THESIS ON INFORMATICS AND SYSTEM ENGINEERING C78

Constraints Solving Based Hierarchical Test Generation for Synchronous Sequential Circuits

TAAVI VIILUKAS

TUT
PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Engineering

Dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer and System Engineering on October 16, 2012.

Supervisor: Prof. Jaan Raik
Department of Computer Engineering
Tallinn University of Technology, Estonia

Opponents: Prof. Heinrich Theodor Vierhaus
Brandenburg University of Technology, Germany

Prof. Matteo Sonza Reorda
Politecnico di Torino, Italy

Defence of the thesis: November 30, 2012

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for any academic degree.

/Taavi Viilukas/

ISSN 1406-4731
ISBN 978-9949-23-383-0 (publication)
ISBN 978-9949-23-384-7 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C78

Kitsenduste lahendamisel baseeruv hierarhiline testigenereerimine sünkroonsetele järjestikskeemidele

TAAVI VIILUKAS

TTÜ
KIRJASTUS

To my lovely wife Kaja
and to my sweet daughters Ella and Paula

Abstract

The developments in nanotechnologies have increased the complexity of digital circuits. While the area of integrated circuits is reducing, the number of elements is increasing rapidly. The growing complexity of digital circuits has made testing one of the most complicated and time-consuming problems in system design and production. A very small manufacture defect in a wire or in a transistor element can easily result in a faulty digital system. Therefore, testing is required to guarantee fault-free products and more efficient testing methods are needed.

This thesis addresses several hierarchical automated test pattern generation techniques. The main emphasis is to increase the fault coverage with reducing the run times with respect to the state-of-the-art.

Firstly, a novel constraint-based approach for hierarchical automated test pattern generator is presented. Deterministic algorithm first activates test path constraint at register-transfer level and subsequently applies a constraint solving package ECLiPSe Prolog assembling the tests. Experimental results show that it provides increased fault coverage for hard-to-test designs with respect to semi-formal approaches and this approach offers short run times.

Secondly, a novel approach was presented combining three different fault models − hierarchical fault model for functional blocks, a functional fault model for multiplexers and a mixed hierarchical-functional fault model for comparison operators. The fault models are integrated into the fast hierarchical decision diagram based automated test pattern generation tool developed in this thesis. According to experiments, the proposed method significantly outperforms state-of-the-art test pattern generation tools. The main new contribution of this work is a formal definition of high-level decision diagram representations and the combination of the three fault models in order to target high gate-level stuck-at fault coverage for sequential cores.

Thirdly, based on the same tool, untestable faults identification method is introduced. The method is based on hierarchical approach where test path constraints extracted at the RTL are applied to proving untestable faults at the gate-level. For the first, the concept of test path constraints for testing a module in the RTL design is presented. Then the procedure of extracting a full set of test path constraints is shown. Experiments show that it is capable of generating tests yielding maximum fault efficiency for modules embedded into the RTL.

All mentioned three approaches were included into hierarchical test generation tool named Decider.

Annotatsioon

Nanotehnoloogiline areng on muutnud digitaalskeemid palju keerulisemaks. Kuigi ühelt poolt digitaalskeemide pindala väheneb, siis teisalt nendes sisalduv lülituste ehk transistoride arv suureneb, mistõttu on digitaalskeemide disainimise ja tootmise juures muutunud nende testimine üheks keerulisemaks ja aeganõudvamaks etapiks. Piisab vaid ühest defektist toodetud mikrokiibis, mille tõttu ei pruugi terve digitaalseade töötada korrektselt. Seega on äärmiselt oluline, et iga digitaalseadet testitakse peale tootmist.

Antud väitekiri käsitleb kolme hierarhilise testigenereerimise meetodit, mille põhieesmärk on suurendada testigeneraatori veakatet ning vähendada genereerimiseks kulunud tööaega.

Esiteks esitletakse uudset kitsenduste lahendamisel põhinevat hierarhilist testigenereerimise meetodit. Selles meetodis kutsutakse kõigepealt välja deterministlik algoritm, mis kirjutab välja kõik kitsendused, mis on vajalikud, et aktiveerida tee sisendist kuni testitava moodulini RTL tasemel. Seejärel kutsutakse välja leitud kitsenduste lahendamiseks ECLiPSe Prolog ning saadud vastuseid kasutades genereeritakse moodulile madalal tasemel test. Katsetulemused näitavad, et selline uudne meetod annab suurema veakatte eelkõige raskesti testitavatele digitaalskeemidele ning lühendab testigenereerimiseks kulunud aega.

Teiseks esitletakse uudset meetodit, kus on ühendatud kolm erinevat rikete mudelit: hierarhiline rikete mudel funktsionaalsetele moodulitele, funktsionaalne rikete mudel multiplekserite jaoks ja kombineeritud hierarhiline-funktsionaalne rikete mudel võrdluste moodulitele. Rikete mudelid on integreeritud kõrgtaseme otsustusdiagrammidesse. Vastavalt katsetulemustele pakutakse välja oluliselt efektiivsem uudne meetod kui nüüdisaegsed testigenereerimise vahendid. Peamine uudsus antud töös on formaalne määratlus nende kolme rikkemudeli kombinatsioonis kõrgetaseme otsustusdiagrammides.

Kolmandaks esitatakse kitsendustel põhineva mittetestitavate rikete tuvastamise meetod. Meetod põhineb samuti hierarhilisel testigenereerimise lähenemisviisil, kus leitakse tee testitava moodulini registersiirde tasemel ning antud teed kasutatakse mittetestitavate vigade tõestuseks loogikalülituste tasemel. Esiteks esitatakse registersiirde disainis testitava mooduli tee aktiveerimine ning seejärel antud tee kitsenduste väljatoomise algoritm. Katsetulemused

näitavad, et antud meetod on võimeline genereerima registersiirde tasemel maksimaalseid tulemusi integreeritud moodulitele.

Kõik kolm esitletud meetodit integreeriti hierarhilisse testigeneraatorisse Decider.

Acknowledgements

I would like to show appreciation to everybody who helped me with advice and support during my PhD studies. My special thanks goes to my supervisor, Professor Jaan Raik, who greatly supported me during my research and showed a lot of patience during the long process of my PhD studies.

I also would like to acknowledge the organizations that have supported my PhD studies − Tallinn University of Technology, Estonian IT Foundation (EITSA), Enterprise Estonia funded ELIKO Development Centre, National Graduate School for Information and Communication Engineering (IKTDK), European Commission's FP6 research project VERTIGO, European Commission's FP7 research project DIAMOND and the European Science Foundation (ESF COST action IC0901).

Finally I would like to warmly thank my family for their patience and support.

Thank you all!

List of Publications

Journals

1. Taavi Viilukas, Anton Karputkin, Jaan Raik, Maksim Jenihhin, Raimund Ubar and Hideo Fujiwara (2012). Identifying Untestable Faults in Sequential Circuits Using Test Path Constraints. *Journal of Electronic Testing: Theory and Applications*, Springer, Volume 28, Issue 4 (2012), pp. 511 – 521.

2. Raik, J.; Ubar, R.; Viilukas, T.; Jenihhin, M. (2008). Mixed hierarchical-functional fault models for targeting sequential cores. *Journal of Systems Architecture*, 54(3-4), pp. 465 – 477.

3. Jenihhin, Maksim; Raik, Jaan; Ubar, Raimund; Viilukas, Taavi; Fujiwara, Hideo (2011). An Approach for Verification Assertions Reuse in RTL Test Pattern Generation. *J. of Shanghai Normal University (Natural Sciences),* Vol. 39(No 5), pp. 441 – 447.

Conferences

4. Drenkhan, Taavi; Tšepurov, Anton; Viilukas, Taavi; Raik, Jaan; Karputkin, Anton; Jenihhin, Maksim; Ubar, Raimund (2012). Generating Directed Tests for C Programs using RTL ATPG. *Proceedings of the IEEE 11th Workshop on RTL and High Level Testing (WRTLT'12),* November 22-23, 2012, Niigata, Japan.

5. Raik, J; Rannaste, A; Jenihhin, M; Viilukas, T; Fujiwara, H; Ubar, R (2011). Constraint-Based Hierarchical Untestability Identification for Synchronous Sequential Circuits. *Proceedings of IEEE European Test Symposium*, (1 - 6). IEEE Computer Society Press

6. Viilukas, Taavi; Jenihhin, Maksim; Raik, Jaan; Ubar, Raimund; Baranov, Samary (2011). Automated Test Bench Generation for High-Level Synthesis flow ABELITE. *Proceedings of IEEE East-West Design & Test Symposium 2011* (1 - 6). Sevastopol, Ukraine: IEEE Computer Society

7. Jenihhin, M.; Raik, J.; Fujiwara, H.; Ubar, R.; Viilukas, T. (2011). An Approach for Verification Assertions Reuse in RTL Test Pattern Generation. In: *Proceedings of the IEEE 11th Workshop on RTL and High Level Testing (WRTLT'10)*, Shanghai, China, December 5-6, 2010.

8. Viilukas, Taavi; Raik, Jaan; Jenihhin, Maksim; Ubar, Raimund; Krivenko, Anna (2010). Constraint-based Test Pattern Generation at the Register-Transfer Level. In: *Proceedings of the 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems* : April 14–16, 2010 Vienna, Austria: IEEE, 2010, pp. 352 – 357.

9. Chepurov, Anton; Di Guglielmo, Giuseppe; Fummi, Franco; Pravadelli, Graziano; Raik, Jaan; Ubar, Raimund; Viilukas, Taavi (2008). Automatic Generation of EFSMs and HLDDs for Functional ATPG. *11th Biennial Baltic Electronics Conference* (pp. 143 – 146).IEEE

10. Raik, Jaan; Ubar, Raimund; Viilukas, Taavi (2006). High-Level Decision Diagram based Fault Models for Targeting FSMs. In: *Proceedings of the 9th IEEE Euromicro Conference on Digital Systems Design* : DSD2006, Cavtat, Aug. 31- Sept. 2, 2006. IEEE Computer Society Press, 2006, pp. 353 – 358.

Doctoral schools

11. Taavi Viilukas, Jaan Raik, Using Test Pattern Generation Tool Decider in Hardware Verification, *2nd IKTDK Conference*, Viinistu, Estoia 2007.

12. Taavi Viilukas, Jaan Raik, Using constraint solver in Test Pattern Generation Tool, *3th IKTDK Conference*, Voore Resort Center, Estonia, 2008, pp. 14 – 17.

13. Taavi Viilukas, Jaan Raik, Automated Test Pattern Generator with Constraint Solver, *4th IKTDK Conference*, Essu mois, Estonia, 2010, pp. 33 – 36.

14. Taavi Viilukas, Jaan Raik, Raimund Ubar, Anna Rannaste, Maksim Jenihhin, Hideo Fujiwara, Constraint-Based Hierarchical Untestability Identification for Syncronous Sequential Circuits, *5th IKTDK Conference*, Nelijärve resort, Estonia, 2011, pp. 139 – 142.

15. Taavi Viilukas, Approaches to improve hierarchical ATPG for synchronous sequential circuits, 6*th IKTDK Conference,* Laulasmaa, Estonia, 2012, pp. 105 – 108.

List of Abbreviations

ADD	Assignment Decision Diagram
ADN	Assignment Decision Node
ATE	Automated Test Equipment
ATPG	Automated Test Pattern Generation
BDD	Binary Decision Diagram
libIC	Interval Constraints library
CLP	Constraint Logic Programming
CUT	Circuit Under Test
DFT	Design for Testability
DIFFEQ	Differential Equation
FFR	Fanout-Free Region
FSM	Finite State Machine
FU	Functional Unit
GCD	Greatest Common Divisor
HLDD	High-Level Decision Diagrams
IC	Integrated Circuit
MULT8x8	8-Bit Sequential Multiplier
MUT	Module Under Test
MOT	Multiple Observation Time
PI	Primary Input
PO	Primary Output
QFBV	Quantifier-Free Bitvector
RISC	ALU based processor
RT	Register-Transfer
RTL	Register-Transfer Level

SAF	Stuck-at-Fault
SOT	Single Observation Time
s-a-0	Stuck-at-0
s-a-1	Stuck-at-1
SSA	Single Stuck-At
VLSI	Very Large Scale Integrated
SDC	Synopsys Design Compiler
ELLIPF	Elliptic Filter
FUTEG	Functional Test Generation

Table of Contents

1 Introduction

This thesis addresses several hierarchical automated test pattern generation improvements. The main emphasis is to increase the fault coverage while reducing the working time.

This chapter presents the motivation behind the presented work, followed by a more detailed problem formulation. This is followed by a summary of the main contributions and an overview of the thesis structure.

1.1 Motivation

Digital circuits become more important in everyday life by controlling very complex systems in which different subsystems intertwine with each other. In such systems, it is very important that all parts function as expected. A simple error in subsystem may propagate through the whole system and affect other modules. For example, in 1997 USS cruiser Yorktown's main power was shut down for about three hours, because of an error in the kitchen inventory application. An arithmetic exception was propagated through the system until it stopped the main power [1]. Or an even more dramatic example from the year 2008 where 154 persons were killed in a civil airplane crash. Due to a fault in the central computer system, no alarm was raised over multiple problems on the plane [2].

As known, the complexity of digital circuits has been increasing very rapidly. It is following the so-called Moore's law [3], which says that the scale of Integrated Circuits (IC) has doubled every 18 months. While a microprocessor in 1989 contained only one million transistors, already in 2005 the number of transistors had raised to more than one billion [4]. Today's architecture feature size is 22nm or less compared with 1μm in 1986. It means the feature size has reduced 45 times! Moreover, the operating frequencies have been increasing dramatically too – Intel 4004 microprocessor ran at 108 KHz in 1971 [5], while current commercially available microprocessors commonly run at several gigahertzes.

Increasing complexity of digital circuits has made testing one of the most complicated and time-consuming problems in system design and production. It

is estimated that 70 % of the design cycle for systems is spent on tests and verification [6]. A very small manufacturing defect, for example, open or bridge, can easily result in a faulty IC [7]. It takes only one faulty transistor or wire to make the entire chip function improperly. Defects that were created during the manufacturing process are unavoidable and a certain number of ICs is expected to be faulty. Therefore, testing is required to guarantee fault-free products, regardless of whether the product is a Very Large Scale Integrated (VLSI) circuit's device or an electronic system composed of many VLSI devices.

When the time for testing grows rapidly, more efficient testing methods are needed. Testing means the checking of circuits for manufacturing correctness for each produced device. There also exists a functional verification to check the correctness of functionality but this is not considered in this research. The keyword for testing is how to generate effective test vectors. The goal is getting a minimum number of test vectors that cover 100 % of faults with a minimum time. The quality of testing relies on algorithms that generate test vectors. For combinational circuit, a method that guarantees 100 % fault detection was approached in 1990s. As a result, Automated Test Pattern Generation (ATPG) for combinational logic is no longer a problem [7].

Test generation for sequential synchronous designs is still a difficult and time-consuming task. A state sequence must be traversed so that it is possible to propagate the fault effect to a primary circuit output. The fault must be observed and detected. Ideally, testing should be fast and reliable. However, in reality test generation times are too long for complex circuits and the achieved fault coverage is unsatisfactory. Several approaches to generating tests for stuck-at faults in sequential circuits have been proposed over the years. For example, at the gate-level deterministic methods [8] [9] are not able to efficiently handle sequential designs with more than a couple of thousands of gates. But simulation-based approaches [10] [11] [12] do not guarantee detection for hard-to-test random pattern resistant faults. And functional test [13] [14] generation does not offer full structural level fault coverage. Still, no satisfactory solution for sequential circuit test generation has been proposed and the problem still lacks a breakthrough.

1.2 Problem formulation

As mentioned above, a lot of different methods have been implemented to generate test vectors for sequential circuits, but none of these give a solution to reach 100 % fault coverage in sequential circuits. In sequential circuits, there are memory elements and feedback loops and these make test pattern generation difficult.

In generating tests for synchronous sequential circuits, a state sequence must be traversed to propagate the fault effect to a primary circuit output and initialize inputs to activate the test path to the Module Under Test (MUT).

During that state traversing, different constraints are created and updated. Created constraints may be very long and hard to solve. To speed up test generator time and increase the fault coverage it is important to solve those constraints quickly and correctly.

Because it is hard to generate tests for real defects, fault models are used. A fault model is a mathematical description of how a defect alters the design behaviour. A good fault model must satisfy two criteria – it should accurately reflect the behaviour of the defect and it should be computationally efficient for the fault simulation and the test pattern generation. No single fault model accurately reflects the behaviour of all possible defects that can occur.

When generating tests, two criteria must be satisfied − the fault must be controllable and observable. If there is a fault and those two criteria cannot be satisfied, then there is an untestable fault. Untestable fault is a fault for which no test exists. Identifying untestable faults in sequential synchronous circuits remains unsolved. ATPG tools spend a lot of effort not only for deriving test vectors for testable faults but also for proving that there exist no tests for the untestable faults. Because of the reason mentioned above, the identification of untestable faults has been an important aspect in speeding up the sequential ATPG.

This thesis is addressing the challenges mentioned above.

1.3 Contributions

The current thesis introduces several approaches to improve hierarchical ATPG for synchronous sequential circuit fault coverage and speed. The main contributions of this thesis are summarised as follows:

- A new deterministic algorithm that extracts constraints for activating test paths at Register-Transfer Level (RTL) and subsequently applies a constraint solving package ECLiPSe assembling the tests in hierarchical ATPG. The current thesis is focused to constraint based ATPG on Chapter 5.

- A new type of fault model based on High-Level Decision Diagrams (HLDD) dedicated to the faults in FSMs embedded into RTL description. The novel contribution of this approach is a formal definition of high-level decision diagram representations and the combination of the three fault models in order to provide a high fault coverage testing of sequential cores. The current thesis Chapter 6 is focused to new type of fault model.

- A new hierarchical untestability identification method for non-scan sequential circuits containing feedback loops is presented. The method is based on extracting and minimizing RTL test path activation constraints that drive a dedicated logic-level deterministic ATPG. The current thesis is focused to improve hierarchical untestability identification on Chapter 7.

All three mentioned approaches were included into top-down hierarchical test generation tool named Decider and are ready for using. Decider is worked out by Department of Computer Engineering by Tallinn University of Technology.

1.4 Thesis organization

The presented thesis consists of eight chapters. It is organized as follows.

Chapter two provides background information required for the discussion of further proposed approaches. Basic definitions and different aspects of digital circuits and testing are introduced in that chapter.

In chapter three automatic test pattern generation and fault simulation overview are given.

Chapter four gives an overview of RTL and high-level decision diagram models. In addition assignment decision diagram is introduced.

In chapter five, constraints based ATPG is presented. It starts with previous work after which the concept of path activation constraints is shown. Then fault effect propagation and constraint justification are shown with an example. Finally, constraints solving methods are presented. The chapter ends with experimental results and conclusions.

In chapter six, high-level decision diagram based fault models are presented. This chapter starts with previous work after which a mixed functional-hierarchical fault model is presented.

In chapter seven, identifying untestable faults in sequential circuits is presented. This chapter starts with previous work in identifying untestable faults in sequential circuits where different methods are shown. It continues with the untestability proof flow. Then, test path constraints extraction and minimization are presented. Finally, constraint-driven ATPG for proving untestability is shown. The chapter ends with experimental results and conclusions.

Chapter eight draws conclusions of this thesis.

2 Fault modelling

This chapter presents basic background knowledge that is necessary to understand the related topic of current research. Starting at the very beginning − what digital circuits are and how to divide them into different classes, should guarantee that a reader is able to track this thesis. After that, elementary knowledge about the digital testing and definitions is presented. And finally, the stuck-at-fault model is introduced.

2.1 Digital circuits

A digital circuits can be defined generally as an interconnection of Boolean logic elements such as AND (&) gates, OR (V) gates and INVERTERS and combinations of these elements and it must be able to process a set of discrete and finite-valued electrical signals. The opposite of the digital circuits is analog devices where only continuous time voltages and currents are used. These devices are not considered in this work. Digital circuits may be classified as combinational or sequential. For example, Figure 1 presents the combinational circuit (A) and sequential circuit (B).

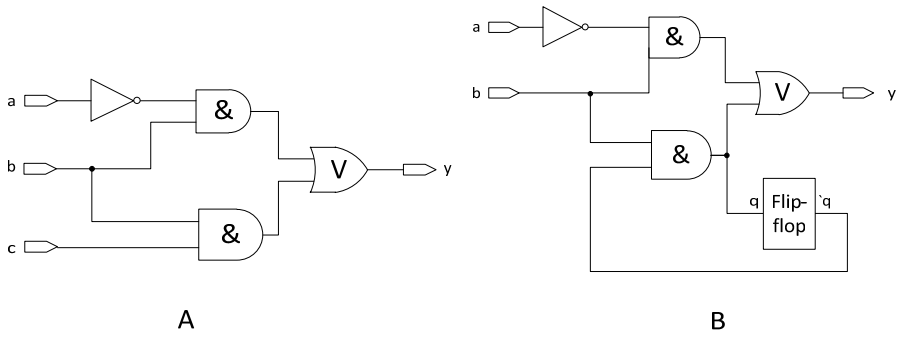


Figure 1. Combinational (A) and sequential (B) circuit example.

In a combinational circuit, the present outputs depend only on present inputs. In a sequential circuit the present outputs depend not only on present inputs, but also on past inputs, i.e. the circuit state. This is because a sequential circuit consists not only of the combinational part, but also of memory elements like flip-flops and registers holding the circuit state. A sequential circuit also contains feedback loops.

Those two classes, combinational and sequential circuits, have different topologies. The first difference between them is, as mentioned earlier, that the combinational circuit does not contain loops. Because of that combinational circuits do not have states like sequential circuits have. The second one is from testing, proposing that a test for a fault in a sequential circuit may consist of several vectors while a combinational circuit to test a fault is a single vector need.

When a combinational circuit does not contain redundant logic, the device may be tested by applying all possible 2^n input patterns for stuck-at faults, where 'n' is the number of inputs. This testing is called exhaustive testing and it is suitable only for small circuits. When the number of inputs gets higher, the exhaustive testing time will grow rapidly. In sequential circuits, it is not possible to use exhaustive testing because that type of testing does not guarantee that all possible states will be covered.

Sequential digital circuits may be further classified as asynchronous or synchronous. A synchronous circuit is a digital circuit in which the parts are synchronized by a clock signal. In contrast to a synchronous, an asynchronous circuit is not governed by a clock signal. Instead they often use signals that indicate the completion of instructions and operations, specified by simple data transfer protocols. This work addresses only synchronous sequential circuits.

2.2 Importance of testing digital circuits

In test generation, the terms ‘defect’, ‘error’ and ‘fault’ are used very often. A **defect** in an electronic system is the unintended difference between the implemented hardware and its intended design. Defects occur either during manufacture or during the use of devices. A wrong output signal produced by a defective system is called an **error**. A **fault** is a mathematical representation of a defect reflecting a physical condition that causes a circuit to fail to perform in a required manner.

When a digital sequential circuit is fabricated, it is not possible to guarantee that all produced circuits work as expected. This is because of material defects. Also, some device behaviour may change during the time because of material changing. To make sure that a fabricated circuit works as the design specification intends, we need to test it. Testing consists of two different processes – test generation and test application.

In the test generation process, test patterns are produced for efficient testing. A test pattern or sequence of input pattern is also called test vectors or the sequence of test vectors. They produce a different output in a faulty circuit compared with the fault-free circuit. The goal of the test generation is to find an efficient set of test vectors that detects all faults considered for that circuit.

Test application is performed by the Automated Test Equipment (ATE). The purpose of this process is to apply test patterns to the Circuit Under Test (CUT) and analyse the output responses with known good results. Circuits that produce correct output responses for all input stimuli pass the test and are considered to be fault-free. Those circuits that fail to produce a correct response at any point during the test sequence are assumed to be faulty. Figure 2 depicts a digital testing process flow.

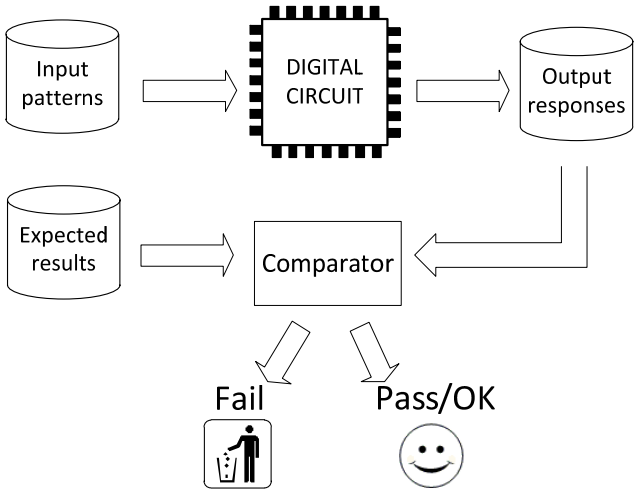


Figure 2. Digital circuits test process flow

Quality and economy are the benefits of testing. It means that digital circuits are tested within an acceptable time frame with minimum costs while satisfying the user's needs. For example, testing all possible stuck-at faults in sequential circuits with only 32 inputs and 100 flip-flops with 2 gigahertz will take 86 322 264 566 448 100 000 000 years. It is obvious that this is not a solution and circuits need to be tested within a reasonable time frame.

$$\text{All SSA test time, in seconds} = \frac{2^{(\text{number of inputs})} \times 2^{(\text{number of flip-flops})}}{\text{test operating frequence}}$$

To measure the quality of a test, fault coverage is used. Fault coverage can be defined as the following ratio:

$$\text{Fault coverage} = \frac{\text{Number of detected faults}}{\text{Total number of faults}}$$

It may be impossible to obtain fault coverage of 100 % because of the existence of undetectable faults in sequential circuits. An untestable fault means there is no test to distinguish the fault-free circuit from a faulty circuit containing that fault. As a result, the fault coverage can be modified and expressed as the fault efficiency, also referred to as the effective fault coverage, which is defined as:

$$\text{Fault efficiency} = \frac{\text{Number of detected faults}}{\text{Total number of faults} - \text{number of untestable faults}}$$

In order to calculate the efficiency of fault detection, let alone reach a 100 % fault coverage, all of the undetectable faults in the circuit must be correctly identified, which is a difficult task [7].

2.3 Yield and Reject Rate

Because it is not possible to produce a 100 % good IC, some percentage of the manufactured ICs are expected to be faulty due to manufacturing defects. The yield of a manufacturing process is defined as the percentage of acceptable parts among all parts that are fabricated:

$$\text{Yield} = \frac{\text{Number of acceptable parts}}{\text{Total number of parts fabricated}}$$

There are two types of yield loss: catastrophic and parametric. Catastrophic yield loss is due to random defects, and parametric yield loss is due to process variations. Parametric variations due to the process fluctuations become the dominant reason for yield loss.

When ICs are tested, the following two undesirable situations may occur:

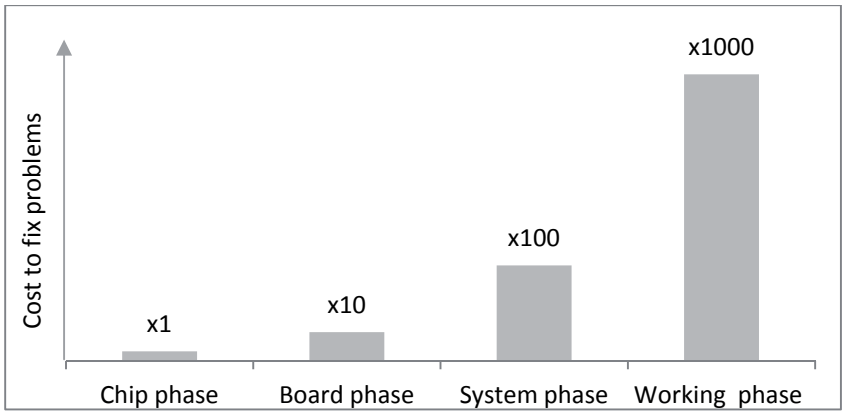
1. A faulty device appears to be a good part passing the test (under-testing).
2. A good device fails the test and appears as faulty (over-testing).

