TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Adeniyi Olanrewaju Adekoya 173219IASM

TESTABILITY ANALYSIS OF DIGITAL CIRCUITS USING THE MODEL OF STRUCTURALLY SYNTHESIZED BDDS

Master's thesis

Supervisor: Prof. Raimund-Johannes Ubar

D.Sc. Institute of Computer Engineering, Tallinn University of Technology
Professor, Chair of Computer Systems Test and Verification

Co. Supervisor: Adeboye Stephen Oyeniran

Tallinn 2019

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Adeniyi Olanrewaju Adekoya

06.05.2019

Tallinn 2019

Abstract

A new approach to testability analysis and evaluation of digital circuits using structurally synthesized binary decision diagrams (SSBDD) is proposed. It focuses on calculating the probabilistic controllability measures in terms of signal probabilities. Different methods of calculating probabilistic controllability measures in terms of signal probabilities of digital circuits are examined and a new method is elaborated. Also, as an added value of the newly developed algorithm of SSBDD processing, a novel method is proposed for identifying the redundancy of faults in digital circuits. The proposed methods are based on true path tracing of Structurally Synthesized Binary Decision Diagrams. It is shown in the thesis that the known methods of calculating signal probabilities, are either inaccurate, due to neglecting signal correlations, or they will have exponential complexity if taking the correlations into account, and hence, are not practical. The novelty of the proposed method stands in the new idea of SSBDD-based elimination of signal correlations, which allowed the elaboration of a new algorithm to speed-up the calculation procedure and achieving exact values of probabilities. The feasibility and efficiency of the method were demonstrated by programming the algorithm and by carrying out the related experiments with different benchmark circuits. The experimental results demonstrated the advantage of the proposed SSBDD-based approach compared to the classical gate-level approach. The results of this research are submitted as a research paper to the conference of IEEE EUROMICRO Digital System Design: L.Jürimägi, A. Adekoya, R.Ubar, MJenihhin. True Path Tracing in Structurally Synthesized BDDs for Controllability Analysis of Digital Circuits.

Annotatsioon

Digitaalskeemide Testitavuse Analüüs Struktuurselt Sünteesitud Otsustusdiagrammide Abil

On välja töötatud uus struktuursetel binaarotsustusdiagrammidel (SSBDD) põhinev lähenemisviis digitaalskeemide testitavuse analüüsiks ja hindamiseks. Uue meetodi fookuseks on signaalide juhitavuse mõõtmine signaalitõenäosuse terminites. Töös on uuritud erinevaid seniseid meetodeid signaalide tõenäosusliku juhitavuse mõõtmiseks digitaalskeemides ja on arendatud välja uus meetod. Uue meetodi realiseerimiseks töötati välja algupärane SSBDD-mudelit skaneeriv algoritm, mille täiendavaks lisaväärtuseks sai töös väljapakutud veel üks uudne meetod liiaste (mitteoluliste) rikete identifitseerimiseks. Väljatöötatud meetodid põhinevad tõeste signaaliteede läbitrasseerimisel SSBDD-graafides. Töös on näidatud, et senised teada olevad signaalide tõenäosuste arvutamise meetodid on kas ebatäpsed, kuna ei võta arvesse signaalide korrelatsiooni, või on korrelatsiooni arvesse võttes liiga keerulised ja seetõttu ebapraktilised. Käesoleva uurimistöö tulemuste uudsus seisneb selles, et töötati välja uus graafiline signaalide korrelatsiooni elimineerimise meetod, mis võimaldas saavutada nii suuremat arvutuskiirust kui ka tagada arvutatud tõenäosuste täpsust. Meetodi realiseeritavust ja efektiivsust on tõestatud selle programmeerimise ja eksperimentide läbiviimisega erinevatel katseskeemidel. Eksperimentaaltulemused näitasid SSBDD-põhise meetodi paremust võrreldes klassikalise loogikaelementide tasemel läbiviidavate arvutustega. Magistritöö tulemused on esitatud artiklina rahvusvahelisele konverentsile IEEE EUROMICRO Digital System Design: L.Jürimägi, A. Adekoya, R.Ubar, MJenihhin. True Path Tracing in Structurally Synthesized BDDs for Controllability Analysis of Digital Circuits.

Acknowledgements

Firstly, I would like to express my greatest gratitude to God for His unconditional love.

I would also like to thank my supervisor, Prof. Raimund-Johannes Ubar, for his guidance and support to complete this research and writing this thesis and my co-supervisor Adeboye Oyeniran Stephen, for his contributions. I am forever grateful.

To my lovely wife, Oluwajoba, who has stood by me, supported and encouraged me through it all, I say a big thank you. I express my very profound gratitude to my parents, my siblings and entire family for always believing in me.

I must say a very big thank you to Sister Kemi, Brother Tunde, and the entire Onakoya family for the profound love and support they have shown me.

Finally, my sincere appreciation goes to my friends Olugbenga Niyi-Leigh and Adeolu Samuel Osidibo, who were immensely helpful during this research and writing this thesis. I am very grateful.

God bless you all.

List of abbreviations and terms

BDD	Binary Decision Diagram
CUT	Circuit Under Test
FFR	Fan-out-free Regions
ROBDD	Reduced Ordered Binary Decision Diagram
SSBDD	Structurally Synthesized Binary Decision Diagram

Table of Contents

List of Figures

List of Tables

1 Introduction

This thesis work focuses on improving testability of combinational circuits through calculation of exact signal probabilities, a measure of probabilistic controllability, using true path tracing of Structurally Synthesized Binary Decision Diagrams (SSBDDs).

The rest of this chapter discusses the importance of testability and the efficiency of the methods used in testability analysis.

1.1 Importance of Testability

We live in a world that depends a lot on digitally controlled systems for everyday activities from business to pleasure. Digital systems also make up the backbone of most critical systems. These systems must be highly reliable and retain their reliability as they evolve without incurring prohibitive costs [1]. This is because a failure in these systems could result in loss of lives, properties, and sensitive data or cause severe damage.

In order to prevent or minimize such failures, developers of these systems have to follow a series of standard procedures such as: validating and verifying designs and implementing fault-tolerant techniques. However, these alone are not enough to make sure that the system is fail-proof. Testing is also done to ensure that the entire system is functioning correctly and no fault is present in the final product.

Testing is the process of performing a set of experiments on a system to analyze its response to determine whether or not it behaves correctly [2]. During testing, a set of test patterns are applied to the circuit under test (CUT) and the output is compared with the expected response [3].

Interestingly, following Moore's law, the number of transistors in integrated circuit doubles in less than two years [3], [4], [5] which has proved accurate for about half a century and is evident in the accelerated technological advancements we have experienced. Many complex digital systems are now being embedded on single chips with the gain of reducing size and power consumption. This has greatly increased the complexity and testing cost of such chips.

Consequently, special design techniques have to be used to make a chip fully testable [6]. Testability refers to the property of a circuit that allows the application of logical test inputs and the ability to observe the internal states from its output. A fault is testable if a well-specified procedure exists to expose it, which is implementable with a reasonable cost using current technologies [7].

Testability may come at the cost of performance degradation, area overhead, and I/O pin demand, and it helps to improve the reliability of systems, manufacturing process, and increase yield.

Although the value of testing and testability cannot be overemphasized, it could pose a serious problem. Test pattern generation algorithms are time costly and this time cost could increase when trying to cover unrecognized redundant faults.

The efficiency of test pattern generation algorithms is an important issue as it plays a significant role in design for testability and built-in self-testing [2], [8], [9] and the results of testability analysis can be used to speed up the process or introduce design changes to improve testability.

1.2 Efficiency of testability analysis methods

Most methods used for analyzing the testability of digital circuits use the controllability and observability properties of the internal nodes to evaluate its testability. Controllability refers to the ease with which a node can be set to a logical value while observability refers to the ease with which the value of a node can be known by inspecting the output of the circuit.

Several methods have been proposed for estimating testability measures like SCOAP, CAMELOT, VICTOR [10], [11], [12] where all signals are assumed to be independent. The accuracy of these methods is generally limited due to the ignorance of signal correlations. Hence, the use of probabilistic measures to define testability [13], [14], [15], in [13], an exact method was proposed for calculating signal probabilities at the primary outputs of the circuit given the signal probabilities at the primary inputs.

However, to overcome the computational complexity, a cutting algorithm was proposed for estimating lower and higher bounds of probabilities [14]. The re-convergent fan-outs problem was addressed using conditional probabilities approach in [15]. These methods are developed for gate-level descriptions and suffer from the high complexity of calculations. Also, the exact method proposed in [13] loses its exactness for circuits with redundancies.

Binary decision diagrams (BDDs) have become state-of-the-art data structure for representing and manipulating Boolean functions [16], [17]. Traditional BDDs do not represent structural information about circuits and may quickly explode in size. SSBDDs, however, present the possibility of a one-to-one mapping between signal paths and the nodes of the BDDs [18], [19], [20]. The possibility of direct mapping of the nodes to signal paths allows for probabilistic analysis.

This thesis explores the use of SSBDDs for calculating exact signal probabilities for both non-redundant and redundant circuits.

1.3 Thesis Structure

The rest of this thesis is organized as follows. In Chapter two, an overview of testability measures is given and the problem of correlation is discussed. Thereafter, the concepts of SSBDDs and how digital circuits are represented as SSBDDs are introduced. Chapter three introduces the approach of true path tracing and its application to the calculation of signal probabilities and the identification of redundant faults. Chapter four covers the implementation of the path tracing algorithm, function for proving redundant faults, and calculation of probabilities. In chapter five, the experimental results are presented and chapter six concludes the thesis.

2 Overview Of Testability Measures

This chapter gives an overview of the testability measures. It begins with a brief description of the Heuristic measures of controllability, observability, and testability. Afterward, the various probabilistic measures are discussed and compared in terms of accuracy and complexity. Finally, we present the modeling of digital circuits as SSBDDs which will be used in the next chapter to optimize the calculation of signal probabilities.

2.1 Heuristic Measures

In this section, we discuss certain heuristics used to analyze the testability of a digital circuit. Several existing methods provide quantitative measures of the difficulty of controlling and observing the logical values of internal nodes [10]. Generally, the higher the number, the greater the difficulty.

2.1.1 Controllability

The measure of the difficulty or ease of controlling a node in a circuit depends on the logical value to be set on the node. Consider the AND and OR gate in figure 1.

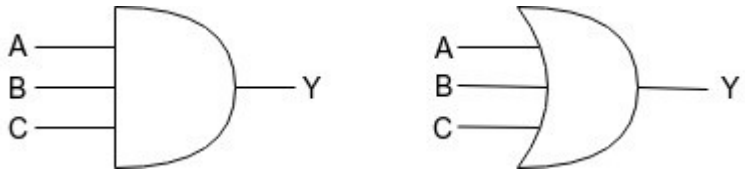


Figure 1. Three input AND and OR gates.

The difficulty of setting a circuit line, Y to a logic 0 or 1 can also be interpreted as the minimum number of nodes that must be set to produce a logic 0 or 1.

Let C_0, C_1 be the difficulty of setting a logic 0 and logic 1 respectively. Therefore the controllability of Y for the AND gate can be given as:

$$C_1(Y)=C_1(A)+C_1(B)+C_1(C)+1$$
$$C_0(Y)=min\left[C_0(A),C_0(B),C_0(C)\right]+1 \quad (1)$$

To get a logic 1 at Y for the AND gate, A, B, C must be set to logic 1, and in order to get a logic 0, at least one of the three inputs must be set to logic 0.

Likewise, the controllability of Y for the OR gate can be given as:

$$C_1(Y)=min\left[C_1(A),C_1(B),C_1(C)\right]+1$$
$$C_0(Y)=C_0(A)+C_0(B)+C_0(C)+1 \quad (2)$$

To get a logic 1 at Y for the OR gate, at least one of the three inputs must be set to a logic 1, and to get a logic 0, A, B, C must be set to a logic 1.

The result is incremented by 1 to reflect the distance to the primary inputs [7].

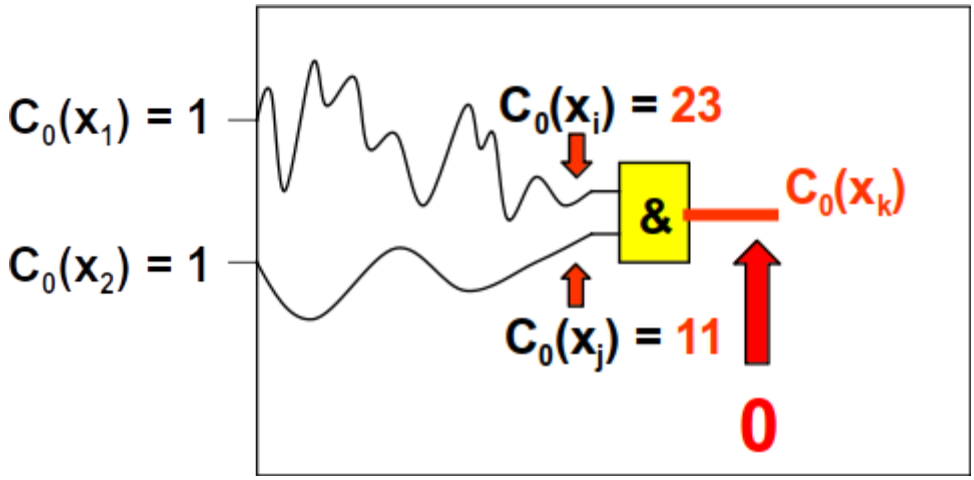


Figure 2. Controllability measure of AND gate.

