*. When a *PDFTestTool* instance is finished with a job it uploads the results to the *JobServer* which forwards the result to the *Command center* that requested the simulation job. Some companies and all universities have access to large rooms filled with computers doing nothing, such grid computing arrangements is thus a way to make use of otherwise wasted CPU time, and improve total simulation time.

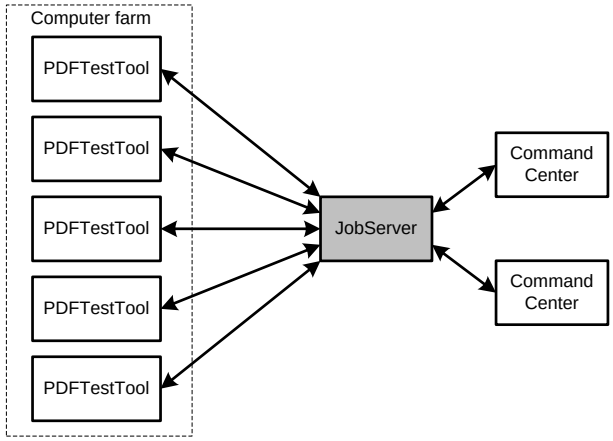


Figure 5.12: Grid computing.

Chapter 6

Automatic test pattern generation for the K-longest testable path-delay faults

It is infeasible to test all path-delay faults in circuits with an exponential number of paths. One common strategy is instead to test a subset of all possible path-delay faults. The fault list is often set equal to the K longest testable paths, or to all testable path-delay faults longer than a predefined limit.

This chapter describes an algorithm for extracting the K-longest testable path-delay faults in a circuit. The engine presented here has been integrated with the fault simulator presented in the previous chapter. In Chapter 7 test pattern generators for use in Built-In Self-Test arrangements are presented. These test generators will be evaluated against the fault lists extracted by the ATPG presented in this chapter.

The earliest attempts at creating an ATPG for path-delay faults that could extract the K-longest testable path-delay faults was very inefficient. ATPGs were often divided into two separate parts. A *structural path extractor* that sorted the structural paths in the circuit according to the length or delay of the paths. Paths where then passed to the *test generator*, one at a time, which decided whether or not a test could be found for that path. Usually a lot of paths are untestable, and the structural path extractor would thus pass on a lot of untestable paths to the test generator. Fortunately, by combining the *structural path extractor* and the *test generator*, it is possible to prune the search space significantly by sorting out untestable set of paths at an early stage. This approach was used by Qiu and Walker [WW03].

First in this chapter, terms that will be used in the presentation of the ATPG algorithm are introduced. Next the path-delay fault ATPG algorithm itself is described. It is based on lessons learned from [WW03]. The final step of the algorithm involves finding a test vector for the path-delay fault, and this is achieved with a version of the FAN algorithm [FS83] together with recursive learning [KP94].

6.1 Terminology

This section presents terms that will be used in the discussion in this chapter. A three value logic, B={0,1,X} is assumed.

6.1.1 Implications

One very important notion in the world of ATPG is the *implication*. Definition 11 defines *implication*. In this context implications are used for describing relations between signal values of different signals in a circuit. An implication, $(x = 1) \rightarrow (y = 0)$, between the two signals x and y means that if the signal x happens to have the value 1, then signal y must have the value 0.

Definition 11 (Implication) *An implication, $p \rightarrow q$, is a logical relation that indicates that if one condition, p, is true, then another condition, q, must also be true. The implication, $p \rightarrow q$, only fails to hold (i.e. is false) if p is true and q is false.*

Implications are used in order to find the necessary value assignments following from an initial value assignment in the circuit. For instance, when finding a test for a net stuck-at 0, it is necessary to assign a 1 to the net in order to activate the fault. An example is shown in Figure 6.1. This initial assignment might force other signals to specific values as indicated in the figure.

When testing for stuck-at faults, it is sufficient with one vector per test. However, tests for path-delay faults require two-pattern tests. According to Figure 4.1 in Chapter 4, a non-robust test for a path-delay fault, requires that all off-inputs have non-controlling values under v_2. An example is shown in Figure 6.2. Here the initial value assignments constitute the rising transition that is propagated on the target path, and the non-controlling values assigned to the off-inputs. These initial value assignments might force other signals to specific values as shown in the figure.

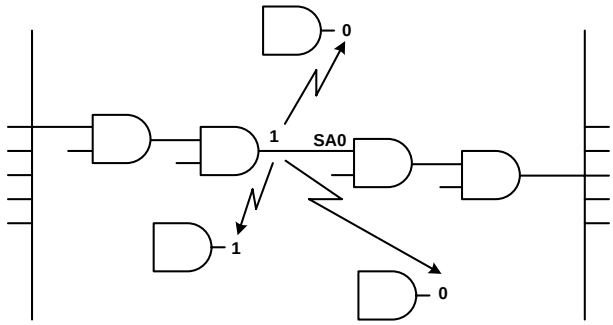


Figure 6.1: Test for a stuck-at fault.

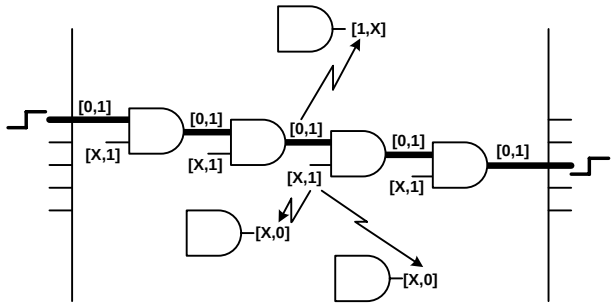


Figure 6.2: Test for a path-delay fault.

Usually one partitions implications into two groups, *direct* implications and *indirect* implications. Some examples of direct implications are shown in Figure 6.3. The first two implications illustrated are examples of direct backward implications. The two last implications are direct forward implications.

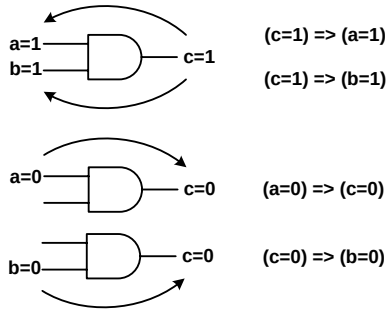


Figure 6.3: Example of direct implications.

Modus tollens is Latin for *mode that denies*, and is the formal name of proof by contra positive. The *denied* object in this context is the consequence of an implication. If for instance $(p = 0) \rightarrow (q = 1)$, then $(q = 0) \rightarrow (p = 1)$ by *modus tollens*. Often the implications derived by employing *modus tollens* lead to indirect implications. One example is shown in Figure 6.4. A method for systematically finding all indirect implications is presented in section 6.4.

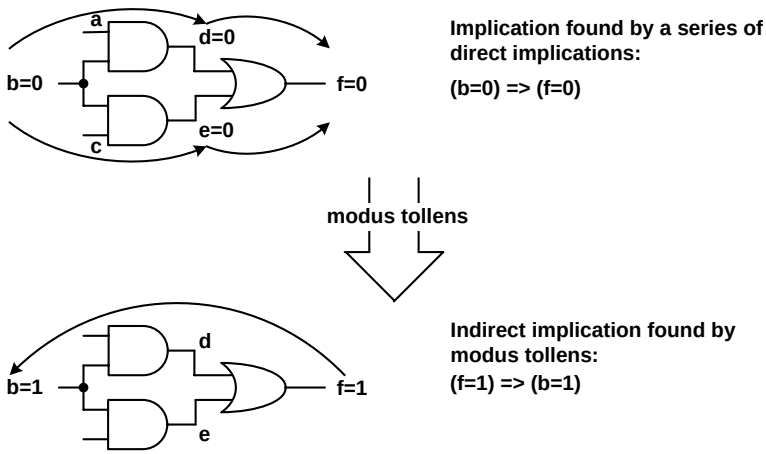


Figure 6.4: An indirect implication.

6.1.2 Specified and unspecified signals

A signal can either be *specified* or *unspecified*. Assuming that the three value alphabet B={0,1,X} is used, a signal will be *specified* if it is assigned either 0 or 1, and *unassigned* if it is assigned the value X.

6.1.3 Consistent and inconsistent signal assignments

The signal assignments in a circuit can either be *consistent* or *inconsistent*. *Consistent* signal assignments does not produce any signal assignment conflicts. A given set of signal assignments are *consistent* if there are no *conflicts*. An *inconsistent* set of signal assignments produces conflicts. In order to create a non-robust test for a path delay fault, the off-inputs must be set to non-controlling values under v_2. Thus, in order to create a non-robust test for the path in Figure 6.5, the signals a and b must both be 1. In this case the two signals a and

b are connected by an inverter, and the current set of signal assignments is thus *inconsistent*.

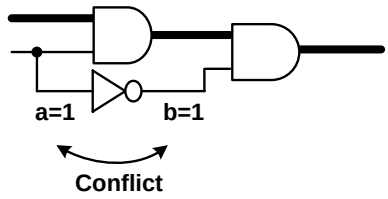


Figure 6.5: Conflicting signal assignments.

6.1.4 Justified, Unjustified and Justification

Justified, unjustified and justification are three frequently used terms in the ATPG field of research. *Justified* and *unjustified* are adjectives used in order to describe the state of the signal assignments of inputs and outputs of a gate.

Assuming the three value logic B={0,1,X}, in order for a gate to qualify as either justified or unjustified, at least one input or output of the gate must be specified. Thus, gates that are assigned the value X on all terminals are neither justified nor unjustified.

A gate is *justified* if it is *impossible* to assign values to remaining unspecified inputs and outputs of the gate so that a conflict is produced at that gate. A gate is *unjustified* if it is *possible* to find values to unspecified inputs and outputs of the gate so that a conflict is produced at that gate. Some examples of justified and unjustified gates are shown in Figure 6.6.

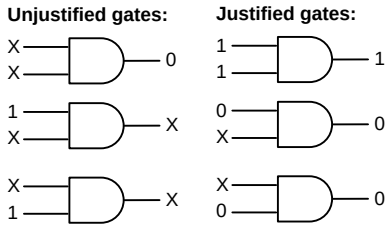


Figure 6.6: Examples of justified and unjustified gates.

A *justification* is a set of additional signal assignments that makes *unjustified* gates *justified*. An example is shown in Figure 6.7.

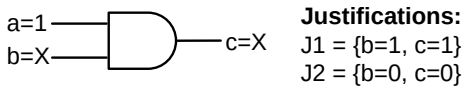


Figure 6.7: Justifications of an unjustified gate.

6.1.5 Complete path and Partial path

A *complete* path in a combinational circuit is a path that starts at an input, runs through the circuit, and ends at an output. A *partial* path on the other hand starts at an input, but is terminated at an internal gate. The partial path highlighted in Figure 6.8 is an example. The ATPG algorithm to be described in section 6.2 maintains a list of such partial paths, and repeatedly extends the most promising, in terms of potential length, until the longest testable complete path is found.

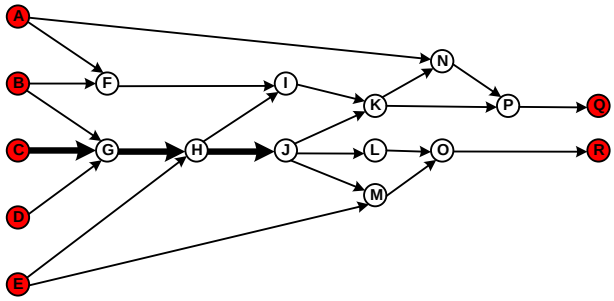


Figure 6.8: A DAG representing a circuit with one partial path highlighted.

6.1.6 VertexDelay, PERTDelay and Esperance.

The ATPG is able to extract the *K longest*, in terms of delay, non-robust testable paths in combinational circuits. There are several ways of modelling the delay through gates and nets. For the ATPG a simple *transport delay* model is assumed. Associated with each gate is a *VertexDelay*. All transitions arriving at the input of each gate is delayed by time equal to the gate's *VertexDelay* before it is propagated to the output of the gate. In the experiments all gates in all circuits have been assigned a *VertexDelay* of 1, but individual *VertexDelay* can be set for each gate if desirable.

PERT delay is a term that stems from the theory of PERT (Program Evaluation and Review Technique) digraphs [Eve79]. In a PERT digraph every edge

represents a process, and for every vertex the processes represented by the outgoing edges of the vertex can be started when all processes represented by the incoming edges of the vertex are finished. One of the interesting problems such PERT digraphs can solve is to find the shortest running time of the program it models. This is determined by the delay of the critical path in the PERT digraph. The delay of the critical path can be found by processing the vertices in reverse topological order, starting at the termination vertex. For each vertex the maximum (PERT) delay from each vertex to the termination vertex is computed, until the start vertex is reached. The *Maximum PERT delay* for the start vertex would then be the delay of the critical path. The same method can also be used in order to compute delay of the shortest path in the program. A gate netlist can also be modelled as a PERT digraph, where the stable value of the output of each gate can be determined first when the inputs of the gates have settled at stable values.

Figure 6.9 shows the partial path from Figure 6.8. The delay of the partial path, see Equation refPartialPathEq, is simply the sum of the *VertexDelay* for each vertex in the partial path. The maximum PERT delay for the last vertex, J, in the partial path is computed according to Equation 6.2.

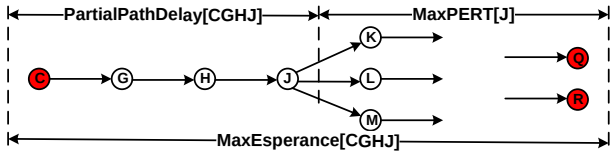


Figure 6.9: Computation of maximum esperance for a partial path.

$$PartialPathDelay[CGHJ] = \begin{array}{l} VertexDelay[C] + VertexDelay[G] + \\ VertexDelay[H] + VertexDelay[J] \end{array} \tag{6.1}$$

$$MaxPERT[J] = max\left\{ \begin{array}{l} VertexDelay[K] + MaxPERT[K] \\ VertexDelay[L] + MaxPERT[L] \\ VertexDelay[M] + MaxPERT[M] \end{array} \right\} \tag{6.2}$$

Associated with each partial path is also a value called *esperance* [BVMCDM90]. The maximum esperance, see Equation 6.3, is the sum of the length of the partial path and the maximum PERT delay from the last vertex of the partial path. The maximum esperance thus represents an *upper bound* of the delay of the partial path when it grows into a complete path.

$$MaxEsperance[CGHJ] = PartialPathDelay[CGHJ] + MaxPERT[J] \quad (6.3)$$

6.2 ATPG Algorithm

The ATPG algorithm presented in this section is based on the algorithm by Qiu and Walker [WW03]. A flow chart [WW03] describing the algorithm is shown in Figure 6.10. The algorithm is capable of finding both the K-longest paths through a particular gate in the circuit as well as the overall K-longest paths in the circuit. In order to find the K-longest paths through a particular gate in the circuit, it is necessary to go through a preprocessing step as shown at the top of Figure 6.10. During the preprocessing step a simple structural reachability analysis is carried out in order to constrain the search to only those paths that passes through that particular gate.

The ATPG algorithm is based on repeatedly extending the most promising partial path, i.e. the partial path with the largest esperance, with one gate at a time until a complete path is found. The partial paths are stored in a *path store*, and sorted according to their maximum esperance. Initially the path store is populated with two partial paths for each input, one partial path for a rising transition applied to the input and one for a falling transition applied to the input.

The algorithm in Figure 6.10 continues with popping the path with the largest maximum esperance and *extends* the partial path with one more gate. This particular process is illustrated in Figure 6.11. The example used is based on the same partial path as shown in Figure 6.8 and Figure 6.9. Figure 6.11 shows a part of the path store at some point. Our partial path $C - G - H - J$ is stored in slot 1. Since a unit delay is used, the delay of the partial path is 4, and the maximum esperance is 8. This means that the longest structural path that can be constructed starting with the vertices $C - G - H - J$ will have a length of 8. The *PathStore* also contains a list of possible fan-outs from the last gate of the partial path. This list may be shorter than the number of fan-outs from that gate in the actual netlist due to search space pruning techniques such as the reachability analysis done initially. Together with each possible fan-out is the *FanOutDelay*. The *FanOutDelay* is equal to the sum of the gates *MaxPERT* and *VertexDelay*. The fan-out with the highest *FanOutDelay* thus represents the gate the partial path should be extended with in order to achieve the highest possible esperance for the new partial path. In the example in Figure 6.11 the

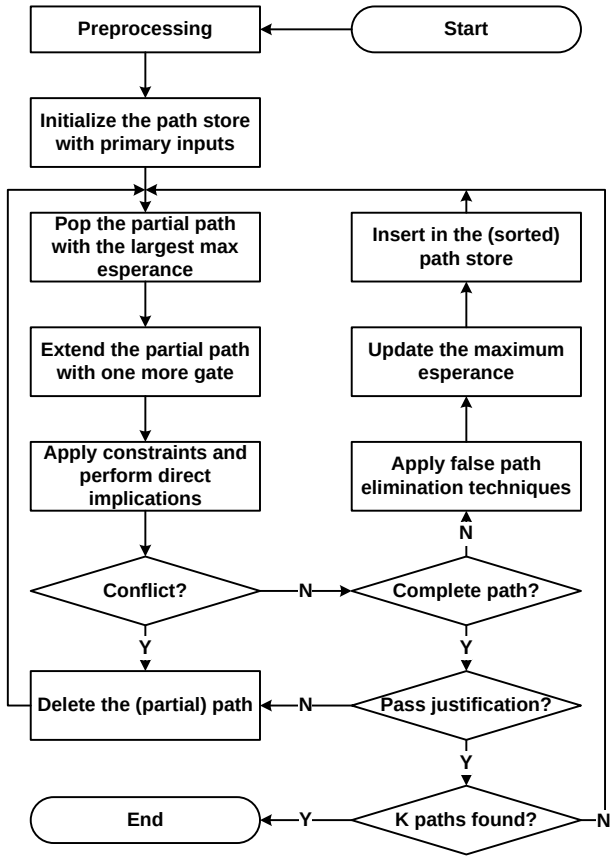


Figure 6.10: Path generation algorithm.

partial path is extended with gate K, since that fan-out has the largest *FanOut-Delay*. In order for the algorithm to be complete, it is necessary to also store the two other optional ways to extend the partial path. This is achieved by copying the original partial path to another location in the *PathStore* and removing the most promising fan-out option.

The next process carried out by the algorithm in Figure 6.10 is to check whether or not it is possible to successfully apply signal value constraints to the newly extended partial path without any conflicts. These constraints are non-controlling values to all off-inputs under v_2 as well as the correct transition to all on-inputs. If a conflict occurs the newly generated partial path is deleted, if not the partial path is checked in order to find out if it is a complete path or not.

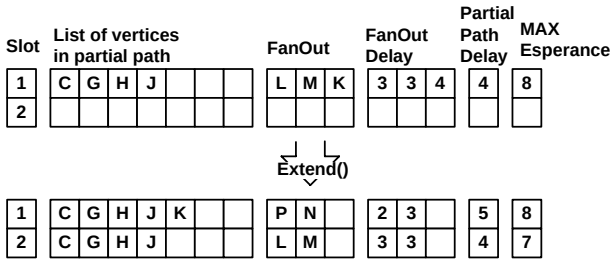


Figure 6.11: Extending the most promising partial path in the partial path store.

If the partial path is complete, some false path elimination techniques is applied. These search space pruning techniques are applied in order to obtain a better estimate of the maximum esperance. When this is done the maximum esperance is updated for the extended partial path and inserted into the path store. The processing of the partial path is now finished and a new partial path can be investigated.

If the partial path turns out to be a complete path it is a very good chance that the path is testable since it no conflicts have been detected through direct implications. But in order to ascertains that the path is testable and hopefully find a test-vector for the path, it is necessary to pass a full justification process. This is done by a version of the FAN algorithm described in Section 6.3.

6.2.1 Forward trimming

After the algorithm is done checking if the partial path is a complete path, some false path elimination techniques are employed. One of these techniques is called *forward trimming*. The purpose of *forward trimming* is to use the information gained from applying signal propagation constraints in order to compute a tighter upper bound of the delay of the partial path. In other words find a more accurate value for the path's esperance. This can in many cases prevent the ATPG from pursuing partial paths in directions that must produce a conflicts at a later stage. In order to explain the problem the example in Figure 6.12 is used.