First outcomes are often due to a poorly designed test or the lack of design for testability (DFT). Second outcomes may occure when DFT is used.

Poorly designed test is a test that does not detect faults in the manufactured circuits and affect to the device quality. Moreover, digital circuits have different phases during its lifetime and in each phase the device should be tested to ensure its quality. The following phases can be recognized: chip manufacture, board manufacture, system manufacture, and working phase of a product. For example if a chip fault is not caught by chip phase testing, then finding the fault costs 10 times as much at the board level as at the chip phase. Similarly, if a board fault is not caught by board level testing, then finding the fault costs 10 times as much at the system level as at the board phase. The relationship of the test and repair costs during each of these phases can be

approximated with the rule-of-ten [15]. Rule of then is illustrated in Table 1. If the test and repair cost in the component manufacturing phase is R, then in the board manufacture phase it is 10R, in the system manufacturing phase it is 100R, and during the working phase it is 1000R. This is due to the increase in the difficulty level of locating the faulty part, the increase in repair effort and the larger volume of units involved. [16]

Table 1. Cost of fixing defect at different stages of the product phases



DFT is design techniques for sequential circuits that add controllability and observability to a hardware design that makes test generation and test application cost-effective. Controllability reflect how difficult is to set a signal line to a required logic value from primary inputs and observability reflects, how difficult is propagate the logic value of the signal line to primary outputs.

There are three main types of DFT approaches for digital circuits:

- ad-hoc techniques
- scan design techniques
- built-in self-test

While DFT techniques are generally used in order to reduce test generation complexity, they may induce over-testing problems. DFT makes a large number of untestable faults testable. Overtesting occurs when a defect that would not be detected under functional operation conditions of a chip, is detected due to non-functional conditions created during test application. The over-testing causes yield loss because good circuits in normal operations may be regarded as faulty ones under the test mode. Moreover, over-testing reduce test generation time. Furthermore, identification of untestable faults could avoiding over-testing and reduce yield loss. [17]

If all products pass acceptance test, some faulty devices will still be found in the manufactured electronic system. The ratio of field-rejected parts to all

parts passing quality assurance testing, is referred to as the reject rate, also called the defect level:

$$\text{Reject rate} = \frac{\text{Number of faulty parts passing final test}}{\text{Total number of parts passing final test}}$$

The reject rate provides an indication of the overall quality of the VLSI testing process [7]. The highest quality refers to the product meeting its requirements at lowest possible cost. In a manufacturing process, both criteria, meeting the requirements and reducing cost, determine the quality. Testing checks conformance to requirements. Therefore, a cost tradeoff is often necessary. A test that can reduce the number of outgoing faulty parts to an acceptably small value can be considered as a good test, especially if the test cost is also acceptable.

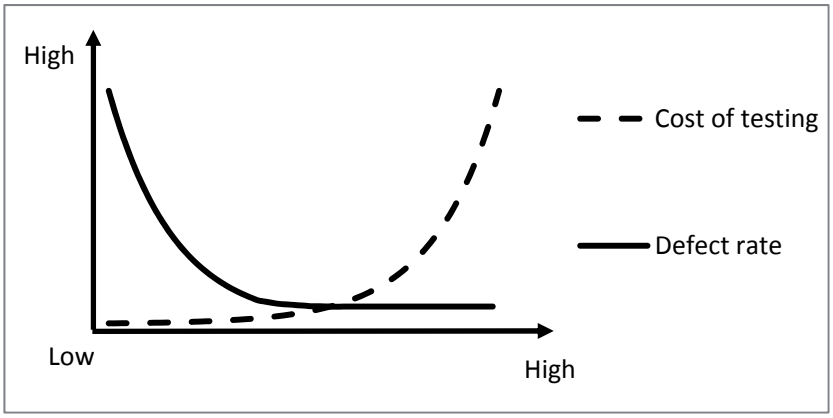


Figure 3. Cost of testing

2.4 Fault Models

It is difficult to generate tests for real defects. For this reason, fault models are used. A fault model could be defined as a mathematical description of how a defect alters design behaviour. Fault models are necessary for generating and evaluating a set of test vectors. Generally, a good fault model should satisfy two criteria:

1) It should accurately reflect the behaviour of defects.

2) It should be computationally efficient in terms of fault simulation and test pattern generation.

Many fault models have been proposed [7], but, unfortunately, no single fault model accurately reflects the behaviour of all possible defects that can occur. Here, in this thesis, only suck-at faults are considered. Other fault models such as transistor faults, open and short faults, delay faults and other fault models are not considered in this work.

The stuck-at fault is a logical fault model that has been used successfully for decades and it has been the industrial standard since 1959. A stuck-at fault affects the state of logic signals on lines in a logic circuit, including the Primary Inputs (PIs), Primary Outputs (POs), internal gate inputs and outputs, fan-out stems (sources), and fan-out branches. The stuck-at fault model assumes that the elementary components are fault-free. A stuck-at fault transforms the correct value on the faulty signal line to appear to be stuck at a constant logic value, either logic 0 or logic 1, referred to as stuck-at-0 (SA0) or stuck-at-1 (SA1), respectively. To generate a test for line SA0, we need to find a vector of PIs that sets signal on that line to 1 so that some primary output differs between the good circuit and the faulty circuit.

There can be several stuck-at faults simultaneously present in a circuit. A circuit with n lines can have 3^n-1 possible stuck combinations [7]. This is because each line can be in one of the three states:

1. SA1
2. SA0
3. Fault-free.

The circuit is fault-free only then when all lines are identified as fault-free. All other combinations mean that the circuit is faulty, because even one faulty line could cause a catastrophic result on the circuit operation. Already a small number of n will give an enormously large number of multiple stuck-at faults. Therefore, it is common practice to model only one single stuck-at (SSA) fault, so an n-line circuit can have at most 2n SSA faults.

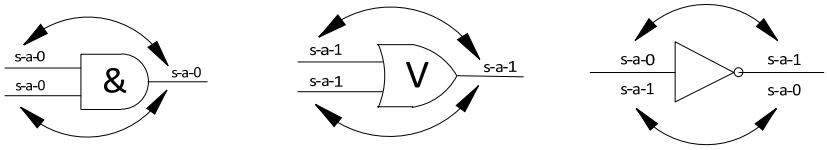
The stuck-at fault model is the most often used fault model in ATPG systems. In SSA fault model, the following presumption is made − only one single and permanent fault is considered at the time. There are three properties that characterize a single stuck-at fault model:

- Only one line is faulty.
- The faulty line is permanently set either logical 1 or logical 0.
- The fault can be assumed at an input or an output of the gate [18]

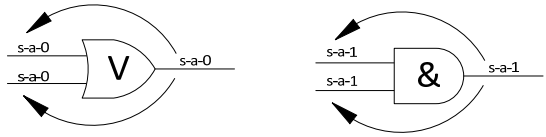
The stuck-at fault model is a logical fault model because no delay information is associated with the fault definition. It is also called a permanent fault model because the faulty effect is assumed to be permanent, in contrast to

intermittent faults which occur (seemingly) at random and transient faults which occur sporadically, perhaps depending on operating conditions (e.g., temperature, power supply voltage) or on the data values (high or low voltage states) on surrounding signal lines.

The number of SSA faults is further reduced by technique called fault collapsing [18]. Traditionally, this is done by implementing two types of relations on the set of faults: fault equivalence and fault dominance. Faults f1 and f2 are said to be equivalent if any test that detects f1 detects f2 and vice versa, any test detecting f2 covers also the fault f1. It is said that fault f1 dominates f2, if any test that detects f2 will also detect f1. Note that the equivalence relationship is symmetrical while the dominance is not. Equivalence relations between stuck-at faults for basic Boolean gates are presented in Figure 4A and the dominance is explained in Figure 4B. Typically, fault collapsing reduces the total number of faults by 50 to 60 % [7] [19].



A. Equivalence relationships of stuck-at faults for basic logic gates



B. Dominance relationships of stuck-at faults for basic logic gates

Figure 4. Fault collapsing for Boolean gates

There are functional and structural tests. In functional testing every entry in the truth table for the combinational logic circuit is tested to determine whether it produces the correct response. For example, to test an adder with 129 inputs and 65 outputs it needed 2^{219} input patterns that produce 2^{219} output responses [18]. ATE that operates at 1 GHz would take $2{,}1580566142 \times 10^{22}$ years to apply all of these patterns to the circuit-under-test (CUT). In structural testing, only selected specific test patterns are used based by circuit structural information and fault models. The total number of test patterns decreases

because the test vectors target specific faults. For testing the above example adder, only 1728 test patterns will be needed and ATE with working the same frequency would apply these patterns in 0.000001728 s and it achieves exactly the same fault coverage as the intractable functional test-pattern set described above.

In Figure 5, one simple circuit example is shown. There are nine signal lines representing potential stuck-at faults. They are labelled alphabetically − A, B, C, D, E, F, G, H and I. For exhaustive testing, eight test vectors will be needed (2^n for all possible input pattern, where n is the number of inputs). Table 2 gives the truth tables for observable stuck-at faults. All the entries where the faulty circuit produces an output response different from fault-free circuit are marked as 1 and highlighted in grey in Table 2. Let us look closely at a two different scenarios generating test on potential stuck-at fault B. First, let us use vector 000 for testing. If line B is faulty and it is stuck-at-0, the line will be spread 0 instead of 1. Despite of stuck-at-0 on the point of B the primary output will get the correct response 0 and the fault will not be discovered. In that scenario the fault is not observable in the primary output and it is marked as 0 in the truth table. Secondly, let us use vector 010 for testing. If line B is faulty and it is stuck-at-0, the primary output will get a faulty response 1 (the correct value in fault-free circuit should be 0) and the fault will be discovered. As a result, the input values for the highlighted truth table entries represent valid test vectors to detect the associated stuck-at faults. The circuit in Figure 5 can be tested with 100 % coverage with four test vectors – 001, 010, 110 and 111.

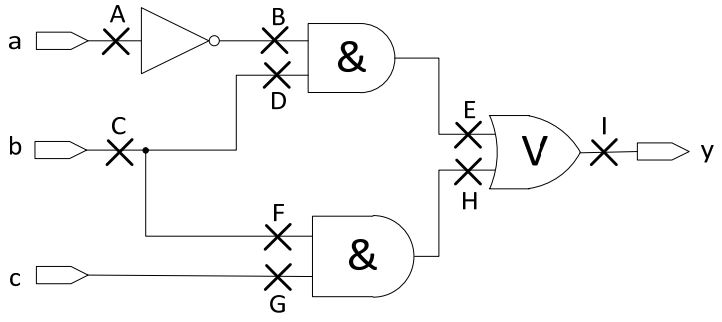


Figure 5. Potential stuck-at fault locations

Table 2. Truth tables for observable stuck-at faults for Figure 5

SAF	000	001	010	011	100	101	110	111
A s-a-0	0	0	0	0	0	0	1	0
A s-a-1	0	0	1	0	0	0	0	0
B s-a-0	0	0	1	0	0	0	0	0
B s-a-1	0	0	0	0	0	0	1	0
C s-a-0	0	0	1	1	0	0	0	1
C s-a-1	1	1	0	0	0	1	0	0
D s-a-0	0	0	1	0	0	0	0	0
D s-a-1	1	1	0	0	0	0	0	0
E s-a-0	0	0	1	0	0	0	0	0
E s-a-1	1	1	0	0	1	1	1	0
F s-a-0	0	0	0	0	0	0	0	1
F s-a-1	0	1	0	0	0	1	0	0
G s-a-0	0	0	0	0	0	0	0	1
G s-a-1	0	0	0	0	0	0	1	0
H s-a-0	0	0	0	0	0	0	0	1
H s-a-1	1	1	0	0	1	1	1	0
I s-a-0	0	0	1	1	0	0	0	1
I s-a-1	1	1	0	0	1	1	1	0

3 Test of sequential circuits

This chapter presents automatic test pattern generator methods and algorithms. Many challenges that exist in this area include reduction of the time and memory with obtaining high fault coverage. Secondly the task of fault simulation, which determines how many of the potential faults are detected with ATPG, is explained.

3.1 Automatic test pattern generation

Testing sequential circuits is more difficult than testing combinational circuits. For example fault propagation and activation is complicated because of the presence of memory elements and feedback paths. To detect a fault, a test sequence is usually required, rather than a single input vector, and the response of a sequential circuit is a function of its initial state [20].

ATPG for sequential circuits is the application of algorithmic-based software to generate sequence of input test patterns that can be used for the purpose of testing a manufactured circuit for defects. [7]. Many challenges existing in this area include reduction in time and memory required to generate the tests, reduction in the number of cycles needed to apply the tests to the circuit, and obtaining high fault coverage. Adding to the complexity of this problem is that an untestable fault is not necessarily redundant in a sequential circuit [16].

General form for fault detection in sequential circuits is like follow. Let T be a test sequence and $R(q,T)$ be the response to T of a sequential circuit N starting in the initial state q. Now consider the circuit N_f obtained in the presence of the fault f. Similarly we denote $R_f(q_f,T)$ the response of N_f to T starting in the initial state q_f. [20]

Definition 1: A test sequence T strongly detects the fault f if the output sequences $R(q,T)$ and $R_f(q_f,T)$ are different for every possible pair of initial states q and q_f [16] [20].

Definition 2: A test sequence T detect the fault f if, for every possible pair of initial states q and q_f, the output sequences $R(q,T)$ and $R_f(q_f,T)$ are different for some specified vector $t_i \in T$ [16] [20].

Two faults f and g are said to be strongly functionally equivalent if the corresponding sequential circuits N_f and N_g have equivalent state tables [16].

The discovered tests are applied then to the CUT, using the ATE, and the output responses are compared with expected results. Circuits that produce the correct output responses for all input stimuli pass the test and are considered to be fault-free. Those circuits that fail to produce a correct response at any point during the test sequence are assumed to be faulty. The traditional goal of the ATPG is to achieve a high fault coverage by producing a small volume of test patterns [21].

The effectiveness of the ATPG is measured by the amount of detected faults and the number of generated patterns. High quality is getting more fault detection in short times with fewer patterns. ATPG efficiency is influenced by the fault model, the type of circuit, the level of abstraction and the required test quality.

The ATPG process for a targeted fault consists of two phases: fault activation and fault propagation. Fault activation establishes a signal value at the fault model site that is opposite to the value produced by the fault model. Fault propagation moves the resulting signal value, or fault effect, forwarded by sensitizing a path from the fault site to a primary output.

The ATPG can fail to find a test for a particular fault in at least two cases. Firstly, the fault may be intrinsically undetectable, such that no patterns exist that can detect that particular fault. The classic example of this is a redundant circuit, designed so that no single fault causes the output to change. In such a circuit, any single fault will be inherently undetectable. Secondly, it is possible that patterns exist but the algorithm cannot find them. Since the ATPG problem is NP-complete there will be cases where patterns exist but the ATPG gives up since it will take too long time to find them. NP-complete is a complexity class of decision problems when any given solution to the decision problem can be verified in polynomial time.

The sequential-circuit ATPG searches for a sequence of vectors to detect a particular fault through the space of all possible vector sequences. Various search strategies and heuristics have been devised to find a shorter sequence and/or to find a sequence faster.

In the past several decades, the most popular fault model used in practice was the SSA fault model. Even a simple stuck-at fault requires a sequence of vectors for detection in a sequential circuit. Also, due to the presence of memory elements, the controllability and observability of the internal signals in a sequential circuit are in general much more difficult than those in a combinational logic circuit. These factors make the complexity of the sequential ATPG much higher than that of the combinational ATPG.

Sequential circuits generally have two modes of operation: synchronization mode and free mode. In the synchronization mode, the

operation always starts with a specified input sequence. If hardware reset is available as a special input, which is always employed to reset the circuit at the beginning of operation. Under the free mode of operation, no synchronization is done, and the sequential circuit starts operating from whatever state it happens to be in at the time. Two test strategies are known, corresponding to the operation modes defined above: restricted and unrestricted. Under the restricted test strategy, all test sequences start with some certain sequence. Under the unrestricted test strategy, any sequence can be generated as a test sequence.

Both of the test strategies above can be used under one of two test generation approaches: single observation time (SOT) and multiple observation time (MOT). Under the single observation time (SOT) approach, a fault *f* is said to be detectable if there exists an input sequence *I* such that for every pair of initial states S and S_f of the fault-free and faulty circuits, respectively, the response z(I, S) of the fault-free circuit to *I* is different from the response $z_f(I, S_f)$ of the faulty circuit at a specific time unit j. Under the multiple observation time (MOT) approach, a fault *f* is said to be detectable if there exists an input sequence *I* such that for every pair of initial states S and S_f of the fault-free and faulty circuits, respectively, z(I, S) is different from $z_f(I, S_f)$ at some time unit. A fault is said to be undetectable if it is not detectable under the specified test approach [16].

Every fault that is detectable under the SOT approach is also detectable under the MOT approach. In fact, a fault may be undetectable under the SOT approach, but may be detectable under the MOT approach [16].

3.1.1 ATPG algorithms for synchronous sequential circuit

The first complete ATPG algorithm for combinational circuits was the D-algorithm. It was published by J. Roth in 1966 [22]. The D-algorithm uses a logical value to represent both the “good” and the “faulty” circuit values simultaneously and can generate a test for any stuck-at fault, as long as a test for that fault exists. The next landmark effort in combinational circuit ATPG was the PODEM algorithm [23], which searches the circuit’s primary input space based on simulation to enhance computational efficiency. Since then, ATPG for combinational circuits has become an important topic for research and development, many improvements have been proposed, and many commercial ATPG tools have appeared. For example, FAN [24] and SOCRATES [25] were remarkable contributions to accelerating the ATPG process.

Test generation for sequential circuits behaves mostly similar to that for combinational circuits, but it is much more difficult. In sequential ATPG algorithm different states and memory elements must be taken into account. A test for a fault in a sequential circuit may consist of several vectors.

In the following, different methods and classification of sequential ATPG are presented.

3.1.1.1 Simulation-based method

The simulation-based test generation approach is as follows. To generate a test for a fault or set of faults, a candidate test vector or test sequence is generated. Candidate tests are generated usually by targeting several faults simultaneously. The fitness of the vector or sequence is evaluated through logic or fault simulation. The vector or sequence with the highest fitness value, based on some specified cost function, is selected and the others are discarded. This process continues until some pre-specified halting condition is met. Disadvantages of this method are that it cannot identify undetectable faults and it is difficulty to detect hard-to-test faults. However, methods have been devised to overcome the last two disadvantages. In simulation-based test generator the complexity of backtracking values through logic gates is avoided, and processing occurs in the forward direction only [16] [18] [7].

In simulation-based ATPG genetic algorithms are widely used. Genetic algorithm starts with a random population of individuals, and a fault simulator is used to calculate the fitness of each individual. The best test vector evolved in any generation is selected and added to the test set. Then, the fault set is updated by removing the detected faults by the added vector(s). The genetic algorithm process repeats itself until no more faults can be detected. The test sequence length depends on the sequential depth. Sequential depth is defined as the minimum number of flip-flops in a path between the primary inputs and the farthest gate in the combinational logic. For example GATEST is a popular academic genetic ATPG tool [16] [18] [7].

3.1.1.2 Deterministic method

In the deterministic method time-frame expansion is widely used (Figure 6). For each time frame, the inputs of memory elements from the previous time frame are often referred to as *pseudo primary inputs* with respect to that time frame, and the output signals to feed the flip-flops to the next time frame are referred to as *pseudo primary outputs*. When the test generation begins, the first time frame is referred to as time frame 0. An ATPG search is carried out, where different decision will be made and when a conflict is encountered a backtrack will be made. Backtrack is going back to some earlier point and redecide on a previous decision. Backtrack is illustrated on Figure 7.

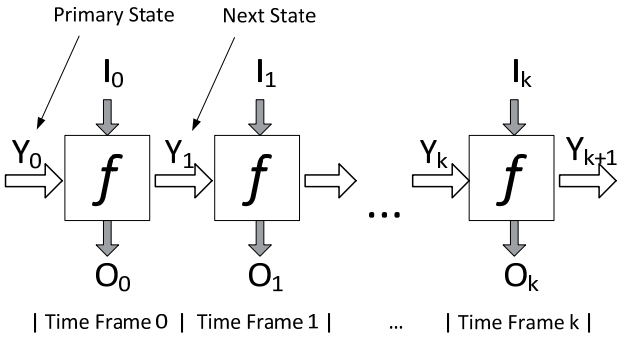


Figure 6. Time-frame expansion

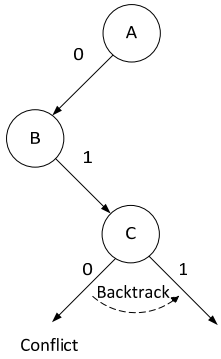


Figure 7. Backtrack

At the end of the search, a combinational vector is derived, where the input vector consists of primary inputs and pseudo primary inputs. The fault-effect for the target fault may be sensitized to either a primary output of the time frame or a pseudo primary output. If at least one pseudo primary input has been specified, then the search must attempt to justify the needed flip-flop values in time frame −1. Similarly, if fault-effects only propagate to pseudo primary outputs, the ATPG must try to propagate the fault-effects across time frame +1. Deterministic algorithms are effective in deriving tests for control-dominant circuits and in identifying untestable faults. For example HITEC is an academic deterministic algorithm based tool [16] [18] [7].

SAT-based ATPG was proposed in the 1990s [26] [27] [28] and does not work on a structural netlist but on a Boolean formula in Conjunctive Normal Form (CNF). The algorithm solves Boolean Satisfiability (SAT). Therefore, the problem must be transformed into CNF. The disadvantage of this method lies in the fact that during transformation, relevant information about the problem might get lost and therefore is not available in the solving process.

3.1.1.3 Hierarchical method

Hierarchical method takes advantage of high level information while generating tests for gate level faults.

In hierarchical RTL test generation approach, top-down and bottom-up strategies are known. In the bottom-up approach [29], tests generated at the low-level will be later assembled at the higher abstraction level. Such algorithms ignore the incompleteness problem: constraints imposed by other modules and/or the network structure may prevent test vectors from being assembled. Thus, while being fast, this type of approach is not really applicable for sequential circuits with difficult to test feedback loops. In the top-down approach [30], constraints are extracted at the higher level as a goal to be considered when deriving tests for modules at the lower level. This approach allows testing modules embedded deep into the RTL structure.

Current work is based on hierarchical top down method.

3.2 Fault simulation

To measure the test patterns' actual fault coverage at the gate level, the test patterns have to be fault-simulated. The simulating process is done on the structural level description and on the whole device. A fault simulator emulates the target fault in a circuit in order to determine which faults are detected by a given set of test vectors. Different methods have been developed to accelerate the fault simulation. Parallel fault simulation uses n-bit parallelism of logical operations where n-1 faults are simulated simultaneously. Deductive fault simulation deduces all signal values in each faulty circuit from the fault-free circuit values and the circuit structure in a single pass of true-value simulation augmented with the deductive procedure. Concurrent fault simulation is essentially an event-driven simulation to emulate faults in a circuit in the most efficient way [18] [7].

Most commonly used sequential test generators and fault simulators are based on a combinational iterative array model of the sequential circuit where

the feedback signals are generated from the copies of the combinational logic of the sequential circuit in the previous time frames.

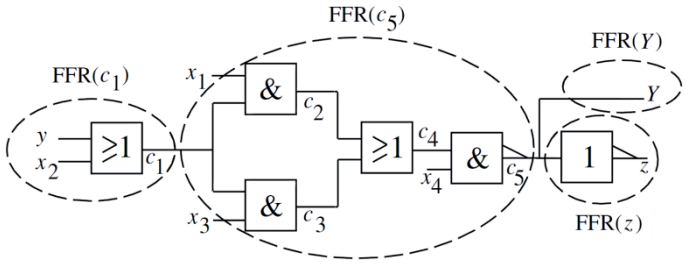


Figure 8. Partition of the combinational logic of a sequential circuit [16]

Consider the combinational logic of the sequential circuit. If the fanout stems of the combinational logic are removed, the logic is partitioned into fanout-free regions (FFRs), Figure 8. Let FFR(i) denote the FFR whose output is i. The output of an FFR can be a stem, a primary output or a next state line. In the combinational logic, if all the paths from a line r to primary outputs and next state lines go through line q, then line q is said to be the dominator of line r. If there is no other dominator between a signal and its dominator, the dominator is said to be an immediate dominator. Stem i is a dominator of all lines within FFR(i). A stem may or may not have a dominator [16].

The behaviour of a sequential circuit can be simulated by repeating the simulation of its combinational logic in each time frame. If the effect of fault f does not propagate to a next state line in a particular time frame, the fault-free and faulty values of the corresponding present state line are the same in the following time frame. If the fault-free and faulty values of each present state line are identical in a time frame, the fault is said to be a single event fault in that time frame. If there exists at least one present state line that's fault-free and faulty values are different, the fault is said to be a multiple event fault. The propagation of the fault effect for a single event fault as well as a multiple event fault.

4 Representing Sequential Digital Circuits using High-Level Decision Diagrams

In this chapter Register-Transfer Level (RTL) is introduced. RTL is a design abstraction for modelling a synchronous digital circuit and it is the most popular level for ATPG-s with SSA fault models. After that two decision diagrams are introduced – High-Level Decision Diagrams (HLDDs) and Assignment Decision Diagram (ADD).

4.1 Register-Transfer Level

Nowadays, the chip density has reached millions of transistors and it is impossible for a human to process this amount of data directly. Moreover, even for computers handling this amount of data may be very time-consuming. A solution to overcome this problem is to describe a system in abstraction – a simplified model of the systems where only selected features are shown and associated details are ignored. The purpose of an abstraction is to reduce the amount of data to a manageable level so that only critical information is presented. While high-level abstraction contains only the most vital data, low-level abstraction is more detailed. Figure 9 shows design levels starting from the behaviour level and moving on to a more detailed level.

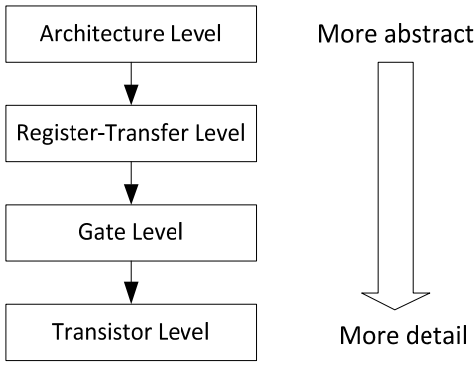


Figure 9. Design abstractions levels

The architectural level is the highest level where the circuit model contains only few implementation details. The main goal at the architectural level is to provide block architecture of the circuits implementing the basic functional specifications. At this level, a complete simulatable model may be built in some high-level language. Those models do not implement any timing information.

Next level, RTL, contains all functional details of the design, together with accurate cycle-level timing information. Here, clocked storage elements such as registers become visible. Basically, the RTL model describes the flow of signals (data) between registers and the logical operations performed on those signals. Signals are grouped together and interpreted as a special kind of data type, such as an unsigned integer or system state. This level does not include detailed timing information such as propagation delays of each block. Here, design can partition into a control part and a datapath. See, for example, Figure 10. Design at the RTL level is a typical practice in modern digital design where stuck-at faults are the most popular fault models [18] [31].