Using equation (1), $C_0(X_k)$ for the circuit given in figure 2 can be calculated as:

$$C_0(X_k)=min\left[C_0(X_i),C_0(X_j)\right]+1$$

$$= min\left[23,11\right]+1 = 12$$

The controllability of other internal nodes are calculated recursively from the inputs. Since the input values can be applied easily, it is assumed that C_0, C_1 for all inputs is taken as 1.

2.1.2 Observability

This is a measure of the number of nodes which must be set for propagating the value of a node. This quantifies how difficult propagating a fault to the observable output will be. To observe a gate input, the output of the gate must be observable, and the other input(s) of the gate must be non-controlling.

Consider the AND and OR gate in the figure 3.

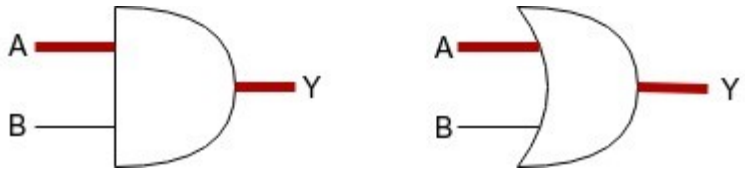


Figure 3. Signal propagation for AND and OR gate.

To observe the input *A* at *Y* for the AND gate, *B* must be set to the non-controlling value 1. Therefore, the observability of *A* in the AND gate is a measure of the observability of *Y* and the controllability of making *B* a logic 1. See equation (3).

$$O(A)=O(Y)+C_1(B)+1 \tag{3}$$

Likewise, to observe the input *A* at *Y* for the OR gate, *B* must be set to the non-controlling value of 0. Therefore, the observability of *A* in the OR gate is a measure of the observability of *Y* and the controllability of making *B* a logic 0. See equation (4).

$$O(A)=O(Y)+C_0(B)+1 \tag{4}$$

The observability of internal nodes is calculated backward recursively from the primary outputs. Since the primary output is easily observable, *O(Y)* is taken as 1.

2.1.3 Testability using heuristic measures

Using the discussed heuristic measures, testability can be quantified. Given the controllability and the observability, the testability can be taken as a measure of controllability and observability. To check the testability for a signal x stuck-at 0 and x stuck-at 1, this can be given as:

$$T(x \equiv 0) = C_1(x) + O(x)$$
$$T(x \equiv 1) = C_0(x) + 0(x)$$

Summary

In this section, we discussed two heuristics measures for testability: controllability and observability. Controllability is a measure of how difficult it is to control the value of a node/gate. It examines the circuit from the input to the nodes and determines the minimum number of nodes that must be set to produce logic 0 (denoted as C_0) or a logic 1 (denoted as C_1). Observability is a measure of how difficult it is to observe the value of a signal at the primary output. It examines the circuit from the output back to the nodes and determines the minimum number of nodes which must be set to propagate the value of a node to the primary output.

These measures are combined to determine how testable a circuit is and can be used to guide the design for testability and test generation.

2.2 Probabilistic Measures

In this section, we present some of the methods of calculating signal probabilities as seen in [13], [14], [15] using gate-level implementations. Signal probabilities give a probabilistic measure of how testable a node is. Given the input probabilities, the output probability can be computed and can be used for fault detection and test generation.

For example, given that all input probabilities is 0.5, if the output probability of a two-input AND gate is computed as 0.25, which is the probability of getting an output of 1, then a test length of 4 can be assumed to test the stuck-at 0 fault at the output of the AND gate. Also, the test generator can be improved to reduce the test length.

The following definition and expressions are used to calculate logic and signal probabilities.

Definition 2.2.1. The probability of a (logic) signal, expressed as $a = P(A=1)$ for signal *A*, is a real number on the interval [0, 1] which expresses the probability that signal *A* equals 1 [13].

Definition 2.2.2. The probability that signal *A = 0* is given as $P(A=0)=1-P(A=1)=1-a$ [13]

Consequently, the following formulas were derived [13]:

$$P(AB)=P(A)P(B)$$

$$P(A\vee B)=1-(1-P(A))(1-P(B))$$

(a) (b)

Figure 4. Multiple input gates with a single output.

Given a circuit depicted by figure 4a we have

$$p_y=\prod_{i=1}^{n} p_{xi} \tag{5}$$

and for figure 4b we have

$$p_y=1-\prod_{i=1}^{n}(1-p_{xi}) \tag{6}$$

These definitions are applied in the following methods.

2.2.1 Gate-by-Gate Calculation

According to the first method in [13], the output probability is computed gate by gate, for each gate the related formula of the probability calculation is used. The result for each gate is then propagated to the next gate or final output.

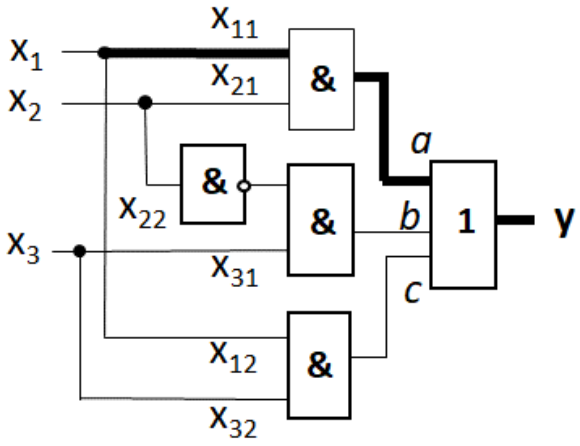


Figure 5. Combinational circuit with redundancy

Given the circuit in figure 5, the probability P_y that the output of the circuit will be $y=1$, assuming that for all inputs $P_k=1/2$, can be calculated as follows:

Since *y* is the output of the OR gate which has three inputs, using equation (6), P_y can be computed as

$$p_y=1-(1-p_a)(1-p_b)(1-p_c)$$

however, the inputs *a, b, c* are outputs of the AND gates as well and therefore should be computed using equation (5) in order to calculate the value for P_y

$$p_a=p_1 p_2, \quad p_b=\overline{p_2}\, p_3\,, \quad p_c=p_1 p_3$$

These values of P_a, P_b, P_c are then substituted

$$p_y=1-(1-\frac{1}{4})(1-\frac{1}{4})(1-\frac{1}{4})=\frac{37}{64}=0.58$$

This result, however, is not exact, because the correlations of signals due to re-convergent fan-outs are not taken into account.

2.2.2 Parker-McCluskey Method

According to the second method in [13], the probability expressions are expanded in the symbolic form with the input probabilities as arguments of the function.

$$p_y = 1-(1-p_1 p_2)(1-\overline{p_2}\, p_3)(1-p_1 p_3)$$

$$1-(1-p_3+p_2 p_3-p_1 p_2+p_1 p_2 p_3-p_1 p_2^2 p_3-p_1 p_3^2-p_1 p_2 p_3^2+p_1^2 p_2 p_3-p_1^2 p_2 p_3^2+p_1^2 p_2^2 p_3^2)$$

thereafter, all the exponents are removed as a method of addressing the correlations of signals.

$$1-(1-p_3+p_2 p_3-p_1 p_2+p_1 p_2 p_3-p_1 p_2 p_3-p_1 p_3-p_1 p_2 p_3^2+p_1 p_2 p_3-p_1 p_2 p_3+p_1 p_2 p_3)$$

$$\frac{27}{64}=0.42$$

This is an improvement over the gate-by-gate calculation as it takes into account the correlation of signals. As the circuits grow, the calculations will explode due to the opening of the embedded parentheses. Although this method has the advantage of providing exact probabilities, it is only valid for circuits without redundancies.

For example, the two functions

$$y_1=\overline{x_1}\, x_2 \vee x_1 \text{ and } y_2=x_2 \vee x_1$$

are equivalent, but the signal probabilities for these functions are different when calculated:

$$P_{y1}=1-(1-(1-p_1)p_2)(1-p_1)$$

$$P_{y2}=1-(1-p_2)(1-p_1)$$

$$P_{y1} \neq P_{y2}$$

2.2.3 Problem of correlations

One major problem of calculating signal probabilities is the presence of re-convergent fan-outs which introduce functional dependencies as well as statistical correlations

among the signals in the network [21], [22]. A simple example is given in the circuit in figure 6.

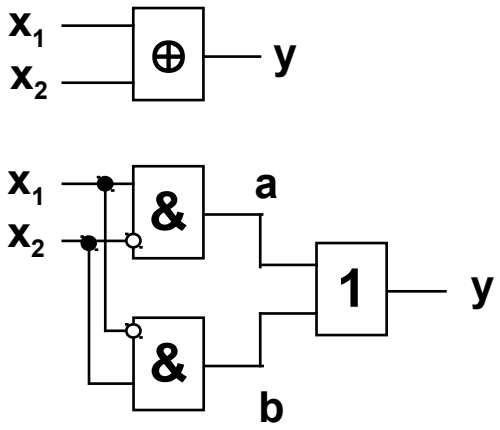


Figure 6. Signal correlations.

Using the gate-by-gate calculation, p_y is calculated as:

$$P_y = 1-(1-p_a)(1-p_b) \;=\; 1-0.75*0.75 = 0.44$$

Since the gate-by-gate method does not take signal correlations into account, the result is not exact. The exact result is 0.5.

2.2.4 Cutting Method

The cutting algorithm was proposed to reduce the computational complexity of exact probabilities. Its objective is to turn the combinational network into a tree by cutting re-convergent fan-out branches and inserting equivalent bounds at the cut points, which will guarantee that all the signal probability bounds computed on this tree will enclose the true values [14].

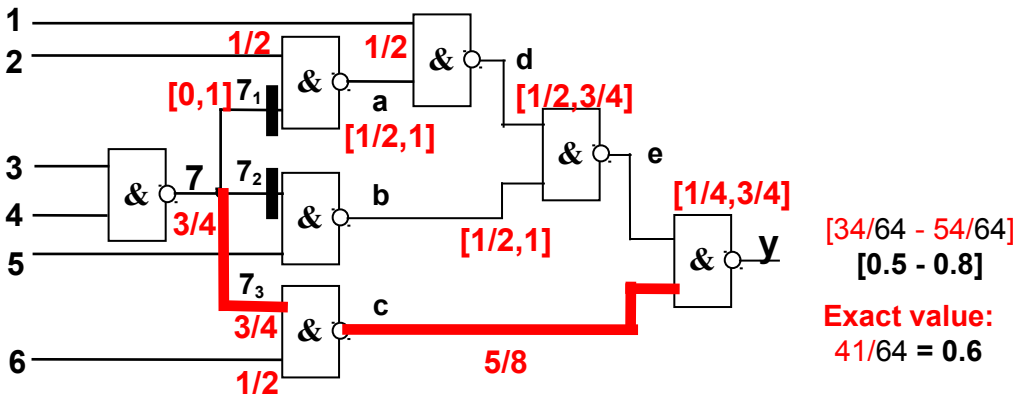


Figure 7. Calculating signal probability using the cutting method.

A signal probability range of [0, 1] is assigned to all the cut points except one. The bounds are propagated to the circuits other lines. The line not cut receives the probability of its immediate ancestor [14].

Consider the circuit in figure 7, assuming that for all inputs $P_k = 1/2$. 7_1 and 7_2 cut points are assigned the probability range [0, 1]. The output probability of gates are calculated and propagated like in the gate-by-gate method but for the cut points, both values in the range are used to calculate a new range.

Take the signal *a*, for example, its probability p_a, can be calculated as:

$$p_a = 1 - p_2 p_7^1$$

if $p_7^1 = 1$ then $p_a = 1/2$, likewise if $p_7^1 = 0$ then $p_a = 1$ therefore

$$p_a = [1/2, 1]$$

The disadvantage of this method is that it does not give an exact value result.

2.2.5 Method of Conditional Probabilities

The conditional probability method was proposed to tackle the problem of re-convergent fan-outs [15].

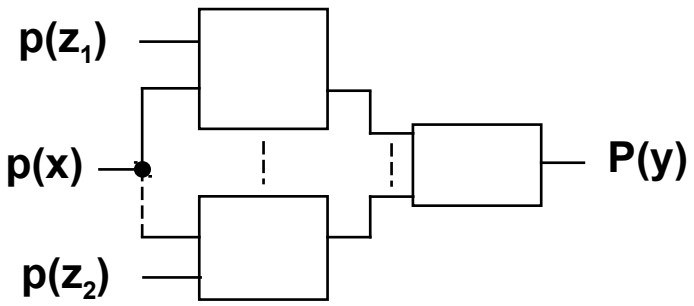


Figure 8. Combinational circuit with re-convergent fan-outs.

This method calculates two probabilities along each of the fan out paths. For example, *p(y)* in figure 8 is calculated with the condition *p(x) = 0* and *p(y)* = 1. Signal

correlations are avoided since there are no longer re-convergent fan-outs on the paths. Unlike the cutting method though, the final result is not a range.

$$p(y)=\sum_{i\in(0,1)} p(y/(x=i))\,p(x=i) \tag{7}$$

The probability of y is given in equation (7), where x set of conditions {0, 1}.