The figure shows the last gate in a partial path (highlighted). No signal conflicts is present and the algorithm is now ready to extend the partial path. Two options are possible: the upper or the lower fan-out. According to the algorithm, it will choose to extend the partial path to the fan-out gate with the largest maximum PERT delay. Of the two possibilities, the lower fan-out ob-

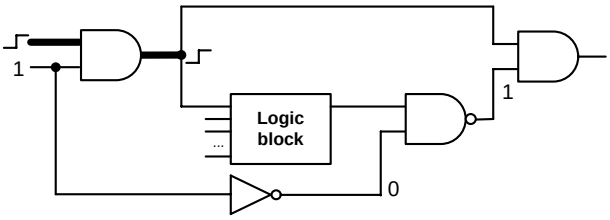


Figure 6.12: Forward trimming.

viously has the largest maximum PERT delay if only structural information is considered. However, extending the path towards the bottom fan-out gate is not an good idea, since the current set of signal value assignments will block the propagation of all transitions at the NAND gate. Thus, the algorithm should choose to extend the path to the upper fan-out. The lower fan-out should be dropped for this partial path.

Forward trimming is a method that recomputes the maximum PERT delay for each fan-out gate of the current partial path. The method is the same as the one explained in Equation 6.2, but the maximum PERT is set to 0 for gates that blocks transition propagation. Signal propagation is blocked by gates that have the same *specified* signal value, i.e. 0 or 1, for both vectors v_1 and v_2. The increased cost of recomputing the maximum PERT for each fan-out of each partial path, is compensated by a the reduced probability of extending partial paths in unfruitful directions.

6.3 FAN algorithm

In the final stage of the path-delay ATPG algorithm, the program is presented with a set of path-delay faults that most likely are testable. However, some untestable path-delay faults might have slipped through the pruning process. In addition a two pattern test must be generated for all path-delay faults. This section presents an adapted version of the FAN algorithm that was used in order to perform justification and find a suitable test vector. The FAN algorithm is given in pseudo-code in Algorithm 2 and Algorithm 3.

Algorithm 2 (FAN algorithm)

```
FANAlg(){
  /*OUTPUT: Status information indicating wherer          */
  /*              the fault is testable, untestable or aborted  */
  DTree;                  // The decision three
  CObj, SObj, HObj;    // Set off current, stem and head objectives
  FObj;                   // Variable holding the final objective
  NoConflict;             // Boolean that is true when no conflicts occurs.

  -Clear all objective containers and the decision three;
  -Mark gates in the cone defined by the current path;

  // Continue the search until a test for the fault is found,
  // the fault is found to be untestable or the search is aborted.
  while(1){
    // Determine the next objective by multiple backtrace if there are any
    // Justify the free lines when all other lines have been justified.
    -Reinitialize multiple backtrace
    if(!MultipleBacktrace()){
      - Justify free lines;
      return TEST_FOUND;
    }
    // Add the a new node to the decision three
    // Assign the value specified in the final objective to the specified
    // gate and perform direct implications. Check for conflicts.
    GName, GValue =  FObj.GateName, FObj.GateValue
    DTree->AddNewNode(GName, GValue)
    NoConflict  = DoDirectImplications(S, GName, GValue)

    // Backtrack until a assignment that doesnt cause conflicts is found
    while(!NoConflict) {
      // Abort if the maximum backtrack limit has been reached
      if(Dtree->BacktrackCnt > MAX_BACKTRACK){
        return TEST_ABORTED;
      // The path is untestable if all decisions have been tried
      } else if(!DTree->UntriedLeft()){
        return UNTESTABLE;
      // Find an untried assignment in the decision three
      } else{
        GName, GValue = DTree->FindUntriedAssignment();
        -Undo signal assignments that stems from old decisions.
        NoConflict  = DoDirectImplications(S, GName, GValue)
} } } }
```

Algorithm 3 (Multiple backtrace)

```
MultipleBacktrace(){
 // OUTPUT: Returns true if a objective is found, false otherwise
 // Gobal variables:
 CObj, SObj, HObj: // Set off current, stem and head objectives
 FObj              :  // Final objective
 n0, n1; // Array of backtrace votes that counts the number of
         // requests of to set signals to 0 (n0) and 1 (n1)

 while(!CObj->IsEmpty()){
  - Return the next objective (G, V) = (Gate name, Value)
    breadth first from CObj.

  // Compute n0 and n1 for each driver of G
  if (G needs controlling value on one input to be justified){
   - Increment n0(n1)[easiest to control input] with n0(n1)[G] if
     the gate is an AND (OR) gate.
  }else{
   foreach(driver D of G){
   - Increment n1(n0)[D] with n1(n0)[G] if
     the gate is an AND (OR) gate.
   }
  }
  - Add drivers of gate G to their appropriate containers. Gates
    that are head objectives are added to HObj, gates that are stems
    are added to SObj, and the rest are added to CObj.
 }

 while(!SObj->IsEmpty()){
  - Return the next objective (G, n0[G], n1[G]) from SObj with
    highest logical level.

  if(n0[G] > n1[G]){ V = 0; } else{ V = 1; }

  if( (n0[G]!=0) && (n1[G] != 0) ){  // If both votes are nonzero there is
   FObj = (G, V);                    // a chanse for conflict, and one
   return true;                      // should try to set the gate to a value.
  } else{
   - Add G to CObj                   // No conflict occurred and the
   return MultipleBacktrace();       // backtrace can continue
  }
 }

 while(!HObj->IsEmpty()){
  - Pick the head objective G with the highes value of n0[G]+n1[G]
    and set the objective value V to the value that have highest
    number of votes.
  FObj = (G, V) ;
  return true;
 }
}
```

6.4 Recursive learning

Inside modern stuck-at fault ATPGs is usually an *implication engine*. In our case we employ recursive learning, developed by Kunz and Pradhan [KP94], since it is a good compromise between ease of implementation and efficiency. Other examples of implication engines are found in [CAR93, TGA00, GF02].

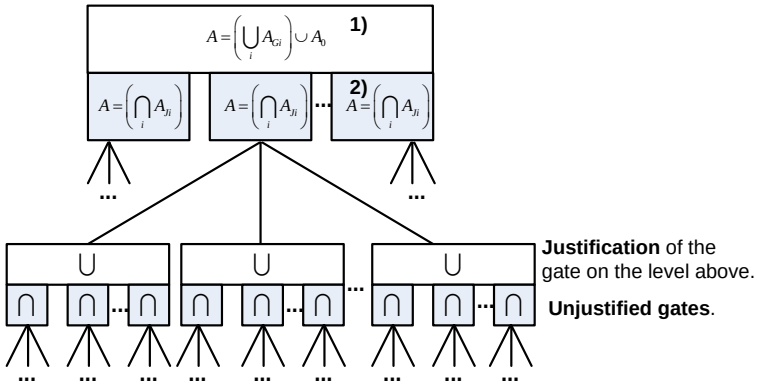
Recursive learning was added to the ATPG in order to prune the search space better before candidate paths were sent to the FAN algorithm. Recursive learning can also be used within the FAN algorithm in order to reduce the number of backtracks.

Recursive learning is a method for extracting all logical dependencies between signals in a circuit and to perform *precise* implications for a given set of value assignments [KP94]. Pseudocode for recursive learning is presented in Algorithm 4 [KP94]. Before the program is started a set of initial value assignments has been applied to the circuit. These initial signal assignments could for instance be assignments to off-inputs of a path in the circuit. Assume the set of signal assignments are stored in a an instance A of type $AssignmentList$. All implications, direct and indirect, following from this initial set of signal assignments will be extracted and added to A by calling the function *MakeAllImplications(A)*.

Algorithm 4 (Finding all implications using recursive learning.)

```
MakeAllImplications(A, r){
  // INPUT : A: An AssignmentList
  // Local variables:
  J: // An array of Justifications. J[i] refers to justification number i.
  T: // An array of AssignmentLists for temporary storage of assignments.
       T[i] refers to assignment list associated by Justification i.
  U: // A one-dimentional array of unjustified gates.

  Make all direct implications, exit routine if conflict detected;
  Store unjustified gates in U;

  if (r > RMAX){
    foreach(unjustified gate g in U){
      Fill J with all justifications of gate g;

      for(i 0 to length(J)-1){
        Make the assignments in J[i] and store them in T[i];
        MakeAllImplications(T[i], r+1);
      }

      if(T[i].IsConflict() is false for all i in range 0 to length(J)-1){
        A.SetConflict(true);
      } else {
        Extract all signal assignments common to all
        consistent justifications and store them in A;
      }
    }
  }
}
```

The function will thus start with making all *direct* implications. If con-



1) Learned set of signal assignments. Computed as the **union** of the set of signal assignments learned through the justification of each initially unjustified gate.

2) Learned set of signal assignments. Computed as the **intersection** between the signal assignments from all **consistent** justifications.

Figure 6.13: Recursion tree.

flicts are detected the function returns to the calling function and indicates that a conflict was detected by setting a flag in A. After all direct implications have been discovered the *unjustified* gates are extracted and stored in U. Next the algorithm will for each unjustified gate, g, perform a series of tasks. First all justifications for the current gate is found and stored in a list J. The justifications are now tried one at a time, with the purpose of extracting all implications resulting from each justification. This is done by the recursive call *MakeAllImplications(T[i])*. When all justifications are tried out the set of signal assignments are compared. If one or more signal assignments are common to all *consistent* justifications, these assignments must be *necessary* assignments at the current recursion level. Such assignments are added to the list A. Further details is found in [KP94].

An illustration of how the signal assignments are learned is presented in the recursion tree in Figure 6.13. Starting at the root of the tree, the set of all signal assignments A, is the union of signal assignments learned through the justification of the unjustified gates and the initial set, A_0, of signal assignments. The learned signal assignments common to all justifications of each unjustified gate is found by intersecting the sets of signal assignments from each justification. This process is repeated at each recursion level as indicated in Figure 6.13.

An example [KP94] of recursive learning is shown in Figure 6.14. The initial set of signal assignments in this case is $\{p = 1\}$, and the example illustrates

how all other signal assignments following from this initial signal assignment are learned. Initially only one gate is unjustified, namely gate G6. Two possible justifications exist for this gate $\{q = 0, r = 0\}$ and $\{q = 1, r = 1\}$. First the justification $\{q = 0, r = 0\}$ is investigated. After applying the signal assignments in the justification and performing direct implications, four gates turn out to be unjustified: G1, G2, G3 and G4. There are two possible justifications of G1, but both turns out to produce inconsistent signal assignments. It is thus impossible to justify G1. Since all gates must be possible to justify, the justification $\{q = 0, r = 0\}$ of G6 turned out to be a dead end.

The other possible justification of G6 is $\{q = 1, r = 1\}$. After applying the signal assignments in the justification and performing direct implications, one gate turns out to be unjustified: G5. There are four possible justifications of G5, and all produce consistent signal assignments. However, when the signal assignments are compared, it turns out that the four justifications have no signal assignments in common other than the signal assignments present before the justification of G5. The justification of G5 thus does not provide any new information. Nevertheless, the justification, $\{q = 1, r = 1\}$, of G6 turned out to be consistent. Since the set of signal assignments common to all consistent justifications of G6 is $\{q = 1, r = 1\}$, this information is learned and added to the set of signal assignments. Since G6 was the only initially unjustified gate, the program finishes. Thus, through recursive learning, it was found that $p = 1$ implies $q = 1$ and $r = 1$.

6.5 ATPG software architecture

Figure 6.15 shows an overview of the architecture of the ATPG. The ATPG is built around the same netlist representation as *PDFSim*. The other components of *PDFAtpg* interacts with this netlist when the *PDFAtpg* is running. The ATPG contains three other main modules: The *Implicator*, the *Path Extender*, and the *PDFAtpg Control*.

The *Implicator* is a large module capable of finding all implications (direct and indirect) from a given set of initial assignments. In order to achieve this, the *Implicator* is equipped with an *Implication Queue*, an *Unjustified Checker*, modules for *StaticLearning* and *Recursive Learning*, and a *Value Vector*.

The current set of value assignments (for two pattern tests) is stored in an array in the *Value Vector*. The *Value Vector* contains several such arrays, and can thus easily both switch between, and undo value assignments. This is frequently utilized by the *Recursive Learning* module, which is capable of finding

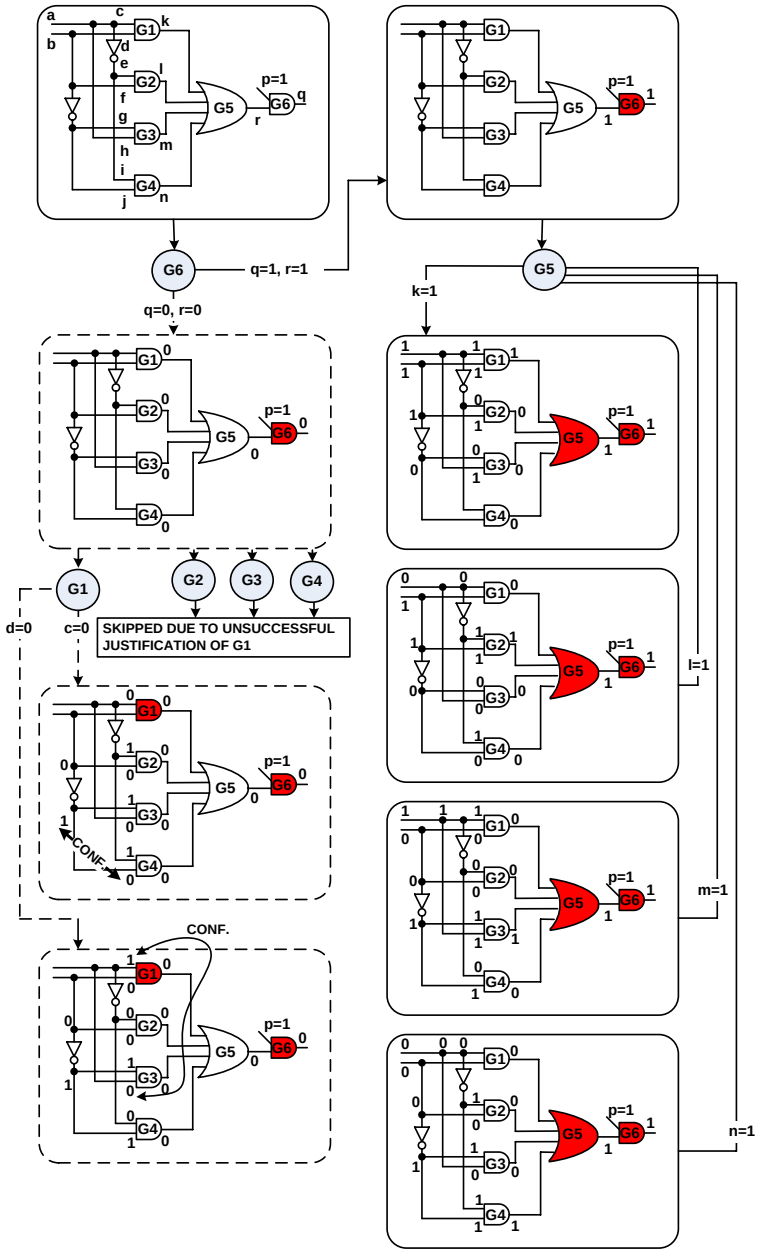


Figure 6.14: Example of recursive learning.

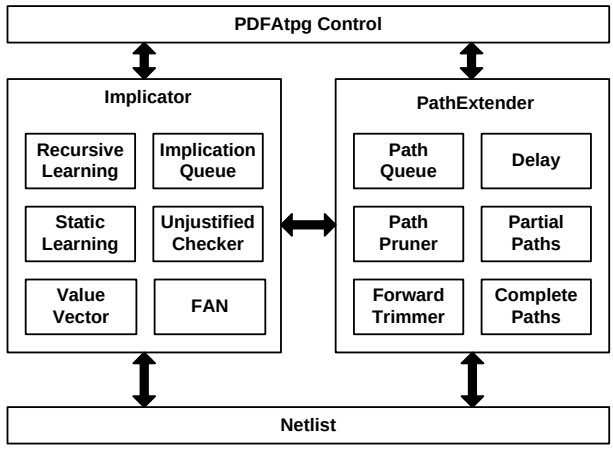


Figure 6.15: Simplified overview of PDFAtpg.

all indirect implications from a given set of initial value assignments. Some indirect implications is also found and stored by the *Static Learning* module. These are extracted once and stored in an implication graph for repeated use later. The *Implicator* also contains an *Implication Queue*, which is the ATPG equivalent of the event queue in a event driven simulator such as *PDFSim*. It is used in order to propagate value assignments back and fourth through the netlist. The purpose of the *Unjustifier Checker* is, as the name suggests, to check if a particular gate in the netlist is justified or not. The last module in the *Implicator* is the *FAN* module which is an implementation of the FAN algorithm. This module is used in order to find a set of assignments to the inputs of the netlist that justifies a given set of value assignments to gates.

The other main module in *PDFAtpg*, is the *Path Extender*. The purpose of this module is to provide efficient pruning of the search space in the search for the K-longest testable paths. In order to achieve this the *Path Extender* is equipped with a *Path Queue*, a *Path Pruner*, a *Forward Trimmer*, a module for storing different *Delay* attributes, a container for *Partial Paths* and a container for *Complete Paths*.

The *Path Queue* is a max heap that keeps the partial paths sorted according to how promising they are with respect to their max esperance. The *Path Pruner* and the *Forward Trimmer* contains different algorithms for pruning the search space. The pruning process relies on different delay parameters, such as esperance. These delay parameters are stored in the *Delay* module. When the search for the longest paths is started, the *Path Store* is populated with the inputs of the

netlist. These short paths are then extended with one gate at a time. If a partial path is extended all the way to an output, it is transferred to the *Complete Store*, but only if it passes final justification by the *FAN* module.

Chapter 7

Stimuli generators

A necessary component in order to implement self-test is a stimuli generator. This chapter presents some stimuli generators for use in a built-in self-test environment for path-delay faults.

7.1 Introduction to stimuli generators

The test generators that will be presented in this chapter exploit pseudo-random stimuli generation. Two different sources of pseudo-random stimuli will be presented. One is an accumulator based pseudo-random generator, and the other is based on a more complex algorithm called Mersenne twister. The advantage of the Mersenne twister is its statistical properties in particular, with respect to correlation between samples. However, this comes at a cost of increased computations compared to the simpler accumulator based scheme.

These two pseudo-random generators will be used as the underlying pseudo-random generator in five different weighting schemes. Each weighting scheme employ a two-phase process. In the first phase weigths are generated. One strategy is to apply uniformly distributed stimuli to the circuit under test and relate the input signal values (0 or 1) to fault detection in various ways. This information is then used in order to generate weights. In the second phase the weights are used together with the two different pseudo-random generators.

The generators will be described in detail later in this chapter. We begin with prerequisite information about hardware and software based BIST, and single-input change patterns which are used in all presented stimuli generators.

7.2 Built-in self-test of system on chip

Successful test of a digital integrated circuit should ensure the quality of the circuit as well as being cost effective. The classical way of testing digital integrated circuits is by means of an external tester (also often referred to as Automated Test Equipment (ATE). Moore's law affects all parts of the semiconductor industry, including the *requirements* for the testers. Among these requirements are support for larger test data volumes, higher pin counts and higher speeds of the testers. Unfortunately, the actual increase in performance of cost efficient testers don't match the increased performance of state-of-the-art systems-on-chip. According to the International Technology Roadmap for Semiconductors [ITR05] one of the problems is the increasing speed gap between state of the art VLSI circuits and cost efficient testers. This speed gap makes it increasingly difficult to apply test vectors at-speed to the circuit under test. Together with the increase in test data volume the increased speed gap gives an increase in test time and thus the cost of test.

One increasingly important tool in order to reduce the cost of test and achieve at-speed testing of large integrated circuits is built-in self-test (BIST) [RT98, Str02]. There are many ways of implementing built-in self-test, but it usually encompasses the following elements:

- Method for generating test vectors (Test vector generator/ Stimuli generator).
- Circuit to be tested (Circuit under test, CUT).
- Method for transporting test vectors to the CUT and the captured response from the CUT.
- Method for compacting the test response (Test response compactor/signature analyzer).
- Method for controlling the test (BIST controller).