Gate level is the level where design is described by using logic gates. All the interconnections between different elements within the design are thoroughly detailed. Complex design at this level can be difficult to simulate, because of the high amount of information that model contains. For example, a 16x16 multiplier contains 2500 gates. This level is still abstract because there is no information about the actual transistors.

The lowest level, transistor level, represents the design in terms of transistors and their interconnecting wires. This level is not considered to be an abstraction level. At this level, only some logic cells are simulated because it is not practical to simulate the whole design at this level [32].

In the RTL, there are two different descriptions – the pure RTL and the behavioural RTL. The pure RTL targets the desired architecture, while the behavioural one describes the design in a more natural way for a human. In this thesis, all contributions rely only on the pure RTL.

Figure 10 presents a structural RTL view of a digital system. The datapath can be viewed as a network consisting of modules or blocks. These include registers (flip-flops), multiplexers and FUs (combinational logic for implementing operations). All the registers and some internal lines of the datapath can be represented by variables in the RTL model (variables X_R and X_L, respectively). Inputs for the datapath are the primary inputs X_i and control signals X_C (e.g., multiplexer addresses and register enable signals). Outputs are the primary outputs X_O as well as conditional signals X_N (e.g., from the comparison operators) leading to the control part FSM. [33]

The control part consists of a Finite State Machine (FSM) with a state register, next state logic and output logic. The input signals to the FSM are the primary inputs of the design (variables X_i), conditional signals originating from the datapath (variables X_N) and the current value of the state register (variable X_S). Outputs of the FSM are the primary outputs of the design (variables X_O), control signals (variables X_C) and the next value of X_S.

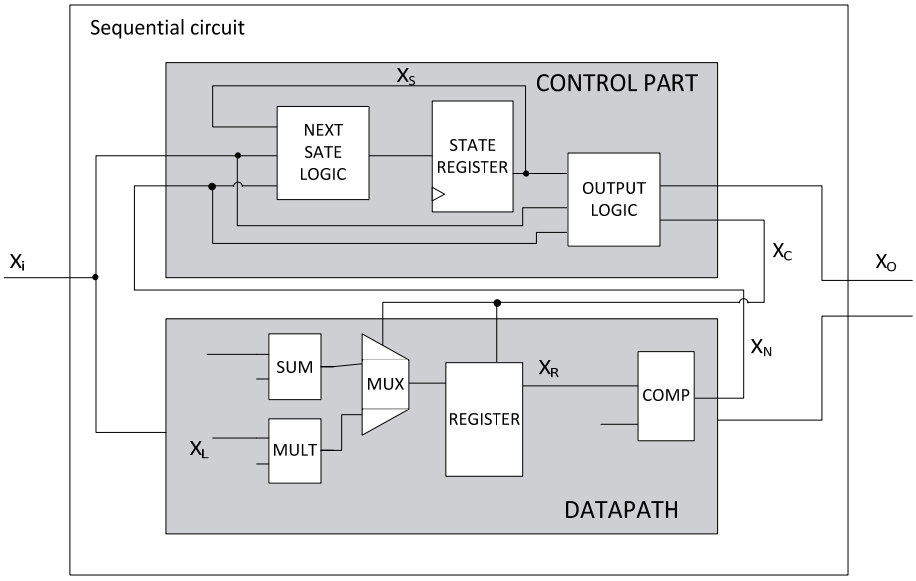


Figure 10. RTL view of a digital circuit

As it can be seen in Figure 10, the RTL descriptions contain synchronous loops as there is a feedback loop within the control part FSM as well as a feedback loop between the control and datapath parts. Furthermore, in general case, the datapath itself contains synchronous loops.

4.2 High-Level Decision Diagram Models

A HLDD is a graph representation of a discrete function. A discrete function y = f(x), where y = (y_1, …, y_n) and x = (x_1, …, x_m) are vectors is defined on X = X_1×…×X_m with values y ∈ Y = Y_1×…×Y_n, and both, the domain X and the range Y are finite sets of values. The values of variables may be Boolean, Boolean vectors, integers. Figure 11 shows an example of function y=(x_1, x_1, x_2, x_4,) represented in HLDD.

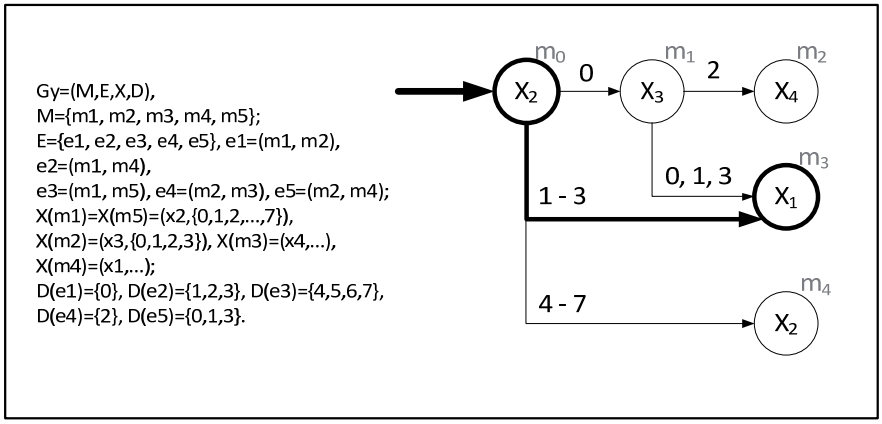


Figure 11. HLDD example for representing function y=f(x1,x2,x3,x4)

Definition 3: A High-Level Decision Diagram is a directed non-cyclic labelled graph that can be defined as a quadruple G= (M, E, X, D), where M is a finite set of vertices (referred to as nodes), E is a finite set of edges, X is a function which defines the variables labelling the nodes and the variable domains, and D is a function on E.

The function X(m_i) returns the variable x_k, which is the labelling node m_i. Each node of a HLDD is labelled by a variable. In special cases, nodes can be labelled by constants or algebraic expressions. An edge e∈E of a HLDD is an ordered pair e=(m_{pc},m_{sc})∈M^2, where M^2 is the set of all the possible ordered pairs in set M. Graphical interpretation of e is an edge leading from node mpc to

node msc. It is said that m_{pc} is a predecessor node of m_{sc}, and m_{sc} is a successor node of the node m_{pc}, respectively. D is a function on E representing the activating conditions of the edges for the simulating procedures. The value of D(e) is a subset of the domain X_k of the variable x_k, where $e=(m_i,m_j)$ and $D(m_i)=x_k$. It is required that $Pm_i = \{ D(e) \mid e = (m_i,m_j) \in E\}$ is a partition of the set X_k.

Figure 11 presents a HLDD for a discrete function $y=f(x_1,x_2,x_3,x_4)$. HLDD has only one starting node (root node) m_0, for which there are no preceding nodes. The nodes that have no successor nodes are referred to as terminal nodes $M^{term} \in M$ (nodes m_2, m_3 and m_4). The design representation by high-level decision diagrams, in general case, is a system of HLDDs rather than a single HLDD. During the simulation in HLDD systems, the values of some variables labelling the nodes of a HLDD are calculated by other HLDDs of the system.

Simulation on decision diagrams takes place as follows. Consider a situation where all the node variables are fixed to some value. For each non-terminal node $mi \notin M^{term}$ according to the value v_k of the variable $x_k=Z(m_i)$ a certain output edge $e = (m_i,m_j)$, $v_k \in D(e)$ will be chosen, which enters into its corresponding successor node m_j. Let us call such connections activated edges under the given values and denote them by bold arrows. Succeeding each other, activated edges form activated paths. For each combination of values of all the node variables, there always exists a corresponding activated path from the root node to some terminal node. This path is referred to as the main activated path. The simulated value of the variable represented by the HLDD will be the value of the variable labelling the terminal node of the main activated path.

Figure 11 presents a simulation on the high-level decision diagram. Assuming that variable x_2 is equal to 2, a path (marked by bold arrows) is activated from node m_0 (the root node) to a terminal node m_3 labelled by x_1. Let the value of variable x_1 be 4, thus, $y=x_1=4$. Note that this type of simulation is event-driven since it has to simulate only those nodes that are traversed by the main activated path.

When representing systems by decision diagram models, in general case, a network of HLDDs are required. During the simulation in HLDD systems, the values of some variables labelling the nodes of a HLDD are calculated by other HLDDs of the system.

4.2.1 HLDD Representation for RTL Circuits

In HLDDs, representing the datapath, the non-terminal nodes correspond to control signals. The terminal nodes represent operations (FUs). Register transfers and constant assignments are treated as special cases of operations. Figure 12 shows a simple example of a HLDD representation for the given datapath fragment [34].

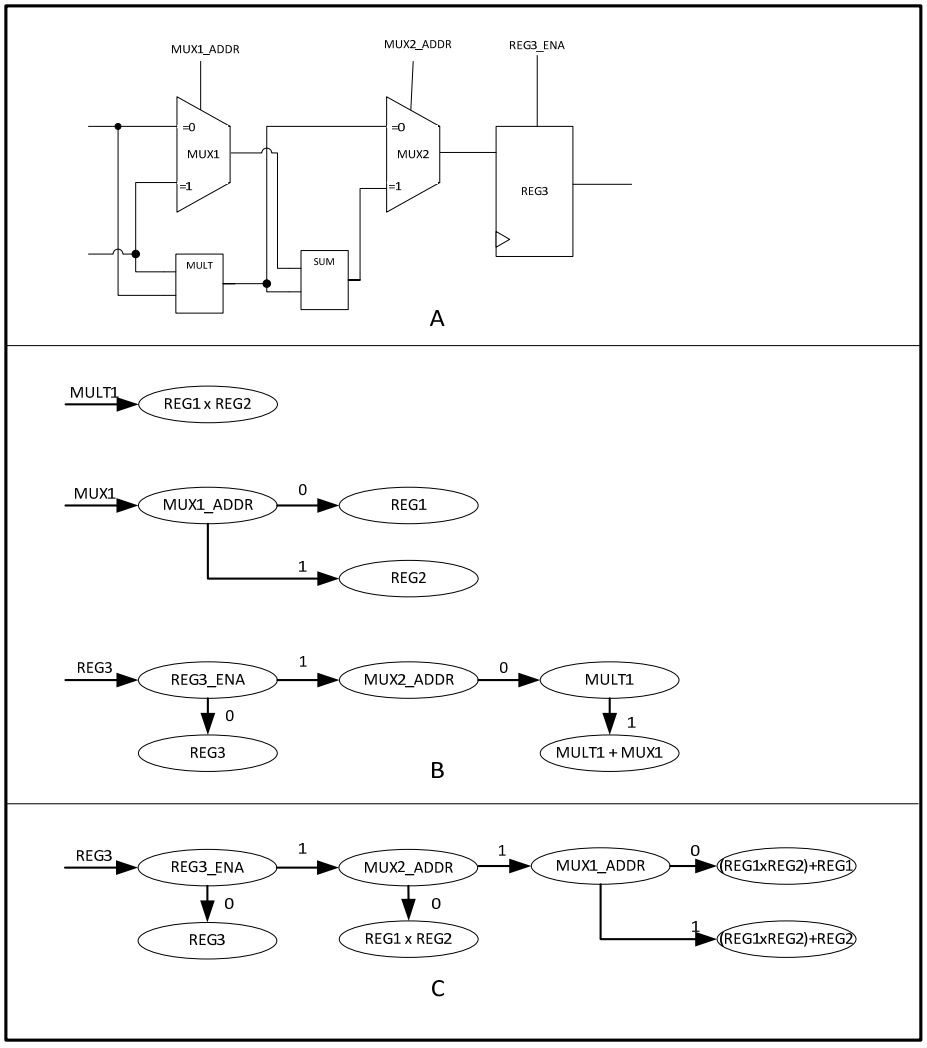


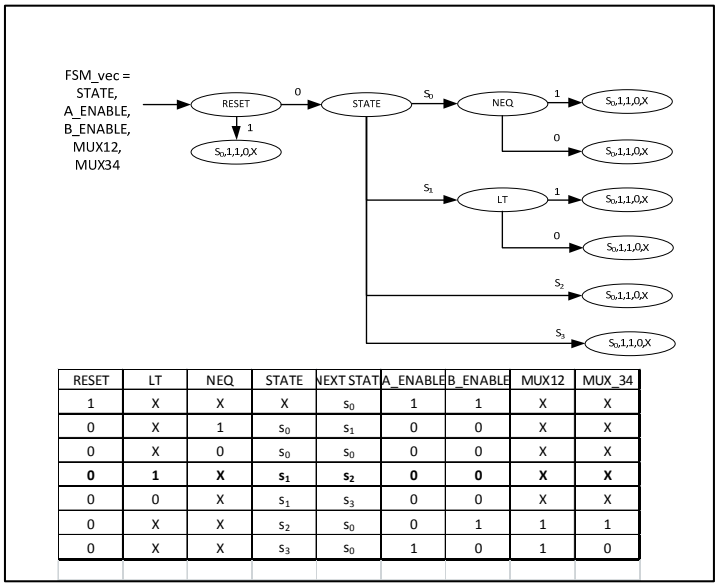
Figure 12. Datapath representing in HLDD and partitioning types

Usually, a datapath is represented by a system of HLDDs. Here, different partitioning strategies are possible. The most commonly used partitioning is the one in which to each primary output, fan-out signal and register a HLDD corresponds. In addition, multiplexers that are connected to inputs of a FU are represented by a separate HLDD. Figure 12B shows this type of HLDD system partitioning for the datapath given in Figure 12A. However, it is possible to use alternative partitioning. For example, Figure 12C shows an approach where exactly one decision diagram corresponds to each register of the datapath. This type of partitioning is sometimes referred to as the register-oriented HLDD.

Other types of HLDD partitioning can be used depending on the target model application [34].

A simple RTL design control part is usually represented by a single HLDD, however, in the case of complex or multiple FSMs different partitioning are possible here as well. The control part HLDD calculates the values for a vector consisting of the state variable and control signals. In the HLDD, the non-terminal nodes correspond to the current state (labelled by variable X_S) and conditional signals originating from the datapath (variables X_N). Terminal nodes hold vectors with the values of next state and control signals X_C [34].

Figure 13 shows a FSM state table and its corresponding HLDD representation. In the HLDD, state denotes the next state and state denotes the current state value. Variables A_enable, B_enable, mux_12 and mux_34 are FSM outputs and belong to the control signals X_C. Variables RESET, LT and NEQ are FSM inputs and belong to X_N. The dashed circles and arrows in Figure 13 depict setting up of the edges and the terminal node corresponding to the fourth row of the state table [34].



RESET	LT	NEQ	STATE	NEXT STATE	A_ENABLE	B_ENABLE	MUX12	MUX_34
1	X	X	X	s_0	1	1	X	X
0	X	1	s_0	s_1	0	0	X	X
0	X	0	s_0	s_0	0	0	X	X
0	**1**	**X**	**s_1**	**s_2**	**0**	**0**	**X**	**X**
0	0	X	s_1	s_3	0	0	X	X
0	X	X	s_2	s_0	0	1	1	1
0	X	X	s_3	s_0	1	0	1	0

Figure 13. Converting FSM state table into HLDD

There exist other word-level decision diagrams such as multiterminal decision diagrams [20], K*BMDs [21] and ADDs [13]. However, in MTDDs the nonterminal nodes hold Boolean variables only. K*BMDs, where additive and multiplicative weights label the edges are useful for compact canonical representation of functions on integers (especially wide integers). The main goal of HLDD representations described in this work is not canonicity but simulation and implications. In HLDD the nodes are divided into nonterminal (control) and terminal (data) ones In HLDDs, the selection of a node activates a path through the diagram, which derives the needed value assignments for variables.

The principal difference between HLDDs and Assignment decision diagram (ADD) [35] lies in the fact that ADDs edges are not labelled by activating values. They are rather used as connecting signals to represent structure. ADD will be explained in the following subsection.

4.3 Assignment Decision Diagram

Assignment decision diagram (ADD) is an acyclic graph that consists of a set of nodes that can be categorized into four types: read node, write node, operation node and assignment decision node (ADN), and a set of edges which contain the connectivity information between two nodes (Figure 14). A read node represents a primary input port, a storage unit or a constant while a write node represents a primary output port or a storage unit. An operation node expresses an arithmetic operation unit or a logic operation unit while an ADN selects a value from a set of values that are provided to it based on the conditions computed by the logic operation units. If one of the condition inputs becomes true, the value of the corresponding data input will be selected

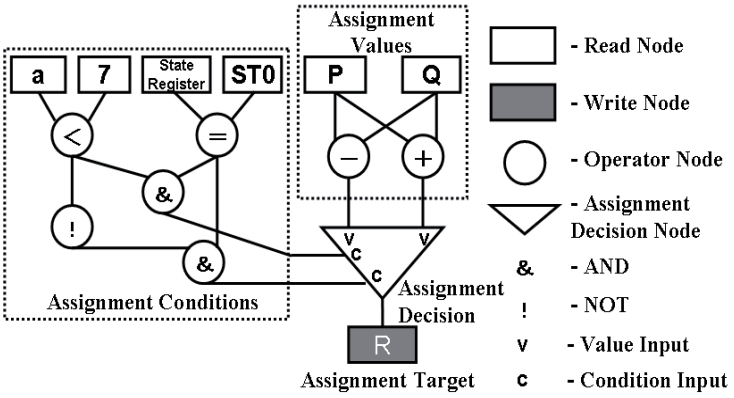


Figure 14. Assignment Decision Diagram (ADD)

When a node N is under test, the testability of the node is guaranteed if:

a) any value can propagate from a read node corresponding to a primary input port to the input of N, and
b) the value at the output of N can propagate to a write node corresponding to a primary output port.

The paths which allow (a) and (b) to occur are called justification path and propagation path, respectively. Justification and propagation can be done through symbolic processing that utilizes nine valued algebra. The series of symbols obtained from the symbolic processing that activates justification and propagation paths is known as the test environment for the node under test. For a given node under test, its test sequence is generated by first extracting a test pattern from the test set library and by substituting the test pattern for the test environment. The test set library is obtained beforehand by first simply taking a logic-level circuit of the node under test, then generating the test patterns for all faults in the circuit using a combinational ATPG algorithm. In the case where the node is synthesized into a circuit which is different, fault simulation must be performed to check the fault efficiency of the test patterns.

The symbols of Ghosh's nine-valued algebra [10], each of which can be assigned true or false, are as follows:

- Cg(v): variable v can be set to any value.
- C0(v): variable v can be set to 0.
- C1(v): variable v can be set to 1.
- Ca1(v): all bits of variable v can be set to 1's.
- Cq(v): variable v can be set to a constant.
- Cz(v): variable v can be set to high impedance Z.
- Cs(v): state variable v can be set to a specific state.

• O(v): any fault effect at variable v can be observed.
• O'(v): fault effect of D' can be observed for a single bit variable v.

To generate a test environment, first an objective has to be set. In order to achieve the test environment objective, the test sequence for each ADD can be generated through the following two phases using justification/propagation rules [36]:

Phase 1: Generate the test environment of the node under test.

Phase 2: Generate the test sequence of the node under test by substituting the test patterns of the logic-level circuit corresponding to the node under test for the test environment.

Without going into details of the symbol propagating rules, consider Figure 15 presenting backward propagation (justification) of two symbols Cq and Cg that converge in a fanout read node. In the strict interpretation of the propagation rules of [36] the two symbols when converging in the fanout result in a conflict. In the weak interpretation the symbols will resolve in assigning Cg to the read node.

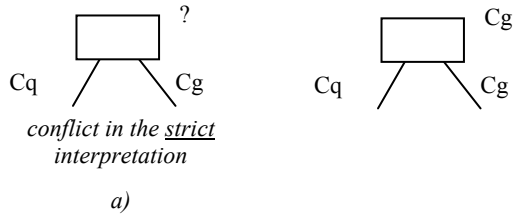


Figure 15. Handling of fanouts during justification

Thus, the strict interpretation of Ghosh's algebra [36] lead to overly pessimistic results because tests for some Modules Under Test (MUTs) are aborted due to justification conflicts. On the other hand, the weak interpretation is too optimistic and can also lead to loss of fault coverage because some of the test patterns that are expected to cover faults in the MUT do not propagate.

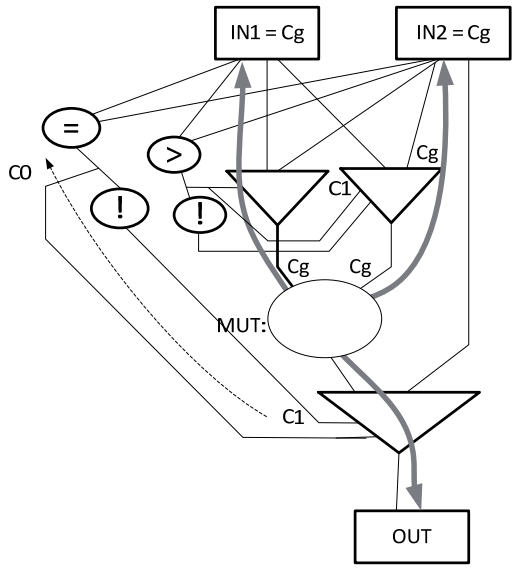


Figure 16. Test environment generation example. An unrolled view.

Consider as an example, a simplification of the ADD for the Greatest Common Division (GCD) benchmark presented in Figure 16. Without loss of generality in this ADD the control state information and the data registers have been removed and the circuit has been unrolled by applying time-frame expansion in order to improve the readability of the diagram. (Note, that the original GCD benchmark still contains a data dependent loop, which has been unrolled in Figure).

Assume that our task is generating a test environment for the subtraction module (MUT) in Figure 16. The output value of MUT will be propagated to the primary output OUT only if the first value input of the corresponding assignment decision is 1. Therefore we set the corresponding condition input of the ADN to C1. When we justify this particular condition input and the symbols at the MUT inputs according to the propagation rules presented in [36], then the strict interpretation of these rules would lead into a contradiction (See Figure 15a). However, the weak interpretation (also used in [37]) would still allow the following test environment: IN1=Cg and IN2=Cg. Note, that in current situation the weak rules are preferable since they at least allow testing part of the MUT while the strict rules would not generate any test environment at all.

However, as it will be shown in Chapter 7 the weak interpretation is overly optimistic and results in tests where the achieved fault coverage is 8-14 % lower than maximum achievable fault coverage. To overcome this restriction a top-down method is proposed in this Thesis.

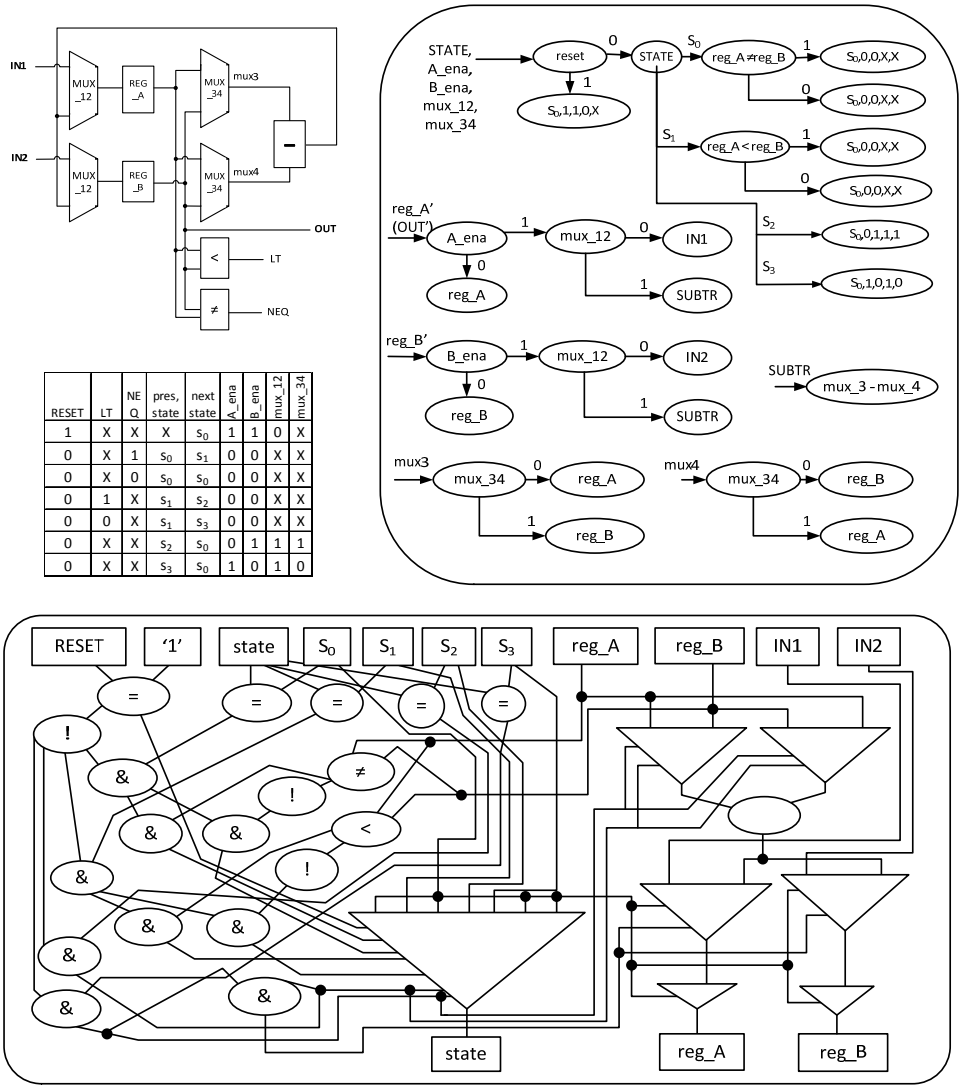
4.4 Differences between HLDDs and ADDs

Figure 17 presents the RTL description of a Greatest Common Division benchmark and its corresponding HLDD and Assignment Decision Diagram (ADD) representations. Apart from the fact that HLDD description contains fewer nodes, there are the following fundamental differences:

1. ADDs structure closely matches the RTL design. Edges of ADD correspond to connecting nets in datapath. ADD for FSM is equivalent to its gate-level implementation. In contrast, HLDDs do not strictly follow the circuit structure. Here, a synthesis to extract data and control relationships from the circuit functionality has been carried out.

2. ADD model includes four types of nodes (read, write, operator, assignment decision). In the HLDD, the nodes are treated uniformly and can be divided into nonterminal nodes (control) and terminal nodes (data).

3. While ADDs do not support decision-making implicitly in the model, in the HLDDs, the selection of a node activates a path through the diagram which derives the needed value assignments for variables. Note that the edges in ADD model have no labels. This is the most significant difference between the two models.



RESET	LT	NEQ	pres. state	next state	A_ena	B_ena	mux_12	mux_34
1	X	X	X	s_0	1	1	0	X
0	X	1	s_0	s_1	0	0	X	X
0	X	0	s_0	s_0	0	0	X	X
0	1	X	s_1	s_2	0	0	X	X
0	0	X	s_1	s_3	0	0	X	X
0	X	X	s_2	s_0	0	1	1	1
0	X	X	s_3	s_0	1	0	1	0

Figure 17. RTL circuit (top left), its HLDD (top right) and ADD (bottom).

5 Constraint-based Automated Test Pattern Generation for Sequential Circuits

This chapter presents a novel constraint-based approach for the hierarchical ATPG [38] [39] where the deterministic algorithm first activates the test path constraint at the RTL and subsequently applies a constraint-solving package ECLiPSe Prolog [40] for assembling the tests. The results of experiments indicate that the proposed deterministic method provides increased fault coverage for hard-to-test designs with respect to semi-formal approaches. In addition, this approach offers short run times.