From figure 8, the probabilty of y, $p(y)$ is given as:

$$p(y)=p(y/x=0)\,p(y)+p(y/x=1)\,p(x)$$

For the circuit in depicted in figure 7, the probabilty of y, $p(y)$ can be calculated as follows:

$$p_y=p(y/x_7=0)(1-p_7)+p(y/x_7=1)\,p_7 \;=\; (1/2*1/4)+(11/16*3/4)=41/64$$

Summary

In this section, various probabilistic measure and their limitations were described. The problem of signal correlation as it affects the gate-by-gate method is discussed. This is addressed in the Parker-McCluskey method, although calculations tend to increase in complexity quickly. The cutting method attempts to reduce the complexity in exact probability calculations by computing a range for probabilities. Finally, the conditional probability approach is presented to tackle the problem of re-convergent fan-outs.

2.3 Modelling of Digital Circuits With SSBDDs

SSBDD is a planar, acyclic BDD generated by superpositioning of logic gates [23], [24]. The main difference between SSBDD and traditional BDDs is in their generation. While traditional BDDs are generated from Shannon's expansion, SSBDDs are generated from by superpositioning.

SSBDD functionalities are set apart from other BDDs by their approach of extractions with the use of fan-out-free regions (FFR) to separate each component of the digital

circuit as a separate entity thereby generating a structural decision diagram for each element [25], [26].

When BDDs were introduced, they were proposed for logic simulation [27] but were later adapted for test generation [28]. Reduced Ordered BDD (ROBDD) was introduced several years later and it helped to simplified the structure of BDDs. It, however, struggled with large designs as it tends to explode when synthesizing digital circuits with exponential data; these problems were never adequately addressed despite the modifications [29]. SSBDD was proposed to take care of the memory explosion problem presented by ROBDD in large designs. In the worst case, SSBDDs are linear in size with respect to the number of gates while traditional BDDs are exponential [30].

Figure 9 shows the BDD generated from the boolean expression $F = a \neg c + a \neg b \neg c + b \neg c$ while figure 10 shows the ROBDD counterpart. The ROBDD model removes repetitive nodes communicated in the BDD model.

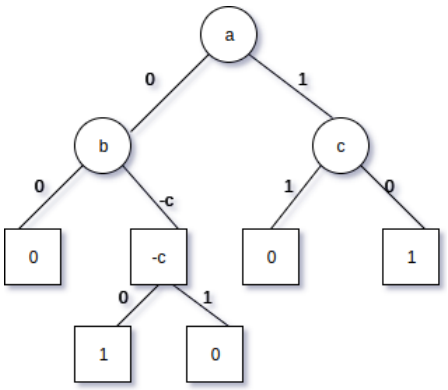
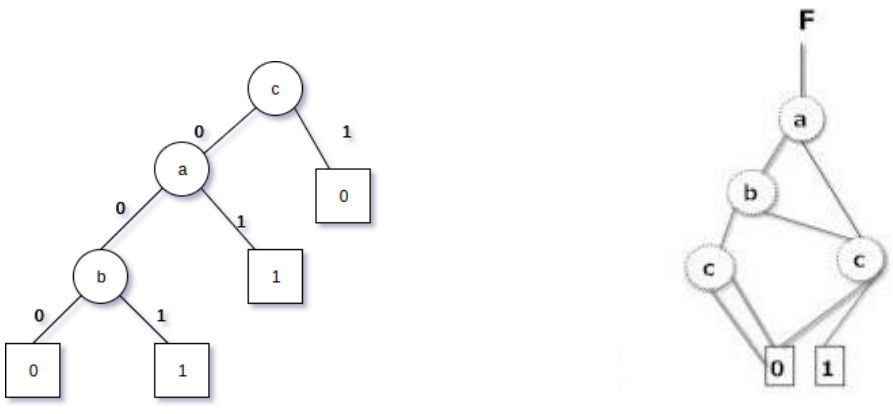


Figure 9. Binary Decision Diagram



a) b)

Figure 10. Reduced Order Binary Decision Diagram.

SSBDDs are formed from fan-out-free regions (FFR) of a combinational circuit using a superposition procedure. Each logic gate is represented with its equivalent elementary BDD recursively until the primary inputs or fan-out branches are reached [30].

Figure 11 shows the elementary BDDs for the AND, OR and NOR gate. Using this elementary BDDs, the SSBDD for a larger circuit can be constructed.

2.3.1 Representing digital circuits as SSBDDs

Consider the combinational circuit shown in figure 12. This can be converted to SSBDD in several steps of the superpositioning procedure illustrated in figures (13 - 17) [31]. The procedure involves working backward from the immediate logic gate preceding the output to the inputs. Input signals that are output signals from predecessor gates are replaced with a variable to simplify the construction and then replaced.

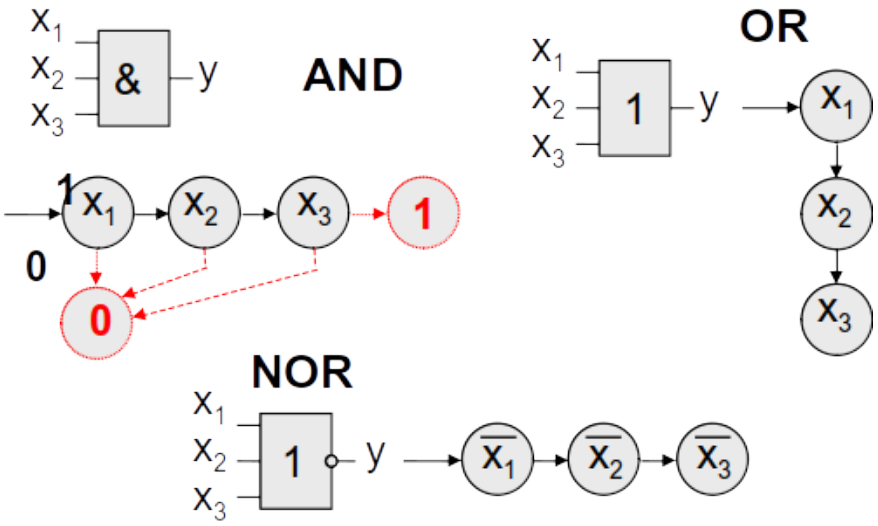


Figure 11. Elementary BDDs for logic gates.

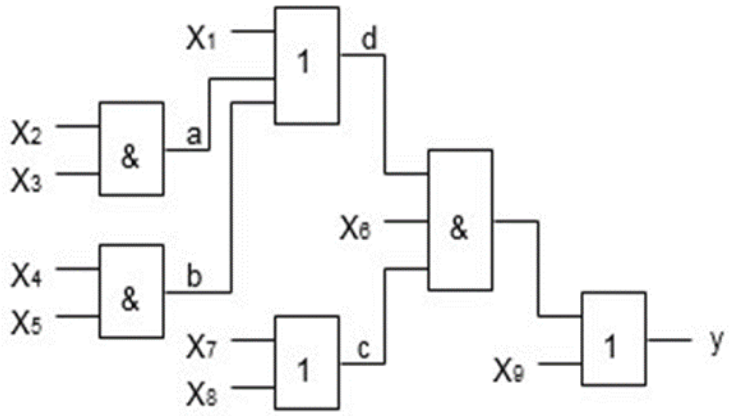


Figure 12. Combinational Circuit before SSBDD representation.

In step one, the immediate logic gate preceding the output *y* is considered as a simple OR gate with two inputs, X_9, and *f*, where *f* is the second input signal.

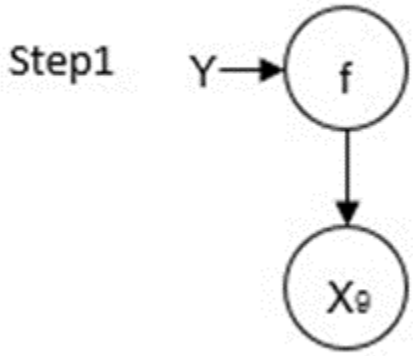


Figure 13. Superposition procedure (Step 1) .

Step two repeats the same step after substituting *f* with the value of the preceding the *f* signal. The substituted value also uses the same simplification step as in step one. Here, the two non-direct inputs are replaced with variables *d* and *c*.

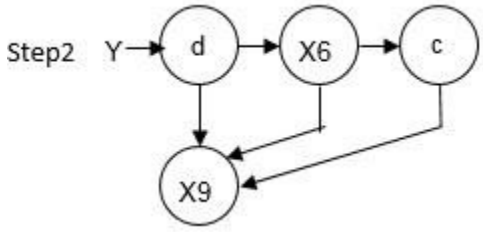


Figure 14. Superposition procedure (Step 2).

In step three, the input signal *d* is substituted in the same manner introducing the input X_1 and two new signal variable signals *a* and *b*.

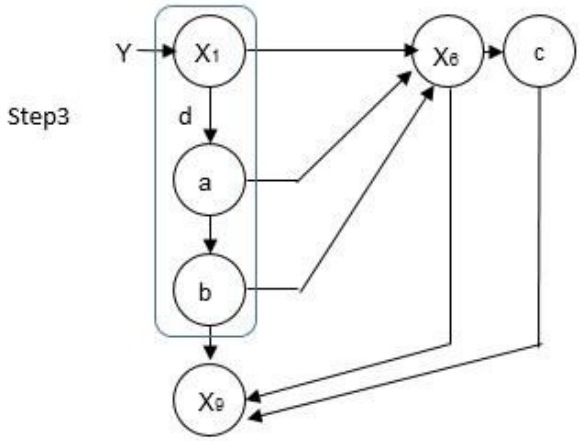


Figure 15. Superposition procedure (Step 3).

Step four follows the same process, substituting the variable signals *a* and *b* with their respective elementary BDDs and inputs X_2, X_3 and X_4, X_5 respectively.

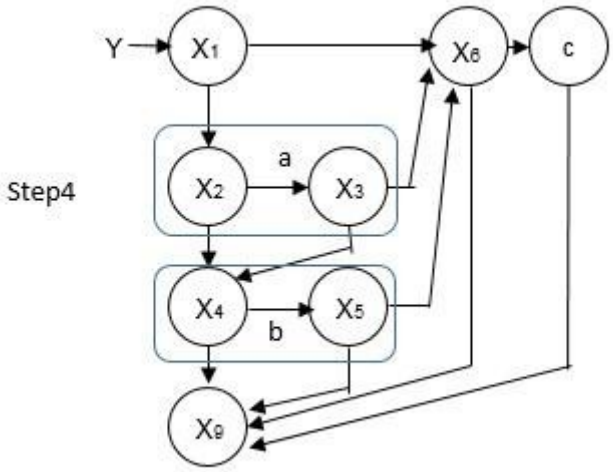


Figure 16. Superposition procedure (Step 4).

In step five, the final variable signal, *c* is substituted with its elementary BDD. Here the final input set, X_7 and X_8 are introduced and the SSBDD construction is complete.

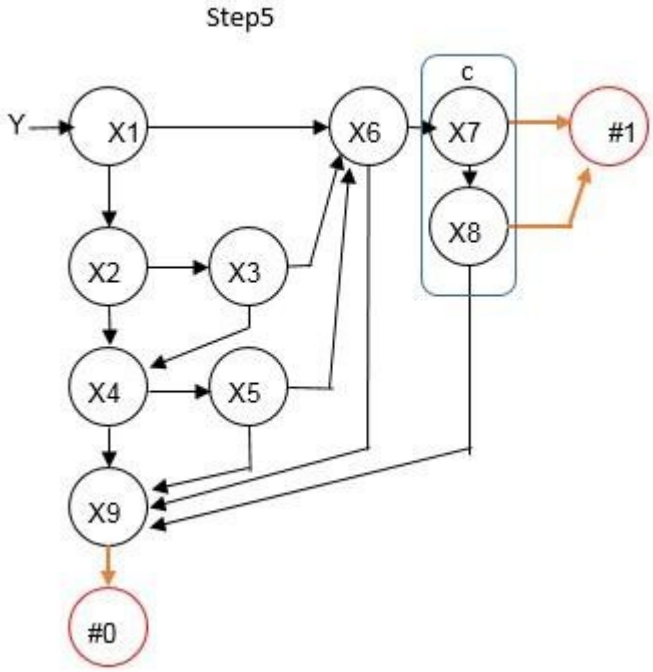


Figure 17. Superposition procedure (Step5).

To demonstrate how SSBDDs handle circuits with fan-outs, two circuits are considered. One circuit with internal fan-out (figure 18) and another with no internal fan-outs (figure 21).