7.2.1 Hardware implementation of BIST

One traditional implementation [BH82] of BIST is the STUMPS (self-testing using MISR and parallel shift register sequence generator) architecture. In this architecture a linear feedback shift register (LFSR) is used as the test vector generator. The test vectors are fed to the CUT through multiple scan chains, and the response is compacted by a multiple-input signature register (MISR)

[Dav84]. Each scan chain is fed from one flip-flop in the LFSR. Additional hardware have to be added to the mission logic in a circuit in order to implement STUMPS. The next section presents a software based approach which does not need additional test hardware. Other examples of hardware implementations of BIST can be found in textbooks such as [BA02, Str02].

7.2.2 Software implementation of BIST

In 1978 Thatte and Abraham presented a methodology, which was the first attempt of software-based self-test (SBST), for *functional level* testing of microprocessors [TA78, BA84]. Since then SBST has been refined in order to support structural fault models and utilize on-chip memories, and is now widely in use for testing processors and large systems-on-chip designs. The usual test flow within systems with software-based self-test today starts by uploading a test program to the on-chip instruction cache. Additional data, such as seeds and deterministic test vectors, may be uploaded to the data cache. When all necessary data is uploaded the test program is started. The purpose of the test program is to generate stimuli, apply the stimuli to the circuit under test, and compact the test response.

Rajski and Tyszer [RT93a, RT93c, RT93b, GRT96, RRT97, RT98] found an approach to software-based self-test that focused on the possibility to reuse the mission logic for test purposes which they called *arithmetic built-in self-test* (ABIST). ABIST is applicable to systems with on-chip microprocessor and cache. Prior to a test a test program is uploaded to the on-chip program cache. When the test is started the instructions are fetched from the cache. Test programs may use the arithmetic logic unit (ALU) [DW98] in the microprocessor to act both as a test vector generator, and a signature analyzer.

The stimuli generators presented are intended for use in software based BIST. Examples of assembly test programs based on the generators presented here are presented in Section 7.8.

7.3 SIC: Single input change patterns

It is possible to toggle a single bit (Single Input Change-SIC) or multiple bits (Multiple Input Change-MIC) when a two-pattern test vector is applied to the circuit under test. Random SIC test sequences were found in [VDG$^+$00, VGL$^+$01] to be *more effective* than random MIC test sequences when both robust and non-robust faults were considered. The focus will therefore be on SIC generators in

this thesis.

The generators produce SIC vectors by first establishing a *basis pattern*. Each bit is then toggled twice in order to generate both rising and falling transitions. This is illustrated in Figure 7.1, where the vectors are encoded using Smith's alphabet [Smi85].

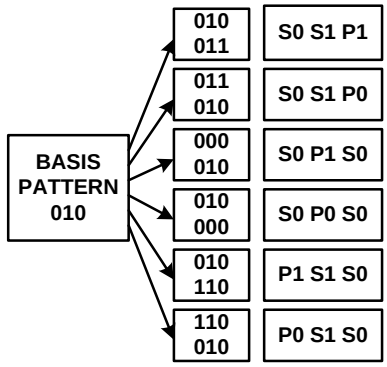


Figure 7.1: An n-bit basis pattern is used in 2n SIC test vectors.

The generators presented in this chapter do all generate SIC patterns, but they differ in the way the basis patterns are generated. Since one n-bit (n is the number of primary inputs to the CUT) basis pattern is re-used in $2n$ test vectors it is important to create high quality basis patterns, i.e. patterns that detects several new faults.

7.4 Pseudo exhaustive patterns

One way to achieve 100% fault coverage is by employing an exhaustive test. Unfortunately, applying 2^n test vectors, in the case of stuck-at faults, is only feasible for circuits with few inputs n. However, sometimes, it is possible to partition the circuit so that each sub circuit can be tested exhaustively. Four common ways of segmenting the CUT is described in [BA02, RT98]:

- *Verification testing* or *cone segmentation* [Mcc84]: The CUT is partitioned into fanin cones. The fanin cones are generated by backtracing from each primary output through the circuit to the inputs that influences the output. A cone thus consist of an output and all gates and inputs driving it. This is illustrated in Figure 7.2. If the number of inputs influencing each output is small enough, the cones may be tested exhaustively.

- *Hardware partitioning* or *physical segmentation* [MB81]: Extra logic is added to the circuit in order to partition it into smaller sub circuits that is directly controllable and observable. Each sub circuit is tested exhaustively.

- *Sensitized path segmentation* [CKMZ83, Che88, Mcc84, UJ86]: The circuit is divided into partitions. For each partition, sensitizing paths are set up from the primary inputs to the input of the partition and from the output of the partition to the primary outputs. Each partition is tested separately, while the remaining partitions are stimulated so that necessary propagating conditions are met.

- *Partial hardware partitioning* [UJM88]: Combines *sensitized path segmentation* to control inputs to the partition under test and *hardware partitioning* for observing signals in the partition under test.

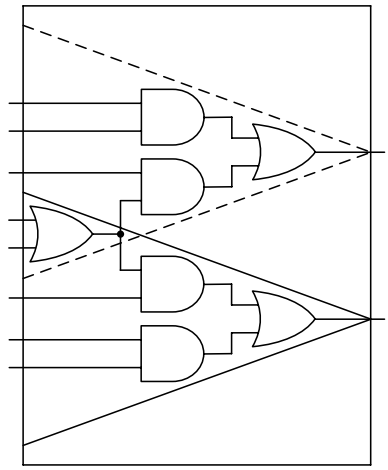


Figure 7.2: Cone segmentation

Pseudo exhaustive test generators may be realized in many ways. Any generator that is guaranteed to generate all 2^k test vectors for a sub circuit with k inputs will suffice. A maximal length LFSR or an binary counter are examples. Two other interesting pseudo-exhaustive generators based on accumulators was presented Rajski and Tyszer in [RT98] as a part of the ABIST methodology. These two generators will be presented next.

7.4.1 ACC-FIXED: Optimal accumulator based generators for single size subspaces

Arithmetic Built-In Self-Test (ABIST) [RT98] is a term coined by Janus Rajski and Jerzy Tyszer. They pointed out that existing components in todays complex integrated circuits often contain ALUs and memory that can be reused for testing purposes. One efficient way to generate stimuli (measured in the number of clock cycles needed to generate a new vector) is to accumulate a constant as shown in Equation 7.1.

$$A_{i+1} = A_i + C \bmod 2^n, A_0 = I \tag{7.1}$$

There are 2^{2n} ways of choosing pairs of C and I, and it is, to some extent, possible to chose C and I so that the resulting generator exhibits some interesting properties.

By carefully selecting the parameters C and I it is possible to cover exhaustively every subinterval of size r within the first 2^r test vectors [RT98]. A pseudo-exhaustive generator like this can be used to test modules with physically adjacent input lines (e. g. adders). The number of inputs driving each partition (often a fanin cone) may be the same for some regular structures, but in general their sizes differ. The value of r should then be set equal to number of inputs to the partition with the largest number of inputs.

7.4.2 ACC-RANGE: The best accumulator based generators for subspaces within a range of sizes

The number of inputs to the partitions often varies, and in such cases a generator made for subspaces with fixed size might be suboptimal. It is not possible to synthesize values for C and I for Equation 7.1 in such cases. A full search through all possible generators is the only option way to find generators based on Equation7.1 that exhaustively covers a *range* of subinterval sizes in the shortest number of test vectors. An efficient method for pruning the search space is presented in [RT98].

7.5 Pseudo random patterns

Pseudo random pattern generators are generators that generate sequences that have the same properties as true random sequences even though the sequence is generated by a deterministic algorithm. The number of test vectors needed

in order to detect all faults are usually much smaller than the number of test vectors generated during an exhaustive or pseudo-exhaustive test. However, the test sequence might still be long due to *random pattern resistant faults*. Pseudo random tests are therefore usually topped by some deterministic tests.

7.5.1 LFSR: Linear feedback shift register

The most popular pseudo random generator is the *linear feedback shift register* (LFSR). LFSRs are very efficient (require little logic) in hardware and are also easy to emulate in software. One example of software emulation of LFSRs is described in [RT98].

7.5.2 TWISTER: Mersenne twister pseudo random generator

Mersenne Twister [MN98] is a pseudo-random generator which has a period of $2^{19937} - 1$. The generator is fairly complex and is not suitable for use in built-in self-test. However, there are many pitfalls when designing pseudo-random generators, and the Mersenne twister may thus be used as a verification tool in the design phase. If, for instance, an LFSR based generator in a BIST environment performs much poorer than the Mersenne twister, it may be caused by some structural or linear dependencies.

7.5.3 MAC: Multiply and accumulate based generator

In order to reduce the test application time of large sequential circuits with scan, the scan chain is usually broken down into several scan chains. These scan chains must then be fed by the test generator. LFSRs may, due to structural and linear dependencies, fail to produce some test patterns. Instead one can use a generator based on multiply and accumulate (MAC) operations. The congruent random number generator algorithm by Knuth cannot be used in its original form [RT98]. It is a poor source for random sequences on designated bit positions, even though the entire numbers are quite random [RT98]. Rajski and Tyszer instead suggests the method in Equation 7.2 where A_i^l and A_i^h are the contents of the n least significant and n most significant bits of A, respectly, after i iterations. M is an n-bit constant. It is assumed that A_{i-1}^h is shifted right by n bits prior to a next addition. A table of initial values that will provide the longest sequences is given in [RT98].

$$A_i = MA_{i-1}^l + A_{i-1}^h \bmod 2^n \tag{7.2}$$

7.6 WEIGHT: Weighted pattern generators

Even simple circuits may contain random pattern resistant faults. Figure 7.3, for example, shows the only *robust* test for the falling path-delay fault starting at the *on-input* of the *AND* gate, and ending at the output of the *AND* gate. If a pseudo-random generator were applied to this circuit it would hit the desired second pattern of the two-pattern test with a probability of 2^{-n}. One method to increase the probability of detecting such faults is to employ a *weighted pattern generator*. In a weighted pattern generator the outputs are biased so that patterns needed for random resistant faults are more likely to occur.

Methods for generating weights based on structural analysis of the CUT and deterministic test sets can be found in [Ber93, KPSW94, MAND90, PR91, WLEF89, Wun87, Wun88, Wun90]. Other approaches is found in [IC02, PN88, Maj96, Sav99, PR93, KPSW96, KTM96, NK97, KT99, LPRWT05, KLK01].

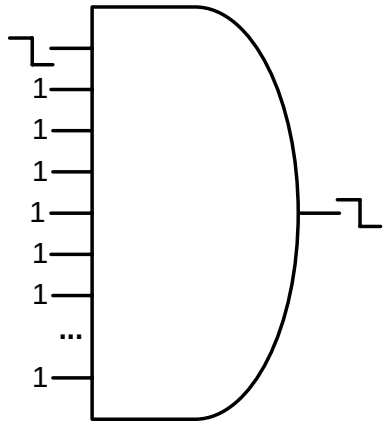


Figure 7.3: Random pattern resistant path-delay fault

7.6.1 DETERM: Weights based on deterministic test set for stuck-at faults

There are several ways of creating weights for a weighted pattern generator. One common method used in conjunction with the single stuck-at fault model, is to create weights based on a deterministic test set for stuck-at faults. Even though the test generators presented here will be evaluated for their efficiency to detect path-delay faults, it is interesting to find out whether or not a weight set based on a deterministic test set for stuck-at faults would yield a good result.

The weights was generated as follows. First a deterministic test set for each of the benchmark circuits was created by TetraMax, an ATPG from Synopsys. Each deterministic test set consists of a number of test vectors as shown in Figure 7.4.

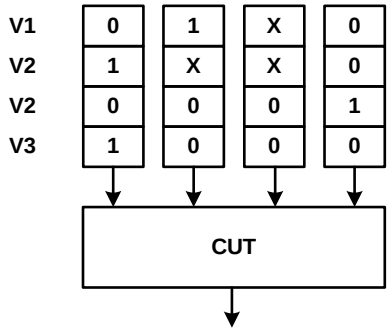


Figure 7.4: A deterministic test set.

The weights were determined by first counting the number of test vectors $n1_i$ with the symbols '1' and 'X' present at input i, as well as the number of test vectors $n0_i$ with the symbols '0' and 'X' present at input i. Values for $n0$ and $n1$ is shown in Figure 7.5 for the test set in Figure 7.4. The probability for observing '1' at the input of the circuit under test at input i can thus be computed as shown in Equation 7.3.

$$p_1 = n_1/(n_1 + n_0) \tag{7.3}$$

n0	2	3	4	3
n1	2	2	2	1
p1	2/4	2/5	2/6	1/4

Figure 7.5: Computation of weights based on a deterministic test set for stuck-at faults.

7.6.2 PDF-DETERM: Weights based on deterministic test set for path-delay faults

The ATPG described in Chapter 6 can be used in order to extract the K-longest testable paths in a circuit. Section 8.2.1 describes an experiment (EX1) where

the 20000 longest non-robust testable paths were extracted together with a valid test vector.

The weights were then computed based on the deterministic test set for path-delay faults in the same way as described in Section 7.6.1.

7.6.3 COUNTING: Weights based on counting of detected faults

Weighs can also be generated based on fault coverage measurements. Figure 7.6 shows a circuit with two inputs and one output. The circuit is attached to a pseudo-random generator that creates uniformly distributed *basis patterns*. Two counters ($S0Ctr$, $S1Ctr$) are associated with each input. The purpose of $S0Ctr$ ($S1Ctr$) is to store the number of faults detected when the input has the value $S0$ ($S1$). When the desired number of basis patterns has been applied the weighting factors can be computed for every input according to the formula in Equation 7.4.

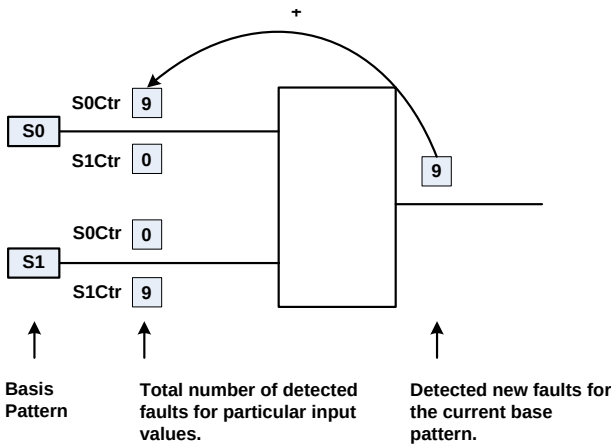


Figure 7.6: Computation of weights based on counting of detected faults.

$$p_1 = S1Ctr/(S1Ctr + S0Ctr) \tag{7.4}$$

Circuits will typically have some paths that are very easy to detect and other paths that are more difficult to detect. The easiest to detect faults are usually detected in the first few vectors anyway, thus it is a good idea to try to tune the weights on to the faults that are a bit more difficult to detect. When weights were generated using this method, the first 100 basis vectors were skipped and did not contribute to the computation of the weights. 10M SIC patterns were

applied to each CUT during computation of the weights, the same number of SIC patterns as will be used in the experiments.

7.6.4 FAULT-SUBSET: One weight for each subset of faults

Each path-delay faults starts at an input-node and ends at an output-node. This can be used to efficiently divide the set of all path-delay faults into smaller disjoint subsets containing only faults that ends at a particular output, starts at a particular input or both. This is illustrated in Figure 7.7 for a circuit with four inputs and eight outputs.

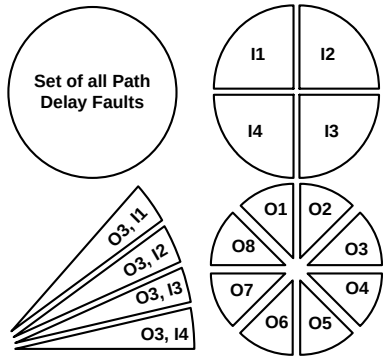


Figure 7.7: Set of all Path Delay Faults divided into disjoint sets with the same input, output or both.

Instead of trying to detect all faults in the fault set using one fixed weight set, it may be more efficient to restart the generator with weights that target one subset of the fault set at a time. The method presented in Section 7.6.3 can easily be adapted in order to achieve this as shown in Figure 7.8.

The figure shows a circuit with two inputs and four outputs. The circuit is attached to a pseudo-random generator that creates uniformly distributed base patterns. Associated with each input there are two arrays of counters, one array of counters ($S0Ctr$, $S1Ctr$) for each of the two possible input values ($S0$, $S1$). The purpose of $S0Ctr[i]$ ($S1Ctr[i]$) is to store the number of faults detected that ends at output i when the input has the value $S0$ ($S1$). When the desired number of base patterns have been applied, the weighting factors can be computed for every input and subset i according to the formula in Equation 7.5.

$$p_1[i] = S1Ctr[i]/(S1Ctr[i] + S0Ctr[i]) \tag{7.5}$$

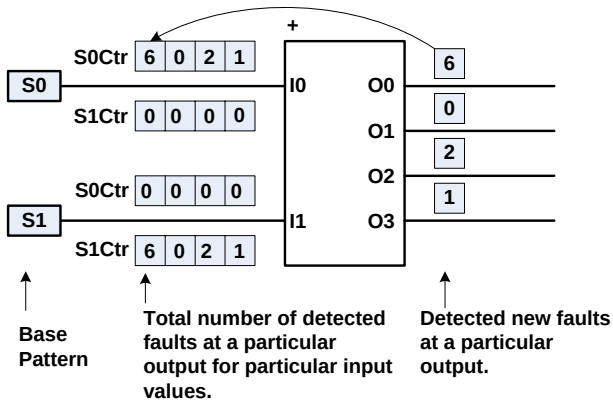


Figure 7.8: Fault set partitioning and computation of weights based on counting of detected faults.

This section has described a method for generating weights optimized for different subsets of the total set of path-delay faults. When these weights are used to bias a pseudo-random generator a strategy for restarting the generator with new weights must be deviced. Two such strategies are presented next in Section 7.6.5 and Section 7.6.6.

7.6.5 REL-SEEDOPT: Weight sequence optimization based on relative fault detection

The procedure in Figure 7.8 produces one weight set for each output in the circuit under test. Associated with each weight set is a fault count d_i that can be used in order to schedule the restarting of the generator. Assume that the available test budget is T test vectors. Each weight set can be given a fair share of this test budget by assigning t_i (Equation 7.6) test vectors to weight set i.

$$t_i = \frac{d_i}{\sum d_i} * T \tag{7.6}$$

Although the weight sets are optimized with respect to their associated output, a generator will also detect some faults from other subsets of the fault set than the subset targeted by the current weight set. In order to climb faster to high fault coverage, the method in Section 7.6.6 was deviced.

7.6.6 SIM-SEEDOPT: Optimizations of weight sequence based on simulation

Figure 7.9 illustrates the number of detected faults as a function of the number of applied vectors for an imagined circuit. The double line represent the test budget T. The other vertical lines indicate places where restarting of the generator with a new weight set may take place.

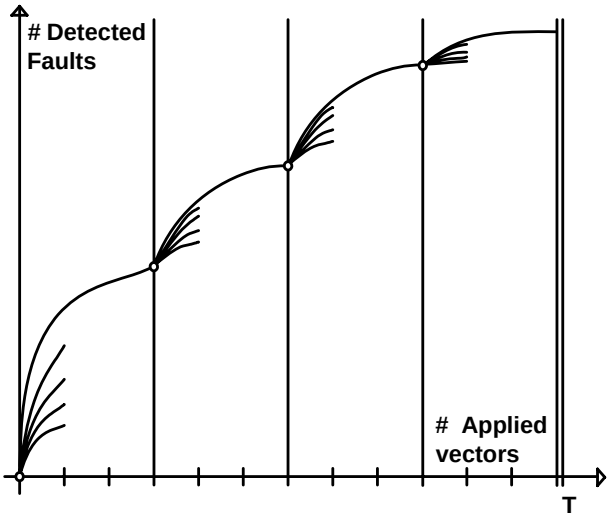


Figure 7.9: Weight sequence optimization based on simulation.

The algorithm assigns one weight set to each of the (in this example) four intervals defined by the vertical lines in Figure 7.9. Thus, the fault coverage increases quicker after each restart.

In order to achieve this, a greedy approach is used. All weight sets are loaded into the generator, and the weight set that yields the highest coverage after the first (in this example) third of an interval, is used as the weight throughout that interval. This procedure is repeated for all intervals.