5.1 Previous work

At present, satisfactory methods for testing sequential circuits are missing and this has led the community to replace the hard test pattern generation task by theoretically much simpler approach that relies on scan paths together with the combinational ATPG. However, the scan-path method has its shortcomings, including increased area, delay and consumed power. It also causes targeting of non-functional failure modes which results in over-testing and yield loss [41].

Several approaches to generating tests for structural faults in sequential cores have been proposed over the years. Despite all the efforts the problem still lacks a breakthrough. At the gate-level, a number of deterministic test generation tools, both academic [8] [9] and commercial, have been implemented. None of these methods can efficiently handle sequential designs of even a couple of thousands of gates. With the further growth of the circuit size fault coverage tends to drop while the run times increase rapidly.

Better performance has been obtained with simulation-based approaches. Here, genetic algorithm-based methods have been widely used [10] [11] [12]. Relatively efficient results have been obtained by spectral methods [42]. However, the simulation-based methods are fast for smaller circuits only and become ineffective when the number of primary inputs and the sequential depth of the circuit increase. Moreover, these methods do not guarantee the detection of hard-to-test random pattern-resistant faults.

Many works on Functional Test Generation (FUTEG) have been published in the past [13] [14]. In this field, an efficient technique based on the Binary

Decision Diagram (BDD) manipulation of data domain partitions has been proposed [43]. However, the fundamental shortcoming of the approaches that rely on functional fault models is that they do not offer full structural level fault coverage.

Hierarchical and RTL test pattern generation has been proposed as a promising alternative to tackle complex sequential circuits. Here, top-down and bottom-up strategies are known. In the bottom-up approach [29], tests generated at the lower level will be later assembled at a higher abstraction level. Such algorithms ignore the incompleteness problem: constraints imposed by other modules and/or the network structure may prevent test vectors from being assembled. In the top-down approach [30], where constraints are extracted at a higher level with the goal to be considered when deriving tests for modules at a lower level. This approach allows testing modules embedded deep into the RTL structure. However, as modules are often tested through highly complex constraints, their fault coverage may be compromised.

The top-down hierarchical ATPG have been developed by some authors [38] [39]. Here, the ATPG operates on the RTL HLDD model of the circuit and as generates test patterns as an output. The output patterns do not offer precise information about the achieved fault coverage. In order to measure the actual gate-level fault coverage of the generated tests, the test patterns have to be fault simulated on the structural level description of the whole device.

A number of works have been published on implementing assignment decision diagram models [35] combined with SAT methods to address register transfer level test pattern generation [44] [36] [37]. All of these are bottom-up methods based on a multivalued algebra for establishing transparent test paths. Therefore they suffer from the incompleteness issue described above.

At the Tallinn University of Technology, the hierarchical ATPG has been developed by Professor Jaan Raik. The hierarchical ATPG operates on the RTL HLDD model of the circuit and generates test patterns as an output. The output patterns do not offer precise information about the achieved fault coverage. In order to measure the actual gate-level fault coverage of the generated tests, the test patterns have to be fault-simulated on the structural level description of the whole device. Because the HLDD themselves are not contributions of this thesis, only a brief description will be presented. The HLDD model description provided in this subsection is mostly based on the description provided in [19] and [45].

5.2 Motivation

In the previous top-down test pattern generation algorithms by the authors [38] [39], random constraint solving was applied. For random constraint solving it is hard task to generate solution for complicated operators like “equal to” or value between some short intervals. Therefore, the ATPG spends a lot of valuable time on trying to generate suitable solutions for those constraints until the solution limit is reached and the module could remain without suitable test patterns. The goal is to improve the hierarchical ATPG fault coverage and short run times by applying a constraint solver to solve the test path constraints.

In the following part of this thesis, the deterministic path activation method and constraints extraction is introduced.

5.3 Hierarchical ATPG algorithm

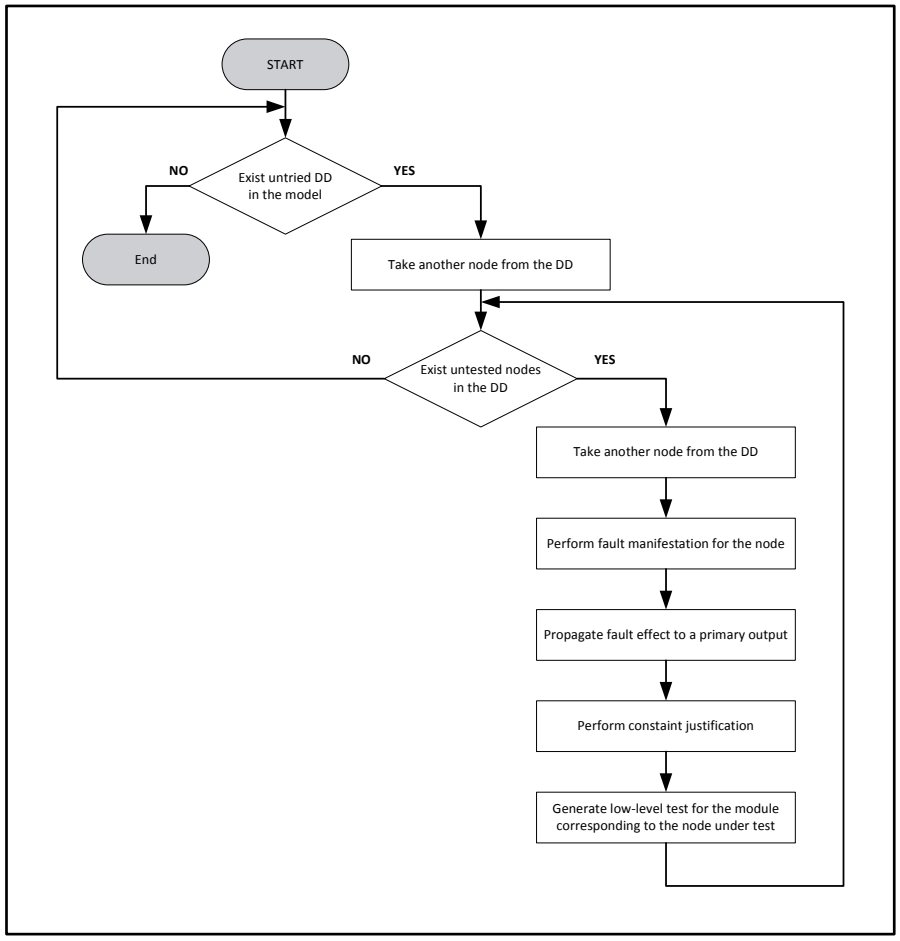


Figure 18. The general flow of the hierarchical test generation

In Figure 18, a general overview of the top-down hierarchical test generation algorithm is presented. It starts at a higher level with the decision-making, which considers whether the nodes in this test scheme, is untested. If untested nodes exist, then go to generate the test run, otherwise exit from the algorithm. In a test run, six different operations will be visited. Firstly, the fault manifestation is followed by the fault effect propagation. During the propagation stage, we move forward in time (clock cycles), the fault effect is propagated towards the primary outputs and path activation constraints are created whenever conditions in the control part HLDD are traversed. Propagation is completed when the fault effect pointer points to variable x corresponding to a primary output of the circuit. Subsequent to the propagation, the constraint justification

starts. Justification moves backwards in time, starting from the clock cycle in which the propagation ended. During this process, the existing constraints are updated and additional path activation constraints are created. Finally, tests are generated at a low level for the node under test.

For each datapath MUT, we extract the control part FSM state sequences in order to propagate fault effects from the output of the MUT to primary outputs and to propagate the values from the primary inputs to the inputs of the MUT. Such state sequences constitute test paths for accessing the MUT. We represent the test paths by sets of constraints. All test paths within a certain cycle limit are activated and the corresponding constraints extracted by the proposed algorithm. In order to extract the RTL test path constraints in this work, a test path activation tool [46] is applied.

5.4 Concept of path activation constraints

The concept of the constraints for a single test path for a datapath MUT is visualized in Figure 19. The test path constraints are divided into three categories. These are the set of path activation constraints C_A, the transformation constraints C_J and the propagation constraint C_P, respectively. Path activation constraints correspond to the conditions in the FSM state transitions that have to be satisfied in order to perform propagation and value justification through the circuit. Transformation constraints, in turn, reflect the value changes along the paths from the inputs of the high-level MUT to the primary inputs of the whole circuit. These constraints are needed in order to derive the local test patterns for the MUT. The propagation constraints show how the value propagated from the output of the MUT to a primary output depends on the values of the primary inputs. The main idea here is to check whether the fault effect will be masked when propagated to a primary output. All the above categories of constraints are represented by common data structures and manipulated by common procedures for creation, update, modelling and simulation. In the following part of this thesis, the data structure and update operations of test path constraints are defined.

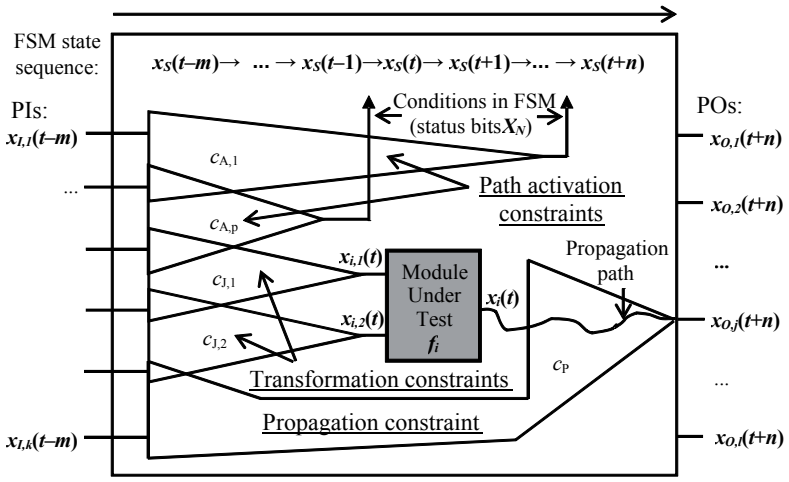


Figure 19. An unrolled RTL circuit with test generation constraints for a test path for a MUT

Definition 4: A condition c that is $d = g(X')$, where d is a bitvector or Boolean constant or a variable $x \in X$, and $g(X')$ is a logic, arithmetic or comparison expression on a subset of variables $X' \subset X$, is referred to as a *test path constraint*. From this point on, we refer to test path constraints as constraints.

Definition 5: Constraint c: $d = g(X')$ is said to be *justified* if $X' \subseteq X_I$, where X_I is the set of primary inputs of the system. Otherwise, c is said to be an *unjustified* constraint.

Definition 6: If a constraint c: $d = g(X')$ is unjustified then all the variables in the set X' that are not input variables X_I are said to be *unjustified variables* of the constraint. The input variables belonging to the constraint are called *justified variables*.

Definition 7: Let X' be the set of justified variables and X'' be the set of unjustified variables of a constraint c: $d = g(X', X'')$. The process, where each variable $x''_i \in X''$ is substituted by an expression $h_i(X'''_i)$ on model variables $X'''_i \subseteq X$, is referred to as *updating the constraint c*.

Consider the general case of test path constraints for a MUT presented in Figure 19. Such constraints are extracted as follows. First, the value from the output variable x_i of the MUT f_i is propagated to a primary output $x_{O,j}$ by activating a state sequence $x_S(t) \rightarrow x_S(t+1) \rightarrow ... \rightarrow x_S(t+n)$ in the control part. Here, by $x(t)$ we denote the value of variable x at the clock cycle t. Thus, the propagation state sequence starts at a time step t, which is referred to as the *manifestation step*, and it ends at a clock cycle $t+n$. During propagation, path activation constraints $c_{A,p} \in C_A$ are created at time steps where the next state value of x_S is depending on the status bits X_N. When the fault effect value propagates from x_i to $x_{O,j}$ at the time step $t+n$ then the propagation constraint c_P is created.

Subsequent to the propagation, the constraint justification process begins. Starting from the time step $t+n$, we move backward in time until the manifestation step t is reached. At each time step we update the propagation constraint c_P and those path activation constraints $c_{A,p}$ whose creation time step is later than current time step. During the update, the unjustified variables $X'' \subseteq X_R$ of the constraint expressions $g(X', X'')$ for all the constraints are substituted by expressions $h_i(X'''_i)$ on model variables $X'''_i \subseteq X_R \cup X_I$, where $h_i(X'''_i)$ are the expressions implemented by functional units F_U selected according to the values of control signal variables X_C at the current time step.

At the manifestation time step t, we create the transformation constraints for each input of the MUT. Without loss of generality, Figure 19 shows a MUT with two inputs $x_{i,1}$ and $x_{i,2}$. Thus, in current case the transformation constraints $c_{J,1} \in C_J$ and $c_{J,2} \in C_J$ are created, respectively. We continue moving backwards in time until at some time step $t-m$ all the variables in the constraints are primary inputs X_I. During this process we update all the created constraints and create new path activation constraints $c_{A,p}$ at time steps where the previous state value of x_S is depending on the status bits X_N.

Note that the extracted constraints contain expressions $g(X)$ on primary inputs X_I and constants. (In the case of the propagation constraint c_P the expression also depends on the MUT output x_i). The expressions are determined by the functions implemented by functional units F_U and, in the case of path activation constraints $c_{A,p}$, also by comparison operations F_N. The exponential size complexity of the constraints expression $g(X)$ is avoided by uniting multiple occurrences of the same variable (i.e., the literals) in the constraints at each time step into one single fan-out variable. Because of this, the size requirements for the constraints are linear with respect to justification time-frames and they represent a subset of the expanded time-frame model of the circuit.

Finally, consistency of test paths $c_{A,p}$ is verified. After one consistent set of test path constraints are extracted by Decider, the fault coverage is measured. If MUT is not tested with 100 %, a backtrack occurs and the tool attempts to use alternative propagation and justification paths. The process ends when all the consistent test paths within a certain time-step limit are activated and respective test path constraints are extracted.

The high-level symbolic path activation is a complete algorithm. If transparent paths for fault effect propagation and value justification exist, they will be activated. The algorithm has been implemented as a systematic search and therefore an inconsistency in any stage causes a backtrack and a return to the last decision. However, due to the NP-complete nature of the problem, in some cases, the search must be terminated after a certain maximal number of solutions have been tried. For the sake of simplicity and speed, only three types of symbolic values are used during the path activation:

D - Line with the fault effect,

X - Line with unassigned value,

V - Line with an assigned value.

5.4.1 Fault manifestation

In first step, which is fault manifestation step, (Figure 20) appropriate tests for the corresponding nodes of the RTL HLDD model have to be set up. The two types of tests are referred to as scanning test and conformity test, respectively. Scanning tests are applied to terminal nodes and their aim is to test the Functional Units (FU), registers and constants of the datapath. Conformity tests are set up for non-terminal nodes and they target the decoding logic of the multiplexers of the datapath as well as the output logic of the control part. The goal here is to set the fault effect to the output of the RTL MUT and determine the current FSM state.

Fault manifestation phase sub-division:

1. Activate the full path to the node under test. (A=1, B=1);
2. Create transformation constraints (D1=J and D2=K);
3. FSM state to create manifestation (Q =2);
4. Create path activation constraints (M<N);
5. Fault effect pointer is set to the variable ($p_D \rightarrow Y$).

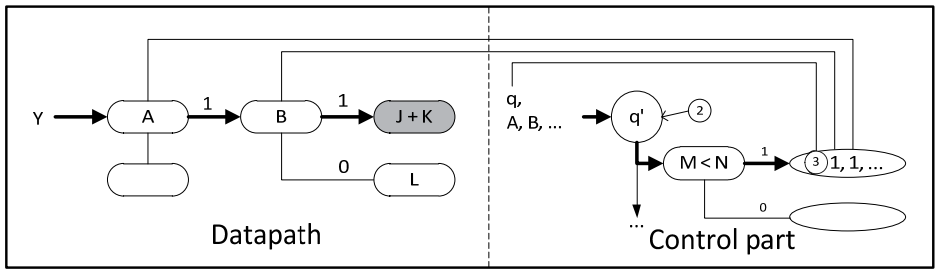


Figure 20. Fault manifestation for node J + K

5.4.2 Fault effect propagation

The purpose of the propagation procedure is to activate a state sequence that propagates the fault effect from the output of the MUT to one of the primary outputs of the design. According to the current approach, propagation along single path is implemented. In order to keep track of the fault effect propagation, a dedicated fault effect pointer is used. During the propagation, high-level test path activation constraints are created. Figure 21 presents the algorithm for fault effect propagation.

In the algorithm descriptions, the term ‘consistent FSM control vector’ is frequently used. By this term we mean a control vector (row) in the control part’s FSM state table whose control signal values are consistent with value assignments made for control signals while propagating (activating) paths in the datapath.

```
/* Fault manifestation for module MUT */
Create constraints from all the module input (M)
Set fault effect pointer to node output (M)

/* Fault effect propagation */
WHILE fault pointer is not propagated to a primary output
  Let α be the node pointer by fault effect pointer
  Choose the most observable fanout branching of α
  Set control signals required to transport fault effect from the
  fanout branch to the next fanout stem or register node β
  /*always only one such path exists!*/
  Set fault effect pointer to β
  IF exist a consistent FSM control vector THEN
    Choose the most observable consistent control vector
    Create constraint of corresponding FSM input vector
    IF β is a register THEN
      move to the next time-frame
    END IF
  ELSE
    Backtrack
  END IF
ENDWHILE
```

Figure 21. Fault effect propagation algorithm

The goal of the second step, the fault effect propagation (Figure 22), is to calculate a state sequence required to propagate this fault effect from the MUT to a primary output of the device. During this process, path activation constraints are created of the conditions traversed in the control part DD. The fault effect propagation ends when a fault effect corresponds to the circuit's primary output.

Fault effect phase sub-division:

1. Select a graph with a node m, where pD→x(m);
2. Activate the full path to the node labelled by the fault effect (A=1, B=0);
3. Select the next FSM state. (q= 4);
4. Create path activation constraints (M=N);
5. Fault effect pointer is set to the variable calculated by the current DD(pD→Y).

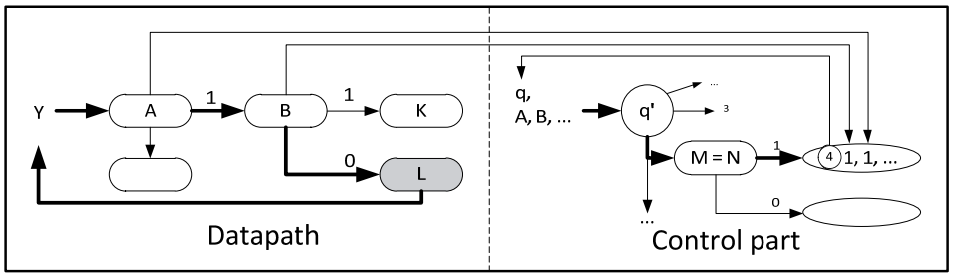


Figure 22. Fault effect propagation

5.4.3 Constraint justification

Subsequent to the propagation, a constraint justification starts. Justification moves backward in time, starting from the clock cycle, where propagation ended. During this process existing constraints are updated and additional path activation constraints are created. Nodes of the circuit, which correspond to primary inputs or constants, are called justified nodes. All other nodes are said to be unjustified. Constraints containing unjustified nodes are referred to as unjustified constraints.

Basically, updating a constraint can be regarded as superposition of the unjustified nodes of the constraint by new datapath nodes determined by paths activated in the datapath by current control vector.

At each justification step, a current justification objective is chosen. In the proposed algorithm implementation, the justification objective is to justify the first unjustified node from the first unjustified constraint. The algorithm for constraint justification is presented in Figure 23.

```
/* Constraint justification */
WHILE exist unjustified constraint
IF current time-frame is earlier then manifestation THEN
  Let current objective be to justify node β
  Choose the most controllable fanout, FU or register node α which
directly precedes β
  Set control signals activating path from α to β
  /*always only one such path exists!*/
  IF exist a consistent FSM control vector THEN
    Choose the most controllable consistent control vector
    Create constraints of corresponding FSM input vector
    IF  α is a register THEN
      move to the previous time-frame
    ENDIF
  ELSE
    Backtrack
  ENDIF
ELSE
  Move to the previous time-frame
ENDIF
Update all active constraints
ENDWHIL
```

Figure 23. Constraint justification algorithm.

In the third step, the constraint justification (Figure 24) traverses backwards the state sequence calculated by the propagation phase until the clock cycle of fault manifestation is reached. During this process, constraints previously created by the propagation are updated. Starting from the clock cycle of the manifestation phase, a reverse state sequence is calculated, the existing constraints are updated and additional constraints are created of the conditions in the FSM that have to be satisfied. Note that at each clock cycle, from all the created constraints only those are considered during justification that were created in a later clock cycle than the current one.

Constraints justification sub-division:

1. Determining current justification objective;
2. Activate the full path to a terminal node (A = 1; B = 1);
3. Select present FSM state (q = 1);
4. Create path activation constraints (M > N);
5. Update the constraints.

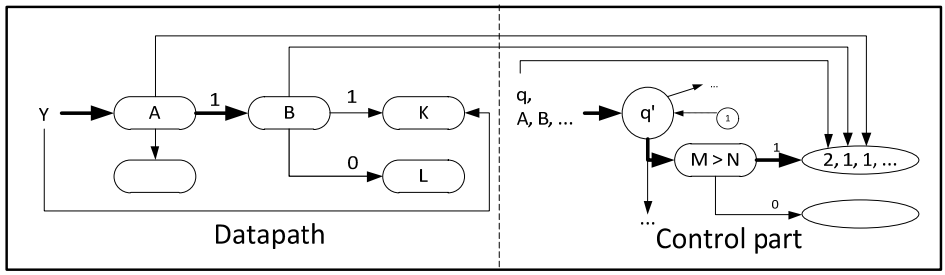


Figure 24. Constraint justification

5.4.4 Constraint solving and low-level test

In the fourth step, the constraint solving and low-level test, the goal is to satisfy the extracted constraints and test the datapath module corresponding to the current node under test at a low level. The extracted constraints are not always satisfiable, because they can also be inconsistent or too complex for the constraint satisfaction algorithm to solve. In these cases, a backtrack occurs and the high-level test generation algorithm attempts to activate an alternative test path. In general case, a single activated path is not enough to reach a 100 per cent fault efficiency for a functional unit, i.e., the test set for a FU can consist of vectors generated during different activated paths and therefore different calls to the low-level part. Thus, a record is kept about the faults detected by the low-level tests during previous activated paths.

5.5 Constraint extraction example

In the following, the test path activation algorithm and constraint extraction is explained basing on the example of the Greatest Common Divisor (GCD). Consider the GCD algorithm described at the behavioural level in a pseudo hardware description language in Figure 25.

```
A := IN1;
B := IN2;
  while (A ≠ B)
    if (A < B) then
    B := B – A;
  else
    A := A – B;
  end if;
end while;
OUT := A;
```

Figure 25. GCD algorithm

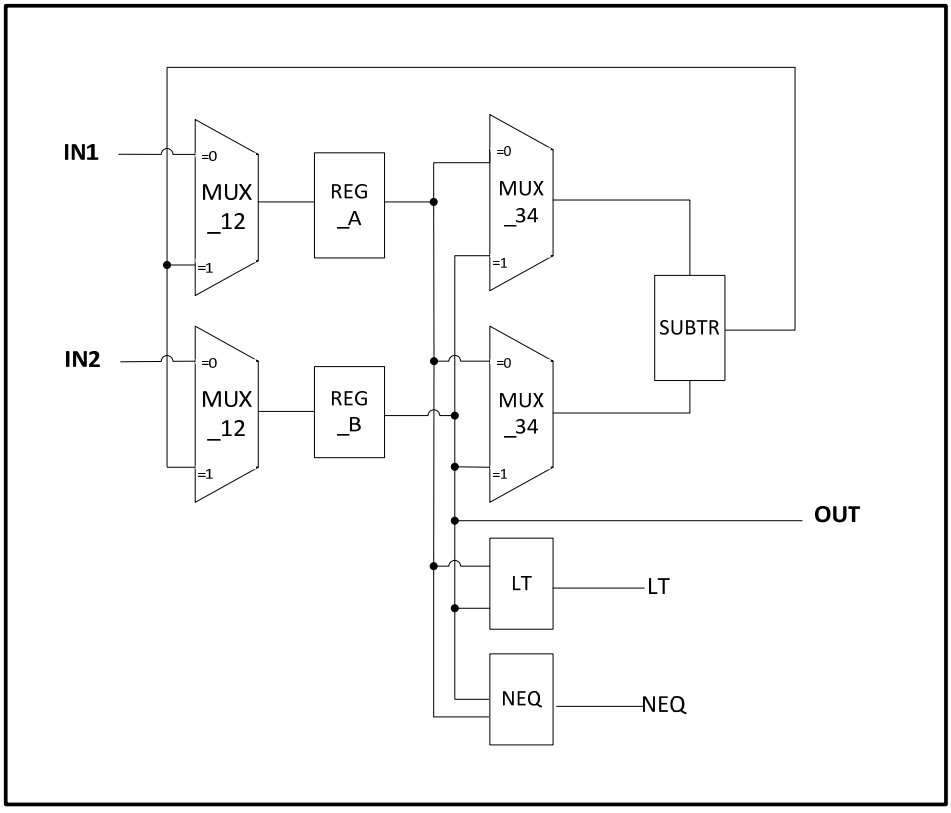


Figure 26. RT-level architecture of the GCD circuit

Table 3. FSM table of the GCD circuit

RESET	LT	NEQ	STATE	NEXT STATE	A_ENABLE	B_ENABLE	MUX12	MUX_34
1	X	X	X	s_0	1	1	X	X
0	X	1	s_0	s_1	0	0	X	X
0	X	0	s_0	s_0	0	0	X	X
0	1	X	s_1	s_2	0	0	X	X
0	0	X	s_1	s_3	0	0	X	X
0	X	X	s_2	s_0	0	1	1	1
0	X	X	s_3	s_0	1	0	1	0

Let us assume that subsequent to applying a high level synthesis to the algorithm description we obtain the RTL architecture presented in Figure 26. This architecture consists of a datapath of 3 FU, 2 registers and 4 multiplexers and a control part FSM of four states. The control part is given as a state table in Table 3. For simplicity, only for module SUBTR generating test paths will be explained.

Fault manifestation.

Set all the variables to 'don't care' values. Create transformation constraints D_0=mux3, D_1=mux4. Set the fault effect pointer to variable SUBTR, i.e., yD := SUBTR.

Fault effect propagation.

Choose a datapath register that reads from the FU SUBTR. There are two possible choices: reg_A and reg_B, respectively. Let us select the first choice. Subsequently, we activate the path from SUBTR to reg_A, which results in the following variable assignments: A_enable := 1, mux_1 := 1. Next, we have to choose a consistent FSM control vector. The only vector consistent with previous variable assignments is the one corresponding to row 7 in the FSM state table (labelled by vector 0, X, X, S3, S0, 1, 0, 1, 0). Based on this vector we obtain the following assignments: reset:=0, B_enable := 0, mux_34 := 0, state := S3 (in the current clock cycle), state := S0 (in the next clock cycle). We move to the next clock cycle and set the fault effect pointer yD to reg_A (i.e., OUT).

We detect that the fault effect pointer points to a variable corresponding to a primary output and thus we have successfully completed the fault propagation process.