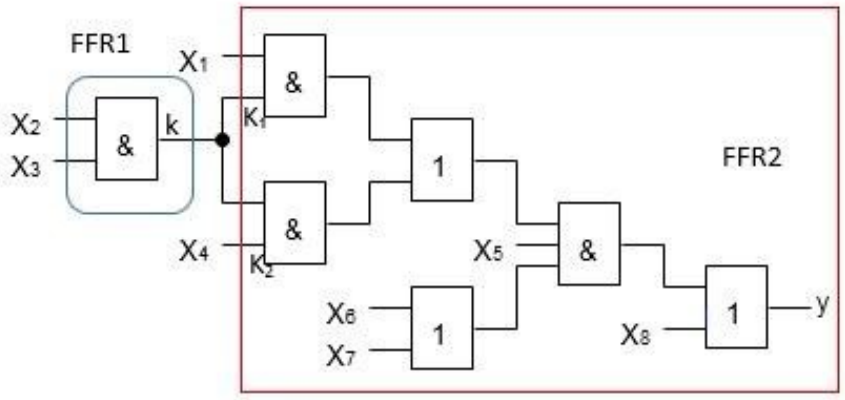


Figure 18. Combinational circuit with internal fan-out.

The circuit in figure 18 has two FFRs, since SSBDDs are formed from FFRs, two SSBDD graphs are generated as shown in figure 19 and 20

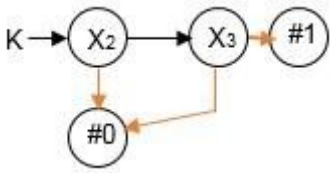


Figure 19. SSBDD for FFR1 in figure 18.

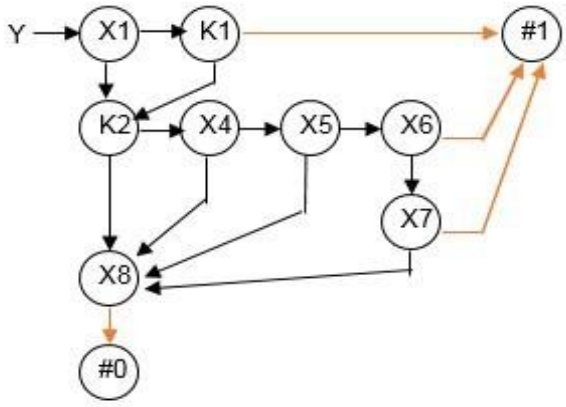


Figure 20. SSBDD for FFR2 in figure 18.

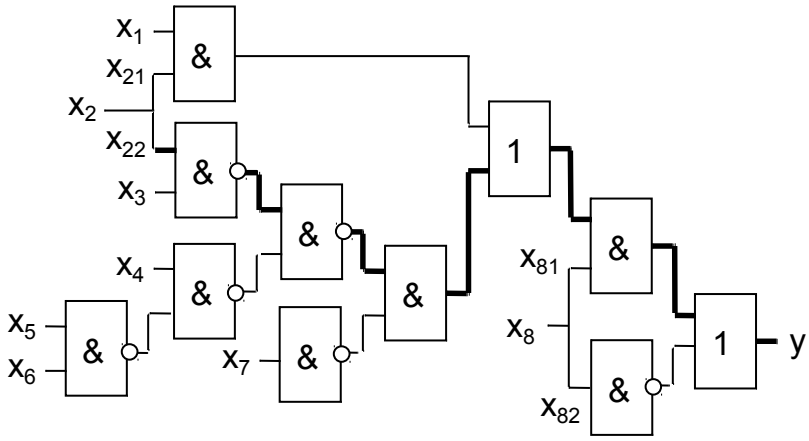


Figure 21. Combinational Circuit with no internal fan-out.

Figure 21 depicts a circuit with no internal fan-outs but with a fan-out branch x_2 with two fan-out stems. Each stem is differentiated from the stem variable In order to differentiate them, a suffix is added to the branch variable to give x_{21} and x_{22} as shown in figure 21. The resulting representation is shown in figure 22.

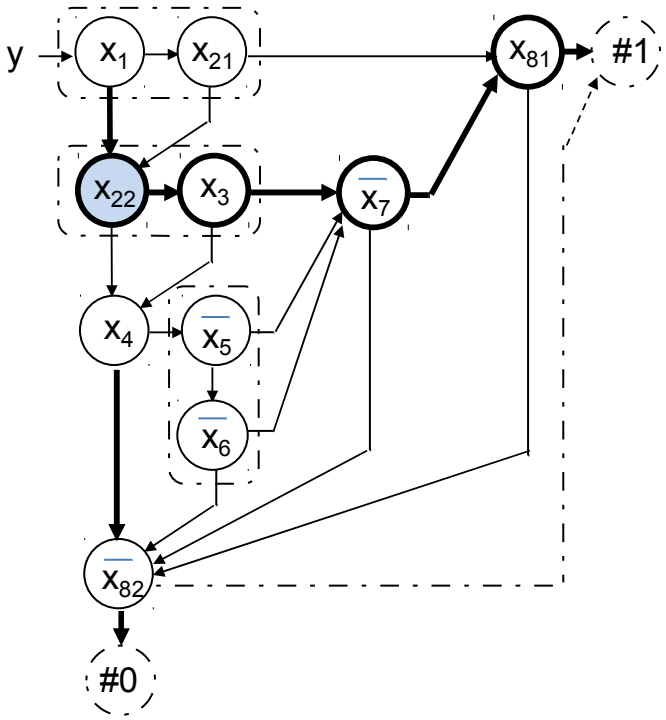


Figure 22. SSBDD representation of the circuit in figure 21

2.3.2 Traversing SSBDDs

Definition 2.3.1. Consider an SSBDD as a directed acyclic graph $Gy = (M,\Gamma,X)$ for representing a Boolean expression $y = F(X)$, where X – is the set of literals of the expression, M – is the set of nodes in the graph and Γ – represents a mapping: $\Gamma(m) \subset M$ denotes the set of successors of m, and $\Gamma^{I}(m) \subset M$ denotes the set of predecessors of m. The graph has a single root node $m_0 \in M$, were $\Gamma^{-1}(m_0) = \varnothing$, and two terminal nodes $m^T \in M_T = \{\#0, \#1\}$, labeled by Boolean constants. The nonterminal nodes $m \in M$ are labeled by literals $x(m) \in X$, and have exactly two successors $m^e \in \Gamma(m)$, $e \in \{0,1\}$ [32].

If there exists an assignment $x(m) = e$ then we say that the *edge* (m, m^e) in G_y is activated. Activated edges connecting nodes m_i and m_j form an activated path $l(m_i, m_j)$. An activated path $l(m_0, m^T)$ is called *full activated path*. In G_y, for all the possible vectors $X^r \in \{0,1\}^n$ there is a path $l(m_0, m^T)$ activated in G_y so that $y=f(X^r) = e(m^T)$ [32].

This implies that if $e = 1$, then we go in the right direction, and if $e = 0$, then we go downwards. Therefore, given an input pattern, the output signal can be simulated.

This is a useful property of SSBDDs as it can be used to simulate faults and generate test patterns for stuck-at faults [31]. For example, given the input pattern in table 1 for the SSBDD in figure 22

Table 1: Test pattern for SSBDD

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	y
0	1	1	0	-	-	0	1	1

By traversing the SSBDD, the output y is calculated. This can be done simply by using the node value from the entry point and following the direction until a terminal 0 or 1 is reached.

Using the input pattern, we have $x_1 \rightarrow x_{22} \rightarrow x_3 \rightarrow \overline{x_7} \rightarrow x_{81} \rightarrow$ #1. Therefore $y = 1$.

Also, the following nodes have been tested x_{22} stuck-at 0, x_3 stuck-at 0, x_7 stuck-at 1, x_{81} stuck-at 0 because if any of the nodes is stuck, then given the input pattern y would have ended up at terminal 0.

Summary

The concept of representing digital circuits as SSBDDs was presented in this section. The major differences between SSBDDs and traditional BDDs were described. While BDDs only represent functional information, SSBDDs represent structural information because they are formed by superpositioning. Also, the problem of exponential complexity of BDD is presented while SSBDD has linear complexity.

The steps of creating an SSBDD was explained for circuits with and without internal fan-out. It is expressed that SSBBDs are formed from fan-out free regions. Lastly, the traversing of SSBDDs is explained, and its value for testing and test generation is already realized.

2.4 Conclusion

This chapter discussed several approaches to testability. Firstly, the use of heuristics such as controllabilty and observability of nodes was presented and how they can be used to derive a quantitative measure for testability and aid in redesign to improve testability.

Secondly, the concept of signal probabilities and how it can benefit testability is presented. Futhermore, various proposed probabilistics measures and their limitations are described. The problem of signal correlation and re-convergent fan-out is described and the work around provided by newer methods.

Finally, the procedure for modeling of digital circuits as SSBDD is discussed and how they differ from traditional BDDs. It is shown that the SSBDD model is advantageous for its linearity in complexity and its ability to represent structural information of circuits. Also, the potential for using SSBDD for fault simulation and generating test patterns is described.

3 SSBDD-Based Method For Measuring Probabilistic Testability of Digital Circuits

In this chapter, the method of calculating signal probabilities and identifying redundant faults using SSBDDs is proposed. The methods are based on true path tracing of SSBDD. True path tracing reduces complexity in calculation by only dealing with nodes that result in a logic 1. Also, it is shown that redundancies do not affect the results of probabilities.

The first section presents the method of true path tracing and how it is used for calculating output probabilities using a proposed algorithm.

The application of the proposed algorithm for calculating probabilities of internal nodes of the circuit is discussed in the second section of this chapter.

In the third section, a method for the identification of redundant faults in digital circuits is proposed.

3.1 True Path Tracing For Calculating Output Probabilities

SSBDDs presents a way to identify signal paths or nodes in a circuit and their direction (which represent their logical value) necessary to attain a specific output. Therefore, to calculate the probability that the output signal equals 1, the required nodes and their direction can be identified and used to compute the probability.

The following procedure describes the steps for the method:

Procedure 1.

1. Trace all possible true paths in SSBDD in 1-direction, entering the terminal 1. Denote the set of all these paths as L.

2. Build for each path $l \in L$ a subset of nodes $M(l) \subset M$, so that each variable $x \in X$ were represented in this set once.

3. Partition for all $l \in L$ the sets $M(l)$ into two disjoint subsets $M(l) = M(l_0) \cup M(l_1)$ where $M(l_0)$ will consist of only 0-nodes, and $M(l_1)$ of only 1-nodes.

4. Calculate the probability p_y using the formula:

5. $$p_y = \sum_{l \, L} [\prod_{ml_0} p_m \prod_{ml_1} (1 - p_m)]$$

Tracing the possible true paths involves trying the different possible patterns that can produce an output of 1. However, using the SSBDD representation of the circuit, the process is simplified. It involves traversing the graph to arrive at terminal 1. Consider the SSBDD representation in figure 23. To get to terminal 1, set a value of 1 to x_{11}. This leads to the node x_{21}, which we can also be set to 1 and arrive at terminal 1. Now the direction of x_{21} can be toggled to see if an alternative true path can be found. This leads to node x_{31}. Toggling node x_{31} leads to one true path and one false path. Figure 23b shows the completely traced SSBDD.

It is important to note that the nodes are individual signal lines from inputs, therefore only the first occurrence of a node with a set input is present in the true path. See figure 25b. This is evident in the second step of procedure 1.

The correlations of signals is taken into account in step 2 by leaving out of the set $M(l)$ all the repeated nodes, which correspond to the re-convergent fan-outs. The values p_m correspond to the probabilities $p(x(m) = 1)$.

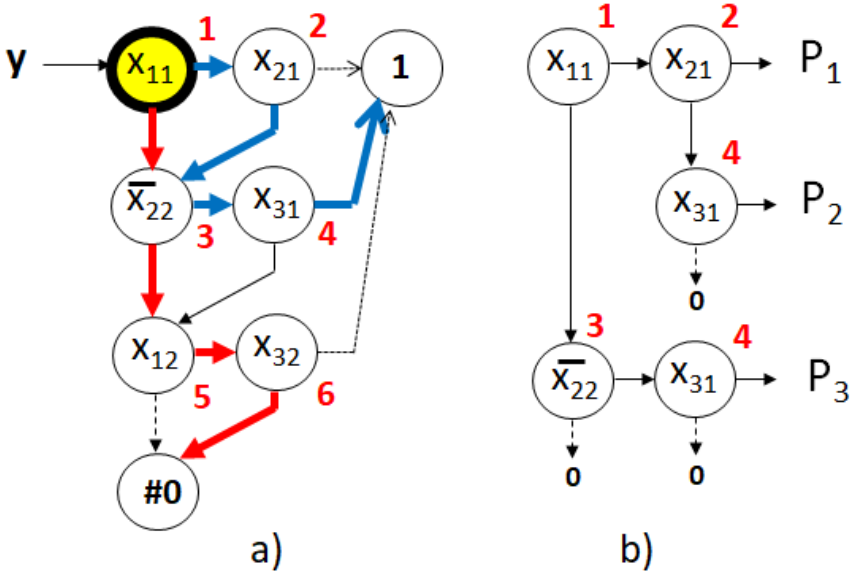


Figure 23. a) SSBDD for the circuit fig. 5 b) Traced SSBDD

The figure 23a shows the SSBDD representation of the circuit in figure 5 and the true path tracing according to step 1 of procedure 1. Figure 23b illustrates the traced paths which will be used for calculation of the probabilities according to step 5. It shows that there are only three paths to #1.