7.7 Test pattern generators

This chapter has described a set of methods and techniques that have been used as tools in designing the test pattern generators. Each technique is identified by the *tag* present in the caption over the section describing the technique. This

section presents the different test pattern generators formed by using these techniques.

Table 7.1 and Table 7.2 give an overview of the different stimuli generators. The first column of each table contains the *ID* of each generator. The second column contains a number of tags or properties associated with the stimuli generator, and the last column gives a short description of the generator.

7.8 Assembly test programs

The purpose of this section is to show how the pseudocode of the test generator algorithms can be translated into efficient assembly programs.

7.8.1 Test application scheme

When the actual test-programs are created it is also necessary to take into account the test application scheme. During simulation it is assumed that the first pattern in the two-pattern test is held until it has propagated through the circuit under test. The second vector creates a transition and the signature compactor must sample the response one clock cycle later in order to capture any path delay faults. The test programs shown here only contain the test generator. They can easily be extended in order to include a response compactor as well. It is assumed that the combinational CUT is connected to the processor through the register $R0$, and that $R0$ has the same width as the number of inputs to the CUT.

7.8.2 Instruction set

The test programs presented in this paper is implemented using the instruction set presented in Table 7.3. It is assumed that every instruction executes within one clock cycle.

7.8.3 Weighting factors

Some of the test generators presented generate weighted pseudo-random test patterns. Such weighting factors can easily be implemented using standard *AND*/*OR* operations as shown in Figure 7.10.

If an *AND* operation is to take place for some bits but not all, it is possible to exploit that $a * (b + 1) = a$. If an *OR* operation is to take place for some bits but not all, it is possible to exploit that $a + (b * 0) = a$. The probabilities in Figure 7.10 can thus be created by: $p = [a*(m_1+b)+(m_2*b)]*(m_3+c)+(m_4*c)$

Table 7.1: Overview of test pattern generators using ABIST as the underlying random generator

ID	Tags	Description
GA1	SIC, ACC-FIXED	Pseudo-exhaustive accumulator based test generator. Optimal for subspaces with size r=8. C and I picked randomly from a determined set.
GA2	SIC, ACC-FIXED, WEIGHT, DETERM	Weighted random generator based on three different accumulator based generators (r=16. C and I picked randomly). Weights are based on a deterministic test set for stuck-at faults.
GA3	SIC, ACC-FIXED, WEIGHT, COUNTING	Same as *GA2* but the weights are computed using a different method. The weights for each input pin is based on the observed coverage for each value assignment taken from the set $V = \{0, 1\}$ while the rest of the inputs are assigned N pseudo-random vectors.
GA4	SIC, ACC-FIXED, WEIGHT, FAULTSUBSET, RELSEEDOPT	Same as *GA2* but the weights are computed using a different method. The generator is reseeded *nPO* times (*nPO* is the number of primary outputs) with different weights. Each weight is computed in the same way as for *GA3*, but only the paths ending at the particular output is considered when the fault coverage is detected.
GA5	SIC, ACC-FIXED, WEIGHT, FAULTSUBSET, SIMSEEDOPT	Same as *GA4*, but the sequence in which the seeds are applied is optimized.
GA6	SIC, ACC-FIXED, WEIGHT, PDFDETERM	Weighted random generator based on three different accumulator based generators (r=16. C and I picked randomly). Weights are based on a deterministic test set for path-delay faults.
GAU	SIC, ACC-FIXED, WEIGHT	Same as GA2/GA3/GA6, but with all weights set to 0.5.

Table 7.2: Overview of test pattern generators using Mersenne twister as the underlying random generator

ID	Tags	Description
GT1	SIC, TWISTER	Same as GA1, but using Mersenne twister as the underlying random generator.
GT2	SIC, TWISTER, WEIGHT, DETERM	Same as GA2, but using Mersenne twister as the underlying random generator.
GT3	SIC, TWISTER, WEIGHT, COUNTING	Same as GA3, but using Mersenne twister as the underlying random generator.
GT4	SIC, TWISTER, WEIGHT, FAULTSUBSET, RELSEEDOPT	Same as GA4, but using Mersenne twister as the underlying random generator.
GT5	SIC, TWISTER, WEIGHT, FAULTSUBSET, SIMSEEDOPT	Same as GA5, but using Mersenne twister as the underlying random generator.
GT6	SIC, TWISTER, WEIGHT, PDFDETERM	Same as GA6, but using Mersenne twister as the underlying random generator.
GTU	SIC, TWISTER, WEIGHT	Same as GT2/GT3/GT6, but with all weights set to 0.5.

with careful selection of the four mask bits m1, m2, m3 and m4 as shown in Table 7.4.

7.8.4 Implementation of GA1 in assembly code

Figure 7.11 shows an assembly implementation of GA1. It consists of three parts. The program starts at the address with the label INIT where the test program generator is initialized. Rising and falling transitions are then created

Table 7.3: Instruction set

Instruction	Description
LOAD Rd, k LOADI Rd, k	Set the content of register Rd to [data stored in memory address k — immediate].
MOV Rd, Rr	Move content of Rr to Rd.
ADD Rd, Rr ADDI Rd, k	Add the content of [reg. Rr — immediate] to Rd. Store the result in Rd.
ADDC Rd, Rr ADDCI Rd, k	Add the content of [reg. Rr — immediate] with carry to Rd. Store the result in Rd.
AND Rd, Rr ANDI Rd, k	Bitwise AND operation of [reg. Rr — immediate] and Rd. Result is stored in Rd.
OR Rd, Rr ORI Rd, k	Bitwise OR operation of [reg. Rr — immediate] and Rd. Result is stored in Rd.
XOR Rd, Rr	Bitwise XOR operation of Rr and Rd. Result is stored in Rd.
NOT Rd	Invert the bits in Rd.
ROL Rd	Rotate left the content of Rd.
BRNE k	Set program counter to k if the equal flag is not set.
CPI Rd, k	Compare register with immediate.

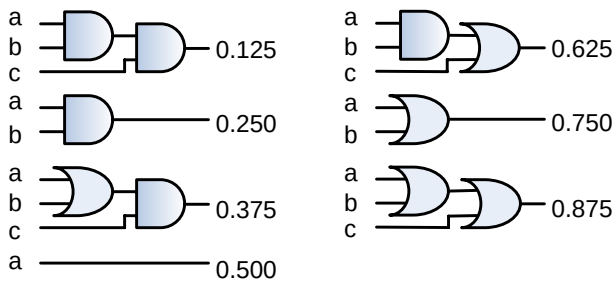


Figure 7.10: Synthesis of weighting factors using AND/OR operations

Table 7.4: Masks for generating the weights in Figure 7.10

Function	m_1	m_2	m_3	m_4
abc	0	0	0	0
ab	0	0	1	0
$(a+b)c$	1	1	0	0
a	1	0	1	0
$ab+c$	0	0	1	1
$a+b$	1	1	1	0
$a+b+c$	1	1	1	1

for each bit as shown from the address with label SIC. When all $2n$ transitions have been created, a new basis vector is created until a total of N basis vectors have been created. The N basis vectors corresponds to $2n * N$ two pattern tests.

```
const  N = 0x5 # No. of basis vectors
const  C = 0xB # Constant to be added
const  I = 0x9 # Initial vector

INIT : LOAD R4, N    # Move N to R4
       LOAD R0, I    # Move I to R0
       LOAD R6, 0x1  # Move 1 to R6
SIC  : XOR  R0, R6   # Create transition
       NOOP
       XOR  R0, R6
       ROL  R6
       CPI  R6, 0x1
       BRNE SIC      # Toggle all bits
BASIS: ADD  R0, C    # Create new Basis
       SUBI R4, 0x1
       BRNE SIC
FINISH: NOOP         # End of test
```

Figure 7.11: Assembly code implementation of GA1.

The output of register $R0$ is assumed connected directly to the CUT, and the sequence generated by the test program is shown in Figure 7.12. Each instruction in the test program in Figure 7.11 corresponds to one line in the listing in Figure 7.12. The vertical wave in the figure illustrates the falling and rising transition on the LSB in the input vector.

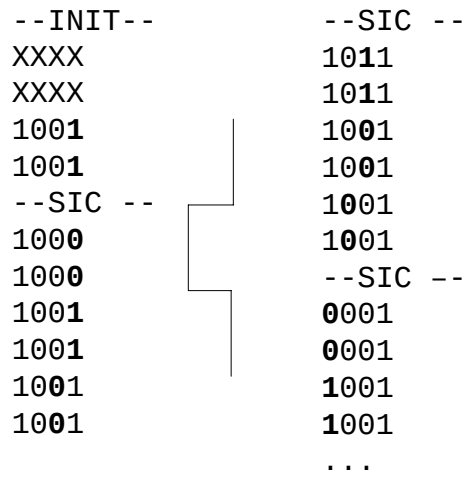


Figure 7.12: Output of GA1 with parameters $N = 4$, $C = 1011$, $I = 1001$.

Chapter 8

Experiments

This chapter describes the experiments carried out in order to evaluate the stimuli generators presented in the previous chapter. Only non-robust faults (including robust faults) are considered in the experiments.

8.1 Benchmark circuit properties

The circuits in the iscas'85 benchmark suite have been used in the experiments presented in this chapter. In order to be able to interpret the results, some information about each benchmark is provided in this section. The information was extracted from the netlists by sending queries to the path-delay fault simulator described in Chapter 5.

The number of inputs, outputs, gates, logical levels and physical paths for each circuit is shown in Table 8.1. Among the things that can be observed is the huge number of physical paths in benchmark c6288 (a 16x16 bit array multiplier).

The paths in the benchmarks are of different length. An overview of the number of paths of different length is illustrated by the histograms shown in Figure 8.1 and Figure 8.2. A histogram is not shown for the small circuit c17, but it has five paths with length three and six paths with length four.

The circuits c432 and c499 are ommitted from most of the experiments because they contain XOR-gates, which is not currently supported by the ATPG. Another circuit that is ommitted from most experiments is c6288. The large number of paths in this circuit cause problems for the both the simulator and the ATPG. The rest of the benchmarks are used in all experiments.

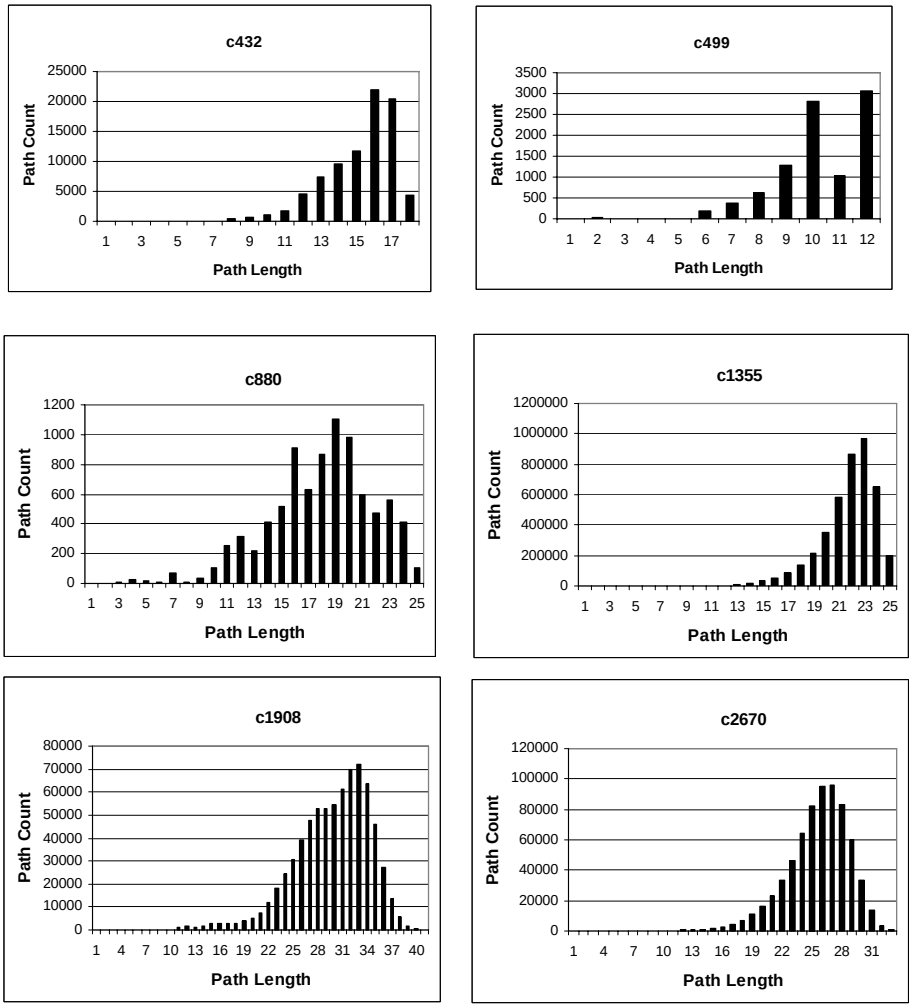


Figure 8.1: The number of physical paths of different lengths.

Table 8.1: Benchmark properties

Circuit	Inputs	Outputs	Gates	Levels	Paths
c17	5	2	13	4	11
c432	36	7	203	18	83926
c499	41	32	275	12	9440
c880	60	26	469	25	8642
c1355	41	32	619	25	4173216
c1908	33	25	938	41	729057
c2670	233	140	1566	33	679960
c3540	50	22	1741	48	28676671
c5315	178	123	2608	50	1341305
c6288	32	32	2480	125	98943441738294937238
c7552	207	108	3827	44	726494

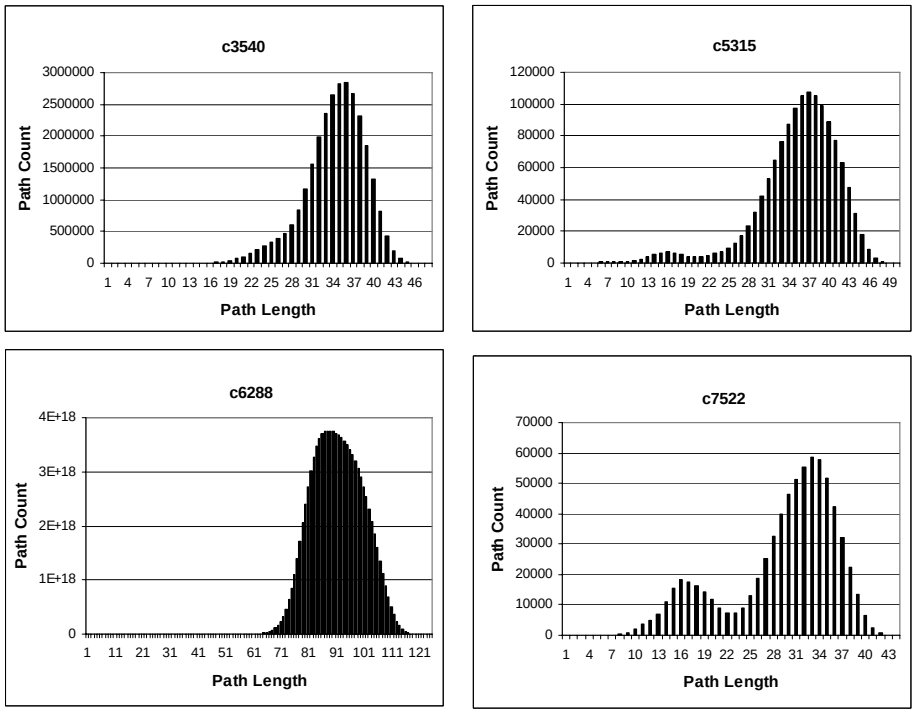


Figure 8.2: The number of physical paths of different lengths.

8.2 Statistical properties of the test generators

This section presents some statistical properties of the sequences generated by the different test generators described in Chapter 7. This information can be used as an aid in the interpretation of the results from the experiments presented in this chapter.

8.2.1 EX1: Find the K-longest testable paths in each circuit.

The objective of this experiment was to find the longest non-robust testable paths of each benchmark circuit. This was achieved by using the ATPG tool described in Chapter 6. If unlimited time and memory had been available, the tool would be able to list all testable faults in the benchmark circuit. Unfortunately some of the benchmarks contains a huge number of testable path-delay faults, and this would cause the size of the data structure inside the ATPG tool to blow up. In order to keep the whole path store inside computer memory, the size of the path store was set to a maximum of 1M. The ATPG was asked to find the 20K longest non-robust testable pathts in each of the benchmarks. The number of non-robust testable paths found for the different circuits are listed in Table 8.2 together with an upper bound[CC93] of all non-robust path delay faults. Since all circuits except c880 contain more than 20K testable paths, the ATPG had no problem finding 20K testable paths. It is also reassuring to observe that the number of testable paths found in c880 indeed matched the upper bound in [CC93].

Table 8.2: The number of testable paths found.

Circuit	U. Bound	Paths found
c880	16652	16652
c1355	1110076	20000
c1908	355197	20000
c2670	1306884	20000
c3540	12330969	20000
c5315	353300	20000
c7552	282752	20000

8.2.2 EX2: Determine how many paths of different length are detected by unweighted pseudo-random stimuli.

In this experiment test vectors were applied to the benchmarks, and the number of detected path-delay faults and their length were logged. The test vectors used in this experiment were generated by an unweighted pseudo-random generator (GT1). The purpose of the experiment was to obtain some information about the number of paths of different lengths typically detected by a standard pseudo-random generator. The results are presented in Figure 8.3 and Figure 8.4.

When these plots are compared with the total number of paths in the circuits of different lengths (Figure 8.1 and Figure 8.2), it is easy to observe that the longer paths are not as easily detected as shorter paths. To make it even more obvious the centre of gravity, i.e. the average path length, was computed for each circuit in Figure 8.3 and Figure 8.4. Table 8.3 lists the average path length of the detected path-delay faults after 100K, 200K, 1M, 2M and 4M applied test vectors. The last column in the table lists the expected length of a randomly selected path-delay fault out of all possible path-delay faults in the circuit.

According to Table 8.3 the expected length of the detected faults increases when more test vectors have been applied to the circuit under test. The table also shows that the average length of a randomly selected physical path is higher than the average length of the paths detected within 4M test vectors. It is possible to conclude from this that the shorter paths are easier to detect with unweighted pseudo-random stimuli than the longer paths. This is also in accordance with what one would expect, since longer paths have more signal-assignment requirements in order to successfully sensitize the path-delay fault than shorter paths have.

8.2.3 EX3: Comparison of GA1, GA2, GA3, GA4, GA5

In this experiment the performance of GA1, GA2, GA3, GA4 and GA5 was evaluated. The experiment was carried out by using the fault simulator described in Chapter 5. 10M test patterns were simulated for each circuit and generator. Each simulation run was repeated 10 times with different seeds in order to cover statistical variations. Table 8.4 presents the average number of detected faults over 10 trials after 10M applied test vectors.