Constraint justification

As there was no path activation constraints created during the manifestation and propagation stages, we move backwards in terms of clock cycles until the clock cycle of manifestation phase is reached. We select the justification objective from the unjustified variables of the transformation constraints (D_0=mux3, D_1=mux4). Let the current objective be to justify variable mux3. Due to the fact that we have already assigned mux_34 := 0 at current clock cycle during the propagation process, then we have no choice but backtracking mux3 to reg_A. We update the constraints, obtaining D_0= reg_A, D_1= reg_B and move to the preceding clock cycle.

Without focusing on further details, we continue executing the constraint justification algorithm until the path presented in Figure 27 is activated as one of possible high-level path solutions. In the Figure we have denoted the manifestation clock cycle by t, the i-th cycle following t is denoted by t+i and i-th cycle preceding t is denoted by t-i, respectively. Below the clock cycle information, the activated state sequence is provided. Then we present graphically the processes of fault propagation and extraction of transformation constraints. Decisions in the high-level path activation are marked by stars (*) in the Figure. Extraction of path activation constraints is depicted below the striped line. Here, t corresponds to Boolean value 'true' and f corresponds to 'false'. As shown in Figure, we have to apply the constraint satisfaction process to the following set of constraints: in1 < in2 is false, in1 ≠ in2 is true.

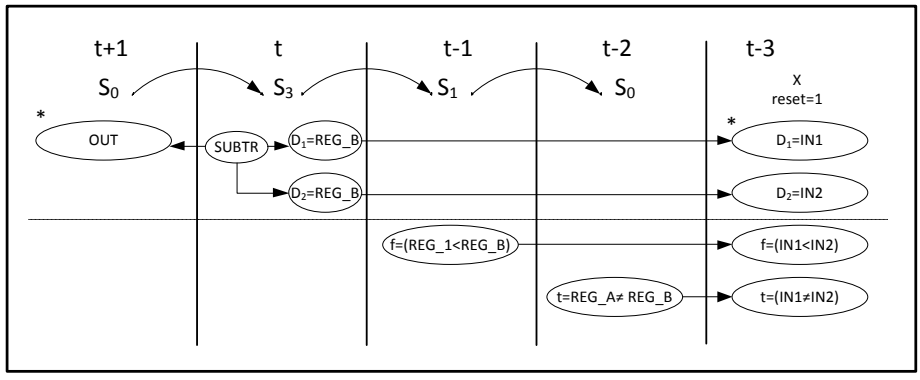


Figure 27. Constraint satisfaction process for GCD circuit

Subsequent to testing the node with the first path, backtrack occurs and the high-level path activation algorithm tries to find alternative path solutions.

5.6 Constraint logic programming and ECLiPSe

The concept of logic programming [47] was first developed in the 1970s, while the first Constraint Logic Programming (CLP) language was Prolog II [48], which was designed by Colmerauer in the early 1980s. The CLP over finite domains was first implemented in the late 1989 by Pascal Van Hentenryck [49] within the language CHIP [50].

The CLP is not a single programming language but a programming paradigm, which is parametric with respect to the class of constraints used in the language. Working with a particular CLP language means choosing a specific class of constraints, for example − finite domains, linear, or arithmetic, and a suitable constraint solver for that class. For example, a CLP over finite domain constraints uses a constraint solver which is able to perform consistency checks and projection over this kind of constraints.

ECLiPSe is a Prolog-based software system for the development and deployment of CLP applications. It includes constraint programming, mathematical programming, local search and various combinations of the above. It has some advantage compared with other CLP tools –

- embed ECLiPSe code to C, C++ or Java environments,
- open source project.

Embedding ECLiPSe means that you do not have to run a constraint solver program separately from your main application. For example the definition "#include <eclipse.h>" should be added into the directive part in C++ code. After that you may start calling ECLiPSe from you C++ application. Eclipse.so library need to link with application during the compile process. At runtime, application must be able to locate libeclipse.so.

ECLiPSe was born in 1991. At the beginning, the constraint programming features were initially based on the CHIP. The first released interface to an external state-of-the-art linear and mixed integer programming package was in 1997. The integration of the finite domain solver came in 2001 and the Interval Constraints (libIC) library was released in 2001. In 2006 ECLiPSe released as open source software [51].

ECLiPSe contains several constraint solver libraries, a high-level modelling and control language. For constraint solver only IC library is used in that work. LibIC supports Boolean constraint, arithmetic constraints, variable declarations (numeric ranges and numeric type declarations).

5.7 Solving the test path constraints

After the first test generation phase where module propagation and justification is performed and constraints for the test path were extracted, the constraints solver phase came. Here, the mission is to satisfy the constraints by using a constraint solver. Constraints that were built by the hierarchical ATPG are in string type. It means that the ATPG builds one and complete constraint string and gives it to the solver instead of sending constraint sub-blocks and the constraint solver itself creates a CLP. For example, a simple constraint string is given in the following:

```
lib(ic), L is (0), R is (2^3)-1, X1 :: L..R, X2 :: L..R, X1 #> X2, X2 #> 4, X2 #< 6, 2 #= X1 / 3,
indomain(X1, random), indomain(X2, random)
```

Figure 28. An example of a constraint string

This constraint string consists of five different groups:

1.	used library declaration	lib(ic)
2.	lower and upper domain boundaries definition	L is (0), R is (2^3)-1
3.	variables boundaries definition	X1 :: L..R, X2 :: L..R
4.	constraints	X1 #> X2, X2 #> 4, X2 #< 6, 2 #= X1 / 3
5.	search criteria for each variable	indomain(X1, random), indomain(X2, random)

Each group is separated from each other with a comma. Variables or constraints are also allocated by a comma inside a group. Note that search criteria are not a mandatory declaration. For example, in Figure 28, a constraint is given that must satisfy the following conditions − variables X1 and X2 must exist, both of them in the range between 0 and 7, where X1 is greater than X2,

X2 is greater than 4 and X1 divided by 3 should be equal with 2. The constraint will be satisfied, if X1 = 6 and X2 = 5.

When the conditions are inconsistent and the constraint is not satisfiable, random constraint solving is applied. As experiments presented in the next chapter show, the deterministic constraint solving has definite advantages over the pseudo-random method.

5.8 Experimental results

In order to evaluate the impact of the deterministic constraint solving, experiments on ITC99 and HLSynth92/95 benchmarks were carried out. By this moment the following three circuits are included in the analysis: b00, 604 and GCD because these circuits contain "equal to" comparison operators which are hard to test by pseudo-random constraint solving. In this experiment, the ECLiPSe constraint solver version 5.10_41 was used.

Table 4 shows the comparison of the semi-formal approach presented in [38] and the proposed fully deterministic approach. Comparison has been obtained by fault-simulating the test sets generated by both generators by a stuck-at fault simulator for sequential circuits. The row '# faults' of the Table shows the number of stuck-at faults in the circuit. The row '# tested' presents the number of tested faults by [38] and the proposed approach. The row 'cover., %' lists the achieved stuck-at fault coverage. 'time, s' stands for the ATPG run times in seconds. Finally, the number of generated test vectors is reported in the row '# vect.'

It can be seen that the fault coverage improvement obtained by the deterministic constraint solving setup ranges from 3 to 34 % for the tested examples. Note that while the fault coverage for the circuits is low, this is a usual case for the sequential ATPG because of the large number of untestable faults.

Table 4. Comparison of semi-formal [38] and the proposed deterministic ATPG methods

	B00		B04		GCD	
	semi-formal	current	semi-formal	current	semi-formal	current
faults	1328	1328	1488	1488	1658	1658
tested	251	714	899	943	1443	1519
cover, %	18,9	53,33	60,42	63,37	87,03	91,62
time, s	0,0053	0,0044	0,002	0,011	2,72	0,02
vectors	534	874	574	572	4471	4756

5.9 Chapter summary

In this chapter, a novel constraint-based hierarchical ATPG for RTL designs was introduced.

The tool combines the test path constraint activation with a constraint solver. First, a deterministic algorithm that extracts constraints for activating test paths at RTL is applied. Subsequently, a constraint solving package ECLiPSe is used for assembling the tests. Experiments on ITC99 and HLSynth92/95 benchmarks show that the proposed deterministic method offers very short run times. In particular, it provides increased fault coverage which ranges from 3 to 34 % for the tested examples with respect to earlier, semi-formal, approaches.

6 High-Level Decision Diagram based Fault Models

This chapter presents a set of fault models allowing a high coverage for sequential cores in Systems-on-a-Chip. A novel approach is presented combining three different fault models – a hierarchical fault model for functional blocks, a functional fault model for multiplexers and a mixed hierarchical-functional fault model for comparison operators. The fault models are integrated into a fast high-level decision diagram based test path activation tool. According to the experiments, the proposed method significantly outperforms state-of-the-art test pattern generation tools. The main new contribution of this approach is a formal definition of high-level decision diagram representations and the combination of the three fault models in order to target high gate-level stuck-at fault coverage for sequential cores.

6.1 Previous work

At present, efficient methods for testing sequential cores inside the Systems-on-a-Chip (SoC) are missing. The hard test pattern generation task is usually replaced by a theoretically much simpler approach relying on scan paths and the combinational ATPG. However, the scan-path method has its obvious shortcomings, including increased area and delay, and it also causes coverage of non-functional failure modes which results in over-testing and yield loss [41].

Several approaches to generating tests for structural faults in sequential cores have been proposed over the years. Despite of all the efforts the problem still lacks a breakthrough. At the gate level, a number of deterministic test generation tools, both academic [8] [9] and commercial, have been implemented. None of these methods can efficiently handle sequential designs of even a couple of thousands of gates. With the further growth of the circuit size fault coverage tend to drop while run times increase rapidly.

Better performance has been obtained with simulation-based approaches. Here, genetic algorithm based methods have been widely used [10] [11] [12]. Relatively efficient results have been obtained by spectral methods [42]. However, the simulation-based methods are fast for smaller circuits only and become ineffective when the number of primary inputs and the sequential depth of the circuit increase. Moreover, these methods do not guarantee detection for hard-to-test random pattern resistant faults.

Many works on FUTEG have been published in the past [13] [14]. In this field, an efficient technique based on BDD manipulation of data domain partitions has been proposed [43]. However, the fundamental shortcoming of the approaches that rely on functional fault models only is that they do not achieve satisfactory structural level fault coverage. Hierarchical and RTL test pattern generation has been proposed as a promising alternative to tackle complex sequential circuits.

Recently, a number of works have been published on implementing assignment decision diagram models [35] combined with SAT methods to address the RTL test pattern generation [44] [36]. An efficient RTL path activation has been previously proposed in [38] and complemented with precision fault models for multiplexers in [52]. The common shortcoming for all the former decision diagram based approaches is that they are targeting modules in the datapath of the circuit.

6.2 Motivation

As mentioned before, a common shortcoming of all the former decision diagram based approaches is that they are targeting modules only in the datapath. But in addition to datapath, there is a control unit. In order to overcome this problem, the goal is to combine three different fault models and integrate them into a fast high-level decision diagram based test path activation tool. Those three fault models are as follows:

1. Hierarchical fault model for functional units;
2. A functional fault model for multiplexers;
3. And a mixed hierarchical-functional fault model for comparison operators.

In the HLDD, both the control unit and the datapath are handled in a uniform manner. Previous research has shown that while deterministic test pattern generation algorithms are in general less powerful for larger circuits, they are still capable of testing a number of faults from the FSM part that the RTL and hierarchical methods are unable to cover. A new type of fault model is proposed based on the HLDD dedicated to faults in FSMs embedded into the RTL descriptions.

6.3 Mixed functional-hierarchical fault models

Next we will explain the fault models implemented in current approach where a combination of three fault models is used. These include a hierarchical fault model for the FU, a functional model for multiplexers and a combined hierarchical-functional model for conditional operations. We will describe each of the above models in more detail.

6.3.1 Hierarchical fault model for functional units

In order to target FU, a traditional top-down hierarchical fault model is implemented. At the high level, the state sequence necessary to propagate the local test data from primary inputs to inputs of the FU under test, and to propagate test responses from the outputs of the FU to the primary outputs is activated. In addition, test path constraints are extracted which need to be satisfied in order to allow the value propagation to take place. Subsequently, the local test pattern values are applied to fault simulation of the MUT at the gate level. In general, a single activated path is not enough to reach 100 % fault efficiency for a functional unit, i.e., a test set for a FU can consist of vectors generated during different high-level paths. Thus, a record is kept about the faults detected by the low-level tests during previous test paths.

6.3.2 Functional fault model for multiplexers

Next we introduce a functional fault model for targeting all the SSA faults in the multiplexers of hierarchical designs. In this thesis, only AND/OR multiplexers are considered but functional models for other multiplexer types can be derived in a similar way. The new functional model is based on distinguishing values at the data inputs of the MUX. For multiplexers having more than two data inputs we have chosen to implement pair-wise distinguishing of data inputs, as opposed to distinguishing all the inputs simultaneously. The main motivation for that is that high-level path justification from all the inputs in parallel may be difficult to achieve, or even inconsistent, in complex sequential architectures. Thus it would result in loss of solutions.

Makar et al. [52] presented the groundwork for deriving minimal tests for AND/OR, OR/AND and nMOS implementations of multiplexers. The functional fault model proposed in this approach is similar to [52] but it extends it with the ability to cover multiple stuck-at faults at the address select inputs of the MUX

under test. This, in turn, provides for better coverage of faults in the output logic of the control part FSM.

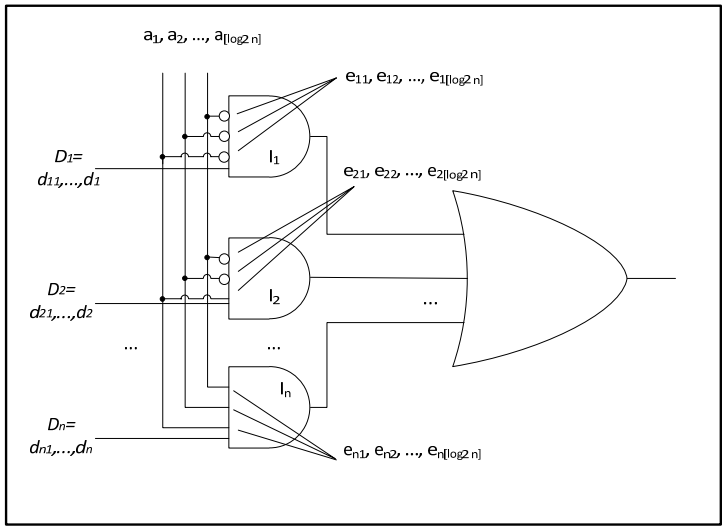


Figure 29. An n-input m-bit AND/OR multiplexer

Let us consider Figure 29 where the general structure of an n-input m-bit AND/OR multiplexer is presented. Inputs of the multiplexer are the data inputs $D_i = d_{i1},\ldots,d_{im}$, i = 1 … n, and the address select inputs A = a_1, …, a[$\log_2$n]. In addition, enabling lines E = $\{e_{ij}\}$, which are (inverted or non-inverted) fan-out's of address select inputs a_j are shown in the Figure.

The sub-circuit of the multiplexer starting with signals D_i and E and ending with outputs forms a fan-out-free cone. Thus, it is sufficient to test the stuck-at faults at the inputs of this cone to cover all the SSA faults in the cone. However, there is no need to explicitly test the data signals D_i in the hierarchical ATPG framework. This is due to the fact that all the SSA faults at these lines will be covered by the tests set up for the FU and signal busses connected to the respective multiplexer data inputs. In order to test the entire multiplexer it is necessary to additionally target the address select inputs a_j. Let us take a look at functional models for targeting faults in the signals a_j and E.

In order to test address select signals A=$\{a_j\}$, a functional fault model based on distinguishing of values of data signals D_i is implemented. In this approach, the distinguishing is carried out on a HLDD as shown in Figure 30. This Figure presents an example of a datapath HLDD for REG2 that contains a multiplexer address signal MUX1_ADDR, which is tested with address value V_k = 1. The constraint to make the behaviour of REG2 sensitive of faults in

MUX1_ADDR is that values at variables labelling the successors of node m must differ and a path to m must be activated (i.e., REG2_ena must have value 1).

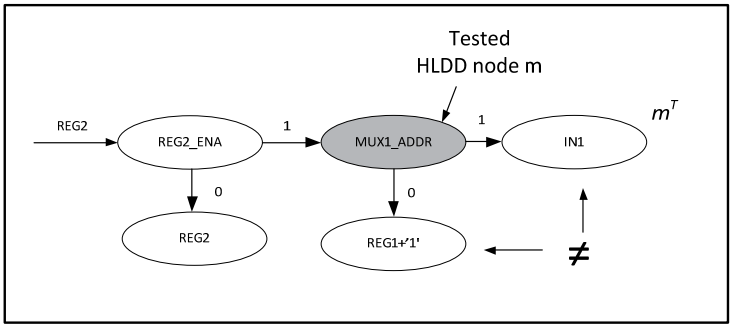


Figure 30. Functional fault model using HLDD node distinguishing

The functional fault model sets up the following constraints that, when satisfied, guarantee a 100 per cent SSA coverage for the corresponding multiplexers:

> Constraint 1. For covering all the SSA faults in signals $A=\{a_j\}$ it is sufficient that with selecting each address select value V_k we distinguish the value of data signal D_k selected by $A=V_k$ and all the data signal values, which are selected by address values, whose Hamming distance from V_k is 1.

The proof is straightforward. It is easy to see that a SSA fault at bus A would lead to a situation where another data input is selected. It is obvious that any SSA at A fault would change the value of A to a faulty value whose Hamming distance from the fault free value is 1. By distinguishing the data signals at corresponding inputs the effect of the SSA fault at A would be made observable at the output of the multiplexer. Performing such distinguishing for each value of A is sufficient to test all the SSA faults at A since both values (zero and one) would be covered at each bit positions by data signal distinguishing.

Let us discuss testing the SSA faults at the enabling lines $E=\{e_{ij}\}$, which are the inverted and non-inverted fan-outs of the address select bus (See Figure 31). It is important to note that these signals are in fact m-bit buses and thus testing the address select bus A does not necessarily cover all the faults in $\{e_{ij}\}$.

```
FOR each value Vi of A
  SSA1_mask := {0}m
  FOR each value Vk of A, Vk ≠ Vi
    A := Vk
      IF Hamming distance of Vi and Vk is 1
      Distinguish Di and Dk
      SSA1_mask := (Di⊕Dk)&Di
      Store the test setup
        WHILE SSA1_mask ≠ {1}m
        Distinguish Di and Dk
        new_mask := ((Di⊕Dk)&Di) | SSA1_mask
          IF new_mask ≠ SSA1_mask
            SSA1_mask := new_mask
            Store the test setup
          END IF
        END WHILE
      ELSE /* To test the state decoder : */
      Distinguish Di and Dk
      Store the test setup
    END IF
  END FOR
END FOR
```

Figure 31. Functional test setup algorithm

Note that stuck-at zero faults at these lines are already covered by the hierarchical test of the modules (FUs) connected to the data inputs of the multiplexer. Since we are considering functional fault model as a supplement to the hierarchical one, these faults do not have to be explicitly targeted.

Constraint 2. In order to test the SSA 1 faults at $\{e_{ij}\}$, the following constraints apply. With all the values V_L of A, for all the values V_k whose Hamming distance form V_L is 1. Signal A has to be assigned value V_k and the values of data signals (denoted by D_k and D_L) selected by the respective values $A=V_k$ and $A=V_L$ have to be distinguished. Since $\{e_{ij}\}$ are m-bit busses and the fault sensitizing values at each bit of D_L is required, it follows that over the set of distinguished D_k and D_L one values have to be distinguished at each bit position of D_L.

So far we have listed the constraints required for testing all the signals in the multiplexer. In Figure 31, an algorithm for test manifestation for a multiplexer in a hierarchical ATPG framework is derived based on the above constraints. In the algorithm description, D_i and D_k denote the values of the data inputs selected by address values V_i and V_k, respectively. SSA1_mask and new_mask are m-bit bit vectors. Characters ⊕, &, | correspond to bitwise XOR, AND and OR operations, respectively.

The algorithm shows how to set up functional tests that can be applied in the test manifestation stage in the HLDD-based path activation approach. Subsequently, propagation and justification procedures have to be executed to the test setups generated by this algorithm.

Note that the functional algorithm presented in Figure 31 distinguishes not only the data signal values selected by values, whose Hamming distance is one. This is done in order to extend the functional model to target the multiplexers to the logic in the control part, which is used for decoding the FSM states to control signals. While we have not developed a fault model to functionally cover the control part, it is possible to target a part of it by this functional fault model.

Note that treating MUXs by a hierarchical fault model similar to the one used for FUs would not have allowed targeting faults in the control part. Furthermore, a slow, three-valued fault simulator would have been needed for evaluating the fault coverage achieved at the low level in the multiplexers, because some of the module inputs would remain unspecified.

6.3.3 Mixed functional-hierarchical fault model for comparison operators

As experimental results presented in Chapter 6.5 show, the traditional hierarchical test approach targeting fault coverage at FU only is not efficient. In the previous subsection we introduced a functional fault model for multiplexers. However, this subsection proposes mixed functional-hierarchical fault models that have to be addressed in the HLDDs in order to set up tests for the conditional operations. According to experiments, this fault model improves the efficiency of the RTL test generation significantly.

The main challenges with testing comparison operators lies in the fact that these modules do not have a path to a primary output that could be activated through the datapath. Comparison operators provide inputs, called status bits, to the control part FSM. Thus, a functional fault model going via the FSM is needed in order to manage fault effect propagation. In addition, a hierarchical

fault model is necessary for targeting the structural faults in the comparison operations themselves.

The functional fault model is based on distinguishing the terminal nodes of the FSM HLDD that are successors to the node labelled by the tested status bit. There are two special cases, which are listed in the following:

1. Distinguishing the registers addressed by a pair of control words (i.e. FSM HLDD terminal nodes).

2. Distinguishing the mux inputs controlled by a pair of control words.

In a case where all the control signals in the two distinguished FSM terminal nodes are equal, we will perform the test setup in the respective next states.

In the case a wrong datapath register is addressed because of a fault in the next state logic, this will be revealed by distinguishing the values in the registers and performing propagation and justification through the circuit. Or, alternatively, if the two control words address the same set of registers, there can be a difference in some multiplexer address signals. In the latter case, the registers connected to inputs of the mux have to be distinguished.

Figure 32 shows an example of the test for comparison operators. Let the current MUT be a conditional operation whose output is connected to the status bit ‘status’. Let us consider the test setup for the module with the output value being one (i.e., status = 1). (Note that separate tests for the MUT have to be set up for output values 0 and 1, respectively.) As a first step, we try to identify possible setup states for the fault manifestation stage. In order to do that we select a nonterminal node in the FSM HLDD labelled by a variable signal. We activate the path to this node obtaining the following value assignments: reset := 0, state := S_j.

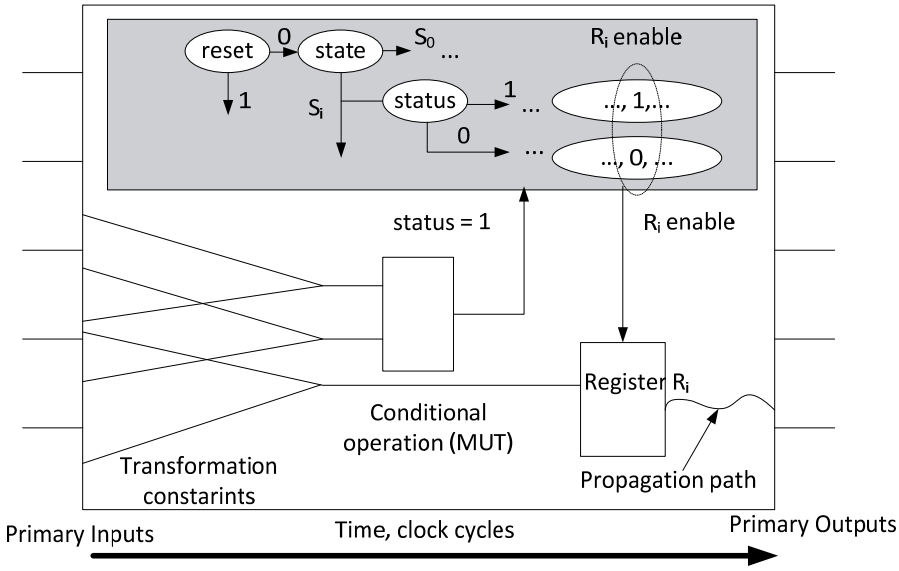


Figure 32. Test setup for conditional operations

The next step is to select the datapath register for fault effect activation. The register is determined by comparing the vectors at the two terminal nodes of FSM HLDD, which lie at the end of the faulty and fault-free paths, respectively. Registers which are selected by the fault free terminal node and deselected by the faulty one are considered to be suitable for fault activation. (In our example, register R_i matched the requirements and was chosen.) Subsequently, the propagation, justification and constraint extraction procedures are performed as explained in the previous section.

6.4 Experimental results

Table 5 presents the characteristics of the example circuits used in test pattern generation experiments in this approach. The following benchmarks were included to the test experiment: a GCD, an 8-bit sequential multiplier (MULT8x8), an Elliptic Filter (ELLIPF), an ALU based processor´(RISC) and a Differential Equation (DIFFEQ). The VHDL versions of GCD and DIFFEQ were obtained from high-level synthesis benchmark suites [53] [54] and the designs of MULT8x8 and RISC from FUTEG benchmarks [55]. The second column "# faults" shows the number of single stuck-at faults in the circuits, the third column "# FSM states" shows the number of states in the control part FSM, and the columns "PI bits" and "PO bits" present the number of primary

input and primary output bits, respectively. Finally, the 6th, 7th and 8th columns show the number of registers, multiplexers and FU respectively. All the experiments were run on a 366 MHz SUN UltraSPARC 60 server with 512 MB RAM under SOLARIS 2.8 operating system.

Table 5. Characteristics of the benchmark circuits

circuit	# faults	# FSM states	PI bits	PO bits	# of reg.	# of mux	# of FU
gcd16	1754	8	33	16	3	4	3
mult8x8	2036	8	17	16	7	4	9
ellipf	5388	28	130	113	17	7	3
risc	6434	4	26	16	8	4	4
diffeq	10008	6	81	48	7	9	5

In Table 6, a comparison of test generation results of four ATPG tools on the hierarchical benchmark designs is presented. Here, five sequential ATPG tools are compared. These are a gate-level deterministic ATPG HITEC [9] and a genetic algorithm based GATEST [10], hierarchical ATPG covering only datapath FUs (column "FU only"), hierarchical test pattern generation for FUs and multiplexers (column "FU only"), and finally, the approach proposed in current approach (column "current method"). Columns "F.C., %" give the single stuck-at fault coverage of the test patterns generated measured by the fault simulator from TURBO TESTER system [56], created at the Tallinn University of Technology. Columns "time, s" stand for test generation run-times achieved on a 366 MHz SUN UltraSPARC 60 server with 512 MB RAM under SOLARIS 2.8 operating system.