Using step 5 of procedure 1, the output probability p_y can be calculated as follows:

$$p_y = p_1 + p_2 + p_3 =$$

$$p_1 p_2 + p_1(1-p_2)p_3 + (1-p_1)(1-p_2)p_3 = 0.5$$

The result 0.5 differs from 0.58 and 0.42 as calculated by the previous methods in chapter two. This can be attributed to the redundancy in the circuit. The lower AND gate can be removed, and the logic function of the circuit remains the same. The circuit in figure 5 represents a multiplexer, where the lower redundant gate is introduced to avoid hazards on the output y during the transition $1 \rightarrow 0$ on the input x_2 in case the other values on the inputs are $x_1 = x_3 = 1$.

The procedure detects and discards the redundancies by excluding from the calculation all the paths which are not true due to inconsistency. This is possible because the algorithm takes into account both the probability functions and the logic circuitry.

3.2 True Path Tracing For Calculating Internal Probabilities

The proposed method in 3.1 which is aimed at calculating only the circuit's output signal probability contains the inherent possibility of calculating also the signal probabilities of internal nodes in the circuit which is not possible for traditional BDDs as they do not represent the structure of circuits. This section describes the possibility.

Each path $l \in L$ in SSBDD, extracted in Step 1 of procedure 1, represent a set of nodes m, where the node variables $x(m)$ have the values $x(m) = 1$. Hence, the probability of controlling the signal $x(m) = 1$ is equal to the probability of controlling the output signal $y = 1$ along this particular path $l \in L$. From that it follows, according to procedure 1, that the controllability of $x(m) = 1$, can be calculated as the sum of the probabilities of all paths $l \in L$ ending in the terminal 1, and which contain the node m in condition of $x(m) = 1$.

As an example of figure 23b, we have

$$p_{x11} = p_1 + p_2 = p_1 p_2 + p1(1 - p_2) p_3$$

which is the by-product of calculating the value of p_y with procedure 1.

Since node 1 with node variable x_{11} in the SSBDD in figure 23b represents the path from the input x_{11} to the output y in the circuit in figure 5, the probability p_{x11} represents the probabilistic controllability of activation of this path, and also the probabilistic controllability of all the signals on this path, consistent with the value $x_1 = 1$.

3.3 Identification of Redundant Faults and Hard-to-detect Faults

The first step of the procedure which performs the tracing of all consistent true paths in a digital circuit can also be used to for identification of redundant faults in circuits.

Consider a boolean function $y = F(X)$ represented in figure 24a as an SSBDD which is the same as the SSBDD in figure 23a.

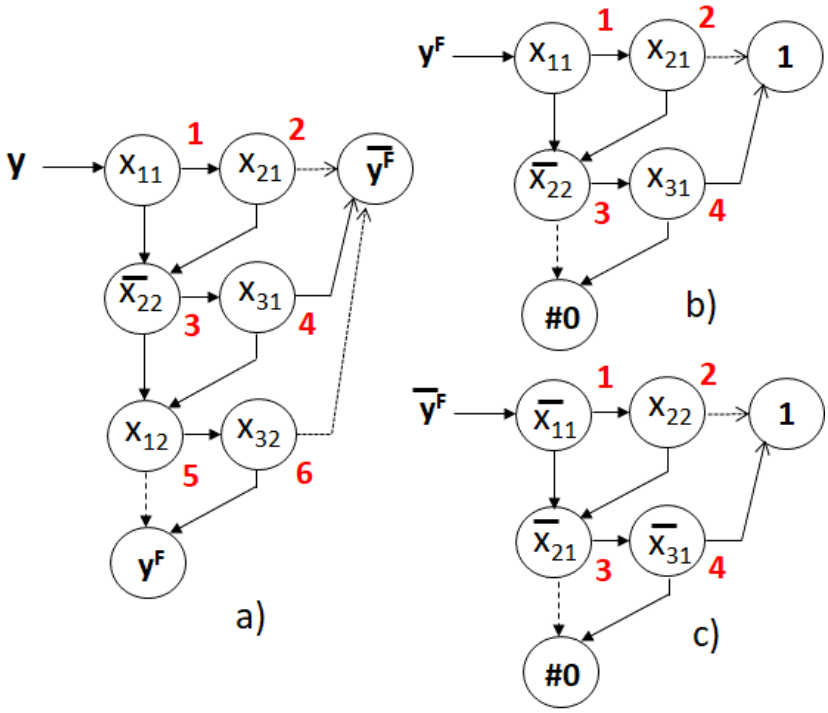


Figure 24. Correct, faulty and inverted faulty SSBDDs

In this circuit, there are three faults x_{12} stuck-at 0, x_{32} stuck-at 0, and c stuck-at 0, which are all redundant. Figure 24b depicts the SSBDD with one of the faults x_{12} stuck-at 0.

The faulty graph is created by using the original graph and the specified stuck-at fault. The general idea is to point both destinations of the faulty node to its stuck-at value. In this case, x12 is stuck-at 0; therefore the direction of x12 goes downwards in both cases. To further simplify the graph as seen in the diagram shown in figure 24b, the node x12 is removed as it has become irrelevant because any node pointing to it will end up in the terminal 0.

An inverted faulty graph is also created using the faulty graph. Each node is simply inverted. This inversion causes a change in destinations to occur as seen in figure 24c.

To prove that the fault x_{12} stuck-at 0 is redundant, we have to prove that the correct function $y = F(X)$ and the faulty function, denoted as y^F are equivalent. Equivalence can be proven by showing that $y \oplus y^F = 0$ or in another form

$$y\,\overline{y^F} \vee \overline{y}\,y^F = 0 \tag{8}$$

The equation (8) is depicted by figure 24a and after substituting y^F and $\overline{y^F}$ the full SSBDD is shown in figure 25a.

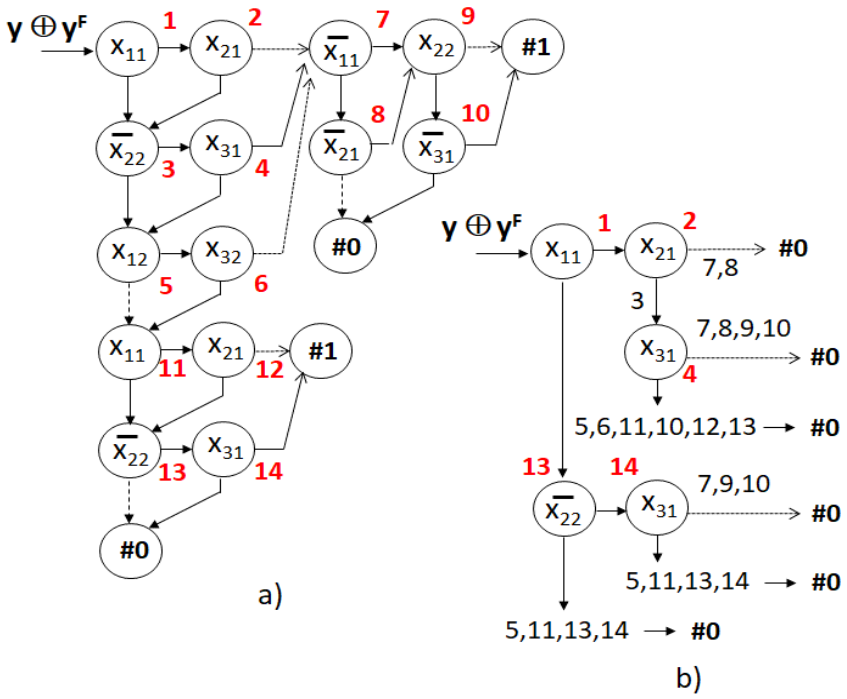


Figure 25. Proof of the redundancy of the fault x_{12} stuck-at 0

By applying the first step of procedure 1 to the SSBDD in figure 25a, all possible consistent true paths in the SSBDD are traced. Figure 25b shows the traced paths. The nodes with red numbers in figure 25b are those where the node variable is met for the first time on the current path. The black numbers on the edges in figure 25b denote the nodes which are forced to pass due to the value of the node variable, that was fixed earlier on the current path under trace.

Figure 25b shows that the result of tracing all end at the terminal 0, which indicates that the equation (8) is valid and correct function, $y = F(X)$ and the faulty function, y^F, are equivalent. Hence the fault in x_{12} stuck-at 0 has been proven to be redundant.

Likewise, the path tracing method can be used to generate tests for hard-to-test faults, for example, those which are resistant to detection by pseudorandom test sequences.

Using the path tracing method in section 3.2 which finds all true paths ending at terminal 1, each path will correspond to a test pattern that can be used to detect the given fault.

3.4 Conclusion

The proposed methods for calculating probabilities and identifying redundant faults are presented in this chapter. These methods rely heavily on true path tracing of SSBDD. The procedure for true path tracing is described and used as the first step in calculating signal probabilities.

The method for calculating output probabilities uses true path tracing to identify all possible nodes and path that can give an output of logic 1. The probability of these paths is then calculated and summed up to give the final output probability. In essence, all unnecessary nodes to get an output value of logic 1 are discarded hence simplifying the complexity.

While calculating output probability, internal node probability paths are inadvertently computed. Since the probability of an internal node measures the probability of controlling that node and the probability of propagating the value of the node to the output, consequently, all true paths with that node present constitute the measure required. Hence, the method presented sums up the probability for these paths to derive internal probabilities.

Lastly, a method for identifying redundant fault is proposed. The method uses a simple boolean equivalence check $y \oplus y^F = 0$, to deduce that the original circuit function y is equivalent to the function with the fault present, y^F. Therefore, an SSBDD constructed with the function $y \oplus y^F$ should have no true paths.

4 Implementation

Following the proposed approaches in section 3, the algorithms in figures 26, 27, 28 were created. The input of the algorithms is a computer-aided diagram (CAD) generated SSBDD model of a given circuit. For identifying redundant faults, the implemented program takes in a fault assignment input and uses the fault assignment to generate a new SSBDD like figure 25a.

The following defined notations are used in the algorithms.

Notations

- m – number of the node
- $Z(m)$ – node variable
- m^1 – neighbor of m if $Z(m) \oplus I(m) = 1$
- m^0 – neighbour of m if $Z(m) \oplus I(m) = 0$
- c – value popped from the stack
- $I(m)$ – inversion mark for $Z(m)$. ($I(m) = 1$ if $Z(m)$ is inverted)
- *#0, #1* – terminal nodes
- P – Output probability
- pp – Path probability

The main principle of the algorithms is the true path tracing in step 1 of the procedure. It is emphasized in the figures using a light blue color shade. The algorithms, simply extend this path tracing algorithm.

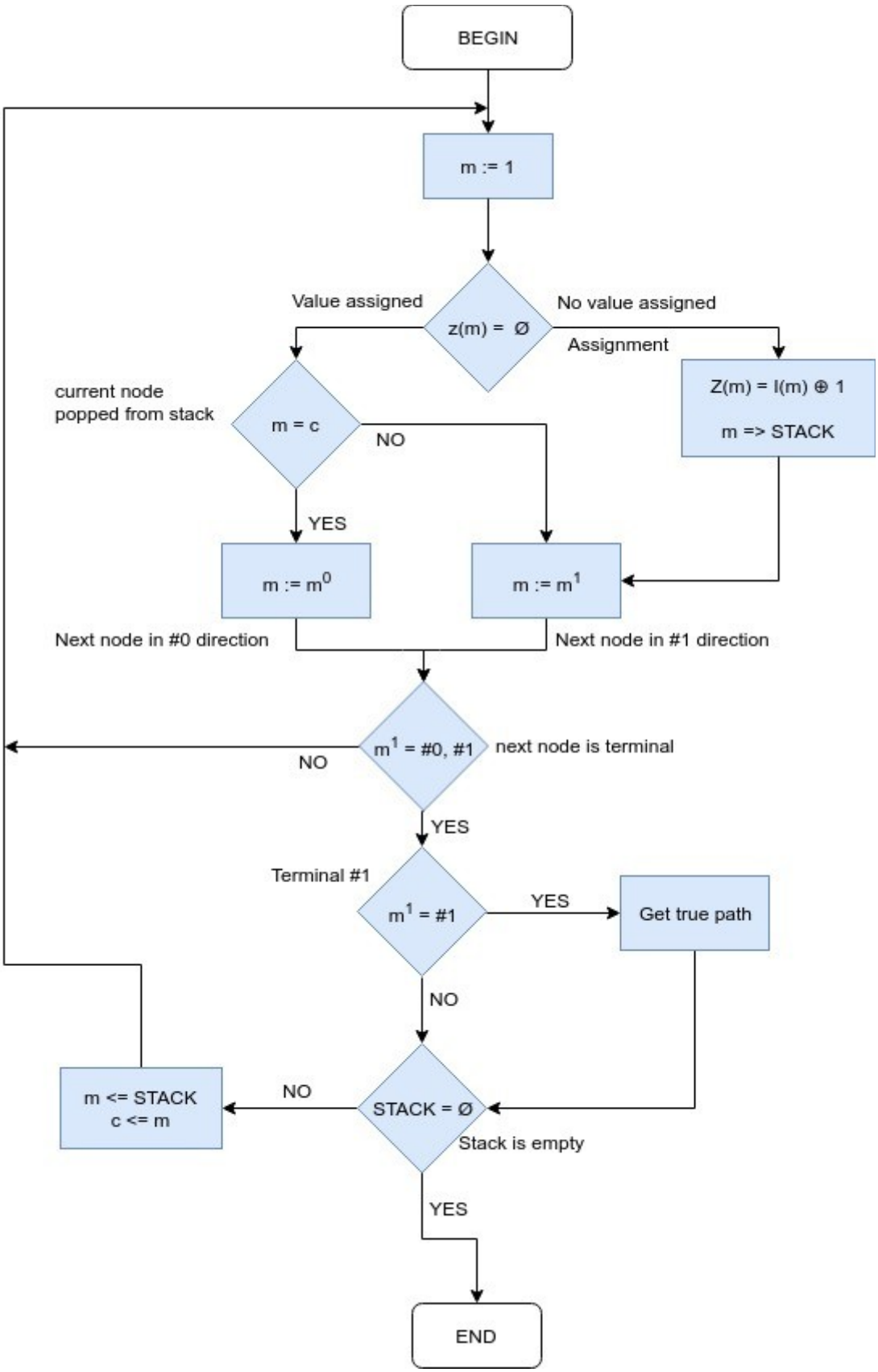


Figure 26. Algorithm for true path tracing in SSBDD.