The best result, i.e. the highest number of detected faults, is shown in bold in Table 8.4 for each circuit. The stimuli generator with the poorest performance is the unweighted pseudo-random generator GA1. This generator detected the fewest number of non-robust path delay faults in all tests. Generator

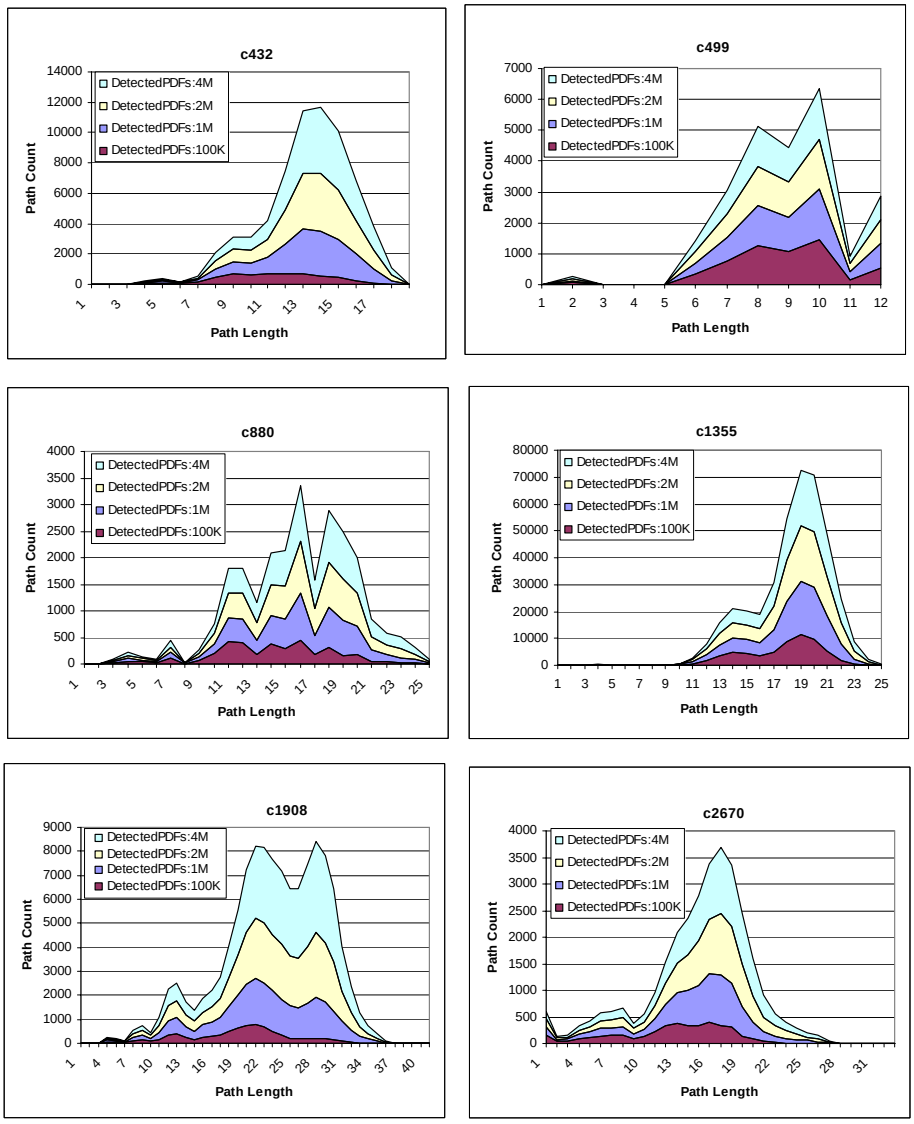


Figure 8.3: Detected paths and their length using GT1.

Table 8.3: Average path length of detected faults.

Circuit	100K	200K	1M	2M	4M	PathCount
c432	10.8	-	12.5	12.7	12.8	15.2
c499	8.8	-	9.0	9.0	9.0	10.3
c880	14.1	-	15.8	16.0	16.3	17.9
c1355	17.6	-	18.4	18.5	18.6	21.8
c1908	19.4	-	22.5	23.3	24.0	29.7
c2670	12.7	-	14.9	15.5	15.8	25.6
c3540	22.5	-	24.0	24.5	24.9	34.4
c5315	14.6	-	18.5	19.6	20.6	35.4
c6288	48.3	50.8	-	-	-	91.6
c7552	17.8	-	20.1	20.5	20.8	29.2

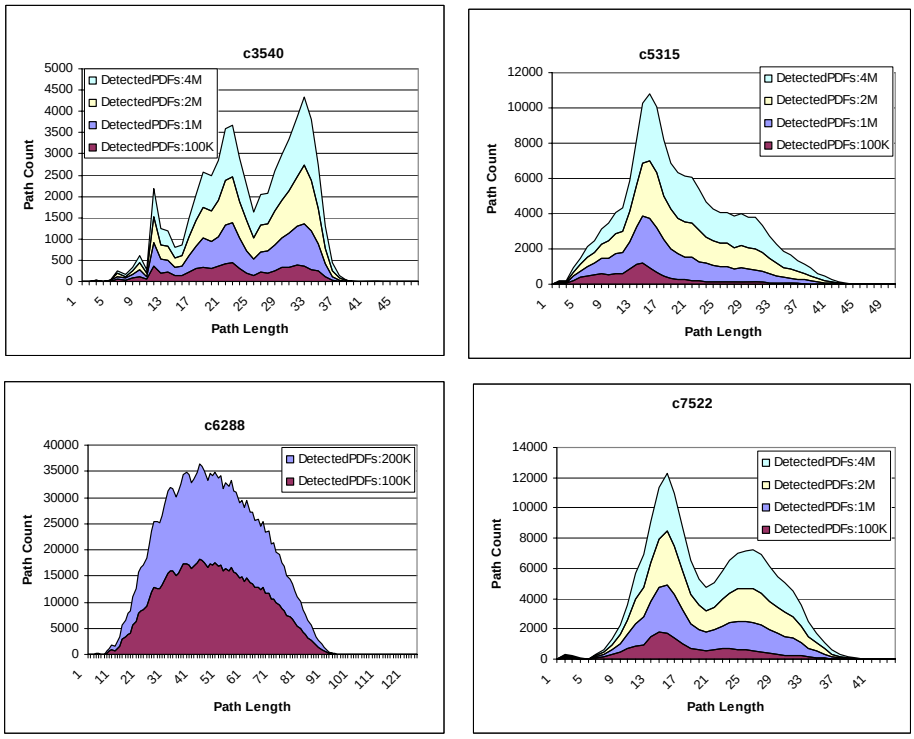


Figure 8.4: Detected paths and their length using GT1.

Table 8.4: Detected faults after 10M applied test vectors

Circuit	GA1	GA2	GA3	GA4	GA5
c880	8714	16194	**16550**	16470	16473
c1355	1050139	1085021	**1110297**	1110264	1110258
c1908	269846	283665	**349613**	349579	349568
c2670	51739	85948	**107711**	102734	104141
c3540	588541	996001	**1062718**	1050384	1050579
c5315	173526	309498	**339396**	339122	339157
c7552	146754	**185983**	185687	185264	185383
Sum	2289259	2962310	**3171972**	3153817	3155559

GA2, which is a weighted pseudo-random generator with weights based on a deterministic test set for stuck-at faults, is somewhat better than GA1. The three best generators are GA3, GA4 and GA5. The performance of GA3, GA4 and GA5 does not differ by much, but the results point in favour of GA3, which detects most path-delay faults for all but one benchmark. GA3 is a weighted pseudo-random generator with weights based on the counting scheme described in Chapter 7.

Figures 8.5- 8.7 show the number of detected faults after 10M applied test vectors for the circuits in the iscas'85 benchmark suite.

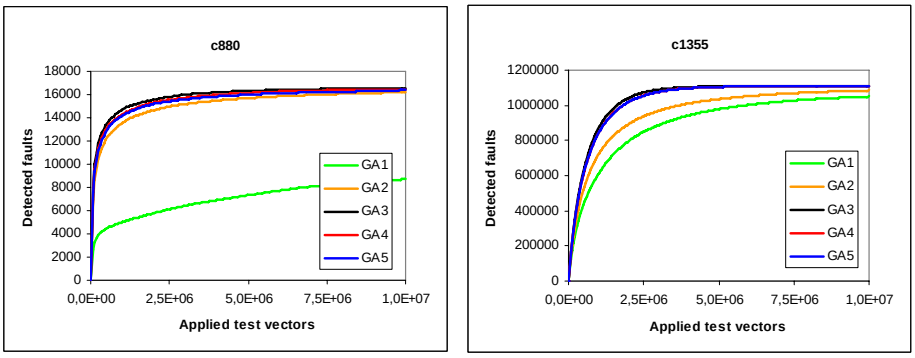


Figure 8.5: Detected faults for c880 and c1355 after 10M applied vectors.

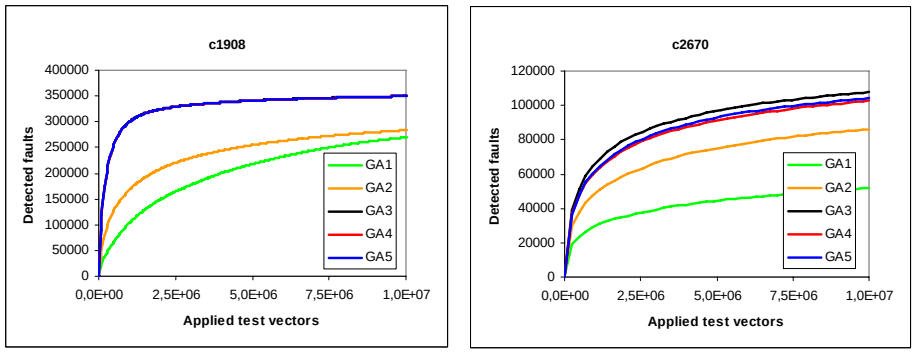


Figure 8.6: Detected faults for c1908 and c2670 after 10M applied vectors.

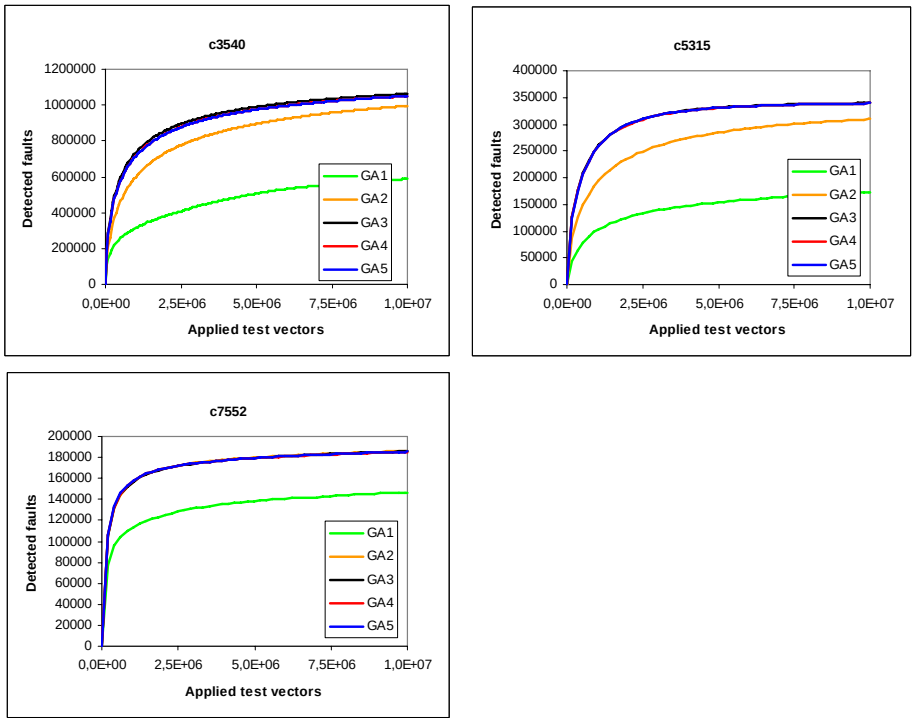


Figure 8.7: Detected faults for c3540, c5314 and c7552 after 10M applied vectors.

8.2.4 EX4: Comparison of GT1, GT2, GT3, GT4 and GT5

In this experiment the performance of GT1, GT2, GT3, GT4 and GT5 was evaluated. The experiment was carried out by using the fault simulator described in

Chapter 5. 10M single-input-change test patterns were applied for each circuit and generator. Each simulation run was repeated 10 times with different seeds in order to cover statistical variations. Table 8.5 presents the average number of detected faults over 10 trials after 10M applied test vectors.

Table 8.5: Detected faults after 10M applied test vectors

Circuit	GT1	GT2	GT3	GT4	GT5
c880	14303	16159	**16571**	16496	16498
c1355	1109963	1068481	**1110297**	1110270	1110289
c1908	317450	263360	**349119**	345722	347978
c2670	88467	87096	**108235**	105236	105873
c3540	1012172	1010114	**1065546**	1060025	1060532
c5315	309562	313712	**340124**	339622	339921
c7552	185595	**186215**	185783	185717	185894
Sum	3037512	2945137	**3175675**	3163088	3166985

The best result, i.e. the highest number of detected faults, is shown in bold in Table 8.5 for each circuit. The stimuli generator with the poorest performance is the weighted pseudo-random generator GT2. This generator detected the fewest number of non-robust path delay faults in most of the tests. Generator GT1, which is the unweighted pseudo-random generator, is somewhat better than GT2. The three best generators are GT3, GT4 and GT5. The performance of GT3, GT4 and GT5 does not differ by much, but the results point in favour of GT3, which detects most path-delay faults for all but one benchmark. GT3 is a weighted pseudo-random generator with weights based on the counting scheme described in Chapter 7.

Figures 8.8- 8.10 show the number of detected faults after 10M applied test vectors for the circuits in the iscas'85 benchmark suite.

8.2.5 EX5: Weighted pseudo-random patterns targeting the K-longest testable path-delay faults

The purpose of this experiment was to find out if proper weighting of pseudo-random stimuli, based on K=20000 deterministic test patterns for path-delay faults, would yield more efficient path delay tests than using uniformly distributed patterns. The experiments was conducted as follows:

First the K=20000 longest testable path-delay faults were extracted for each circuit as described in EX1. For each detected path, the path number was stored

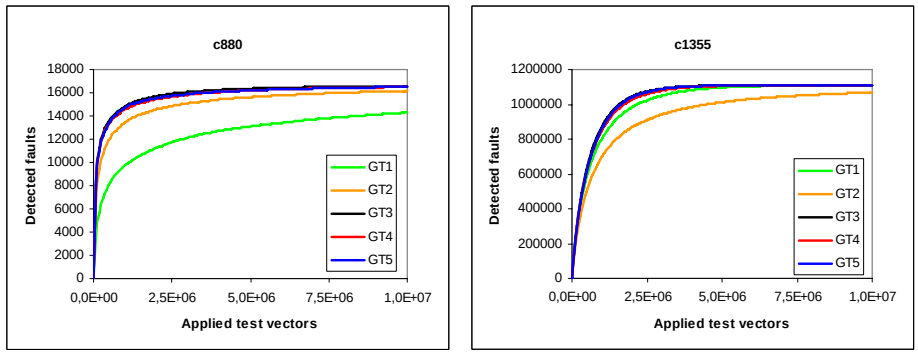


Figure 8.8: Detected faults for c880 and c1355 after 10M applied vectors.

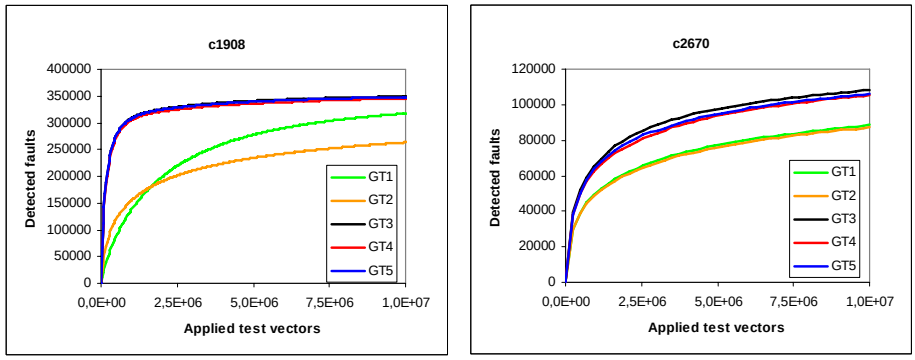


Figure 8.9: Detected faults for c1908 and c2670 after 10M applied vectors.

in a file together with the correspondning path length and test vector. Weights for GA6 and GT6 were then extracted based on each test set as described in described in 7.6.2.

Prior to each simulation run a fault list with the 20000 longest testable path-delay faults was uploaded to the simulator. 10M single-input-change test patterns were then applied to each circuit for each generator. Each simulation run was repeated 10 times with different seeds in order to cover statistical variations.

Six different generators were used: GAU, GA3, GA6, GTU, GT3 and GT6. The three generators GAU, GA3 and GA6 are using the exact same underlying accumulator based pseudo-random generator. GA3 and GA6 are weighted pseudo-random generators and will be compared against GAU, which has all weights set to 0.5. The three generators GTU, GT3 and GT6 are using the exact

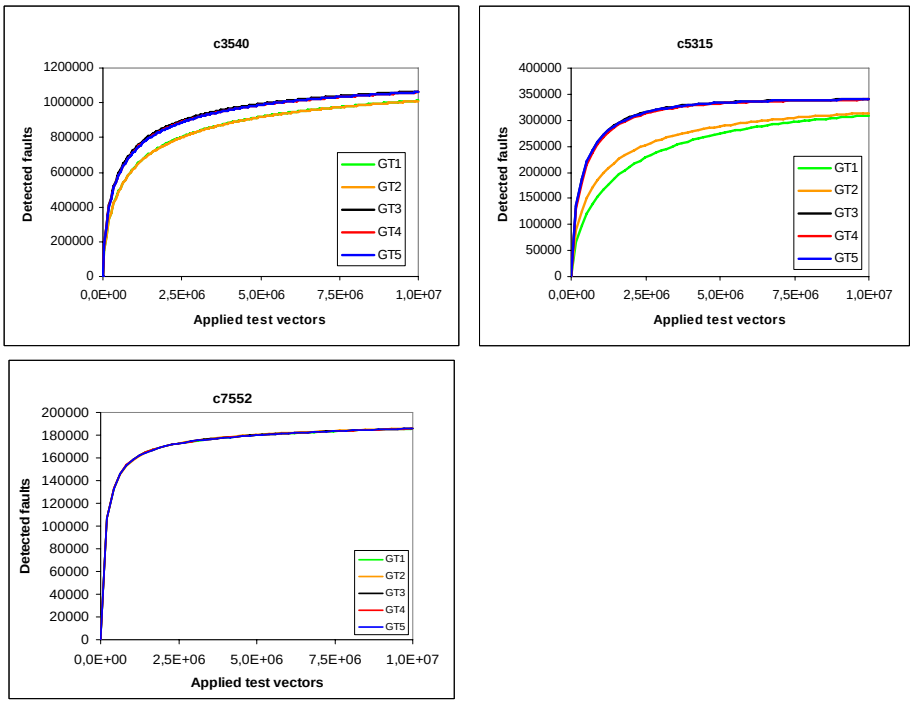


Figure 8.10: Detected faults for c3540, c5314 and c7552 after 10M applied vectors.

same underlying mersenne twister pseudo-random generator. GT3 and GT6 are weighted pseudo-random generators and will be compared against GTU, which has all weights set to 0.5.

Three measures were recorded:

- Fault coverage in relation to the size, K, of the fault list. (K=20000 for all circuits except c880 which has only 16652 non-robust testable paths).
- Test time speedup, defined as the ratio $R_{imp} = NTP(uniform)/NTP(method_x)$. ($NTP$ represents the number of test patterns. The name of the stimuli generator is used as argument).
- Standard deviation in fault coverage, C, after 10M applied SIC test patterns computed over the N=10 different runs according to Equation 8.1.

$$\sigma = \frac{\Sigma(C_i - \overline{C})}{\sqrt{N-1}} \tag{8.1}$$

During simulation the fault coverage, FC, was sampled from time to time until 10M test pattern had been applied. Table 8.6 shows the fault coverage after 10M applied test vectors. The numbers in the second coloumn represent fault coverage achieved with the best generator of GA3 and GA6. The numbers in the third coloumn represent the fault coverage achieved with the best generator of GT3 and GT6.

We observe from Table 8.6 that the GT methods are slightly better than the GA methods. Furthermore, 5 out of 7 circuits attain 97.6% fault coverage, or more. Two circuits exhibit inferior fault coverage, and need more test patterns or other methods of path delay fault detection.

Table 8.6: Fault coverage, FC, of best method after 10M applied test vectors

Circuit	$FC(GA_x)$	$FC(GT_x)$
c880	99.3% ($GA3$)	99.5% ($GT3$)
c1355	100% ($GA3$)	100% ($GT3$)
c1908	97.6% ($GA3$)	98.2% ($GT3$)
c2670	67.9% ($GA6$)	71.0% ($GT6$)
c3540	86.9% ($GA6$)	88.2% ($GT6$)
c5315	96.7% ($GA6$)	98.7% ($GT6$)
c7552	99.8% ($GA3$)	99.7% ($GT6$)
Average	92.6%	93.6%

The standard deviation of the sample fault coverage after 10M applied test patterns over the 10 trials is given in Table 8.7. The variation is modest. If all possible test patterns were applied to the circuits, all faults in the fault set would eventually be detected, and the standard deviation would be zero. This is the situation for circuit c1355. The two circuits with lowest fault coverage (c2570 and c3540) did not reach saturation even after 10M test patterns, and for these circuits the standard deviation is higher than for the other circuits.

In order to measure the speedup of a weighted generator over that of a uniformly distributed pseudo-random generator, one can compare the number of test vectors needed in order to achieve the same fault coverage. The target coverage in our case was set to the fault coverage achieved with the unweighted generator after 10M applied stimuli. The improvement factor of the best weighted generator over uniformly distributed stimuli is listed in Table 8.8.