Table 6. Comparison of sequential circuit test generation tools

	HITEC		GATEST		FU only	FU+MUX		current method	
circuit	F.C, %	time, s	F.C, %	time, s	F.C, %	F.C, %	time, s	F.C, %	time, s
gcd16	59,11	365	86,13	190,7	62,55	85,71	497,4	90,95	677,4
mult8x8	65,9	1243	69,2	821,6	69,4	74,2	76,9	74,7	93,7
ellipf	87,9	2090	94,7	6229	86,7	95,04	1258,9	95,04	1258,9
risc	52,8	49020	96	2459	83,9	96,5	150,5	96,5	150,5
diffeq	96,2	13320	96,4	3000	96,44	96,52	441,7	97,09	453,7
average F.C:	72,4		88,4		79,8	89,6		90,9	

The results show that the proposed method is very efficient for testing sequential designs. It achieves on the average 2.5 % higher fault coverage than the genetic tool GATEST on the given benchmark set. For the sake of

comparison, an experiment with complete tests for datapath FU was performed. This resulted in a poor overall coverage of the design, thus, showing the need for FSM and multiplexer testing in the RTL ATPG. Note that the mixed fault model did not improve fault coverage for circuits ELLIPF and RISC. This was due to the fact that these circuits do not contain any comparison operators. One of the main reasons for consistently low fault coverage in case of all the five approaches for MULT8X8 was mainly due to a large number of sequentially untestable faults. This issue was revealed by the deterministic ATPG HITEC that was able to prove a large number of faults to be untestable. A redesign for testability of this circuit would have increased the fault coverage for all the tools.

6.5 Chapter summary

In the chapter we defined the HLDD as an efficient model for RTL test pattern generation for sequential cores. The HLDD model was compared to currently popular assignment decision diagrams. It was pointed out the core benefits of the former and presented the HLDD based test path activation algorithm. The main novel contribution of this approach is the combination of the three HLDD-based fault models in order to provide for efficient and fast testing of sequential designs. Experiments show that these fault models allow reaching higher stuck-at fault coverage when compared to other approaches. In addition, our experimental results prove quite clearly that RTL test methods targeting datapath FU only cannot guarantee a high fault coverage for the overall design.

7 Identifying Untestable Faults in Sequential Circuits Using Test Path Constraints

In this chapter a constraint based untestable faults identification method is introduced. The method is based on the hierarchical approach where test path constraints extracted at the RTL are applied to prove untestable faults at the gate level. First, the concept of test path constraints for testing a module in the RTL design is presented. Then the procedure of extracting test path constraint by algorithm is shown.

7.1 Previous work

A number of works have been proposed in order to tackle the problem of identifying sequentially untestable faults. The first methods [57] were fault-oriented and based on applying the combinational ATPG to the expanded time-frame model of the sequential circuit. However, such an approach does not scale because of the size-explosion of the unrolled sequential models. Thus, a fault-independent method was introduced by Iyer et al. in [58]. The new algorithm was called FIRES and it implemented illegal state information to complement redundancy analysis. This was followed by a number of fault independent methods including MUST [59], FUNI [60], FILL [60] and others. Liang [61] proposed a simulation-based approach for sequential untestable fault identification. However, it was shown in [60] that this method may result in 'false positives', i.e., a fault may be declared untestable when there actually is a test for it. The common limitation of the above methods is that they operate at the logic-level representation of the design. Thus, a considerable amount of effort is put on the implication process carried out at the level of logic netlists.

In their previous work [62], the authors introduced a specific subclass of sequentially untestable faults, called register enable stuck-on faults and a method for proving them untestable using a model checker.

Early hierarchical methods on bottom-up RTL testing relied on the assembly of module tests and were applicable of the simplest systems only [29]. A more solid basis for the bottom-up paradigm was laid by Ghosh et al. in [44]. In their work, test environments are generated for each functional module of a

given functional RTL circuit described in an ADD [35] using symbolic justification/propagation rules using a nine-valued algebra. In this method, a test sequence is then formed by substituting the corresponding test patterns in the test environment. However, the proposed nine-valued algebra cannot guarantee the generation of a test environment, even if it exists. To overcome this drawback, Zhang et al. upgraded the nine-valued algebra to a ten-valued algebra by taking the variable line value range into consideration. This algebra is able to generate much more test environments [36]. In [37], Zhang's approach has been further improved by introducing additional propagation rules.

Lee and Patel introduced a top-down constraint-based test pattern generation for microprocessors in [30]. Several constraint-based top-down approaches followed, including [63] [64]. [65] proposed a bottom-up approach based on a HLDD engine and on applying a commercial constraint solver SICStus. As experiments show, the tool achieves a lower fault coverage in comparison to a commercial logic-level ATPG. In [46], a top-down approach including a constraint solving package ECLiPSe [40] has been proposed.

None of the previous methods apply RTL constraints in order to prove logic-level untestable faults. Thus, the fault efficiency reported by the approaches [30] − [46] is often low, which decreases the test engineer's confidence in the test. Here, by fault efficiency we mean the ratio of the number of tested faults to the number of testable faults.

7.2 Motivation

Test generation for sequential synchronous circuits is a time-consuming task. ATPG tools spend a lot of effort not only on deriving test vectors for testable faults but also on proving that there exist no tests for the untestable faults. Because of this reason, the identification of untestable faults has been an important aspect in speeding up the sequential ATPG.

The percentage of untestable faults in sequential circuits tends to be considerably higher than in the combinational ones. For combinational circuits, untestable faults occur due to the redundant logic in the circuit, while for sequential circuits untestable faults may also result due to unreachable states or due to impossible state transitions.

The main goal of this work is to process the set of constraints in order to derive conditions for a dedicated logic-level ATPG in proving untestability. The new method allows detecting sequential untestability in combinational modules (functional units, multiplexers) embedded into a hierarchical circuit and is based on path activation constraints extracted by a RTL ATPG.

7.3 Constraint-based untestability proof flow

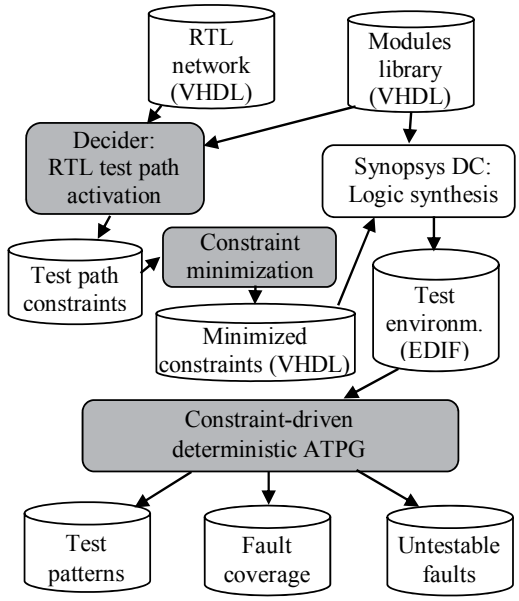


Figure 33. Constraint-based untestability proof flow

Figure 33 presents the corresponding top-down test flow for targeting a MUT in a hierarchical RTL design. The flow contains three main phases that are marked with grey. During the first phase, RTL test path activation, the full set of constraints for setting up a test path to test an RTL module is extracted on the RTL design representation. We apply the RTL hierarchical ATPG [46] in order to extract the constraints for accessing the MUT. Hierarchical ATPG activates as many sets of constraints as there are test paths for that module in a bounded limit of clock cycles. In [46], test constraints were utilized to propagate test patterns to and from the MUT.

During the second phase this set of constraints is minimized as presented in Section 6.5 resulting in a compact test environment for accessing the MUT. The test environment is translated into VHDL and synthesized to logic-level using Synopsys Design Compiler (SDC).

The third phase generates deterministic tests to the logic-level module taking into account the minimized path constraints. Here, the constraint-driven logic-level ATPG is run on the logic-level description of the MUT instantiated into the synthesized test environment. As a result we obtain the list of

sequentially untestable faults in the MUT as well as test patterns for the entire design.

7.4 Test path constraints extraction at the RTL

For each datapath MUT, we extract control part FSM state sequences in order to propagate fault effects from the output of the MUT to primary outputs and to propagate the values from the primary inputs to the inputs of the MUT. Such state sequences constitute test paths for accessing MUT. We represent the test paths by sets of constraints. All test paths within a certain cycle limit are activated and the corresponding constraints extracted by the proposed algorithm. This cycle limit is first set to 1 and then gradually incremented until the obtained constraints will be non-empty after the minimization. In order to extract the RTL test path constraints in this approach, a test path activation tool DECIDER [46] is applied.

Consider the general case of test path constraints for a MUT presented in Figure 19 in Section 5.4. Such constraints are extracted as follows. First, the value from the output variable x_i of the MUT f_i is propagated to a primary output $x_{O,j}$ by activating a state sequence $x_S(t) \rightarrow x_S(t+1) \rightarrow ... \rightarrow x_S(t+n)$ in the control part. Here, by $x(t)$ we denote the value of variable x at the clock cycle t. Thus, the propagation state sequence starts at a time step t, which is referred to as the *manifestation step*, and it ends at a clock cycle $t+n$. During the propagation, path activation constraints $c_{A,p} \in C_A$ are created at time steps where the next state value of x_S depends on the status bits X_N. When the fault effect value propagates from x_i to $x_{O,j}$ at the time step $t+n$ then the propagation constraint c_P is created.

Subsequent to the propagation, the constraint justification process begins. Starting from the time step $t+n$, we move backwards in time until the manifestation step t is reached. At each time step we update the propagation constraint c_P and those path activation constraints $c_{A,p}$ whose creation time step is later than current time step. During the update, the unjustified variables $X'' \subseteq X_R$ of the constraint expressions $g(X', X'')$ for all the constraints are substituted by expressions $h_i(X'''_i)$ on model variables $X'''_i \subseteq X_R \cup X_I$, where $h_i(X'''_i)$ are the expressions implemented by functional units F_U selected according to the values of control signal variables X_C at the current time step.

At the manifestation time step t, we create the transformation constraints for each input of the MUT. Without loss of generality, Figure 19 shows a MUT with two inputs $x_{i,1}$ and $x_{i,2}$. Thus, in current case the transformation constraints $c_{J,1} \in C_J$ and $c_{J,2} \in C_J$ are created, respectively. We continue moving backwards in time until at some time step $t-m$ all the variables in the constraints are primary inputs X_I. During this process we update all the created constraints and create

new path activation constraints $c_{A,p}$ at time steps where the previous state value of x_S is depending on the status bits X_N.

Note that the extracted constraints contain expressions $g(X)$ on primary inputs X_I and constants. (In the case of the propagation constraint c_P the expression also depends on the MUT output x_i). The expressions are determined by the functions implemented by functional units F_U and, in the case of path activation constraints $c_{A,p}$, also by comparison operations F_N. The exponential size complexity of the constraints expression $g(X)$ is avoided by uniting multiple occurrences of the same variable (i.e., the literals) in the constraints at each time step into one single fan-out variable. Because of this, the size requirements for the constraints are linear with respect to justification time frames and they represent a subset of the expanded time-frame model of the circuit.

After one consistent set of test path constraints are extracted by Decider, a backtrack occurs and the tool attempts to use alternative propagation and justification paths. The process ends when all the consistent test paths within a certain time step limit are activated and respective test path constraints are extracted.

7.5 Minimization of test path constraints

The minimization step is required due to the fact that the full set of test path constraints may become large considering their representation in the VHDL and performing logic synthesis on them. The latter is needed to handle the constraints by the gate level ATPG in order to prove untestable faults.

Every test path $p_i \in P$, with P being the set of all the test paths for a MUT within a given time frame, may be represented as a triple $\langle c_{P,i}, C_{J,i}, C_{A,i} \rangle$, where $c_{P,i}$ is the propagation constraint, $C_{J,i}$ is the set of justification constraints and $C_{A,i}$ is the set of path activation constraints extracted for the test path p_i, respectively. We can represent the full set of test paths P by a DNF formula Φ, where terms correspond to the test paths p_i and literals are the constraints $c_{i,j}$ belonging to the test paths and represented as quantifier-free bitvector (QFBV) predicates. The three groups of constraints $c_{P,i}$, $C_{J,i}$ and $C_{A,i}$ are each minimized separately.

Minimization of the DNF formula Φ takes place as follows. First of all, we minimize the propositional skeleton of the formula (a Boolean expression where all predicates are replaced by propositional variables) using a state-of-the-art algorithm ESPRESSO [66].

Second, some constraints in the test paths can be redundant. In order to remove such redundancies we apply a method presented in [66] and briefly described here. Consider a first-order logic formula Φ given in a negation normal form. First, we build a tree where intermediate nodes represent either $\vee$ or $\wedge$ operations and leafs represent QFBV predicates. The idea is to test each

leaf L against a special formula α_L, called the *critical constraint*. If $\alpha_L \Rightarrow L$ then L can be replaced by true, and if $\alpha_L \Rightarrow \neg L$ then L can be replaced by false. Assume, for example, that Φ is presented in DNF:

$$\Phi = \vee_{i=1}^{n} \wedge_{j=1}^{m_i} L_{ij}$$

Then, for a leaf L_{kl}, $1 \leq k \leq n$, $1 \leq l \leq m_k$,

$$\alpha_{L_{kl}} = \left(\vee_{\substack{i=1 \\ i \neq k}}^{n} \wedge_{j=1}^{m_i} \neg L_{ij} \right) \wedge \left(\wedge_{\substack{i=1 \\ i \neq l}}^{m_k} L_{ki} \right)$$

To test whether α_L implies L or $\neg L$ we use an SMT solver Z3 [67].

7.6 Constraint-driven ATPG for proving untestability

We use the assignment decision diagram ADD [35] data structures in order to illustrate the test path constraints.

Consider Figure 34, which gives the ADD for the full set of constraints P extracted for the GCD example. In other words, the MUT can only be tested using one of the two test paths presented in Figure 34A and 34B. The two test paths contain only path activation constraints and the paths are identical except for the fact that the primary inputs IN1, IN2 are swapped in them.

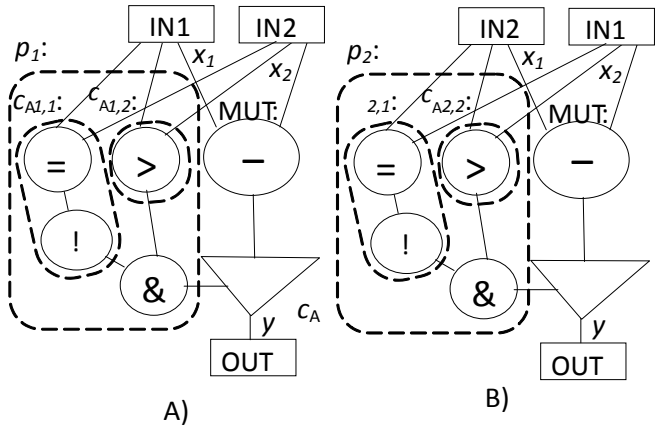


Figure 34. A full set of test path constraints for the MUT

Note that from the point of view of accessing the MUT these two environments are equivalent. It is irrelevant which primary input is used in applying the test patterns when representing the constraint-based test environment for proving untestability. Therefore, we denote the value justified from the *k*-th input of the MUT by x_k and the value propagated from the MUT output by y.

The test paths p_1 and p_2 both consist of two path activation constraints $c_{A_{1,1}}$, $c_{A_{1,2}}$ and $c_{A_{2,1}}$, $c_{A_{2,2}}$, respectively. $c_{A_{1,1}}$ (which is equivalent to $c_{A_{2,1}}$) states that x_1 must not be equal to x_2. $c_{A_{1,2}}$ (equivalent to $c_{A_{2,2}}$) states that x_1 must be greater than x_2. Since all the path activation constraints $c_{i,j}$ within a test path should hold simultaneously they are combined using the conjunction operator. In turn, all the test paths p_i are combined using the disjunction operation because any one of them may be applied for accessing the MUT. Therefore, we can combine the constraints into a DNF as follows:

$$\bigvee_i \bigwedge_j c_{i,j}.$$

Subsequent to combining the test path constraints, the constraint minimization is performed. Using the method presented in previous Section we obtain for the example in Figure 34:

$$(x_1 \neq x_2) \wedge (x_1 > x_2) \vee (x_1 \neq x_2) \wedge (x_1 > x_2) = x_1 > x_2.$$

Figure 35 shows the ADD for the minimized test environment resulting for testing the MUT of the example presented in Figure 34. The constraint shows that the MUT (a subtractor) may only be accessed when the first input of it, i.e., x_1 is greater than the second one, x_2.

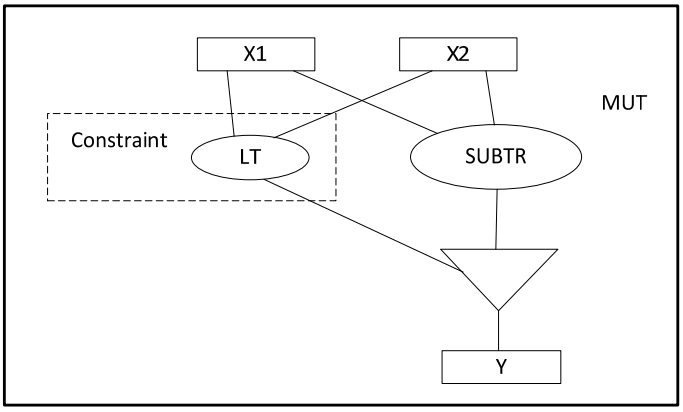


Figure 35. Constraint-based test environment for MUT

The obtained test environment, excluding the MUT, is automatically translated into the VHDL and synthesized to a logic level using SDC. The MUT is linked by instantiating its logic level description into the VHDL of the test environment. Subsequently, the constraint-driven logic-level ATPG is run. As a result we obtain the list of sequentially untestable faults in the MUT as well as test patterns for testing the MUT.

7.7 Discussion on the effect of the top-down proof

As a side effect, the test environment allows us to evaluate the accuracy of bottom-up hierarchical ATPG. In particular, the strict interpretation of Ghosh's algebra [8] leads to overly pessimistic results because tests for some MUTs are aborted due to justification conflicts. On the other hand, the weak interpretation is too optimistic and can also lead to loss of fault coverage because some of the test patterns that are expected to cover faults in the MUT do not propagate.

Consider the case where in a bottom-up scenario we have a deterministic test T_q generated for the MUT in a stand-alone mode reaching the maximum fault coverage W_q for the MUT. Then, we generate the test environment for the module and substitute T_q into this test environment. Due to the test path constraints the actual fault coverage that can be achieved for the MUT embedded inside the network is W_a, which is generally lower than the stand-alone fault coverage W_q. However, when we fault simulate T_q substituted into the test environment we obtain a fault coverage W_r, where $W_r \leq W_a \leq W_q$.

In other words, the bottom-up approach may lose some fault coverage with respect to the top-down one because the set of the tests to choose from is restricted to T_q. If the bottom-up test generation algorithm for the MUT had had some knowledge about the test path constraints it would have generated a different test T_a whose fault coverage would have been equal to W_a. Thus, a deterministic ATPG taking into account the test path constraints is necessary in order to achieve maximum fault coverage and also to prove untestability within sequential circuits. Experiments with the constraint-driven deterministic ATPG presented in Section 6.9 show that the difference between the coverage W_r and W_a may be even as high as 8-14 per cent of stuck-at coverage.

7.8 Limitations and threats to validity

One of the main limitations of the current implementation of the hierarchical untestability identification tool is the fact that the RTL circuits considered are strictly divided into a control and datapath parts. Vast majority of real-world RTL designs are not restricted to the single control part concept. However, this limitation is related to the path activation engine applied [46] and it is not a principal one for the presented method. For example the steps of minimization of constraints and the constraint-driven gate-level ATPG are completely independent of this restriction. Furthermore, it is possible to extend [46] into an RTL path activation tool that would support a network of control part FSMs as opposed to a single one.

Another limitation is the requirement that the modules selected for untestability analysis from the RTL design must be combinational. The method could be easily extended to support pipelined modules. In addition, there exists an efficient top-down method for proving untestable faults in register modules based on bounded model-checking [62]. However, the method cannot be currently applied to arbitrary sequential modules.

Finally, the complexity of the DNF of the constraints in the minimization step of the method grows exponentially with the increase of the cycle limit k of the path activation. Table 7 shows the dependency between the numbers of leaves in the constraints DNF as a function of the cycle limit k. Three modules have been included to the analysis: a subtraction function from the GCD

benchmark and two additional modules from the b04 circuit from the ITC99 benchmark family [68]. Figure 36 visualizes that dependency on a logarithmic scale. The benchmarks GCD and b04 were selected to analyse the complexity because the curve cannot be explored on DIFFEQ and MULT8X8 examples tested in the experimental results section. This is due to the fact that for both the DNF is empty until a certain cycle limit is reached.

Table 7. Number of leaves in the DNF as a function of the cycle limit k

k	Number of leaves in the DNF		
	b04 (AVERAGE1)	b04 (AVERAGE2)	gcd (SUBTR)
1	0	0	0
2	20	10	2
3	1216	610	14
4	50867	25408	54
8	N/A	N/A	444

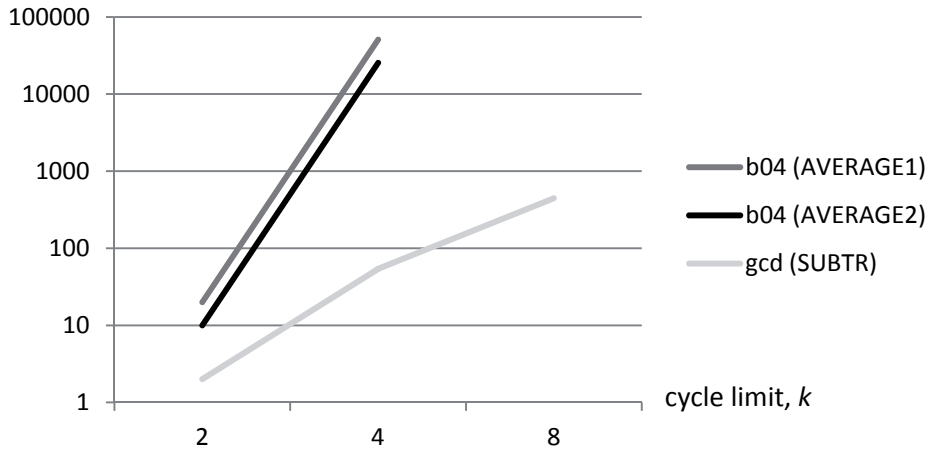


Figure 36. Number of leaves in the DNF as a function of the cycle limit k.

7.9 Experimental results

In order to evaluate the hierarchical untestability identification and test generation method, experiments on HLSynth92 [53] and HLSynth95 [54] benchmarks were run. In addition, to compare the solution with the traditional

bottom-up approach (e.g., [36]) and assess its fault efficiency, a comparative study was carried out.

Table 8 presents the characteristics of the example circuits used in test pattern generation experiments in this approach. The following benchmarks were included to the test experiment: a GCD, MULT8X8, and a DIFFEQ. In the Table, the number of single stuck-at faults, the number of primary input and primary output bits, and the number of registers F_R, multiplexers F_M and functional units F_U in the RTL code are reported, respectively. The final column presents the upper limit for control part FSM cycles (i.e., the maximum times the same control state is traversed) as a time-step bound for the untestability proof. This bound is dependent on the design functionality and can be set by the test engineer.

Table 8. Benchmark characteristics

circuit	#faults	PI bits	PO bits	# reg. ($\|F_R\|$)	# Mux ($\|F_M\|$)	# FU ($\|F_U\|$)	time limit
gcd	472	33	16	3	4	3	5
mult8x8	2356	17	16	7	4	9	8
diffeq	10326	81	48	7	9	5	7

Table 9 shows experiments reporting the time spent by different stages of the constraint-driven untestability identification flow. Note that not all the modules (multiplexers F_M and functional units F_U) in the RTL designs are affected by sequential untestability. Our method identified one module from GCD, three modules from MULT8X8 and two modules from DIFFEQ whose minimized constraints were a non-empty set. Thus, only the above-mentioned six modules were considered in the hierarchical untestability proof by the constraint-driven logic-level ATPG. As it can be seen from the Table 8, the extraction of test path constraints required up to one minute of run time. As discussed in Section 6.5 the constraint minimization step is very much dependent on the time-step bound. In the case of ADD2 the time-step bound k is 7 and the time for minimizing the constraints is accordingly more than 4000 seconds. The test environment synthesis from VHDL to logic-level using SDC remained almost constant and was around 5 to 10 seconds per module while the deterministic constraint-based ATPG spent less than 0.02 seconds per MUT.

Constraint extraction was performed on a 2.5 GHz, Intel Core2 Duo T9300 PC with 4 GB of RAM, constraint minimization on a 2 GHz, Intel Core 2 Duo P7350, 3GB RAM on Windows 7 Pro OS and the synthesis and test experiments were carried out on a Sun-Fire-V250 station with 1.28 GHz sparcv9 processor on Solaris 2.9 OS.

Table 9. Constraint-driven top-down ATPG versus bottom-up ATPG

circuit	**gcd**	**mult8x8**			**diffeq**	
module	SUBTR	ADD2	ADD3	SUBTR	MUX3	MUX4
constraint extraction, s	2,9	47,86			9,18	
constraint minimization, s	0,05	4710	< 0,01	52	14	82
synthesis, s	5,38	5,33	9,52	5,25	5,1	5,1
ATPG, s	0,01	0,01	< 0,01	0,02	< 0,01	< 0,01

The experiments in Table 10 present comparison of the proposed method to the bottom-up paradigm [36]. For creating the test library for the bottom-up approach, the modules were first tested by the ATPG in a stand-alone mode. As a result, a test sequence T_q yielding 100 % stuck-at fault coverage W_q was obtained. The proposed top-down constraint-driven ATPG reached fault coverage W_a which was less than W_q because of the constraints when accessing the MUT that was embedded into the network. However, the fault efficiency of the proposed approach was always 100 % for all the modules.

When test T_q was substituted to the test environment in a bottom-up manner then fault coverage W_r was reached, which was always lower than W_a because some of the tests were invalidated by sequential dependencies. In fact, W_r was considerably lower (by 8-14 %) for all the four modules analysed. Thus, the proposed top-down method was capable of reaching maximum fault coverage for the analysed modules with respect to the test path constraints and proving all of the sequentially untestable faults in them.

Table 10. Constraint-driven top-down ATPG versus bottom-up ATPG results for circuit modules

circuit	gcd	mult8x8			diffeq	
module	SUBTR	ADD2	ADD3	SUBTR2	MUX3	MUX4
W_q, %	100	100	100	100	100	100
W_a, %	95,74	86,64	55,88	85,33	75	75
W_r, %	85,11	72,49	47,06	74,07	64,71	64,71

Table 11 presents detailed statistics of the circuits analysed. The table lists the total number of stuck-at faults in the whole circuit, the number of tested faults, number of unobservable/uncontrollable faults, untestable register faults from [62], the number of faults proven sequentially untestable by the proposed

constraint-based approach and finally the number of all the remaining faults. The experiments show the efficiency of the constraint-driven engine in untestability identification. Though the method quickly classifies untestable faults caused by sequential untestability in the considered modules with 100 % efficiency, there remain a number of faults in other modules, including in the control part, which are still neither tested nor proven untestable. Some of these remaining faults can be tested or proven untestable by ATPG approaches at the logic-level.

Table 11. Breakdown of faults

	gcd	*mult8x8*	*diffeq*
# total faults	472	2356	10326
# tested faults [46]	439	1737	9867
# unobs./uncontr. faults	28	195	252
# untestable register faults [62]	0	130	130
# sequentially untestable faults	**4**	**156**	**68**
# remaining faults	1	138	9

In order to evaluate the fault efficiency (i.e., the ratio of the number of tested faults to the number of testable faults) of the proposed approach it was compared with a commercial ATPG from a major CAD vendor. The commercial ATPG is based on a deterministic gate-level algorithm. The results of the experiments are shown in Table 12. As it can be seen, the gate-level tool obtained comparable fault efficiency only in the case of the MULT8X8 example. In the case of GCD and DIFFEQ benchmarks there was a large percentage of faults aborted by the tool.