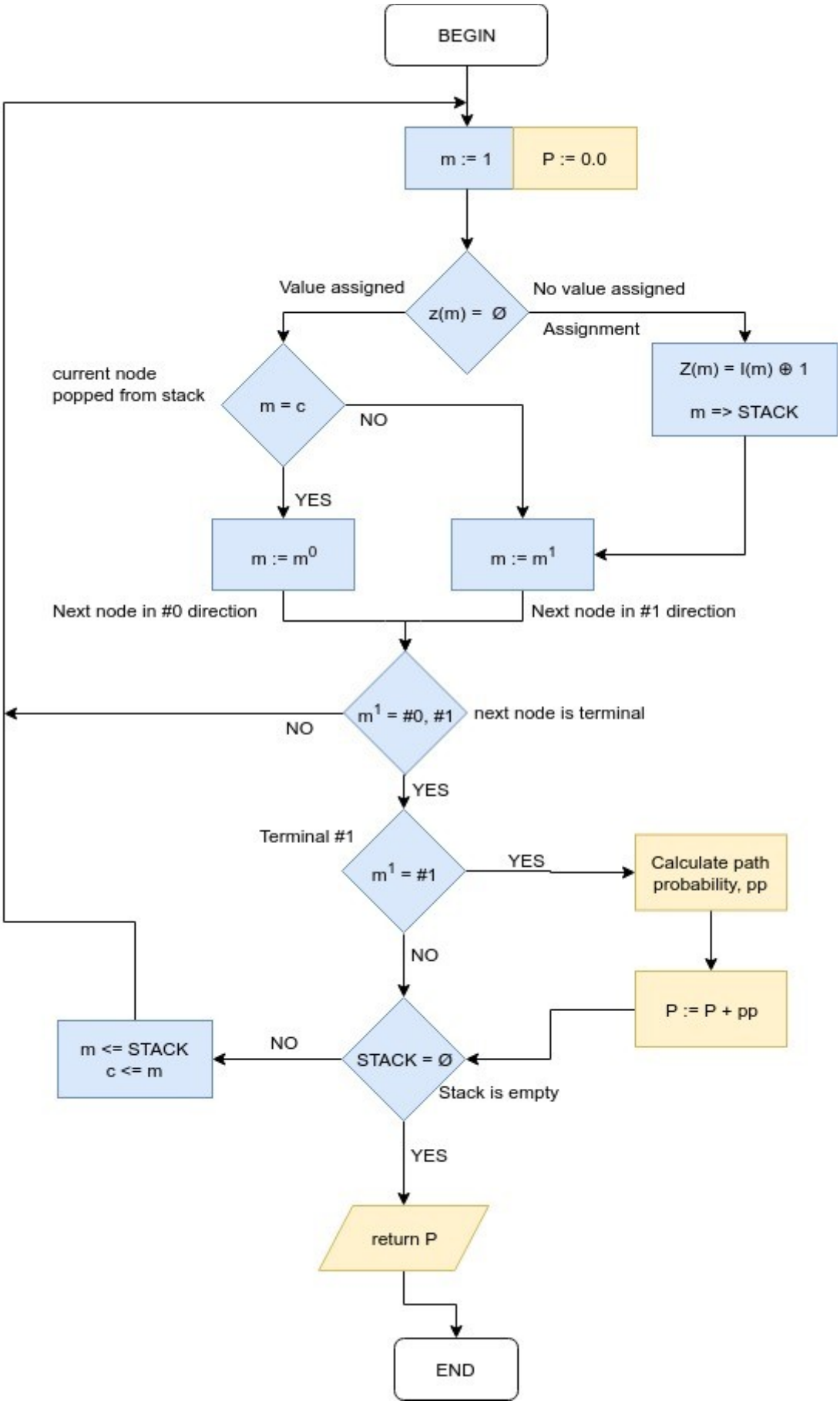


Figure 27. Algorithm for output probability calculation.

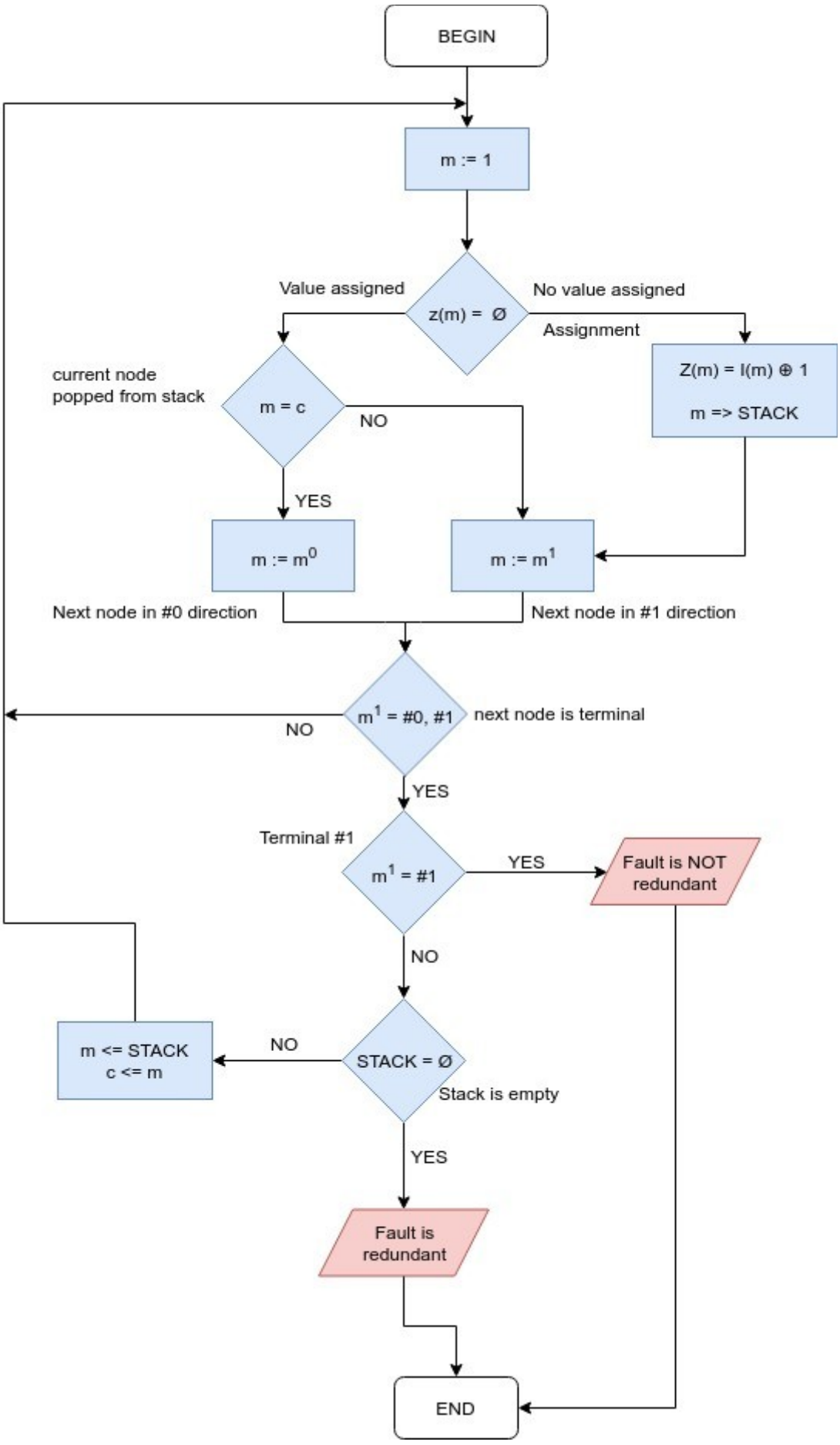


Figure 28. Algorithm for checking fault redundancy

Using the SSBDD circuit given in figure 23, the algorithm for path tracing can be explained as follows.

The tracing begins at $m := 1$ as shown in the first step of the algorithm depicted by figure 26. At the second step, it checks if *Z(m)* has been assigned a value. This gives the "no value assigned" response and the algorithm proceeds to assign a logic value of 1 which will activate the path toward the #1 and put *m* in a stack. Since the rightward path was activated in the previous step, the next node is will be node 2. The next step checks if the next node is a terminal. Since this is not the case we move to the next node. Table 2, shows the values for each iteration.

Table 2. Algorithm workflow

a) Algorithm iteration

#	*m*	*Z(m)*	m^1	m^0	*value*	*next*	*c*
1	1	-	2	3	1	2	-
2	2	-	#1	3	1	#1	2
3	**2**	**1 → 0**			**0**	**3**	**2**
4	4	-	#1	5	1	#1	4
5	4	**1 → 0**			0	5	4
6	1	**1 → 0**			0	3	1
7	3	-	4	5	0	4	1
8	4	-			1	#1	4
9	4	**1 → 0**			0	5	4
10	3	**0 → 1**			1	5	3

b) Stack content and Path

#	**Stack Content**	**Path Stack**
1	1	1
2	**1, 2**	**1, 2**
3	1	1, 2
4	**1, 4**	**1, 2, 4**
5	1	1, 2, 4
6	-	1, 2, 4
7	3	1, 3
8	**3, 4**	**1, 3, 4**
9	3	1, 3, 4
10	-	1, 3

At iteration 3, the value of *m* is already assigned, as a result, the downward path from node 2 is activated. Once the direction is changed, it is important to note that the top of the path stack is changed to the node that initiated the direction change, and only nodes in the path stack have assigned values. Also, there is a skip in the iteration for the value of *m*. This is because, node 2 and 3 are branches and since node 2 already has the value 0, then the rightward path is automatically activated for node 3 to node 4. Similar cases are also skipped in the table since they do not affect the stack content and path stack. The true paths traced are highlighted in green on table 2b.

Conclusion

Two algorithms were developed and implemented as software prototypes: 1) Algorithm for calculation of signal probabilities as controllability measures for a circuit on SSBDDs and 2) Algorithm for proving the redundancy of a given fault by showing the non-existence of a true path for the given SSBDD-based Boolean equation.

For using the second algorithm, the methods proposed in section 3.3, how to translate the given SSBDD to a faulty SSBDD and an inverted faulty SSBDD were used.

5 Experimental Results

The goals of the experimental research were to investigate *the difference between the accuracy of calculating signal probabilities*, and *the speed of calculations for combinational circuits represented at two different levels.* At the lower level as gate networks, and at higher level as networks of FFR-modules. Experiments were carried out with Intel Core i5 3570 Quad Core 3.4 GHz, 8 GB RAM.

Table 3. Comparison of speed and accuracy of signal probability calculation in a macro (FFR) and gate level SSBDDs

Circuit		**Time cost, µs**			**Probabilities**		
No	# Nodes	Gate [13]	Proposed Macro	Gain times	Gate [13]	Proposed Macro	Differ %
1	34	108	31	3.5	0.68	0.77	13.3
2	32	146	95	1.5	0.90	0.92	2.9
3	32	150	82	1.8	0.50	0.61	21.4
4	35	231	72	3.2	0.24	0.23	-2.1
5	33	208	70	3.0	0.63	0.70	11.4
6	29	168	59	2.8	0.53	0.53	-0.4
7	34	234	82	2.9	0.60	0.66	9.2
8	38	264	103	2.6	0.51	0.59	16.3
9	36	174	36	4.8	0.71	0.81	13.7
10	33	248	80	3.1	0.28	0.25	-9.5

In table 3, the results for a set of 10 simple benchmark circuits, represented by SSBDDs as a single FFR were compared with SSBDDs represented as a network of gates. The first method [13] of gate-by-gate probability calculation was used for the gate level circuits, where the signal correlations were not taken into account. There was no

experiment carried out with the second method [13] due to its high complexity. Instead, the experiments were carried out with exact probability calculation method developed in the thesis.

The numbers of nodes in the SSBDDs were from 29 to 38, and the benchmark circuits are characterized through different configurations of re-convergent fan-outs. The gain in the time cost of calculating signal probabilities was in the range of 1.5 – 4.8 and the difference in the probability values spanned in the range of -9.5 – 21.4, i.e. with nearly 30% difference in the worst case. This high percentage dispersion underlines the importance of the need for a more accurate method of probability calculation.