We observe time speedups from nothing to a factor 15! It is not possible to device an a priori metric to predict speedup. But given some substantial savings in test time, and thus savings of test cost, it can be recommended to try GA3

Table 8.7: Standard deviation of relative fault coverage

Circuit	$R_{imp}(GA_x)$	$R_{imp}(GT_x)$
c880	0.11% ($GA3$)	0.12% ($GT3$)
c1355	0.00% ($GA3$)	0.00% ($GT3$)
c1908	0.11% ($GA3$)	0.15% ($GT3$)
c2670	1.50% ($GA6$)	1.48% ($GT6$)
c3540	0.60% ($GA6$)	0.43% ($GT6$)
c5315	0.22% ($GA6$)	0.09% ($GT6$)
c7552	0.12% ($GA3$)	0.05% ($GT6$)

Table 8.8: Time speedup of best method over uniformly distributed stimuli

Circuit	$R_{imp}(GA_x)$	$R_{imp}(GT_x)$
c880	11.9	15.1
c1355	1.5	2.7
c1908	8.0	10.8
c2670	10.7	14.3
c3540	7.1	9.1
c5315	4.7	7.0
c7552	1.0	1.0

and GA6 for a newly designed circuit, and use this method if beneficial.

8.2.6 EX6: Distributed simulation utilizing idle CPU time on 100 machines at the same time.

This thesis has described 14 different stimuli generators which have been applied to 7 different circuits through various experiments. Each experiment have usually been repeated several times (usually 10) in order to cover some statistical variations. This adds up to a total of 9800 different simulation jobs for each experiment. These simulations take quite some time. As an example, consider the experiment (only one trial) of applying 10M SIC patterns to all circuits with the generator GT1. The running times are listed in Table 8.9.

Since these simulations take quite some time, 27 hours for 9800 jobs to be precise, it was tempting to find a way to speed up the simulations. Fortunately, since all these simulation jobs can run independent of each other, the jobs can be distributed to several computers. The simulator and the ATPG described in this thesis have the option of operating in distributed mode and download jobs

Table 8.9: Running time for simulating 10M test patterns on an Intel E6600@3GHz

Circuit	Sim. time
c880	34 s
c1355	108 s
c1908	138 s
c2670	39 s
c3540	211 s
c5315	82 s
c7552	105 s
Sum	12 m

from a central server. During the summer of 2005, experiments were conducted where as many as 100 computers participated in simultaneous simulation, reducing the simulation time with a factor equal to the number of computers.

Chapter 9

Conclusion

This thesis has presented the implementation of a simulator and an ATPG for path delay faults. In Chapter 8 these tools were used in order to evaluate the performance of the test vector generators presented in Chapter 7. In this Chapter we will look closer at what we have learned from these experiments, and present some thoughts on future research.

9.1 Discussion

This thesis has described 14 pseudo-random generators. Ten of these are weighted generators, and the rest generate uniformly distributed pseudo-random stimuli. Each experiment described in goes through one or more phases:

- In the first phase, the ATPG is used in order to find the K-longest testable paths. The corresponding path numbers are then saved to a file together with the corresponding test vector for repeated use later on. The paths can then be used as the target fault list during simulation. Experiments that consider all possible faults skip this phase.

- In the second phase, weights are generated for the weighted pseudo-random generators. These weights are stored for repeated use later on. This phase is skipped for experiments where the generator is unweighted.

- In the third phase, the actual fault simulation takes place. In all experiments 10M single-input-change test patterns were applied and repeated ten times for each generator and circuit in order to cover some statistical

variations. Only non-robust faults (including robust faults) were considered.

Two groups of pseudo-random generators have been evaluated. The first group, GA, consists of accumulator based pseudo-random generators, and includes the following generators: GA1, GA2, GA3, GA4, GA5, GA6 and GAU. Of these GA1 and GAU are unweighted generators, the rest are weighted. The second group, GT, consists of mersenne-twister based pseudo-random generators, and includes the following generators: GT1, GT2, GT3, GT4, GT5, GT6 and GTU. Of these are GT1 and GTU unweighted generators, the rest are weighted.

The result has shown that the GT group of pseudo-random patterns gives marginally better results than the GA group. Since GA generators are much less computationally intensive, GA generators are reccommended over GT generators in practical applications. Experiments have also been conducted in order to evaluate the benefit of weighted stimuli compared to unweighted stimuli. The results show that test time can be reduced with a factor of up to 15 for the circuits in the ISCAS'85 benchmark suite.

The results from the experiments will now be discussed in detail in the following order: EX3, EX4, EX1, EX2, EX5, and EX6.

In experiment EX3 the performance of the five generators GA1, GA2, GA3, GA4 and GA5 in the GA group were evaluated. The performance of GA3, GA4 and GA5 did not differ by much, but the results pointed in favour of GA3, which detects most path-delay faults for all but one benchmark. GA3, GT4 and GT5 are all weighted pseudo-random generators with weights based on the counting schemes described in Chapter 7. The difference is that GT4 and GT5 partition the set of all path-delay faults into disjunct subsets, one for each output, and creates one weight set corresponding to each subset of faults. It is not easy to understand why one generator is better than the other. One way to go about in order to at least get a better understanding of the problem is to look at the topology of the circuit under test. We have not performed such topology analysis, but it is relatively easy to devise examples of circuit topologies that would favor one generator over the other.

One such example is shown in Figure 9.1. The circuit to the left consists of two small cones with one AND gate with many inputs in each cone. Each cone has separate inputs. If weights were generated for this circuit using GA3, the result would be a weight set that will make it more likely to generate '1' than '0' on all inputs of all cones, since that will increase the probability of generating a good basis vector for the path-delay faults in the circuit. GA4 (and GA5), on

the other hand, will end up with two weight sets, one for each output cone. The weight set targeting the subset of faults associated with each cone will result in weights with a higher probability of setting the input of that cone to '1' than to '0', the other inputs would get uniform weights. In such situations it would be preferred to use GA3 since weights in GT4 (and GT5) would only target one cone at a time when it is possible to target both at the same time without penalty.

The circuit to the right in Figure 9.1 shows a situation where it would be meaningful to employ GT4 (or GT5). In this circuit the cones are driven by the same inputs. One of the cones is an AND gate with many inputs and the other is a OR gate with the same inputs. In order to achieve good coverage of the path delay faults in this circuit it is necessary with some patterns that have high probability of '1', and equally many patterns that that have high probability for all inputs to be set to '0'. This can be achieved using generators such as GT4 (and GT5), but not GT3.

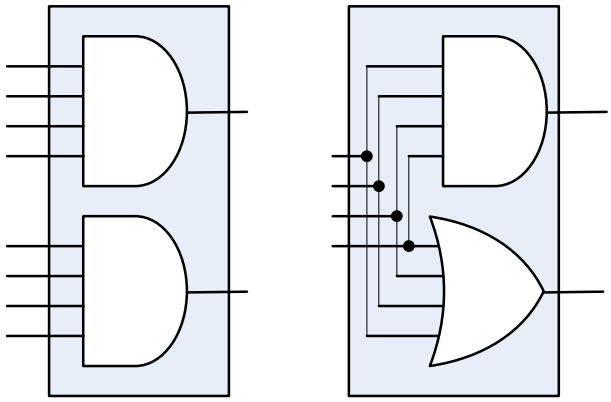


Figure 9.1: Two circuits with two output cones driven by different or the same inputs.

In experiment EX4, the performance of the five generators GT1, GT2, GT3, GT4 and GT5 in the GT group was evaluated. The performance of GT3, GT4 and GT5 did not differ by much, but the results pointed in favour of GT3, which detects most path-delay faults for all but one benchmark. The results for these generators were slightly better that for the corresponding accumulator based test vector generators from EX3, but is not recommended for use due to the high computational complexity. The reason why the Mersenne twister has been included in the experiment, is because we needed a pseudo-random source with

known good properties, in order to verify that the accumulator based generators had adequate performance as pseudo-random sources. Using the GT generators as benchmarks, we can conclude from EX3 and EX4 that the unweighted accumulator based generator GA1 has much poorer performance than GT1. This indicates that a single accumulator is a relative poor source for pseudo-random stimuli. But by putting three accumulators inside each of the generators GA2, GA3, GA4, GA5, GA6, GAU, and use these to generate weights as explained in Section 7.8.3, the performance becomes comparable to the Mersenne twister.

In experiment EX1, the K(20000) longest non-robust testable paths for each circuit were extracted and saved to file. These paths formed the fault lists that were used as target in EX5. The deterministic test set returned from the ATPG was also used in order to generate weights for GA6 and GT6. The results from this experiments showed that all circuits except c880 contained more than 20000 non-robust testable paths. c880 has only 16652 such paths. Allthough the ATPG only returned non-robust tests of these paths, some of the tested paths might also be robust testable. Thus, in these experiments the set of non-robust faults has been considered a superset of the set of robust faults. Both views regarding the set of robust and non-robust faults as disjoint sets, and non-robust faults a superset of the set of robust faults, are common in the literature.

In experiment EX2, it was determined how many paths of different length are detected by unweighted pseudo-random stimuli. From Table 8.3 one can see that the expected length of the detected faults increases when more test vectors have been applied to the circuit under test. This is a well-known fact [LR87] and has been confirmed through this experiment. The experiment was conducted running the simulator in a mode where it considered only a subset of the set of non-robust paths of which higher quality non-robust tests can be found. Thus, the total number of detected faults will be somwhat lower compared to other experiments in this chapter, when considering non-robust tests of any quality. (A higher quality non-robust test has lower probability of being invalidated by exessive delays on other paths than the target path).

The reason why the ATPG was constructed, was that we wanted to be able to find the fault-coverage of the K-longest testable path delay faults. This lead to experiment EX5, which combines the forces of the ATPG and the simulator. The purpose of the experiment was to find out if proper weighting of pseudo-random stimuli, based on K=20000 deterministic test patterns for path-delay faults, would yield more efficient path delay tests than using uniformly distributed patterns. According to the results listed in Table 8.8, the answer must be yes. It shows that it is possible to achieve a reduction in test time of up to

a factor of 12 for the GA class of generators and up to 15 for the GT class of generators, for circuits in the iscas'85 benchmark suite.

Weighted pseudo-random stimuli yield higher coverage than unweighted pseudo-random stimuli for all circuits. Except for circuit c7552, where all methods perform equally well. The reason why all generators perform equally well for circuit c7552 is that all weights were very close to 0.5 for all inputs, thus the weighted generators "degenerate" to ordinary unweighted pseudo-random stimuli.

This experiment also introduced the generators GA6 and GT6 which has the same structure as GA3 and GT3, but the weights are instead based on the test vectors returned from the ATPG. It is not easy to predict in advance which of the two weighting schemes that would yield the best result so it is recommended to try out both methods and choose the best one for the circuit at hand.

Experiments such as these require a lot of CPU time. In order to reduce the simulation time by as much as possible, a umbrella system was devised, where the simulator was encapsulated in a screen saver. This screensaver was distributed on the campus to all the computers in several computer rooms, in order to utilize free CPU time when the students were sleeping. If the computers are equal it is possible to achieve a speedup equal to the number of computers. As explained in EX6, around 100 computers have been running path-delay simulations simultaneously.

Finally, we would like to mention that the experiments that have been conducted in this thesis would not have been possible without a reliable system for keeping track of all conducted experiments and results. A computer program was written for this purpose, and with this it is possible to browse through the different experiments, plotting graphs and extracting important information.

9.2 Future research

Great effort has been invested in creating the simulator and the ATPG for path-delay faults. Future research would involve using these tools in order to create better generators. The more information known about a particular CUT, the easier it is to find a way to create an efficient test program for that circuit. An ultimate goal could be to devise a program that among other things utilizes the simulator and the ATPG to extract as much relevant information about the CUT as possible. This information can then be used to develop a good test program or test generator for that particular circuit. Substantial research must be done in order to get there, including identifying and learning what kind of information

that is relevant in this context. Analysis of the topology of the circuits would probably yield valuable information. The following will sketch one possible first step towards this ambitious goal:

The generators presented in this thesis do all generate basis vectors. Each input are then toggled twice, in order to create single-input-change patterns with rising and falling transitions on the inputs of the paths. Each basis pattern will thus provide twice as many single input change patterns as there are inputs for each basis vector. This raises the following questions: Are all these single-input-change patterns equally valuable? Can it be that all path-delay faults emanating from a particular input pin is detected very early, so that all effort invested in subsequent SIC patterns on that input is wasted? Is it better to use time spent on SIC patterns of a particular input pin on another input pin instead?

A simple experiment that may answer these questions is to use one of the 14 generators presented in this thesis, simulate 10M SIC patterns and record the fault coverage associated with each input pin. If some of the input pins reach 100% fault coverage early on, all subsequent SIC patterns applied to that input are wasted, and should be applied elsewhere.

For the field of path-delay testing as such, it is necessary to develop better tools in order to handle the exponential number of paths found in some circuits such as benchmark c6288. In particular, we need better pruning algorithms for the ATPG, and more efficient fault grading algorithms in the simulator.

Appendix A

Directed Acyclic Graphs

Combinational circuits can be modelled as directed acyclic graphs (DAGs). The GFault path-delay fault simulator is built around a DAG model of the circuit under test. In order to make it easier to follow the discussion of the GFault path-delay fault simulator in the subsequent chapters, this chapter first provides a short introduction to basic graph theory. A more throughout description can be found in textbooks such as [Sed03] and [CLRS01].

None of the basic data structures and algorithms presented in this chapter is new and invented by the author. A C++ graph library was implemented using data structures and algorithms described in this chapter. An overview of the features of the library is given at the end of this chapter.

A.1 Introduction

This section provides a short introduction to graphs and relevant graph terminology necessary in order to follow the discussion in the subsequent sections and chapters.

Definition 12 (Graph) *A graph $G = (V, E)$ consists of a set of vertices, V, and edges, E, connecting pairs of vertices.*

The names 0 through $|V| - 1$ are used for the vertices in a graph with $|V|$ vertices throughout this text. Graph algorithms usually have to associate information with each vertex. This way of naming vertices makes it possible to store and retrieve such information quickly (time complexity $O(1)$) through vertex indexed arrays. Edges are labelled in a similar fashion for the same reason.

Graphs can be divided into two main categories: directed graphs and undirected graphs. In a directed graph all edges have a direction associated with them. Figure A.1 shows an example of undirected and directed graphs. A formal definition, as it appears in [Sed03], is given in Definition 13.

Definition 13 (Directed graph) *A digraph (or directed graph) is a set of vertices plus a set of directed edges that connect ordered pairs of vertices. We say that an edge goes from its first vertex to its second vertex.*

[I have used start vertex and stop vertex instead of start vertex and stop vertex].

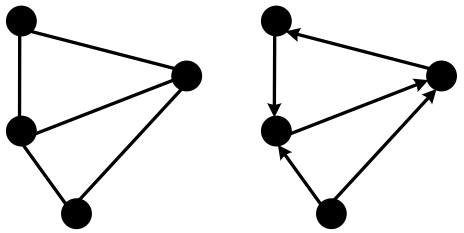


Figure A.1: Undirected and directed graphs

In the special case that a directed graph contains no cycles it is referred to as directed acyclic graph (DAG). A formal definition, as it appears in [Sedg03], is given in Definition 14.

Definition 14 (Directed acyclic graph) *A directed acyclic graph (DAG) is a digraph with no directed cycles.*

A path in a graph is a sequence of vertices in which each successive vertex (after the first) is adjacent to its predecessor in the path. In a simple path, the vertices and edges are distinct. A cycle is a path that is simple except that the first and final vertices are the same.

[Parallel edges (two edges connecting the same two vertices) make it impossible to uniquely identify a path by means of a sequence of vertices. Define path in terms of edges instead.]

Definition 15 (Directed path in digraph) *A directed path in a digraph is a list of vertices in which there is a digraph edge connecting each vertex in the list to its successor in the list. We say that a vertex t is reachable from a vertex s if there is a directed path from s to t.*

A vertex with no incoming edges is called a source, while a vertex with no outgoing edges is called a sink. A DAG has at least one source and one sink.

Sparse graphs are graphs where $|E| = O(|V|)$ or put in another way: A graph with relatively few edges is called a sparse graph.

Dense graphs are graphs where $|E| = T(|V|2)$ or put in another way: A graph with relatively many edges is called a dense graph.

A static graph representation is a graph representation which does not change once the representation is built. No edges or vertices can be added or removed from a static graph representation once it is built.

A dynamic graph representation is a graph representation which can change after the representation is built. Edges or vertices can be added and removed from a dynamic graph representation at any time.

A.2 Data structures for representation of graphs

There are many things to consider when selecting the underlying data structure for representing graphs. On must take into account:

- The different properties of the graphs that the data structure is intended used for. Whether the graph is sparse or dense, contains cycles or not, is directed or undirected, and whether it static or dynamic will play a role on the decision.
- Which operations and algorithms that must be supported efficiently. Typical operations include edge addition and removal, vertex addition and removal, breadth first search, depth first search, topological sorting and test for different properties.
- The properties of the computational platform(s) on which the graph library is intended to run. Memory and cache considerations may come into account.

The data structures used in different graph representations are often defined in terms of arrays, matrixes and lists.

An array is a one-dimensional container with a fixed size once the memory for the array is allocated. Element number i can be retrieved in constant time $O(1)$.

A matrix is a two-dimensional container with fixed dimensions once the memory for the matrix is allocated. Element (i, j) can be retrieved in constant time $O(1)$.

A linked list is an abstract data type (ADT) that consists of a chain of nodes. Each node has a pointer to the next node in the list. The last node contains a *NULL*-pointer. A particular element in a list with n elements can be found in time $O(n)$.

A list is an ADT often implemented as a linked list, but it may also be implemented using an array.

Figure A.2 shows a simple digraph with 4 vertices (labelled with numbers in the range [0,3]) and 5 edges. This graph will be used as an example in the next sections in order to illustrate the different graph representations.

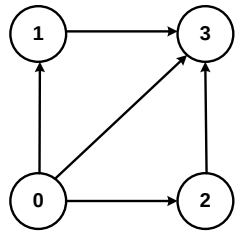


Figure A.2: A simple digraph with 4 vertices and 5 edges

A.3 List of edges

The simplest way to represent a graph is to use a list of edges. The data structure is very simple, and has modest memory requirements. Figure A.3 shows a list of edges representing the graph in Figure A.2.

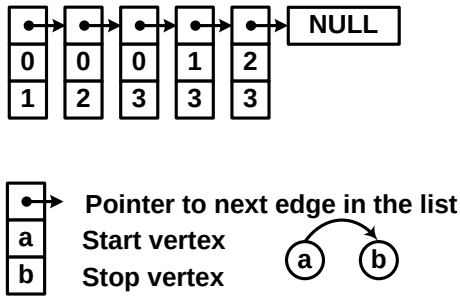


Figure A.3: List of edges representation

If the list is implemented as a single-linked list it will have the following basic properties:

- In order to find out if there exist an edge starting at node a and ending at node b, the whole list must be traversed in worst case. Thus this operation has a time complexity of $O(|E|)$.
- In order to find the edges starting at a given node, the whole list must be traversed in worst case. This operation has a time complexity of $O(|E|)$.
- Deletion of a particular edge has time complexity $O(|E|)$.
- Insertion of a new edge has time complexity $O(1)$ if the edges are unsorted and new edges can be appended to the end of the list. Insertion of a new edge has time complexity $O(|E|)$ if the edges are sorted and the new edge must be inserted at the correct position.

Due to the poor time complexity of important operations this graph representation is not much used. By breaking the large linked list of edges into an array of smaller linked lists a new representation is found called the adjacency list. This is presented in the next section.

A.4 Adjacency matrix

The adjacency matrix representation consists of a two-dimensional $|V|x|V|$ matrix, $AdjMatrix$. An edge connecting node a and b is represented by a nonzero entry in $AdjMatrix[a][b]$. An adjacency matrix representation of the graph in Figure A.2 is shown in Figure A.4.

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

Figure A.4: Adjacency matrix representation

The adjacency matrix has the following basic properties:

- In order to find out if there exist an edge starting at node a and ending at node b, one only have to check if $AdjMatrix[a][b]$ is nonzero. This operation has a time complexity of $O(1)$.
- In order to find the edges starting at a given node, one row in the matrix must be traversed. This operation has a time complexity of $T(|V|)$.
- Deletion of a particular edge has time complexity $O(1)$.
- Insertion of a new edge has time complexity $O(1)$. Multiple edges connecting two nodes is not supported without altering the data structure.