Table 12. Comparison of fault efficiency

	Fault efficiency, %	
Circuit	Commercial ATPG	Constraint-based + register untestability
gcd	76,55	99,79
mult8x8	89,06	89,90
difeq	97,25	99,91

7.10 Chapter summary

A new method for hierarchical untestable stuck-at fault analysis of non-scan sequential circuits is presented. The method is based on extracting and minimizing RTL test path activation constraints that drive a dedicated logic-level deterministic ATPG. Experiments show that it is capable of generating tests yielding maximum fault efficiency for modules embedded into the RTL.

In addition, our study shows that traditional test generation at RTL based on symbolic test environment generation is too optimistic due to the fact that constraints in accessing the modules under test have been ignored. Experiments showed that bottom-up strategies caused a decrease of stuck-at fault coverage up to the range of 8−14 % in the modules tested when compared to the proposed approach. This short-coming is overcome by the proposed top-down constraint-based method which obtains 100 per cent stuck-at fault efficiency with respect to the sequential testability constraints for all the modules considered.

8 Thesis conclusions

This thesis concentrates on the hierarchical ATPG for synchronous sequential circuit that has been proposed as a promising alternative to tackle complex sequential circuits.

To summarize the main contributions of the thesis are:

- **A novel deterministic constraint-based method for hierarchical ATPG for RTL.**

The method combines test path constraint activation with a constraint solver where a deterministic algorithm that extracts constraints for activating test paths at RTL is applied. Subsequently, a constraint solving package ECLiPSe is used for assembling the tests. Experiments show that the proposed deterministic method offers very short run time. In particular, it provides increased fault coverage which ranges from 3 to 34 % for tested examples with respect to earlier, semi-formal, approaches.

- **A novel fault model combined together with hierarchical fault model, functional fault model and a mixed hierarchical-functional fault model.**

The main novel contribution of this approach is the combination of the three HLDD-based fault models in order to provide for efficient and fast testing of sequential designs. The method defined the HLDD as an efficient model for RTL test pattern generation for sequential cores. The HLDD model was compared to currently popular assignment decision diagrams. Experiments show that these fault models allow reaching higher stuck-at fault coverage when compared to other approaches for testing sequential. It achieves on the average 2.5 % higher fault coverage than the genetic tool GATEST on the given benchmark set.

- **A novel method for identifying untestable faults in sequential circuits.**

The method is based on extracting and minimizing RTL test path activation constraints that drive a dedicated logic-level deterministic ATPG. Experiments show that the tool is capable of generating tests yielding maximum fault efficiency for the embedded modules under test. To the best of the authors knowledge this is the first method that can prove sequential untestability starting from the RTL. In addition, our study shows that traditional test generation at RTL based on symbolic test environment generation is too optimistic due to the fact that constraints in accessing the modules under test have been ignored.

Experiments presented in this work showed that bottom-up strategies caused a decrease of stuck-at fault coverage up to the range of 8-14 % in the modules tested when compared to the proposed approach. This short-coming is now overcome by the proposed constraint-based method which obtains 100 per cent stuck-at fault efficiency for all the modules considered.

All three approaches were included into hierarchical test generation tool named Decider. The feasibility of all proposed methods was proven by the presented experimental results by ITC, HLSynth92 and HLSynth95 benchmarks.

Reference

[1] Sunk by Windows NT. *WIRED.* [Online] 24 07 1998.

[2] **Leyden, John.** Malware implicated in fatal Spanair plane crash. *TechNewsDaily.* [Online] 20 08 2010. http://www.msnbc.msn.com/id/38790670/ns/technology_and_science-security/.

[3] **G. Moore.** *Cramming more components onto integrated circuit.* Electronics, 38(8), 114–117, 1965.

[4] **Peter Clarke.** *Intel enters billion-transistor processor era.* EE Times, 14 October 2005.

[5] /microprocessor_timeline.pdf. [Online] http://www.intel.com/pressroom/kits/core2duo/pdf/microprocessor_timeline.pdf.

[6] **R.Klein, T.Piekarz.** *Accelerating Functional Simulation for Processor Based Designs.* Mentor Graphics Corporation, 2005.

[7] **Laung-Terng Wang. Cheng-Wen Wu, Xiaoqing Wen.** *VLSI Test Principles and Architectures: Design for Testability.* Morgan Kaufmann Publishers is an imprint of Elsevier, 2006.

[8] **H.-K.T Ma, S. Devadas, A.R. Newton, A. Sangiovanni-Vincentelli.** *Test Generation for Sequential Circuits.* IEEE Trans. on CAD, Vol. 7, No. 10 pp. 1081-1093, Oct. 1988.

[9] **T. M. Niermann, J. H. Patel.** *HITEC: A test generation package for sequential circuits.* Proc. European Conf. Design Automation (EDAC), pp.214-218, 1991.

[10] **Elizabeth M. Rudnick, Janak H. Patel, Gary S. Greenstein, Thomas M. Niermann.** *Sequential circuit test generation in a genetic algorithm framework.* Proc. DAC, pp. 698-704, 1994.

[11] **F. Corno, P. Prinetto, et al.** *GATTO: A genetic algorithm for automatic test pattern generation for large synchronous sequential circuits.* IEEE Trans. CAD, pp.991-1000, Aug. 1996.

[12] **M. S. Hiao, E. M. Rudnick, J. H. Patel.** *Sequential circuit test generation using dynamic state traversal.* Proc. European Design and Test Conf., pp. 22-28, 1997.

[13] **D. Brahme, J. A. Abraham.** *Functional Testing of Micro-processors.* IEEE Trans. Comput., vol. C-33, 1984.

[14] **A. Gupta, J. R. Armstrong.** Functional fault modeling. 30th ACM/IEEE DAC, pp. 720-726, 1985.

[15] **Davis, B.** *The Economics of Automatic Testing.* London, United Kingdom, McGraw-Hill, 1982.

[16] **Niraj Jha, Sandeep Gupta.** *Testing of Digital Systems.* Cambridge University Press, 2003.

[17] **Pomeranz, Irith.** *To Overtest Or Not To Overtest - More Questions Than Answers.* Test Symposium. ATS '06. 15th Asian, 2006.

[18] **Michael L. Bushnell, Vishwani D. Agrawal.** *Essentials Of Electronic Testing For Digital, Memory And Mixed-Signal Vlsi Circuits.* Kluwer Academic Publishers, 2000.

[19] **Raik, Jaan.** *Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams.* Tallinn: Tallinn University of Technology Press, 2001.

[20] **Miron Abramovici, Melvin A.Breuer, Arthur D. Friedman.** *Digital Systems testing and Testable Design.* IEEE Press, 1990.

[21] **Kruus, Helena.** *Optimization of Built-in Self-Test in Digital Systems.* Tallinn: Tallinn University of Technology Press, 2011.

[22] **Roth, J.** *Diagnosis of automata failures: A calculus and a method.* IBM J. Res. Develop., 10(4), 278–291, 1966.

[23] **Goel, P.** *An implicit enumeration algorithm to generate tests for combinational.* IEEE Trans. Comput., C-30(3), 215–222,1981.

[24] **Shimono, H. Fujiwara and T.** *On the acceleration of test generation algorithms.* IEEE Trans. Comput., C-32(12), 1137–1144, 1983.

[25] **M. H. Schulz, E. Trischler, and T. M. Serfert.** *SOCRATES: A highly efficient automatic test pattern generation system.* IEEE Trans. Computer-Aided Design, CAD-7(1)1988.

[26] **P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli.** *Combinational test generation using satis.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 15(9):1167 -1176, 1996.

[27] **Larrabee, T.** *Test pattern generation using Boolean satisfability.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,11(1):4-15., 1992.

[28] **Sakallah, J. P. Marques-Silva and K. A.** *Robust search algorithms for test pattern generation.* In International Symposium on Fault-Tolerant Computing, pp. 152-157, 1997.

[29] **B. T. Murray, J. P. Hayes.** *Hierarchical test generation using precomputed tests for modules.* Proc. ITC, pp. 221- 229, 1988.

[30] **J. Lee, J.H. Patel.** *Architectural level test generation for microprocessors.* IEEE Trans. CAD, pp. 1288-1300, Oct. 1994.

[31] Register-transfer level - Wikipedia, the free encyclopedia. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Register-transfer_level.

[32] **Dhiraj K. Pradhan, Ian G. Harris.** *Practical Design Verification.* Cambridge University Press, 2009.

[33] **Chu, Pong P.** *RTL Hardware Design.* John Wiley & Sons, Inc.,2006.

[34] **Jenihhin, Maksim.** *Simulation-Based Hardware Verification with High-Level Decision Diagrams.* Tallinn: Tallinn University of Technology Press, 2008.

[35] **V. Chayakul, D. D. Gajski, L. Ramachandran.** *High-Level Transformations for Minimizing Syntactic Variances.* DAC, pp. 413-418, 1993.

[36] **L. Zhang, I. Ghosh, M. Hsiao.** *Efficient Sequential ATPG for Functional RTL Circuits.* Int. Test Conf., pp.290-298, 2003.

[37] **H. Fujiwara, C. Y. Ooi, Y Shimizu.** *Enhancement of Test Environment Generation for Assignment Decision Diagrams.* 9th IEEE Workshop on RTL and High Level Testing, Nov. 27-28, 2008.

[38] **J. Raik, R. Ubar.** *Fast Test Pattern Generation for Sequential Circuits Using Decision Diagram Representations.* JETTA, Kluwer, Vol. 16, No. 3, pp. 213-226, June, 2000.

[39] **Guglielmo, G., et al.** *On the Combined Use of HLDD and EFSMs for Functional ATPG.* East-West Design and Test Symposium, IEEE Computer Society, pp. 503 - 508, 2007.

[40] **System, The ECLiPSe Constraint Programming.** http://eclipse-clp.org/. [Online]

[41] **Maxwell, P., Hartanto, I. ja Bentz, L.** *Comparing functional and structural tests.* Proceedings. International Test Conference ,3-5 Oct. 2000, pp. 400 – 407.

[42] **A. Giani, et al.** *Efficient Spectral Techniques for Sequential ATPG.* Proc. IEEE DATE Conf., March 2001, pp. 204-208.

[43] **F. Ferrandi, F. Fummi, D. Sciuto.** *Implicit Test Generation for Behavioral VHDL Models.* Int. Test Conf., pp. 587- 596, 1998.

[44] **I. Ghosh, M. Fujita.** *Automatic Test Pattern Generation for Functional RTL Circuits Using Assignment Decision Diagrams.* Proc. of ACM/IEEE DAC, pp. 43-48, 2000.

[45] **Jenihhin, Maksim.** *Simulation-Based Hardware Verification with High-Level Decision Diagrams.* Tallinn: Tallinn University of Technology Press, 2008.

[46] **T. Viilukas, J. Raik, M. Jenihhin, R. Ubar, A. Krivenko.** *Constraint-based test pattern generation at the register-transfer level.* 13th IEEE DDECS Symposium, 2010, pp. 352-357.

[47] **Lloyd, J. W.** *Foundations of Logic Programming.* Springer Verlag, 1993.

[48] **Colmerauer, A.** *Prolog II reference manual and theoretical model.* Technical report, Universit´e Aix-Mareseille, 1982.

[49] **Hentenryck, P. Van.** *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[50] **M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier.** *The constraint logic programming language CHIP.* In Proc. International Conference on Fifth Generation Computer Systems, Tokyo, Japan, 1988.

[51] **Krzysztof R. Apt, Mark Wallace.** *Constraint Logic Programming using ECLiPSe.* Cambridge University Press, 2007.

[52] **S.R.Makar, E.J.McCluskey.** *On the testing of multiplexers.* Proc. ITC, pp. 669-679, 1988.

[53] **benchmarks, HLSynth92.** Index of /benchmarks/HLSynth92. [Online] http://www.cbl.ncsu.edu:16080/benchmarks/HLSynth92/.

[54] **benchmarks, HLSynth95.** Index of /pub/hlsynth/HLSynth95. [Online] http://ftp.ics.uci.edu/pub/hlsynth/HLSynth95/.

[55] **E. Gramatova, M. Gulbins, M. Marzouki, A. Pataricza, R. Sheinauskas, R. Ubar.** *FUTEG Benchmarks.* Tech. Report FUTEG-1/1997.

[56] Turbo Tester. *TTU.* [Online] http://www.pld.ttu.ee/tt.

[57] **V. D. Agrawal and S. T. Chakradhar.** *Combinational ATPG theorems for identifying untestable faults in sequential circuits.* IEEE Trans Comput.-Aided Des., vol. 14, no. 9, pp. 1155–1160, Sep. 1995.

[58] **M. A. Iyer, D. E. Long, and M. Abramovici.** *Identifying sequential redundancies without search.* Proc. 33rd Annu. Conf. DAC, LasVegas, NV, Jun. 1996, pp. 457–462.

[59] **Q. Peng, M. Abramovici, and J. Savir,.** *MUST:Multiple stem analysis for identifying sequential untestable faults.* Proc. Int. Test Conf., 2000, pp. 839–846.

[60] **D. E. Long, M. A. Iyer, M. Abramovici.** *FILL and FUNI: Algorithms to identify illegal states and sequentially untestable faults.* ACM Trans. Des. Automat. Electron. Syst., vol. 5, no. 3, pp. 631–657, Jul. 2000.

[61] **H.-C. Liang, C. L. Lee, and E. J. Chen.** *Identifying untestable faults in sequential circuits.* IEEE Des. Test. Comput., vol. 12, no. 3, pp. 14–23, Sep. 1995.

[62] **J. Raik, H. Fujiwara, R. Ubar, A. Krivenko.** *Untestable fault identification in sequential circuits using model-checking.* ATS, pp. 667-672, 2008.

[63] **J. Raik, R. Ubar.** *Sequential Circuit Test Generation Using Decision Diagram Models.* Proceedings of the DATE Conference, pp. 736-740, 1999.

[64] **V. Vedula, J. Abraham.** *FACTOR: A Hierarchical Methodology for Functional Test Generation and Testability Analysis.* IEEE Computer Society Washington, pp. 730 - 734, 2002.

[65] **G.Jervan.** *High-Level and Hierarchical Test Sequence Generation.* IEEE HLDVT, Cannes, 2002.

[66] **Brayton, R. K., Hachtel, G. D., McMullen, C. T., Sangiovanni-Vincentelli, A. L.** *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[67] **De Moura, L., Bjørner, N.** *Z3: An Efficient SMT Solver.* TACAS, 2008, pp. 337-340.

[68] ITC benchmarks. [Online] http://www.cerc.utexas.edu/itc99-benchmarks/bench.html.

Curriculum Vitae

Personal Data

Name	Taavi Viilukas
Date of birth	16.01.1981
Place of birth	Estonia
Citizenship	Estonian

Contact Data

Address	Raja 15, Tallinn, 12618
Phone	+372 56156788
E-mail	taavi@viilukas.ee

Education

2006–…	Ph.D. studies in Information and Communication Technology, Tallinn University of Technology (TUT)
2004–2006	M.Sc. in Computer and Systems Engineering, TUT
1999–2004	Diploma in Computer and Systems Engineering, TUT
1996–1999	Secondary Education from Tallinna Laagna Gümnaasiun

Carrier

2008–…	CEO, Termnet Eesti OÜ
2004–2008	Systems Administrator, Aero Airlines AS
2001–2004	Systems Administrator, Estonian Academy of Security Sciences
2001–2004	Teacher, Estonian Academy of Security Sciences

Academic Degree

Master of Science in Engineering, Computer and Systems Engineering, TUT, “Development and Analysis of the Fault Propagation Method in a Hierarchical Test Generator”

Diploma, Computer and Systems Engineering, TUT, „Develop Fault Propagation in Hierarchical Test Generation“

Awards

2008–2009 Scholarship of Estonian Information Technology Foundation (EITSA)

Research topics

Digital testing, automated test pattern generators

Elulookirjeldus

Isikuandmed

Ees- ja perenimi	Taavi Viilukas
Sünniaeg	16.01.1981
Sünnikoht	Eesti
Kodakondsus	Eesti

Kontaktandmed

Aadress	Raja 15, Tallinn, 12618
Telefon	+372 56156788
E-post	taavi@viilukas.ee

Hariduskäik

2006–…	Doktorantuur, Info- ja kommunikatsioonitehnoloogia Tallinna Tehnikaülikool
2004–2006	Tehnikateaduste magister, Tallinna Tehnikaülikool
1999–2004	Diplom, Tallinna Tehnikaülikool
1996–1999	Keskharidus, Tallinna Laagna Gümnaasium, Tallinn

Teenistuskäik

2008–…	Tegevjuht, Termnet Eesti OÜ
2004–2008	Süsteemiadministraator, Aero Airlines AS
2001–2004	Süsteemiadministraator, Sisekaitseakadeemia
2003–2004	Õpetaja, Sisekaitseakadeemia

Kaitstud lõputööd

2006 – magistritöö „Rikete levitamise meetodi analüüs ja arendus hierarhilisel testigenereerimisel“, TTU. Juh. J. Raik

2004 – diplomitöö „Andmestruktuuride väljatöötamine ja realiseerimine rikete levitamiseks hierarhilisel testigenereerimisel“, TTU. Juh. J. Raik

Teaduspreemiad ja -tunnustused

EITSA „Tiigriülikooli“ stipendium, 2008

Teadustöö põhisuunad

Digitaalsüsteemide test, automaatsed testide genereerijad

Appendix A

PAPER I

Viilukas, Taavi; Raik, Jaan; Jenihhin, Maksim; Ubar, Raimund; Krivenko, Anna (2010). Constraint-based Test Pattern Generation at the Register-Transfer Level. In: Proceedings of the 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems : April 14–16, 2010 Vienna, Austria: IEEE, 2010, pp. 352 – 357.

Constraint-based Test Pattern Generation at the Register-Transfer Level

Taavi Viilukas, Jaan Raik, Maksim Jenihhin, Raimund Ubar, Anna Krivenko
Tallinn University of Technology
Raja 15, 12618 Tallinn, Estonia
E-mail: jaan@pld.ttu.ee

***Abstract*— The paper introduces a novel constraint-based automated test pattern generator for Register-Transfer Level (RTL) designs. The tool combines test path constraint activation with a constraint solver. First, a deterministic algorithm that extracts constraints for activating test paths at RTL is applied. Subsequently, a constraint solving package ECLiPSe is used for assembling the tests. Experiments on ITC99 and HLSynth92/95 benchmarks show that the proposed deterministic method offers short run times. In particular, it provides increased fault coverage for hard-to-test designs with respect to earlier, semi-formal, approaches.**

Keywords-Register-transfer level; automated test pattern generation; constraint satisfaction problems; decision diagrams

I. INTRODUCTION

At present, satisfactory methods for testing sequential circuits are missing and this has led the community to replace the hard test pattern generation task by theoretically much simpler approach relying on scan paths together with combinational Automated Test Pattern Generation (ATPG). However, the scan-path method has its shortcomings including increased area, delay and consumed power. It also causes targeting of non-functional failure modes, which results in over-testing and yield loss [1].

Several approaches to generating tests for structural faults in sequential cores have been proposed over the years. Despite of all the efforts the problem still lacks a breakthrough. At the gate-level, a number of deterministic test generation tools, both academic [2, 3] and commercial, have been implemented. None of these methods can efficiently handle sequential designs of even a couple of thousands of gates. With the further growth of the circuit size fault coverages tend to drop while run times increase rapidly.

Better performance has been obtained with simulation-based approaches. Here, genetic algorithm based methods have been widely used [4, 5, 6]. Relatively efficient results have been obtained by spectral methods [7]. However, the simulation-based methods are fast for smaller circuits only and become ineffective when the number of primary inputs and the sequential depth of the circuit increase. Moreover, these methods do not guarantee detection for hard-to-test random pattern resistant faults.

Many works on functional test generation have been published in the past [8, 9]. In this field, an efficient technique based on BDD manipulation of data domain partitions has been proposed [10]. However, the fundamental shortcoming of the approaches that rely on functional fault models is that they do not offer full structural level fault coverage.

Hierarchical and RTL test pattern generation has been proposed as a promising alternative to tackle complex sequential circuits. Here, top-down and bottom-up strategies are known. In the bottom-up approach [11], tests generated at the lower level will be later assembled at the higher abstraction level. Such algorithms ignore the incompleteness

problem: constraints imposed by other modules and/or the network structure may prevent test vectors from being assembled. In the top-down approach [12], where constraints are extracted at the higher level with the goal to be considered when deriving tests for modules at the lower level.

A number of works have been published on implementing assignment decision diagram models [13] combined with SAT methods to address register-transfer level test pattern generation [14, 15, 16]. All of these are bottom-up methods based on a multi-valued algebra for establishing transparent test paths. Therefore they suffer from the incompleteness issue described above.

In current paper, we propose a new algorithm for constraint-based ATPG on High-Level Decision Diagram (HLDD) models. [17] proposed a bottom-up approach based on an HLDD engine and a commercial SICStus constraint solver. As experiments show, the tool achieves lower fault coverage in comparison to a commercial gate-level ATPG. In [18], a top-down approach DECIDER was introduced, which relied on random constraint solving. The method was recently combined with Extended Finite State Machine (EFSM)-based engine LAERTE++ from the University of Verona, which resulted in a semi-formal setup [19].

Current paper introduces a deterministic algorithm that extracts constraints for activating test paths at RTL and subsequently applies a constraint solving package ECLiPSe [20] assembling the tests. Experiments on ITC99 and HLSynth92/95 benchmarks show that the proposed deterministic method offers short run times. In particular, it provides increased fault coverage for hard-to-test designs with respect to earlier, semi-formal, approaches listed above.

The paper is organized as follows. In Section 2 we introduce the concept of test generation constraints. Section 3 presents the high-level path activation algorithm. In Section 4, the algorithm and constraint extraction is explained on an example. Section 5 introduces the constraint solving setup. Finally, experimental results and conclusions are presented.

II. CONCEPT OF PATH ACTIVATION CONSTRAINTS

The test generation approach proposed in current paper contains two main phases. During the first phase, high-level test path activation, an untested module is selected and for this module propagation and justification is performed as explained in Section 3. In addition, constraints for the test path are extracted. The goal of the second phase is to satisfy the constraints by using a constraint solver and to compile the test patterns by assigning the values obtained by the constraint solver to the primary input signals (See Section 4).

The high-level test generation constraints considered in current paper are divided into three categories. These are path activation constraints, transformation constraints and propagation constraints. *Path activation constraints* correspond to the logic conditions in the control flow graph that have to be satisfied in order to perform propagation and value justification through the circuit. *Transformation constraints*, in turn, reflect the value changes along the paths from the inputs of the high-level Module Under Test (MUT) to the primary inputs of the whole circuit. These constraints are needed in order to derive the local test patterns for the module under test. *Propagation constraints* show how the value propagated from the output of the MUT to a primary output is depending on the values of the signals in the system. The main idea here is to guarantee that fault signals will not be masked when propagated.

All the above categories of constraints are represented by common data

structures and manipulated by common procedures for creation, update, modeling and simulation.

Let us explain the role of these constraints in test generation on an example test path activation for a circuit module shown in Figure 1. In the Figure there are two path activation constraints: *true* = $f_1(x_1,x_2)$ and *false* = $f_2(x_2,x_3)$. The first one is necessary to propagate the value from the output of the module to the primary output y_3 of the circuit. The latter is required for justification of the first input (D_1) of the module under test. Both these constraints are extracted from the conditional nodes traversed in the control flow graph of the circuit during high-level path activation. The Figure also presents two transformation constraints. These constraints are applied for computing the value of the corresponding module input depending on the values of primary inputs of the circuit. Finally, there is a propagation constraint, which states that the value propagated from the module to the primary output y_3 is dependent on the primary input x_6. Thus, in order to avoid fault masking the value of x_6 must be chosen such that the fault free and faulty values of D_{out} would differ. Note, that the subsets of the primary input variables included into the different types of constraints may overlap.

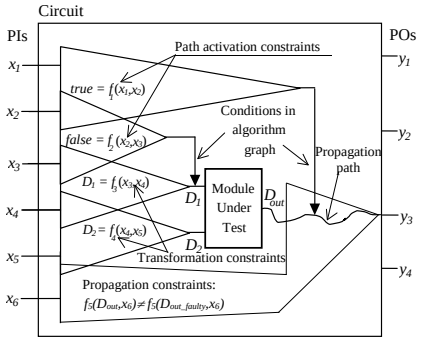


Figure 1. An example of test generation constraints

In the following, the data structure and update operations of high-level test generation constraints are defined.

Definition 1: A condition $C = g(x)$, where C is an integer, Boolean or symbolic value, and $g(x)$ is an expression on a subset of variables of the model representing the system under test, is refered to as constraint.

In current approach, symbolic values that can be used for C in a constraint $C = g(x)$ are D_i and D_{out} which correspond to the values of the i-th input and the output of the current Module Under Test (MUT), respectively (See Figure 1).

Definition 2: Constraint $C = g(x)$ is said to be *justified* if $x \subseteq x_I$, where x_I is the set of primary inputs of the system. Otherwise, $C = g(x)$ is an *unjustified* constraint.

If a constraint $C = g(x)$ is unjustified then all the variables in the set $x^U \subseteq x$ that are not input variables x_I are said to be *unjustified variables* of the constraint.

Definition 3: Let x^J be the set of justified variables and x^U be the set of unjustified variables of a constraint $C = g(x^J, x^U)$.

The process, where each variable x^U_i is substituted by expressions on model variables $x'_i \subseteq x$, is refered to as *updating the constraint* $C = g(x^J, x^U)$ and it creates a new constraint $C' = g'(x^J, x')$, where g' can be regarded as a superposition of functions on a set of variables in the system model representation. Section 4 presents an example of constraint update in test path activation.

Note, that justified constraints consist of operations on primary inputs x_I and constants x_C. Furthermore, the exponential size complexity of the constraints $g(x)$ is avoided by uniting multiple occurrences of the same variable (i.e. the literals) in the constraints at each time step into one single fanout variable. Because of this, the size requirements for the constraints are linear with respect to justification time-frames and they

represent a small subset of the expanded time-frame model of the circuit. Thus, the high-level test constraint extraction procedure is scalable.

III. DETERMINISTIC TEST PATH ACTIVATION

The high-level symbolic path activation, proposed in current paper is a complete algorithm, i.e. if transparent paths for fault effect propagation and value justification exist, they will be activated. The algorithm has been implemented as a systematic search and therefore an inconsistency in any stage causes a backtrack and a return to the last decision. However, due to the NP-complete nature of the problem, in some cases, the search must be terminated after a certain maximal number of solutions have been tried. For the sake of simplicity and speed, only three types of symbolic values are used during the path activation:

D - line with the fault effect,

X - line with unassigned value,

V - line with an assigned value.

In the following the propagation and justification principles of the proposed RT level ATPG are presented.

3.1 Fault effect propagation

The purpose of the propagation procedure is to activate a state sequence that propagates the fault effect from the output of the module under test to one of the primary outputs of the design. In current approach, propagation along single path is implemented. In order to keep track of the fault effect propagation a dedicated fault effect pointer is used. During propagation, high-level test path activation constraints are created. Fig. 2, presents the algorithm for fault effect propagation.

In the algorithm descriptions the term consistent FSM control vector is frequently used. By this term we mean a control vector (row) in the control part's FSM state table whose control signal values are consistent with value assignments made for control signals while propagating (activating) paths in the datapath.

```
/* Fault manifestation for module M */
Create constraints from all the module inputs input_i(M)
Set fault effect pointer to node output(M)
/* Fault effect propagation */
While fault pointer is not propagated to a primary output
Let a be the node pointed by fault effect pointer
Choose the most observable fanout branch of a
  Set control signals required to transport fault effect from the
    fanout branch to the next fanout stem or register node b
    /* always only one such path exists! */
  Set fault effect pointer to b
  If exists a consistent FSM control vector then
    Choose the most observable consistent control vector
    Create constraints of corresponding FSM input vector
    If b is a register then
      move to the next time-frame
    Endif
  Else
    Backtrack
  Endif
Endwhile
```

Figure 2. Fault effect propagation algorithm.