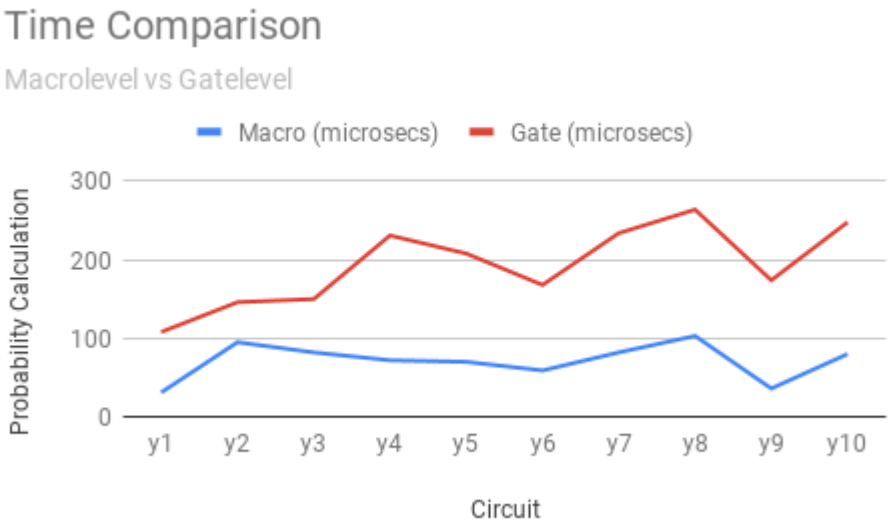


Figure 29. Comparison of the time cost of the calculation of signal probabilities at macro and gate level models.

In figure 7, the time cost of calculating signal probabilities for circuits represented as single SSBDDs (macros) and as networks of gates.

In this thesis, procedure 1 was implemented for the case of single SSBDD. For the general case, if a circuit is not represented as a single SSBDD, but as a network of many SSBDDs, the procedure should be extended as future work.

However, to investigate the practical use of the procedure for the general case, when a circuit consists of a network of FFRs, the experiments were carried out with gate-by-gate method with the extension to a general case where the components of the network can be both gates of FFRs.

The difference of both approaches is the number of correlations taken into account during the probability calculation. In the case of gate-by-gate probability calculation [13], no correlations are considered, whereas, in the proposed new method, all correlations of signals inside the FFRs are taken into account and removed.

To compare the accuracy of the new macro-level approach against the traditional gate-level approach, the data controllabilities calculated by the simulation-based Monte-Carlo method [14] was taken as reference.

For experimental research, the standard ISCAS'85 benchmark circuits [33] were used to estimate the feasibility and efficiency of the method proposed. In table 4, the characteristic data of the SSBDD models for the ISCAS'85 family of circuits in terms of the numbers of nodes (nodes), numbers of variables (var), and numbers of SSBDDs (graphs) are depicted.

Table 4: Benchmark circuits ISCAS'85

Circuit	**Macro-level**			**Gate-level**		
	nodes	var	graphs	nodes	var	graphs
C432	308	132	96	487	311	275
C499	601	228	187	1097	724	683
C880	497	211	151	775	489	429
C1908	866	281	248	1394	809	776
C2670	1313	663	430	2075	1425	1192
C3540	1648	428	378	2784	1564	1514
C5314	2712	811	633	4319	2418	2240
C6288	3872	1520	1488	4864	2512	2480
C7552	3552	1127	920	5795	3370	3163

Since the re-convergence of fan-outs on the macro-level component network, and hence the related signal correlations are still neglected in the proposed method, the following hypothesis was set: due to removing signal correlations inside all FFR-macros in the circuit, the remaining amount of high-level correlations between macros should be less than the amount of all gate-level signal correlations. As a result, if this hypothesis would be correct, the proposed method in average should work more precisely on the macro-level. Experimental results confirmed this assumption.

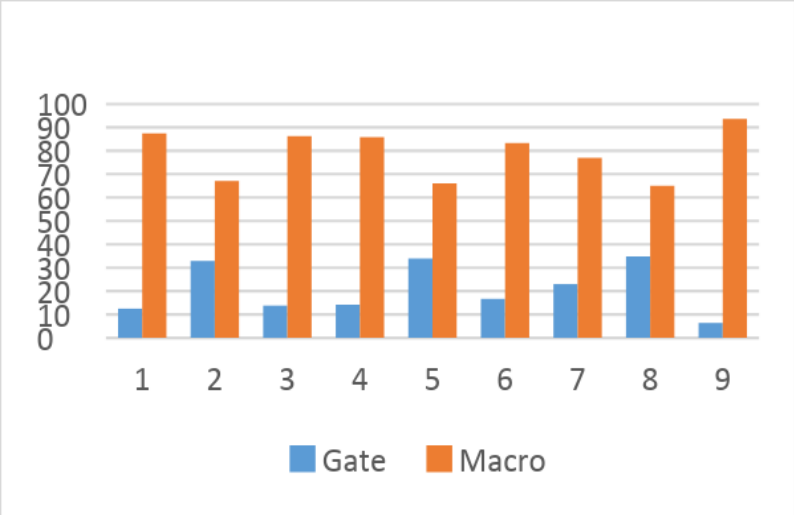


Figure 30. Comparison of the proposed method of controllability calculation for digital circuits at macro and gate levels with Monte-Carlo method

Figure 30 shows that the percentages of the circuit nodes in 9 Benchmark circuits, where the macro-based method of controllability calculation is used, is more exact than using the gate-level calculation. For example, for the circuit C7552 in 93.5% cases, the macro-level approach works better than the gate-level method (Table 5). On average, the overall benchmarks of the macro-level approach is five times more accurate than the gate level approach.

Note that the accuracy of the macro-approach may be increased by extracting bigger macros and representing them as single SSBDDs, which will result in an additional reduction of the fan-out re-convergence (and signal correlations), i.e., in the less number of "sources of inaccuracies." However, since this effect can be achieved by increasing the time cost of calculations, the finding of trade-offs could be interesting for future research.

Table 5: Comparison of the Macro and Gate approaches

Circuit	**C432**	**C499**	**c880**	**C1908**	**C2670**
Macro %	87.5	67.1	86.2	85.8	66.0
Gate %	12.5	32.9	13.8	14.2	34.0
Circuit	**C3540**	**C5315**	**c6288**	**C7552**	**Average**
Macro %	83.3	77.0	65.1	93.6	79.1
Gate %	16.7	23.0	34.9	6.4	20.9

Conclusion

This chapter presents the experimental results and most importantly shows that the signal probabilities for the nodes of digital circuits can be calculated exactly for circuits which may contain redundancy.

Also unlike traditionally BDDs, the probabilistic controllability can be calculated not only for the output signals of circuits represented by BDDs but also for all nodes in the related circuit during the same procedure of true paths tracing on the SSBDD. The reason for the latter possibility is the exact mapping between the SSBDD nodes and the signal paths in the circuit.

6 Conclusion

This thesis aimed to propose a method for exact calculation of signal probabilities as a controllability measure for combinational circuits represented by structurally synthesized binary decision diagrams (SSBDD) as well as present an identification technique for redundant faults.

Chapter one emphasized the importance of testability and discussed the efficiency of various testability analysis approach. In chapter two, an overview of some known testability measures and their limitation was given. Chapter three covers in detail the proposed approach. Chapters four and five cover the implementation and experimental results, respectively.

The proposed method represents a circuit as a set of macros, which is a hierarchical network of fan-out free regions (FFR) or SSBDDs instead of gates.

Experimental results showed that the proposed method is capable of computing exact signal probabilities faster with SSBDDs than the gate level. Additionally, it was shown that taking into account signal correlations increases the accuracy of signal probabilities by, an average, four times compared to gate-level results.

The algorithm for calculation of signal probabilities as controllability measures for a circuit on SSBDDs and the algorithm for proving the redundancy of a given fault algorithms were developed and implemented as software prototypes.

References

[1] M.Hinchey, L.Coyle. Evolving Critical Systems: a Research Agenda for Computer-Based Systems, 17th IEEE Int. Conference and Workshops on Engineering of Computer-Based Systems. 2010, pp. 430–435.

[2] Abramovici, Miron; Breuer, Melvin A.; Friedman, Arthur D., "Digital Systems Testing and Testable Design", John Wiley & Sons, Inc., Hoboken, New Jersey, 1990.

[3] L-T.Wang, C-W.Wu, X.Wen., "VLSI Test Principles and Architectures Design for Testability," Morgan Kaufmann Publishers, 2006.

[4] G. Moore. Cramming More Components onto Integrated Circuits – Reprint from IEEE proceedings on Electronics, Vol. 38, No. 8, 1965.

[5] R. W. Keyes, "The Impact of Moore's Law. – IEEE Solid-State Circuits, Issue", Sept 2006.

[6] R. Ubar. Design for Testability- TalTech, [Online]. Available: http://www.pld.ttu.ee/~raiub/web_0103/disain_ja_test/SLIDES/1_Introduction.pdf

[7] Jin-Fu Li. Testability Measures - National Central University Jhongli, Taiwan [Online]. Available: http://www.ee.ncu.edu.tw/~jfli/test1/lecture/ch03.pdf

[8] M. L. Bushnell, V. D. Agrawal, Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits, Kluwer Academic Publisher, 2000.

[9] S. Hurst. "VLSI Testing Digital and Mixed Analogue-Digital Techniques." The Institution of Electrical Engineers, 1998.

[10] L. H. Goldstein and E. L. Thigen, "SCOAP: Sandia Controllability/Observability Analysis Program." Proc. 17th Design Automation Conf., pp. 190-196, June, 1980.

[11] R. G. Bennetts, C. M. Maunder, and G. D. Robinson, "CAMELOT: A Computer-Aided Measure for Logic Testability." IEEE Proc., Vol. 128, Part E, No. 5, pp. 177-189,1981.

[12] I. M. Ratiu, A. "Sangiovanni-Vincentelli, and D. O.Peterson, VICTOR: A Fast VLSI Testability Analysis Program." Proc. Intn'l Test Conf., pp. 397-401, November, 1982.

[13] K.P.Parker, E.J.McCluskey. Probabilistic Treatment of General Combinational Networks. IEEE Trans. on Computers, Vol. C-24, No. 6, 1975, pp.668-670.

[14] J.Savir, G.S.Ditlow, P.H.Bardell. "Random Pattern Testability." IEEE Trans. on Computers, Vol. C-33, No. 1, 1984, pp.79-90.

[15] S.Seth, L.Pan, V.D.Agrawal. "PREDICT - Probabilistic Estimation of Digital Circuit Testability." IEEE 15th Int. Symp. on Fault-Tolerant Computing, 1985, pp.220-225.

[16] R.E.Bryant. Graph-based algorithms for Boolean function manipulation, IEEE Trans. on Comp., Vol.C-35, No 8, 1986, pp.667-690.

[17] T.Sasao, M.Fujita (eds.). Representations of Discrete Functions, Kluwer Acad. Publ., 1996.

[18] R.Ubar. Test Synthesis with Alternative Graphs, IEEE Design & Test of Computers, Spring, 1996, pp.48-57.

[19] R.Ubar. Overview about Low-Level and High-Level Decision Diagrams for Diagnostic Modeling of Digital Systems. Facta Universitatis (Nis) Ser.: Elec. Energ. vol.24, no.3, Dec. 2011, 303-324

[20] R.Ubar, J.Raik, H.-T.Vierhaus. Design and Test Technology for Dependable Systems-on-Chip. IGI Global, 2011, 550 p.

[21] S. Ercolani, M. Favalli, M. Damiani, P. Olivo, B. Riccò. "Testability Measures in Pseudorandom Testing," IEEE Trans. on Comp., Vol.11, 1992.

[22] S. Ercolani, M. Favalli, M. Damiani, P. Olivo, B. Riccò. "Estimate of signal probability in combinational logic network." Proceedings of the 1st European Test Conference, 1989.

[23] J. Raik, R.Ubar, S. Devadze, A. Jutman. "Efficient Single-Pattern Fault Simulation on Structurally Synthesized BDDs", Tallinn University of Technology, Department of Computer Engineering, Raja 15, 12618 Tallinn, Estonia.

[24] R. Ubar. Test Generation for Digital Circuits Using Alternative Graphs (in Russian), in Proc. Tallinn Technical University, 1976, No.409, Tallinn TU, Tallinn, Estonia, pp.75-81.

[25] M.Gorev, R.Ubar, S.Devadze. "Fault Simulation with Parallel Exact Critical Path Tracing in Multiple Core Environment". Proc. of DATE, Grenoble, France, 2015, pp. 1-6.

[26] R.Ubar, S.Devadze, J.Raik, A.Jutman. "Parallel X-Fault Simulation with Critical Path Tracing Technique." Proc. of DATE, Dresden, Germany, 2010, pp. 1-6.

[27] C.Y. Lee. "Representation of Switching Circuits by BDDs", in Bell System Techn. J., 1959, v.38, No7, pp.985-999.

[28] S.Akers. "Binary Decision Diagrams," IEEE Trans. on Comp., Vol.27, 1978, pp.509-516.