Since the size of the matrix is independent on the number of edges in the graph, it has a lower cost pr. edge for dense graphs than for sparse graphs, and is thus best suited for dense graphs.

A.5 Adjacency list

The adjacency list representation is probably the most used graph representation. It consists of an array, $AdjList$, of $|V|$ lists, one for each vertex in V. The lists are often implemented as linked list in order to support easy addition of new edges. An adjacency list representation of the graph in Figure A.2 is shown in Figure A.5.

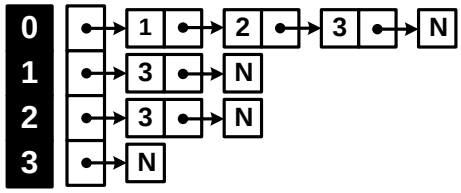


Figure A.5: Adjacency list representation

If the adjacency list is implemented as an array of single-linked list it will have the following basic properties:

- In order to find out if there exist an edge starting at node a and ending at node b, only the list pointed to by $AdjList[a]$ must be traversed. If there are at most one edge between any two vertices, this operation will have a worst case a time complexity of $O(|V|)$.

- In order to find the edges starting at a given node, it is enough to lookup the pointer to the corresponding list. This operation has a time complexity of $O(1)$.
- Deletion of a particular edge has time complexity $O(|V|)$.
- Insertion of a new edge has time complexity $O(1)$ if the edges are unsorted and new edges can be appended to the end of each list. Insertion of a new edge has time complexity $O(|V|)$ if the edges are sorted and the new edge must be inserted at the correct position in each list.

The adjacency list representation is suitable for sparse graphs. A netlist can be represented as a directed acyclic graph and the resulting graphs are often sparse. Netlist has also in practice limited fan-in and fan-out which in practice will reduce the time complexity from $O(|V|)$ to $O(1)$ for all of the above operations.

A.6 An Abstract Data Type (ADT) for static DAGs

Netlist do usually not change during simulation. They are static DAGs. Static DAGs does not need to support edge and node addition/removal once the graph is built. If this property is taken advantage of, it can lead to a data structure that is faster and less complex than what is necessary for dynamic graphs. An efficient data structure for static graphs can be found by using arrays instead of linked lists as shown in Figure A.6.

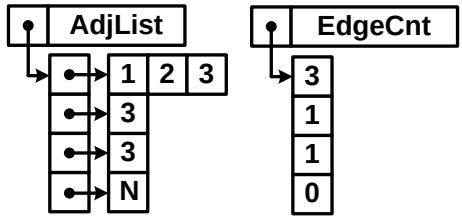


Figure A.6: Adjacency list representation using arrays instead of linked lists

In addition to the adjacency list an additional array (*EdgeCnt*) is necessary in order to store the length of the arrays in the adjacency list.

If a graph contains $|V|$ vertices, each vertex is given a unique integer label in the range $[0, |V| - 1]$, as shown in the graph in Figure A.2. By using vertex

indexed arrays, it is easy to add data/parameters to the vertices. The adjacency list representation uses this to add a list of edges to each vertex.

Sometimes it is necessary to associate information to each edge as well. If a graph contains $|E|$ edges, each edge can be given a unique integer label in the range $[0, |E| - 1]$, as shown in Figure A.7. By using edge indexed arrays, data/parameters can be associated with each edge in the same manner as with the vertices.

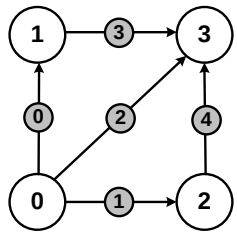


Figure A.7: A simple digraph with four vertices and five edges

Figure A.8 shows a representation of the graph in Figure A.7. This representation differs from the one in Figure A.6 by the three new arrays $AdjList2EdgeId$, $EdgeId2Source$ and $EdgeId2Position$. The purpose of these new arrays is to add a mapping between the edges and their position in the adjacency list and vice versa.

$AdjList2EdgeId$ is an array of arrays that maps an edge in the $AdjList$ to the edges unique edge label. $AdjList[0]$ contains a pointer to a list of edges with start vertex 0. This list contains three edges as indicated by $EdgeCnt[0] = 3$. The last edge in this list has the stop vertex $AdjList[0][2] = 3$. This edge also has a unique edge ID which is $AdjList2EdgeId[0][2] = 2$.

$EdgeId2Source$ is an array that maps an edge ID to the corresponding start vertex. $Edge2Source[2] = 0$ indicates that edge 2 starts at vertex 0.

$EdgeId2Position$ is an array that maps an edge ID to the corresponding position in the corresponding list associated with is start vertex. Together with $EdgeId2Position[2] = 2$ indicates that the destination vertex for edge 2 is found at position 2 in the array pointed to by $AdjList[EdgeId2Source[2]]$.

A.7 Vertex object design

If each vertex have ten parameters associated with them, there are two common ways to organize these parameters:

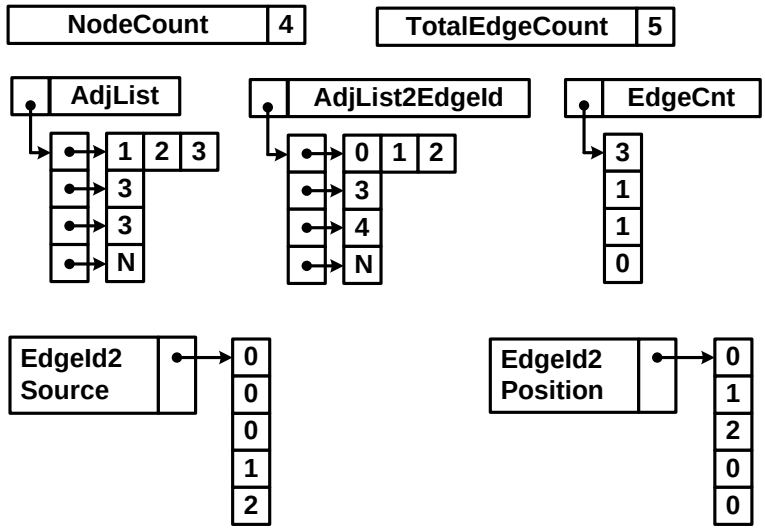


Figure A.8: An ADT for static graphs

- Create one vertex indexed array of objects/structs. The vertex object contains the ten parameters.

- Create ten vertex indexed arrays. One for each parameter.

The first solution will group the different parameters associated with a vertex together in memory. This might lead to fewer cache misses if the algorithm uses several of the parameters when visiting the vertices. On the other hand, if the vertex object has to support many different algorithms, this may result in a large and complex object. When new algorithms are added which uses new parameters, the vertex object must be updated to include the new parameters.

The second solution will cause parameters associated with a vertex to be spread in memory. This might lead to more cache misses compared to if the parameters associated with a vertex were grouped. On the other hand, the vertex indexed arrays can be implemented as a part of the algorithm. This makes it easier to add new algorithms, since the addition of a new algorithm does not interfere with existing algorithms.

Both approaches have been explored, but the latter gave code that was easier to maintain when new features was added.

A.8 Graph algorithms for DAGs

This section presents some algorithms that are implemented on top of the graph representation presented in Figure A.8.

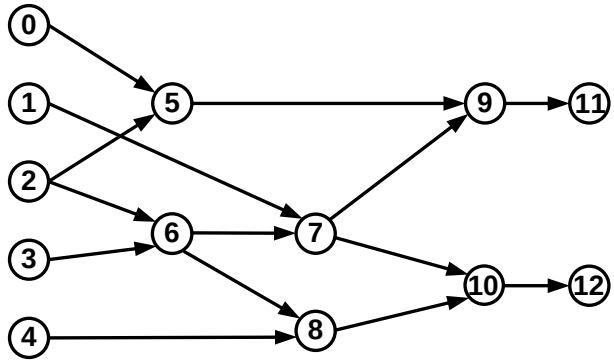
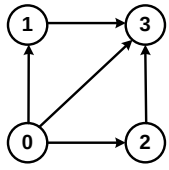


Figure A.9: A sample digraph

A.9 Degree computation

In some applications it is necessary to know the in-degree and out-degree of the vertices in the graph. The in-degree of a vertex in a digraph is the number of incoming edges for that vertex. The out-degree of a vertex in a digraph is the number of outgoing edges for that vertex. Figure A.10 shows a small graph with in-degree and out-degree listed for each vertex.



V	IN DEG.	OUT DEG.
0	0	3
1	1	1
2	1	1
3	3	0

Figure A.10: A small graph with in- and out-degree listed for each vertex

The in-degree and out-degree can for instance be used to find the sources and the sinks in a DAG. This particular graph has one source (vertex 0) and one sink (vertex 3).

A.10 Reverse of a graph

The reverse of a digraph is a graph that can be derived from the original graph by changing the direction of all edges in the original graph. Figure A.11 shows a digraph and its reverse. Storing a graph and its reverse provides an efficient way to find both all outgoing edges and all incoming edges of a given vertex. If the graph represents a gate level netlist this will provide a way to find both all fan-ins and all fan-outs of a given gate.

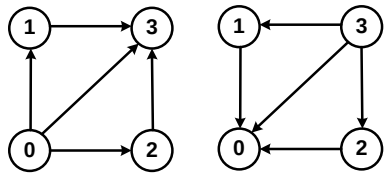
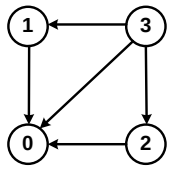


Figure A.11: A digraph and its reverse

A.11 Topological sort

The goal of topological sorting is to be able to process the vertices of a DAG such that every vertex is processed before all the vertices to which it points.

A topological sort of a directed acyclic graph G is a partial ordering of the vertices such that $i < j$ for every edge (ai, bj). Here a and b is the name of the start and stop vertices, and i and j their place/index in the linear ordering. Figure A.12 shows a small DAG. If the following order is used: $[3, 2, 1, 0]$, we can see that this ordering yields a topological sort of the graph in Figure A.12.



(a,b)	(a_i,b_j)	i < j
(3,0)	(3_0,0_3)	0 < 3
(3,1)	(3_0,1_2)	0 < 2
(3,2)	(3_0,2_1)	0 < 1
(2,0)	(2_1,0_3)	1 < 3
(1,0)	(1_2,0_3)	2 < 3

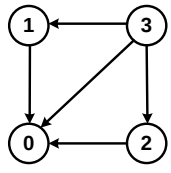
Figure A.12: Topological sorting of DAG

One intuitive algorithm for topological sorting is based on repeated source removal. An outline of the algorithm is given in Algorithm 5.

Algorithm 5 (Algorithm for topological sort by repeated source removal)

```
TopoSort(G){
  /*INPUT  : G, a directed acyclic graph          */
  /*OUTPUT: order, a list of vertices in top. order */
  order = [];
  while(vertices left in G){
    remove a source s and all its outgoing edges;
    order.append(s);
  }
  return order;
}
```

If the algorithm is applied to the graph in Figure A.12, the following steps will be taken (see Figure A.13). First vertex 3 is removed (STEP0), since this is the only source in the original graph. After vertex 3 is removed together with its outgoing edges, only three vertices remain. Two of these (vertex 1 and vertex 2) are sources. The algorithm proceeds by removing one of them, say vertex 2, and all of its edges (STEP1). Only two vertices remains and only one of them is a source. The algorithm thus removes vertex 1 (STEP2) and finally vertex 0 (STEP3). This yield the topological order $[3, 2, 1, 0]$ which is consistent with the order in Figure A.12.



(a,b)	(a_i,b_j)	i < j
(3,0)	$(3_0,0_3)$	0 < 3
(3,1)	$(3_0,1_2)$	0 < 2
(3,2)	$(3_0,2_1)$	0 < 1
(2,0)	$(2_1,0_3)$	1 < 3
(1,0)	$(1_2,0_3)$	2 < 3

Figure A.13: Topological sort by repeated source removal

A topological sort is a partial ordering of the directed acyclic graph, thus there may exist more than one correct topological sort of a given DAG. If STEP1 and STEP2 is switched in Figure A.12 another correct topological order $[3, 1, 2, 0]$ will be the result.

Topological sorting is the key ingredient in many graph algorithms, for instance when it comes to compute the number of paths in a graph.

A.12 Path counting

The number of paths (starting at any source vertex and ending at any sink) in a DAG can be found by on pass of the reverse topological sorted vertices in the DAG. An algorithm for doing path counting is outlined in Algorithm 6.

Algorithm 6 (Algorithm for path counting)

```
PathCount(G){
  /*INPUT    : G, a directed acyclic graph                                  */
  /*OUTPUT: pathcnt, the number of paths in G                               */
  cnt = [0,0,…,0];   /* Vertex indexed array for temporary results  */
  foreach(vertex a in G visited in reverse topological order){
    if (a is a sink){
      cnt[a] = 1;
    }
    foreach(outgoing edge b from a){
      cnt[a] += cnt[b];
    }
  }
  pathcnt = 0;
  foreach(source s in G){
    pathcnt += cnt[s];
  }
  return pathcnt;
}
```

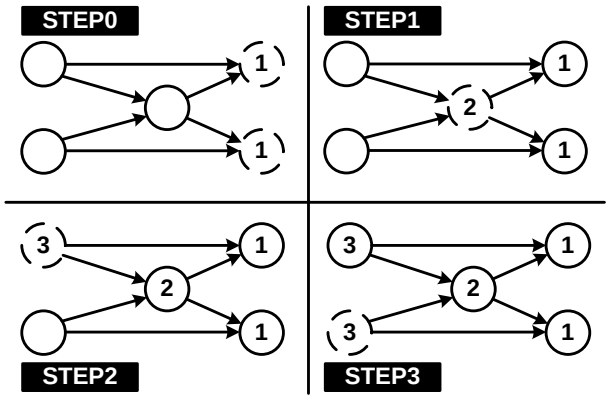


Figure A.14: Steps taken during path counting

If the algorithm is applied to the circuit in Figure A.14, the following will happen: The algorithm will visit each vertex at a time until all vertices are

visited. The vertices will be visited in reverse topological order. The two first vertices to be visited (STEP0) are recognized as sinks and their cnt variable given the value 1. In the next step (STEP1) the *cnt* variable for the vertex in the center is assigned the sum of the *cnt* variables for its fan-out vertices. The algorithm proceeds in the same manner by computing the *cnt* value for the two sources (STEP2 and STEP3). The *cnt* variable associated with a vertex is the number of paths starting at that vertex and ending at any sink. The total number of paths starting at any source and ending at any sink can thus be computed as the sum of the sinks *cnt* variables. The number of such paths is six in this example.

A.13 Longest path length extraction

The length of the longest path in a DAG can also be found by visiting the vertices in reverse topological order. An algorithm for computing the length of the longest path in a DAG is outlined in Algorithm 7. The program is identical in structure with the program in Algorithm 6.

Algorithm 7 (Algorithm for finding length of longest path in DAG)

```
LongestPath(G){
  /*INPUT    : G, a directed acyclic graph                          */
  /*OUTPUT: pathlen, the length of the longest path in G            */
  len = [0,0,…,0];   /* Vertex indexed array for temporary results */
  foreach(vertex a in G visited in reverse topological order){
    if (a is a sink){
      len[a] = 0;
    }
    foreach(outgoing edge b from a){
      len[a] = max(len[a], len[b]+1);
    }
  }
  pathlen = 0;
  foreach(source s in G){
    pathlen = max(pathlen, len[s]);
  }
  return pathlen;
}
```

A.14 Path length histogram extraction

The *LongestPath* algorithm from Algorithm 7. can find the length of the longest path in a DAG. There must be at least one path with length equal to the length of the longest path, but are there more paths with the same length? How many different lengths have the paths in the DAG and how many paths are there of each length? These are questions that can be answered by the program presented in this section.

This algorithm combines the path counting algorithm from Algorithm 6 with the algorithm for finding the length of the longest path in Algorithm 7. The result is an algorithm that counts the number of paths in the DAG with equal length.

Algorithm 8 (Path length histogram extraction)

```
PathLengthHistogram(G){
  /*INPUT   : G, a directed acyclic graph                          */
  /*OUTPUT: hist, Vertex indexed array of dictionaries.            */
  /*            hist[i] stores the number of paths of different    */
  /*            lengths starting at vertex i.                      */
  hist = [{},{},…,{}];    // Initialization of dictionaries
  foreach(vertex a in G visited in reverse topological order){
    if (a is a sink){
      hist[a][0] = 1;     // One path of length 0
    }
    foreach(outgoing edge b from a){
      foreach(pathlen and pathcnt in dictionary hist[b]){
        if(pathlen+1 is a new key in hist[a]){
          hist[a][pathlen+1] = 0;
        }
        hist[a][pathlen+1] += pathcnt;
      }
    }
  }
  return hist;
}
```

If the algorithm is applied to the circuit in Figure A.15, the following will happen: The algorithm will visit each vertex at a time until all vertices are visited. The vertices will be visited in reverse topological order. The two first vertices to be visited (STEP0) are recognized as sinks and their *hist* dictionary each initialized with one path of length zero. In the next step (STEP1) the *hist* dictionary for the vertex in the center is processed. The fan-out vertices *hist* dictionaries sums up to two paths of length zero. The *hist* dictionary for the

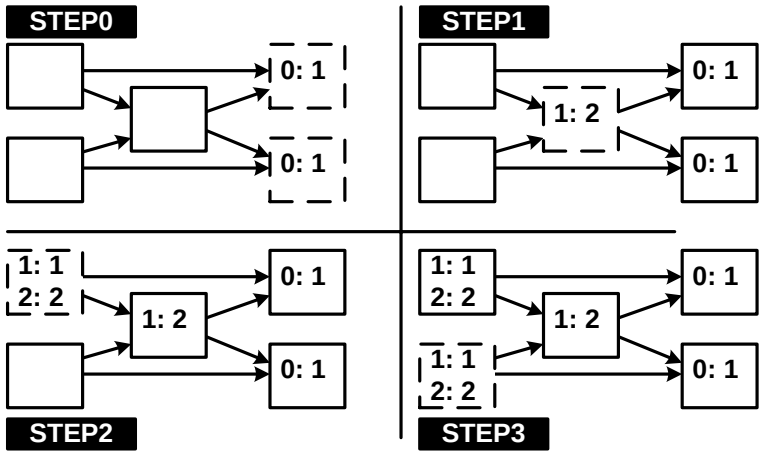


Figure A.15: Steps taken during path length histogram extraction

center node will thus contain two paths of length one. The algorithm proceeds in the same manner by computing the *hist* dictionaries for the two sources (STEP2 and STEP3).

A.15 Transitive closure

The transitive closure of a digraph is a digraph with the same vertices but with an edge from s to t in the transitive closure if and only if there is a directed path from s to t in the given digraph [Sed03]

The transitive closure is also an operation that can be computed by traversing the vertices in reverse topological order. An outline of the program is given in

Algorithm 9 (Transitive closure)

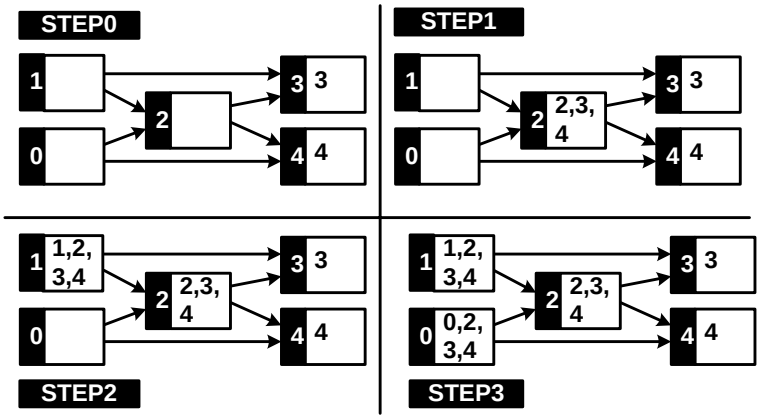


Figure A.16: Steps taken during computation of transitive closure

```
TransitiveClosure(G){
  /*INPUT    : G, a directed acyclic graph                          */
  /*OUTPUT: T, the transitive closure of G                          */
  foreach(vertex a in G visited in reverse topological order){
    T.AdjList[a] = [a];    // Initialize with edge to itself
    foreach(outgoing edge b from a){
      T.AdjList[a] = union(AdjList[a], T.AdjList[b]);
    }
  }
  return T;
}
```

A.16 Cone graph extraction

Algorithm 10 (Cone graph extraction)

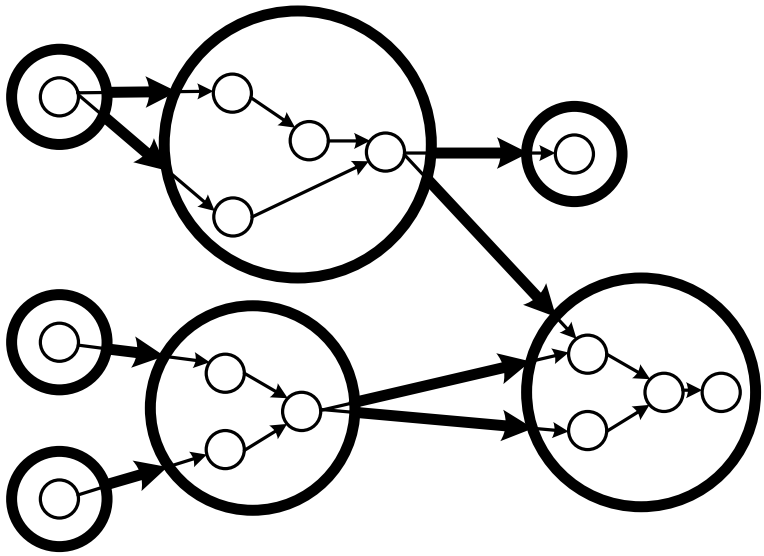


Figure A.17: A graph with extracted cones

```
ConeGraphExtractor(G){
  /*INPUT    : G, a directed acyclic graph                    */
  /*OUTPUT: C, the cone graph of G                            */
  C2G = {};    // Find vertex in G corresponding to vertex in C
  G2C = {};    // Find vertex in C corresponding to vertex in G
  foreach(vertex ga in G){
    if (ga is source, sink or stem){
      Create a corresponding vertex ca in C;
      Add mappings: C2G[ca]=ga, G2C[ga]=ca;
    }
  }

  frontier = [];  // A queue
  foreach(vertex ca in C){
    ga = C2G[ca];
    frontier.append(G.AdjList[ga]);
    while(frontier is not empty){
      gb = frontier.pop();
      cb = G2C[gb];
      if(gb is source, sink or stem){
        C.AddEdge(ca,cb);
      } else {
        frontier.append()
      }
    }
  }

  return C;
}
```

Bibliography

[BA84] D. Brahme and J. A. Abraham. Functional testing of microprocessors. *IEEE Transactions on Computers*, C-33(6):475–485, 1984.

[BA95] S. Bose and V. D. Agrawal. Sequential logic path delay test generation by symbolic analysis. In *Proc. of the Fourth Asian Test Symp.*, pages 353–359, 1995.

[BA02] Michael L. Bushnell and Vishwani D. Agrawal. *Essentials of electronic testing for digital, memory, and mixed signal VLSI circuits elektronisk ressurs*. Frontiers in electronic testing. Kluwer Academic, New York, 2002.

[BAA93] S. Bose, P. Agrawal, and V. D. Agrawal. Path delay fault simulation of sequential circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 1(4):453–461, 1993.

[BAA98] S. Bose, P. Agrawal, and V. D. Agrawal. Deriving logic systems for path delay test generation. *IEEE Transactions on Computers*, 47(8):829–846, 1998.

[Ber93] M. Bershteyn. Calculation of multiple sets of weights for weighted random testing. In *Proc. of the International Test Conf.*, pages 1031–1040, 1993.

[BH82] P. H. Bardell and McAnney W. H. Self-testing of multichip logic modules. In *Proc. of the International Test Conf.*, pages 200–204, 1982.

[BVMCDM90] J. Benkoski, E. Vanden Meersch, L. J. M. Claesen, and H. De Man. Timing verification using statically sensitizable

paths. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(10):10723–10784, 1990.

[CAB92] T. J. Chakraborty, V. D. Agrawal, and M. L. Bushnell. Delay fault models and test generation for random logic sequential circuits. In *Proc. of the 29th Design Automation Conf.*, pages 165–172, 1992.

[CAR93] S. T. Chakradhar, V. D. Agrawal, and S. G. Rothweiler. A transitive closure algorithm for test-generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(7):1015–1028, 1993.

[CC93] K. T. Cheng and H. C. Chen. Delay testing for non-robust untestable circuits. In *Proc. of the International Test Conf.*, pages 954–961, 1993.

[CC96] K. T. Cheng and H. C. Chen. Classification and identification of nonrobust untestable path delay faults. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):845–853, 1996.

[CDK93] K. T. Cheng, S. Devadas, and K. Keutzer. Delay-fault test generation and synthesis for testability under a standard scan design methodology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1217–1231, 1993.

[Che88] C. L. Chen. Exhaustive test pattern generation using cyclic codes. *IEEE Transactions on Computers*, 37(2):225–228, 1988.

[CKC96] K. T. T. Cheng, A. Krstic, and H. C. Chen. Generation of high quality tests for robustly untestable path delay faults. *IEEE Transactions on Computers*, 45(12):1379–1392, 1996.

[CKMZ83] A. K. Chandra, L. T. Kou, G. Markowsky, and S. Zaks. On sets of boolean n-vectors with all k-projections surjective. *Acta Informatica*, 20(1):103–111, 1983.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, Mass., 2nd edition, 2001.

[Dav84] R. David. Signature analysis of multi-output circuits. In *Proc. of the International Symp. on Fault-Tolerant Computing*, pages 366–371, 1984.

[DS91] B. I. Dervisoglu and G. E. Stong. Design for testability using scanpath techniques for path-delay test and measurement. In *Proc. of the International Test Conf.*, page 365, 1991.

[DT01] J. Deodhar and S. Tragoudas. Color counting and its application to path delay fault coverage. In *Proc. of the International Symp. on Quality Electronic Design*, pages 378–383. IEEE Computer Society, 2001.

[DW98] R. Dorsch and H. J. Wunderlich. Accumulator based deterministic bist. In *Proc. of the International Test Conf.*, pages 412–421, 1998.

[Eve79] Shimon Even. *Graph algorithms*. Computer software engineering series. Computer Science Press, Potomac, Md., 1979.

[FS83] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, C-32(12):1137–1144, 1983.

[GA05a] O. Gjermundnes and E. J. Aas. Design of a path delay fault simulator for evaluation of abist generated stimuli. In *Proc. of the Conf. on PhD Research in Microelectronics and Electronics*, volume 2, pages 107–110, 2005.

[GA05b] O. Gjermundnes and E. J. Aas. Efficient stimuli generators for detection of path delay faults. In *Proc. of the 48th Midwest Symp. on Circuits and Systems*, pages 1709–1712, 2005.

[GA05c] O. Gjermundnes and E. J. Aas. Remote path delay fault simulation. In *Proc. of the 8th Euromicro Conf. on Digital System Design*, pages 428–434, 2005.

[GBA95] M. A. Gharaybeh, M. L. Bushnell, and V. D. Agrawal. Classification and test generation for path-delay faults using single stuck-fault tests. In *Proc. of the International Test Conf.*, pages 139–148, 1995.

[GBA96] M. A. Gharaybeh, M. L. Bushnell, and V. D. Agrawal. An exact non-enumerative fault simulator for path-delay faults. In *Proc. of the International Test Conf.*, pages 276–285, 1996.

[GF02] E. Gizdarski and H. Fujiwara. Spirit: A highly robust combinational test generation algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1446–1458, 2002.

[GRT96] S. Gupta, J. Rajski, and J. Tyszer. Arithmetic additive generators of pseudo-exhaustive test patterns. *IEEE Transactions on Computers*, 45(8):939–949, 1996.

[HG96] Y.-C. Hsu and S. K. Gupta. A simulator for at-speed robust testing of path delay faults in combinational circuits. *IEEE Transactions on Computers*, 45(11):1312–1318, 1996.

[HPA96a] K. Heragu, J. H. Patel, and V. Agrawal. Sigma: A simulator for segment delay faults. In *Proc. of the International Conf. on Computer-Aided Design*, pages 502–508, 1996.

[HPA96b] K. Heragu, J. H. Patel, and V. D. Agrawal. Segment delay faults: a new fault model. In *Proc. of the 14th VLSI Test Symp.*, pages 32–39, 1996.

[HPA97] K. Heragu, J. H. Patel, and V. D. Agrawal. Fast identification of untestable delay faults using implications. In *Proc. of the International Conf. on Computer-Aided Design*, pages 642–647, 1997.

[IC02] M. K. Iyer and K.-T. Cheng. Software-based weighted random testing for ip cores in bus-based programmable socs. In *Proc. of the 20th VLSI Test Symp.*, pages 139–144, 2002.

[IRS88a] V. S. Iyengar, B. K. Rosen, and I. Spillinger. Delay test generation. i. concepts and coverage metrics. In *Proc. of the International Test Conf.*, pages 857–866, 1988.

[IRS88b] V. S. Iyengar, B. K. Rosen, and I. Spillinger. Delay test generation. ii. algebra and algorithms. In *Proc. of the International Test Conf.*, pages 867–876, 1988.

[IRW90] V. S. Iyengar, B. K. Rosen, and J. A. Waicukauski. On computing the sizes of detected delay faults. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(3):299–312, 1990.

[ITR05] International technology roadmap for semiconductors, 2005.

[Kap95] B. Kapoor. An efficient method for computing exact path delay fault coverage. In *Proc. of the European Design and Test Conf.*, pages 516–520, 1995.

[KC98] A. Krstic and K.-T. Cheng. *Delay Fault Testing for VLSI Circuits*. Kluwer Academic Publishers, Boston, 1998.

[KCC96] A. Krstic, K.-T. Cheng, and S. T. Chakradhar. Identification and test generation for primitive faults. In *Proc. of the International Test Conf.*, pages 423–432, 1996.

[KG05] F. Kocan and M. H. Gunes. On the zbdd-based nonenumerative path delay fault coverage calculation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(7):1137–1143, 2005.

[KGT04] F. Kocan, M. Gunes, and M. A. Thornton. Static variable ordering in zbdds for path delay fault coverage calculation. In *Proc. of the 47th IEEE International Midwest Symposium on circuits and Systems.*, 2004.

[KLK01] H.-S. Kim, J.-k. Lee, and S. Kang. A new multiple weight set calculation algorithm. In *Proc. of the International Test Conf.*, pages 878–884, 2001.

[KM94] W. Ke and P. R. Menon. Delay-verifiability of combinational circuits based on primitive faults. In *Proc. of the International Conf. on Computer Design*, pages 86–90, 1994.

[KM95] W. Ke and P. R. Menon. Synthesis of delay-verifiable combinational circuits. *IEEE Transactions on Computers*, 44(2):213–222, 1995.

[KP94] W. Kunz and D. K. Pradhan. Recursive learning - a new implication technique for efficient solutions to cad problems

- test, verification, and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(9):1143–1158, 1994.

[KPSW94] R. Kapur, S. Patil, T. J. Snethen, and T. W. Williams. Design of an efficient weighted random pattern generation system. In *Proc. of the International Test Conf.*, pages 491–500, 1994.

[KPSW96] R. Kapur, S. Patil, T. J. Snethen, and T. W. Williams. A weighted random pattern test generation system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):1020–1025, 1996.

[KT93] Cheng Kwang-Ting. Transition fault testing for sequential circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(12):1971–1983, 1993.

[KT99] D. Kagaris and S. Tragoudas. On the design of optimal counter-based schemes for test set embedding. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(2):219–230, 1999.

[KT02] D. Kagaris and S. Tragoudas. On the nonenumerative path delay fault simulation problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(9):1095–1101, 2002.

[KTM96] D. Kagaris, S. Tragoudas, and A. Majumdar. On the use of counters for reproducing deterministic test sets. *IEEE Transactions on Computers*, 45(12):1405–1419, 1996.

[LMB97] Z. C. Li, Y. Min, and R. K. Brayton. Efficient identification of non-robustly untestable path delay faults. In *Proc. of the International Test Conf.*, pages 992–997, 1997.

[LPRWT05] Lai Liyang, J. H. Patel, T. Rinderknecht, and Cheng Wu-Teng. Hardware efficient lbist with complementary weights. In *Proc. of the International Conf. on Computer Design*, pages 479–481, 2005.

[LR87] C. J. Lin and S. M. Reddy. On delay fault testing in logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):694–703, 1987.

[LSBSV95] W. K. Lam, A. Saldanha, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Delay fault coverage, test set size, and performance trade-offs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(1):32–44, 1995.

[Maj96] A. Majumdar. On evaluating and optimizing weights for weighted random pattern testing. *IEEE Transactions on Computers*, 45(8):904–916, 1996.

[MAND90] F. Muradali, V. K. Agarwal, and B. Nadeau-Dostie. A new procedure for weighted random built-in self-test. In *Proc. of the International Test Conf.*, pages 660–669, 1990.

[MB81] E. J. Mccluskey and S. Bozorguinesbat. Design for autonomous test. *IEEE Transactions on Circuits and Systems*, 28(11):1070–1079, 1981.

[Mcc84] E. J. Mccluskey. Verification testing - a pseudoexhaustive test technique. *IEEE Transactions on Computers*, 33(6):541–546, 1984.

[Min93] S.-I. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proc. of the 30th international conf. on Design automation*, Dallas, Texas, United States, 1993. ACM Press.

[MJPA96] A. K. Majhi, J. Jacob, L. M. Patnaik, and V. D. Agrawal. On test coverage of path delay faults. In *Proc. of the Ninth International Conf. on VLSI Design*, pages 418–421, 1996.

[MN98] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, pages 114–117, April 19 1965.

[NK97] D. J. Neebel and C. R. Kime. Cellular automata for weighted random pattern generation. *IEEE Transactions on Computers*, 46(11):1219–1229, 1997.

[PABS98] C. G. Parodi, V. D. Agrawal, M. L. Bushnell, and Wu Shianling. A non-enumerative path delay fault simulator for sequential circuits. In *Proc. of the International Test Conf.*, pages 934–943, 1998.

[PMT03] S. Padmanaban, M. K. Michael, and S. Tragoudas. Exact path delay fault coverage with fundamental zbdd operations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(3):305–316, 2003.

[PN88] V. Pitchumani and V. Narayanan. Weighted random pattern testing based on test counts. In *Proc. of the International Symp. on Circuits and Systems*, pages 433–436 vol.1, 1988.

[PR91] S. Pateras and J. Rajski. Cube-contained random patterns and their application to the complete testing of synthesized multile. In *Proc. of the International Test Conf.*, page 473, 1991.

[PR93] I. Pomeranz and S. M. Reddy. 3-weight pseudo-random test generation based on a deterministic test set for combinational and sequential circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(7):1050–1058, 1993.

[PS92] S. Patil and J. Savir. Skewed-load transition test: Part ii, coverage. In *Proc. of the International Test Conf.*, page 714, 1992.

[PT04a] S. Padmanaban and S. Tragoudas. A critical path selection method for delay testing. In *Proc. of the International Test Conf.*, pages 232–241, 2004.

[PT04b] S. Padmanaban and S. Tragoudas. Using bdds and zbdds for efficient identification of testable path delay faults. In *Proc. of the Conf. on Design, Automation and Test in Europe*, volume 1, pages 50–55 Vol.1, 2004.

[PT05] S. Padmanaban and S. Tragoudas. Efficient identification of (critical) testable path delay faults using decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1):77–87, 2005.

[Pug90] W. Pugh. Skip lists - a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[Pyt06] Python. Python, 2006.

[RRT97] K. Radecka, J. Rajski, and J. Tyszer. Arithmetic built-in self-test for dsp cores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(11):1358–1369, 1997.

[RT93a] J. Rajski and J. Tyszer. Accumulator-based compaction of test responses. *IEEE Transactions on Computers*, 42(6):643–650, 1993.

[RT93b] J. Rajski and J. Tyszer. Recursive pseudoexhaustive test pattern generation. *IEEE Transactions on Computers*, 42(12):1517–1521, 1993.

[RT93c] J. Rajski and J. Tyszer. Test responses compaction in accumulators with rotate carry adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(4):531–539, 1993.

[RT98] Janusz Rajski and Jerzy Tyszer. *Arithmetic built-in self-test for embedded systems*. Prentice Hall, Upper Saddle River, N.J., 1998.

[Sav92] J. Savir. Skewed-load transition test: Part i, calculus. In *Proc. of the International Test Conf.*, page 705, 1992.

[Sav99] J. Savir. Distributed generation of weighted random patterns. *IEEE Transactions on Computers*, 48(12):1364–1368, 1999.

[Sed03] Robert Sedgewick. *Algorithms in Java*. Addison-Wesley, Boston, 3rd edition, 2003.

[SFF89] M. H. Schulz, F. Fink, and K. Fuchs. Parallel pattern fault simulation of path delay faults. In *Proc. of the26th Conf. on Design Automation*, pages 357–363, 1989.

[SLCR95] U. Sparmann, D. Luxenburger, K. T. Cheng, and S. M. Reddy. Fast identification of robust dependent path delay faults. In

Proc. of the 32nd Design Automation Conf., San Francisco, California, United States, 1995. ACM Press.

[Smi85] G. L. Smith. Model for delay faults based upon paths. In *Proc. of the International Test Conf.*, pages 342–349, 1985.

[SP94] J. Savir and S. Patil. Broad-side delay test. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):1057–1064, 1994.

[SRKP01] Y. Shao, S. M. Reddy, S. Kajihara, and I. Pomeranz. An efficient method to identify untestable path delay faults. In *Proc. of the 10th Asian Test Symp.*, pages 233–238, 2001.

[SS97] M. Sivaraman and A. J. Strojwas. Primitive path delay fault identification. In *Proc. of the Tenth International Conf. on VLSI Design*, pages 95–100, 1997.

[Str02] Charles E. Stroud. *A designer's guide to built-in self-test elektronisk ressurs*. Frontiers in electronic testing. Kluwer Academic Publishers, Boston, 2002.

[TA78] S. M. Thatte and J. A. Abraham. A methodology for functional level testing of microprocessors. In *Proc. of the International Symp. on Fault-Tolerant Computing*, pages 90–95, 1978.

[TGA00] P. Tafertshofer, A. Ganz, and K. J. Antreich. Igraine - an implication graph-based engine for fast implication, justification, and propagation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(8):907–927, 2000.

[UJ86] J. G. Udell Jr. Test set generation for pseudo-exhaustive bist. In *Proc. of the International Conf. on Computer-Aided Design*, pages 52–55, 1986.

[UJM88] J. G. Udell Jr. and E. J. McClusky. Partial hardware partitioning: A new pseudo exhaustive test implementation. In *Proc. of the International Test Conf.*, page p. 1000, 1988.

[VDG+00] A. Virazel, R. David, P. Girard, C. Landrault, and S. Pravossoudovitch. Delay fault testing: choosing between random sic and random mic test sequences. In *Proc. of the European Test Workshop*, pages 9–14, 2000.

[VGL+01] A. Virazel, P. Girard, C. Landrault, S. Pravossoudovitch, and R. David. Delay fault testing: Choosing between random sic and random mic test sequences. *Journal of Electronic Testing-Theory and Applications*, 17(3-4):233–241, 2001.

[WLEF89] J. A. Waicukauski, E. Lindbloom, E. B. Eichelberger, and O. P. Forlenza. Wrp: A method for generating weighted random test patterns. *IBM Journal of Research and Development*, 33(2):149–161, 1989.

[WLRI87] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar. Transition fault simulation. *IEEE Design & Test of Computers*, 4(2):32–38, 1987.

[Wun87] H. J. Wunderlich. On computing optimized input probabilities for random tests. In *Proc. of the 24th Conf. on Design Automation*, pages 392–398, 1987.

[Wun88] H. J. Wunderlich. Multiple distributions for biased random test patterns. In *Proc. of the International Test Conf.*, pages 236–244, 1988.

[Wun90] H. J. Wunderlich. Multiple distributions for biased random test patterns. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(6):584–593, 1990.

[WW03] Qiu Wangqi and D. M. H. Walker. An efficient algorithm for finding the k longest testable paths through each gate in a combinational circuit. In *Proc. of the International Test Conf.*, volume 1, pages 592–601, 2003.