[29] A. Jutman, J. Raik, R. Ubar "SSBDDs: Advantageous Model and Efficient Algorithms for Digital Circuit Modeling, Simulation & Test" A. Jutman, J. Raik, R. Ubar 2003.

[30] G. Jervan, A. Markus, J. Raik, R.Ubar. Hierarchical Test Generation with Multi-Level Decision Diagram Models. Tallinn University of Technology, Department of Computer Engineering. 1998

[31] Oyeniran Adeboye Stephen "Double Phase Fault Collapsing With Linear Complexity in Digital Circuit " 23-30 May 2015.

[32] R. Ubar, L. Jürimägi, A. Adekoya, M. Jenihhin. "True Path Tracing in Structurally Synthesized BDDs for Controllability Analysis of Digital Circuits." (Submitted) 2019.

[33] A.Abdollahy. Probabilistic Decision Diagram for exact probabilistic analysis. IEEE/ACM Int. Conference on Computer-Aided Design, 2007.

[34] L. H. Goldstein. Controllability/Observability Analysis of Digital Circuits, IEEE Trans. on Circuits and Systems, Vol. CAS-26, No. 9, Sept. 1979, pp. 685 – 693

[35] O.Novak, E.Gramatova, R.Ubar. Handbook of Testing Electronic Systems.

Appendix 1 – Program Description and Manual

The program is a linux terminal application. The following instructions will assist in running the program.

1. Open the terminal and navigate to where the program is located.

2. Run the program as shown below and pass the agm file path as the argument.

```
:~/workspace/FaultRedundancyChecker/cmake-build-debug$ ./FaultRedundancyChecker ~/Desktop
/thesis/macrolevel/y1.agm
```

Figure 31. Running the program

3. The results are displayed as shown depending on the program mode .

```
:~/workspace/FaultRedundancyChecker/cmake-build-debug$ ./FaultRedundancyChecker ~/Desktop
/thesis/macrolevel/y2.agm

Reading SSBDD-model file /home/noadek/Desktop/thesis/macrolevel/y2.agm... OK

P(X4): 0.500000
P(X6): 0.500000
P(X3): 0.500000
P(X1): 0.500000
P(X5): 0.500000
P(X2): 0.500000
P(Y2): 0.921875

Time(s): 0.000033

Deleting SSBDD-model from memory... OK
```

Figure 32. Probability calculation result

```
:~/workspace/FaultRedundancyChecker/cmake-build-debug$ ./FaultRedundancyChecker ~/Desktop
/thesis/macrolevel/y1.agm

Reading SSBDD-model file /home/noadek/Desktop/thesis/macrolevel/y1.agm... OK

!6- -> 7- -> 9- -> #0
!6- -> 7- -> !9- -> 15- -> !16- -> #0
!6- -> 7- -> !9- -> 15- -> 16- -> #0
!6- -> 7- -> !9- -> !15- -> #0
!6- -> !7- -> 8- -> 9- -> !43- -> #1
Fault Redundant: False


Time(s): 0.000003

Deleting SSBDD-model from memory... OK
```

Figure 33. Fault redundany result

Appendix 2 – Source

```
/*
 * Created by: Adeniyi Adekoya
 *
*/
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "checker.h"
#include "mystack.h"
#include "nodemem.h"
/**
 * Prints the message specified and terminates the program.
 * @param message
 */
void exception(char* message)
{
    printf("%s", message);
    exit(EXIT_FAILURE);
}
/**
 * Creates an array of faults at specified nodes.
 */
void injectFault(fault_t* faults, int n, int node, unsigned
stuckValue)
{
    faults[n].nodeIndex = node;
    faults[n].stuckAt = stuckValue;
}
int isOutputGraph(int graphIndex)
{
    /** outputBound: graphs with index >= this bound are output graphs
*/
    int outputBound = GrpCount - OutCount;
    return (graphIndex >= outputBound) ? 1 : 0;
```

```
}
/**
 * Uses the fault array to create a faulty graph. Then it attaches the
faulty graph
 * at the #0 terminal and the inverse faulty graph at the #1 terminal
creating
 * an xor of the original graph and the faulty graph
 */
void attachFaultyGraph(fault_t* faults, int noOfFaults)
{
    // attach faulty graph
    int outputBound = GrpCount - OutCount;
    // change terminal #0 to new node
    for (int g = outputBound; g < GrpCount; ++g) {
        unsigned graphBegin = GBEG(g);
        unsigned graphLength = GLEN(g);
        int current = 0;
        // since we are adding two new graphs
        unsigned addedLength = graphLength * 2;
        // change zero terminal nodes to new node
        while (current < graphLength) {
            int nodeIndex = graphBegin + current;
            if (!NDST(nodeIndex, 0)) { // if terminal #0 link to
faulty graph
                NDST(nodeIndex, 0) = graphLength;
            }
            if (!NDST(nodeIndex, 1)) { // if terminal #1 link to
inverse faulty graph
                NDST(nodeIndex, 1) = addedLength;
            }
            ++current;
        }
        /* Re-allocate memory for new nodes */
        if(!(Nods = (node_t*) realloc(Nods, NSIZ * (NodCount +
addedLength)))) {
            exception("Out of memory: cannot reallocate nodes");
        }
        /* Re-allocate memory for new node names */
        if(!(NodNames = (char**) realloc(NodNames, sizeof(char*) *
(NodCount + addedLength)))) {
            exception("Out of memory: cannot reallocate node names");
        }
```

```
        // update count variables
        NodCount += addedLength;
        GLEN(g) += addedLength;
        // set current to new node
        current = graphLength;
        unsigned faultyGraphBound = GLEN(g) - graphLength;
        // add faulty graph
        while (current < faultyGraphBound) {
            int nodeIndex = graphBegin + current;
            int previousIndex = nodeIndex - graphLength;
            // copy previous graph node
            NFLG(nodeIndex) = NFLG(previousIndex);
            NDST(nodeIndex, 0) = NDST(previousIndex, 0);
            NDST(nodeIndex, 1) = NDST(previousIndex, 1);
            NVAR(nodeIndex) = NVAR(previousIndex);
            NNAM(nodeIndex) = NNAM(previousIndex);
            // check if node is faulty and update graph node
            for (int i = 0; i < noOfFaults; ++i) {
                if (faults[i].nodeIndex == previousIndex) {
                    // determine stuck path
                    int stuckDestination;
                    if (faults[i].stuckAt) { // stuck at 1
                        stuckDestination = INV(nodeIndex) ? 0 : 1;
                    } else { // stuck at 0
                        stuckDestination = INV(nodeIndex) ? 1 : 0;
                    }
                    // set the other branch also to stuck path
                    NDST(nodeIndex, 1 - stuckDestination) =
NDST(nodeIndex, stuckDestination);
                }
            }
            // update node destination with new index
            NDST(nodeIndex, 0) = graphLength + NDST(nodeIndex, 0);
            if (NDST(nodeIndex, 0) == faultyGraphBound ||
NDST(nodeIndex, 0) == GLEN(g)) { // set new terminal node
                NDST(nodeIndex, 0) = 0;
            }
            // since we are attaching at terminal #0 we must go to
terminal from right
            NDST(nodeIndex, 1) = graphLength + NDST(nodeIndex, 1);
```

```
            if (NDST(nodeIndex, 1) == GLEN(g) || NDST(nodeIndex, 1) ==
faultyGraphBound) {
                NDST(nodeIndex, 1) = 0;
            }
            // move to next node
            ++current;
        }
        // set current node to new node
        current = faultyGraphBound;
        // add inverted faulty graph
        while (current < GLEN(g)) {
            int nodeIndex = graphBegin + current;
            int faultyGraphIndex = nodeIndex - graphLength;
            // copy previous graph node
            NFLG(nodeIndex) = NFLG(faultyGraphIndex);
            NDST(nodeIndex, 0) = NDST(faultyGraphIndex, 0);
            NDST(nodeIndex, 1) = NDST(faultyGraphIndex, 1);
            NVAR(nodeIndex) = NVAR(faultyGraphIndex);
            NNAM(nodeIndex) = NNAM(faultyGraphIndex);
            // invert nodes and destinations
            NFLG(nodeIndex) = 1 - NFLG(nodeIndex);
            unsigned temp = NDST(nodeIndex, 0);
            NDST(nodeIndex, 0) = NDST(nodeIndex, 1);
            NDST(nodeIndex, 1) = temp;
            // skip updating terminal nodes
            NDST(nodeIndex, 0) = graphLength + NDST(nodeIndex, 0);
            if (NDST(nodeIndex, 0) == graphLength) { // set new
terminal node
                NDST(nodeIndex, 0) = 0;
            }
            NDST(nodeIndex, 1) = graphLength + NDST(nodeIndex, 1);
            if (NDST(nodeIndex, 1) == graphLength) {
                NDST(nodeIndex, 1) = 0;
            }
            // move to next node
            ++current;
        }
    }
}
/**
```

```
 * Scans all output paths to see if terminal #1 can be reached. It
returns 1 if
 * not output path terminates at #1 and 0 otherwise.
 * Effectively proving that Y xor YF = 0
 */
int isRedundantFault()
{
    // allocate a control register for all nodes
    int* controlReg;
    if (!(controlReg = (int*) malloc(VarCount * sizeof(int)))) {
        exception("Out of memory: cannot allocate int*");
    }
    int result = 1;
    // holds the value retrieved from the stack so we can invert it
    int changeNodeDirection = -1;
    stack_t* stack = createStack(VarCount);
    /** outputBound: graphs with index >= this bound are output graphs
*/
    int outputBound = GrpCount - OutCount;
    // transverse each graph
    for (int g = outputBound; g < GrpCount; g++) {
        if (!result) {
            break;
        }
        int graphBegin = GBEG(g);
        int current = 0;
        scan_path scanPath = createScanPath(NodCount);
        int scanPathCount = 0;
        char str[1000];
        // node transverse
        while (1) {
            int nodeIndex = graphBegin + current;
            int controlVariable = NVAR(nodeIndex);
            int destination, next;
            int assignedValue;
            /*printf("Before: "); printScanPath(&scanPath);
printf("\n");
            printf("Cur Index: %i, Value: %i, Fixed: %i, Toggle:
%i\n\n", nodeIndex,
                    VVALUE(controlVariable),
isControlVariableFixed(&scanPath, controlVariable),
VTOGGLE(controlVariable) );*/
```

```
            // if no value is assigned to the node control variable
            if (!isControlVariableFixed(&scanPath, controlVariable)) {
                // assign 0 if inverted and 1 if not (0 xor 1/1 xor 0)
                assignedValue = 1 - INV(nodeIndex);
                controlReg[controlVariable] = assignedValue;
                push(stack, current);
                // since we forced the value, next destination will be
to the right
                destination = 1;
                // add node index to path node
                path_node pathNode;
                pathNode.nodeIndex = nodeIndex;
                pathNode.isInverted = 1 - assignedValue;
                pathNode.controlVariable = controlVariable;
                // add path node to scan path
                scanPath.top = scanPathCount;
                scanPath.pathNodes[scanPathCount++] = pathNode;
            } else {
                // check if current node was just retrieved from stack
then we have to invert it
                if (current == changeNodeDirection) {
                    int currentValue = controlReg[controlVariable];
                    controlReg[controlVariable] = 1 - currentValue;
                    // since node changed direction, we should set
invert signal
                    int pathIndex = getPathIndex(&scanPath,
nodeIndex);
                    // if current value is 1 then a change indicates
it was inverted
                    scanPath.pathNodes[pathIndex].isInverted =
currentValue;
                }
                // determine destination node
                if (controlReg[controlVariable]) { // variable value
is 1
                    destination = INV(nodeIndex) ? 0 : 1;
                } else { // variable value is 0
                    destination = INV(nodeIndex) ? 1 : 0;
                }
            }
            next = NDST(nodeIndex, destination);
            //printf("current: %i\n", graphBegin + current);
```

```
            //printf("Stack: ");
            /*peep(stack);
            sprintf(str, "%i | %i | %i", controlVariable,
VVALUE(controlVariable), VFIXED(controlVariable));
            printf("%s\n", str);
            printf("%i --> %i\n\n", nodeIndex, graphBegin + next);
            printf("After: Cur Index: %i, Value: %i, Fixed: %i,
Toggle: %i\n\n", nodeIndex,
                    VVALUE(controlVariable),
isControlVariableFixed(&scanPath, controlVariable),
VTOGGLE(controlVariable) );*/
            if (!next) { // terminal reached
                if (debug) {
                    printScanPath(&scanPath, destination);
                    printf("\n");
                }
                if (destination) { // terminal node is 1
                    result = 0;
                    break;
                }
                if (!isEmpty(stack)) {
                    // set current as value from stack then we can
change direction
                    current = changeNodeDirection = pop(stack);
                    // node where the change in direction was done
                    int nodeIndexChanged = graphBegin + current;
                    int pathIndex = getPathIndex(&scanPath,
nodeIndexChanged);
                    // set flipped node as new top.
                    scanPath.top = pathIndex;
                    // so next path node will begin from one after the
changed index
                    scanPathCount = pathIndex + 1;
                } else {
                    break;
                }
            } else { // move to next node
                current = next;
            }
        }
        // free scan path memory
        free(scanPath.pathNodes);
```

```
    }
    // free memory
    free(controlReg);
    free(stack);
    return result;
}
```