3.2 Constraint justification

Subsequent to propagation, constraint justification starts. Justification moves backward in time, starting from the clock-cycle, where propagation ended. During this process existing constraints are updated and additional path activation constraints are created. Finally, constraints solving procedure is applied to the extracted constraints and module under test is fault simulated by constraint-driven, local test data.

Nodes of the circuit, which correspond to primary inputs or constants are called justified nodes. All other nodes are said to be unjustified. Constraints containing unjustified nodes are refered to as unjustified constraints.

Updating the test generation constraints is defined in Section 2 and shown in more detail on an example presented in Section 4. Basically, updating a constraint can be regarded as superposition of the unjustified nodes of the constraint by new datapath nodes determined by paths activated in the datapath by current control vector.

At each justification step, current justification objective is chosen. In the proposed algorithm implementation the justification objective is to justify the first unjustified node from the first unjustified constraint. The algorithm for constraint justification is presented in Fig 3.

```
/* Constraint justification */
While exist unjustified constraints
If current time-frame is earlier than manifestation then
  Let current objective be to justify node b
  Choose the most controllable fanout, F.U. or register node a,
          which directly precedes b
  Set control signals activating path from a to b
  /* always only one such path exists! */
  If exists a consistent FSM control vector then
     Choose the most controllable consistent control vector
     Create constraints of corresponding FSM input vector
     If a is a register then
       move to the previous time-frame
     Endif
  Else
    Backtrack
  Endif
Else
  Move to the previous time-frame
Endif
Update all active constraints
Endwhile
/* Solve constraints (See Section 5!) */
```

Figure 3. Constraint justification algorithm.

IV. CONSTRAINT EXTRACTION EXAMPLE

In the following, the test path activation algorithm and constraint extraction is explained basing on the example of the Greatest Common Divisor (GCD). Consider the GCD algorithm described at behavioral level in a pseudo hardware description language:

```
A := IN1;
B := IN2;
while (A ≠ B)
  if (A < B) then
    B := B – A;
  else
    A := A – B;
  end if;
end while;
OUT := A;
```

Let us assume that subsequent to applying high-level synthesis to the algorithm description we obtain the RTL architecture presented in Figure 4. This architecture consists of a datapath of 3 Functional Units (FU), 2 registers and 4 multiplexers and a control part Finite State Machine (FSM) of four states. The datapath architecture is depicted in Figure 4a and the control part is given as a state table in Figure 4b, respectively.

We further explain the test generation algorithm described in Section 3 by the example of generating test paths for the module *SUBTR*.

Fault manifestation. Set all the variables to 'don't care' values. Create transformation constraints D_0=mux3, D_1=mux4. Set the fault effect pointer to variable SUBTR, i.e. y_D := SUBTR.

Fault effect propagation. Choose a datapath register that reads from the FU SUBTR. There are two possible choices: reg_A and reg_B, respectively. Let us select the first choice. Subsequently, we activate the path from SUBTR to reg_A, which results in the following variable assignments: A_enable := 1, mux_12 := 1. Next, we have to choose a consistent FSM control vector. The only vector consistent with previous variable assignments is the one corresponding to row 7 in the FSM state table (labeled by vector 0, X, X, S3, S0, 1, 0, 1, 0). Based on this vector we obtain the following assignments: reset:=0, B_enable := 0, mux_34 := 0, state := S3 (in current clock cycle), state := S0 (in the next clock cycle). We move to the next clock cycle and set the fault effect pointer y_D to reg_A (i.e. OUT).

We detect that the fault effect pointer points to a variable corresponding to a primary output and thus have successfully completed the fault propagation process.

Constraints justification. As there were no path activation constraints created during manifestation and propagation stages, we move backwards in terms of clock-cycles until the clock-cycle of manifestation phase is reached. We select

the justification objective from the unjustified variables of the transformation constraints (D_0=mux3, D_1=mux4). Let current objective be to justify variable mux3. Due to the fact that we have already assigned mux_34 := 0 at current clock-cycle during the propagation process, then we have no choice but backtracing mux3 to reg_A. We update the constraints, obtaining D_0= reg_A, D_1= reg_B and move to the preceding clock cycle.

Without focusing on further details, we continue executing the constraint justification algorithm until the path presented in Figure 9 is activated as one of possible high-level path solutions. In the Figure we have denoted the manifestation clock cycle by t, the i-th cycle following t is denoted by t+i and i-th cycle preceding t is denoted by t-i, respectively. Below the clock-cycle information, the activated state sequence is provided. Then we present graphically the processes of fault propagation and extraction of transformation constraints. Decisions in the high-level path activation are marked by stars (*) in the Figure. Extraction of path activation constraints is depicted below the striped line. Here, t corresponds to Boolean value 'true' and f corresponds to 'false'. As shown in Figure 5, we have to apply the constraint satisfaction process to the following set of constraints: in1 < in2 is false, in1 ≠ in2 is true.

Subsequent to testing the node with the first path, backtrack occurs and the high-level path activation algorithm tries to find alternative path solutions.

V. SOLVING THE TEST PATH CONSTRAINTS

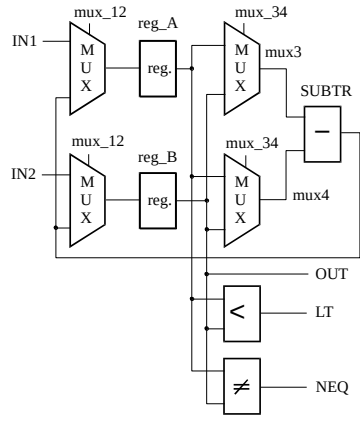
In the previous top-down test pattern generation algorithms by the authors [18, 19], random constraint solving was applied. In this paper we have selected the open source ECLiPSe constraint solver (ECLiPSe5.10_41) to solve the test path constraints. ECLiPSe supports most of the common techniques used in solving constraint problems. It includes constraint programming, mathematical programming, local search and various combinations of the above. We have embedded the solver into the C++ code of the ATPG and use the string-based approach.

An example of a constraint string is given in the following:

```
lib(ic),L is (0),R is(8^2)-1,
X_41::L..R, (7 >> 1 #= X_41),
indomain(X_41,random)
```

This constraint string is divided into 5 groups. First you have to define the library. In our case it is the finite domains library *ic*. In the second group you will define domain boundaries, then variables and domain is defined. The constraints and finally search criteria are given. (Search criteria are not mandatory).

As experiments presented in the following Section show the deterministic constraint solving has definite advantages over the pseudo-random method.



a)

RESET	LT	NEQ	pres. state	next state	A_enable	B_enable	mux_12	mux_34
1	X	X	X	S_0	1	1	0	X
0	X	1	S_0	S_1	0	0	X	X
0	X	0	S_0	S_0	0	0	X	X
0	1	X	S_1	S_2	0	0	X	X
0	0	X	S_1	S_3	0	0	X	X
0	X	X	S_2	S_0	0	1	1	1
0	X	X	S_3	S_0	1	0	1	0

b)

Figure 4. RT-level architecture of the GCD circuit.

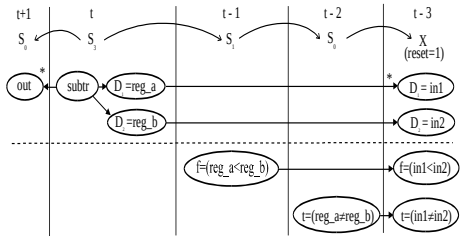


Figure 5. RT-level architecture of the GCD circuit.

VI. EXPERIMENTAL RESULTS

In order to evaluate the impact of the deterministic constraint solving experiments on ITC99 and HLSynth92/95 benchmarks were carried out. By this moment we have included the following three circuits into the analysis: b00, 604 and gcd because these circuits contain "equal to" comparison operators which are hard to test by pseudorandom constraint solving.

Table 1 shows the comparison of the semi-formal approach DECIDER presented in [18] and the proposed top-down tool. Comparison has been obtained by fault simulating the test sets generated by both generators by a stuck-at fault simulator for sequential circuits. The row '# faults' of the Table shows the number of stuck-at faults in the circuit. The row '# tested' presents the number of tested faults by [18] and the proposed approach. The row 'cover., %' lists the achieved stuck-at fault coverages. 'time, s' stands for the ATPG run times in seconds. Finally, the number of generated test vectors is reported in the row '# vect.'

It can be seen that the fault coverage improvement obtained by the deterministic constraint solving setup ranges from 3 to 34 % for the tested examples. Note, that while the fault coverages for the circuits are low, this is a usual case for the sequential ATPG because of the large number of untestable faults.

TABLE I. COMPARISON OF SEMI-FORMAL [18] AND THE PROPOSED DETERMINISTIC ATPG METHODS

	b00		b04		gcd	
	[18]	current	[18]	current	[18]	current
#faults	1328	1328	1488	1488	1658	1658
#tested	251	714	899	943	1443	1519
cover, %	18.90	53.77	60.42	63.37	87.03	91.62
time, s	0.0053	0.0044	0.002	0.011	2.72	0.02
#vect.	534	874	574	572	4471	4756

VII. CONCLUSIONS AND FUTURE WORK

The paper introduces a novel constraint-based automated test pattern generator for Register-Transfer Level (RTL) designs. The tool combines test path constraint activation with a constraint solver. First, a deterministic algorithm that extracts constraints for activating test paths at RTL is applied.

Subsequently, a constraint solving package ECLiPSe is used for assembling the tests. Experiments on ITC99 and HLSynth92/95 benchmarks show that the proposed deterministic method offers very short run times. In particular, it provides increased fault coverage which ranges from 3 to 34 % for the tested examples with respect to earlier, semi-formal, approaches.

Note, that while the fault coverages for the circuits are low, this is a usual case for the sequential ATPG because of the large number of untestable faults. As a future work we plan to integrate untestable fault analysis for sequential circuits (e.g. [21]) into the constraint-based ATPG to improve fault efficiency estimation.

ACKNOWLEDGEMENTS

The work has been supported by European Commission Framework Program 7 projects FP7-2009-IST-4-248613 DIAMOND and FP7-REGPOT-2008-1 CREDES, by European Union through the European Regional Development Fund, and by Estonian Science Foundation through grants 8478, 7068 and 7483.

REFERENCES

[1] Maxwell, P. Et al., Comparing functional and structural tests, *ITC 2000*. 3-5 Oct. 2000 Page(s):400 – 407.

[2] H.-K.T. Ma, S. Devadas, A.R. Newton, A. Sangiovanni-Vincentelli, “Test generation for sequential circuits”, *IEEE Trans. on CAD*, 7(10), pp. 1081-1093, Oct. 1988.

[3] T. M. Niermann, J. H. Patel, "HITEC: A test generation package for sequential circuits", *Proc. European Conf. Design Automation (EDAC)*, pp.214-218, 1991.

[4] E. M. Rudnick, et al. "Sequential circuit test generation in a genetic algorithm framework", *DAC*, 1994.

[5] F. Corno, P. Prinetto, et al., "GATTO: A genetic algorithm for automatic test pattern generation for large synchronous sequential circuits", *IEEE Trans. CAD*, pp.991-1000, Aug. 1996.

[6] M. S. Hiao, E. M. Rudnick, J. H. Patel, "Sequential circuit test generation using dynamic state traversal", *Proc. European Design and Test Conf.*, pp. 22-28, 1997.

[7] A. Giani, et al., “Efficient Spectral Techniques for Sequential ATPG,” *Proc. IEEE DATE Conf.*, March 2001, pp. 204-208.

[8] D. Brahme, J. A. Abraham, "Functional Testing of Micro-processors", *IEEE Trans. Comput.*, vol. C-33, 1984.

[9] A. Gupta, J. R. Armstrong, "Functional fault modeling", *30th ACM/IEEE DAC*, pp. 720-726, 1985.

[10] F. Ferrandi, F. Fummi, D. Sciuto, “Implicit Test Generation for Behavioral VHDL Models,” *Int. Test Conf.*, pp. 587-596, 1998.

[11] B. T. Murray, J. P. Hayes, "Hierarchical test generation using precomputed tests for modules", *Proc. ITC*, pp.221-229, 1988.

[12] J. Lee, J.H. Patel, "Architectural level test generation for microprocessors", *IEEE Trans. CAD*, pp.1288-1300, Oct. 1994.

[13] V. Chayakul, D. D. Gajski, L. Ramachandran, “High-Level Transformations for Minimizing Syntactic Variances”, *DAC*, pp. 413-418, June 1993.

[14] I. Ghosh, M. Fujita, “Automatic Test Pattern Generation for Functional RTL Circuits Using Assignment Decision Diagrams”, *DAC*, pp. 43-48, 2000.

[15] L. Zhang, I. Ghosh, M. Hsiao, “Efficient Sequential ATPG for Functional RTL Circuits”, *Int. Test Conf.*, pp.290-298, 2003.

[16] H. Fujiwara, C. Y. Ooi, Y. Shimizu, "Enhancement of Test Environment Generation for Assignment Decision Diagrams", *WRTLT*, 2008.

[17] G. Jervan et al., "High-Level and Hierarchical Test Sequence Generation", IEEE HLDVT, Cannes, 2002.

[18] J. Raik, R. Ubar, "Fast test pattern generation for sequential circuits using decision diagram representations", *JETTA*, Kluwer, 16(3), 2000.

[19] G. Di Guglielmo et al., "On the Combined Use of HLDDs and EFSMs for Functional ATPG", *East-West Design & Test Symposium*, Yerevan, 2007.

[20] The ECLiPSe Constraint Programming System http://eclipse-clp.org/

[21] Jaan Raik, Hideo Fujiwara, Raimund Ubar, Anna Krivenko. Untestable Fault Identification in Sequential Circuits Using Model-Checking. *The 17th Asian Test Symposium (ATS 2008)*, IEEE, pp. 667-672, November 24-27, 2008, Sapporo, Japan.

PAPER II

Raik, Jaan; Ubar, Raimund; Viilukas, Taavi (2006). High-Level Decision Diagram based Fault Models for Targeting FSMs. In: Proceedings of the 9th IEEE Euromicro Conference on Digital Systems Design : DSD2006, Cavtat, Aug. 31 – Sept. 2, 2006. IEEE Computer Society Press, 2006, pp. 353 – 358.

High-Level Decision Diagram based Fault Models for Targeting FSMs

Tallinn University of Technology, Department of Computer Engineering
jaan@pld.ttu.ee

***Abstract*— Recently, a number of works have been published on implementing assignment decision diagram models combined with SAT methods to address register-transfer level test pattern generation. Those methods have proven efficient. However, all of them target modules inside the datapath of the circuit. In this paper, we show by experiments that the fault coverage achieved by full datapath tests is often lower than what can be achieved if faults in the control part FSM were additionally considered. We also propose a new type of fault model for targeting faults in FSMs embedded to RTL descriptions. In addition, we present an alternative for traditional assignment decision diagrams, which provides for a more general representation of RTL circuits. We show that our model, called high-level decision diagrams, allows efficient high-level test path activation. According to experiments the proposed approach outperforms state-ofthe- art test pattern generation tools..**

I. INTRODUCTION

Several approaches to generating tests for structural faults in sequential circuits have been proposed over the years. However, the problem still lacks a breakthrough. At the gate-level, a number of deterministic test generation tools, both academic [1, 2] and commercial, have been implemented. None of these methods can efficiently handle sequential designs of even a couple of thousands of gates. With the further growth of the circuit size fault coverages tend to drop while run times increase rapidly.

Better performance has been obtained with simulationbased approaches. Here, genetic algorithm based methods have been widely used [3, 4, 5]. Recently, efficient results have been obtained by spectral methods [6]. The approaches belonging to this class are fast for smaller circuits only but become ineffective when number of primary inputs and the sequential depth of the circuit increase.

Many works on functional test generation have been published in the past [7, 8]. In this field, an efficient technique based on BDD manipulation of data domain partitions has been proposed [9]. However, the main principal shortcoming of the approaches that rely on functional fault models only is that they cannot guarantee satisfactory structural level fault coverage.

Hierarchical and RTL test pattern generation has been a promising alternative to tackle complex sequential circuits. Here, top-down and bottom-up strategies are known. In the bottom-up approach [10], tests generated at the lower level will be later assembled at the higher abstraction level. Such algorithms ignore the incompleteness problem: constraints imposed by other modules and/or the network structure may prevent test vectors from being assembled. In the top-down approach [11], where constraints are extracted at the higher level with the goal to be considered when deriving tests for modules at the lower level.

Recently, a number of works have been published on implementing assignment decision diagram models [12] combined with SAT methods to address register-transfer level test pattern generation [13, 14]. The authors of this paper have been relying on a different kind of representation, called high-level decision diagrams [15, 16], where, both, control unit and datapath are handled in a

uniform manner. An efficient RTL path activation has been previously proposed in [17] and complemented with precision fault models for multiplexers in [18].

The common shortcoming for all the former decision diagram based approaches is that they are targeting modules in the datapath of the circuit. Our previous research has shown that while deterministic test pattern generation algorithms are in general less powerful for larger circuits, they are still capable of testing a number of faults from the FSM part that the RTL and hierarchical methods are unable to cover.

In this paper, we propose a new type of fault model based on high-level decision diagrams dedicated to faults in FSMs embedded into RTL descriptions. The paper is organized as follows. In Section 2 we introduce the circuit model of high-level decision diagrams. Section 3 gives a short overview of the high-level path activation process. Section 4 defines the new fault models used in current approach. Finally, experimental results and conclusions are presented..

II. HIGH-LEVEL DECISION DIAGRAMS

This Section defines the concept of high-level decision diagrams (HLDD) and compares the model to commonly used assignment decision diagram (ADD) approach. We will point out the main differences between the two models and show how HLDDs can be used in efficient RTL test path activation. The following Sections will explain in detail the HLDD based path activation method and fault models for embedded FSMs.

2.1 Basic definitions

Decision diagrams are graph representation of discrete functions. A discrete function y=f(x), where y=(y1,…yn) and x=(x1,…xm) are vectors is defined on X=X1x…xXm with values y ∈ Y = Y1x…xYn, and both, the domain X and the range Y are finite sets of values. The values of variables may be Boolean, Boolean vectors, integers.

For representing the functions y=f(x) we use decision diagrams Gy. A Decision Diagram (DD) is a directed noncyclic labeled graph that can be defined as a quadruple G=(M,E,X,D), where M is a finite set of vertices (referred to as *nodes*), E is a finite set of *edges*, X is a function which defines the *variables labeling the nodes* and the variable domains, and D is a function on E.

The function X(mi) returns a pair (xi,Xi), where xi is the variable letter which is labeling node mi and Xi is the domain of xi. Each node of a DD is labeled by a variable. A single variable can label multiple nodes. In special cases of DDs, nodes are labelled by constants, arithmetic expressions or vectors.

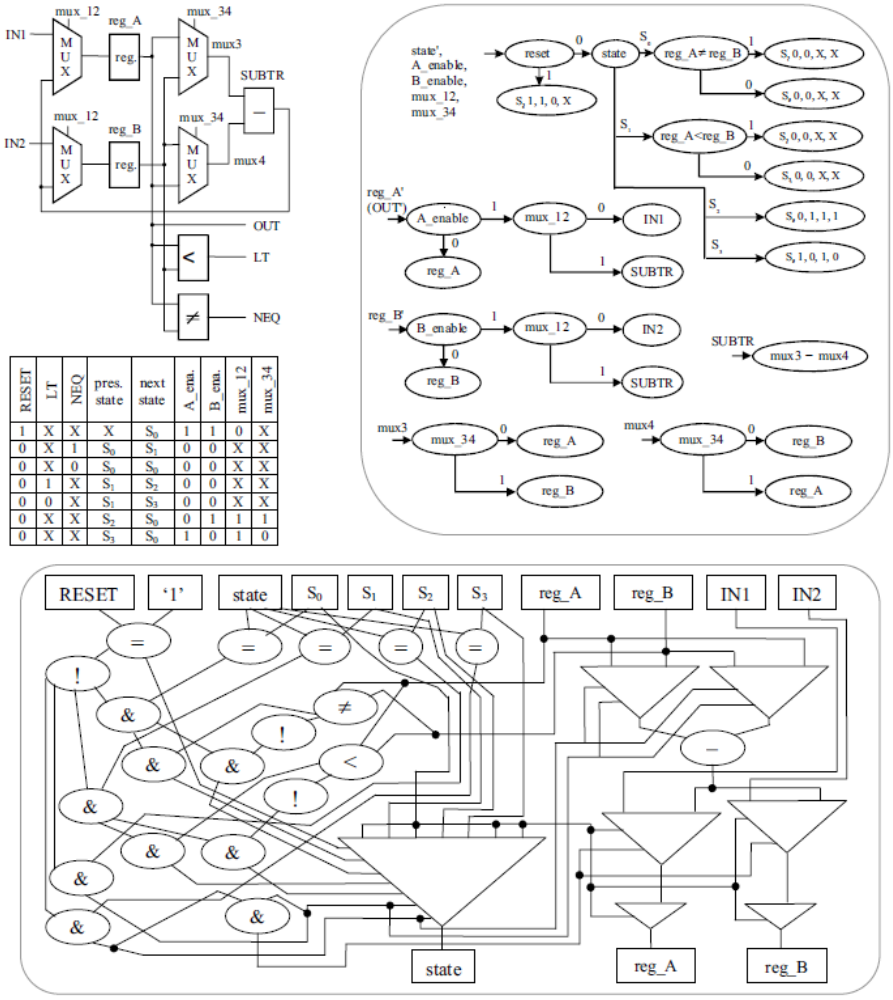
An edge e∈E of a DD is an ordered pair e=(m1,m2) ∈E2, where E2 is the set of all the possible ordered pairs in set E. Graphical interpretation of e is an edge leading from node m1 to node m2. It is said that m1 is a *predecessor node* of m2, and m2 is a *successor node* of the node v1, respectively.

D is a function on E representing the activating conditions of the edges for the simulating procedures. The value of D(e) is a subset of Xi, where e=(mi,mj) and X(mi)=(xi,Xi). It is required that Pmi={D(e) | e=(mi,mj) ∈E } is a partition of the set Xi. In other words, the subsets of the set Xi labeled on the edges starting from a node mi must not overlap and their union must be equal to Xi.

DD has only one starting node (*root node*) m0, for which there are no preceding nodes. The nodes, for which successor nodes are missing are referred to as *terminal nodes* MT. Simulation on decision diagrams takes place as follows. Consider a situation, where all the node variables are fixed to some value. According to these values, for each non-terminal node a certain output edge will be chosen which enters into its corresponding successor node. Let us call

such connections *activated edges* under the given values. Succeeding each other, activated edges form in turn *activated paths*. For each combination of values of all the node variables there exists always a corresponding activated path from the root node to some terminal node. We refer to this path as the *main activated path*. The simulated value of variable represented by the DD will be the value of the variable labeling the terminal node of the main activated path.

When representing systems by decision diagram models, in general case, a network of DDs rather than a single DD is required. During the simulation in DD systems, the values of some variables labeling the nodes of a DD are calculated by other DDs of the system.



RESET	LT	NEQ	pres. state	next state	A_ena.	B_ena.	mux_12	mux_34
1	X	X	X	S_0	1	1	0	X
0	X	1	S_0	S_1	0	0	X	X
0	X	0	S_0	S_0	0	0	X	X
0	1	X	S_1	S_2	0	0	X	X
0	0	X	S_1	S_3	0	0	X	X
0	X	X	S_2	S_0	0	1	1	1
0	X	X	S_3	S_0	1	0	1	0

Figure 1. RTL circuit (top left), its HLDD (top right) and ADD (bottom).

2.2 HLDD representation for RTL circuits

Datapath can be viewed as a network consisting of modules or blocks. These include registers, multiplexers and Functional Units (FUs). Inputs for the datapath are the primary inputs xI and control signals xC (e.g. multiplexer addresses and register enable signals). Outputs are the primary outputs xO as well as status bits xN (primarily from comparison operators) leading to the control part FSM.

In HLDD models representing the datapath, the nonterminal nodes correspond to control signals (labeled by variables xC). The terminal nodes represent operations (functional units). Register transfers and constant assignments are treated as special cases of operations.

At the RT-level, datapath is represented by a system of DDs. In this paper we use partitioning, where for each primary output, fanout signal and register a DD corresponds. In addition, multiplexers that are connected to FU inputsare also represented by a separate DD.

Similar to the datapath, the control part of an RTL design can be represented by a HLDD. This DD calculates the values for a vector consisting of the state variable and control signals. In the DD, the non-terminal nodes correspond to current state (labeled by variable xS) and status bits originating from the datapath (variables xN). Terminal nodes hold vectors with the values of next state and control signals xC.

2.3 Differences between HLDDs and ADDs

Figure 1 presents the RTL description of a Greatest Common Division benchmark and its corresponding HLDD and Assignment Decision Diagram (ADD) representations. Apart from the fact that HLDD description contains less nodes, there are the following fundamental differences:

1.ADDs structure closely matches the RTL design. Edges of ADD correpond to connecting nets in datapath. ADD for FSM is equivalent to its gate-level implementation. In contrast, HLDDs do not strictly follow the circuit structure. Here, a synthesis to extract data and control relationships from the circuit functionality has been carried out.

2.ADD model includes four types of nodes (read, write, operator, assignment decision). In HLDD the nodes are treated uniformly and can be divided into nonterminal nodes (control) and terminal nodes (data).

3.While ADDs do not support decision-making implicitly in the model, in HLDDs, the selection of a node activates a path through the diagram, which derives the needed value assignments for variables. Note, that the edges in ADD model have no labels. This is the most significant difference between the two models.

III. RTL TEST PATH ACTIVATION USING HLDDS

This Section presents the top-down hierarchical test generation framework used as a basis in the paper. The concept of hierarchical test generation constraints is explained and different stages of the test pattern generation algorithm on HLDD models are explained.

3.1 Constraint-based hierarchical test generation

The hierarchical test generation constraints considered in current paper can be divided into two categories: path

activation constraints and transformation constraints. Path activation constraints correspond to the logic conditions in the control flow graph that have to be satisfied in order to perform propagation and value justification through the circuit.

Transformation constraints, in turn, reflect the value changes along the paths from the inputs of the high-level module under test to the primary inputs of the whole circuit. These constraints are needed in order to derive the local test patterns for the module under test. Both types of constraints can be represented by common data structures and manipulated by common procedures for creation, update, modeling and simulation.

Figure 2 explains the role of these constraints in test generation for a circuit module. In the Figure there are two path activation constraints: true = f(x1,x2) and false = g(x2,x3). The first one is necessary to propagate the value from the output of the module to the primary output y3 of the circuit. The latter is required for justification of the first input (D1) of the module under test. Both these constraints are extracted from the conditional nodes traversed in the control flow graph of the circuit during high-level path activation.

In addition, the Figure presents two transformation constraints. These constraints represent the function for computing what will be the value of the corresponding module input depending on the values of primary inputs of the circuit.

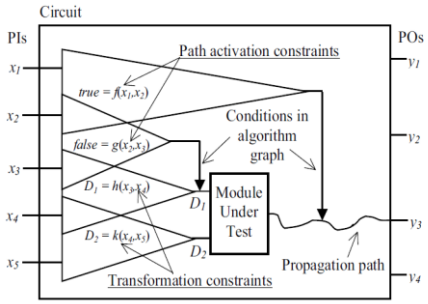


Fig. 2. Hierarchical test generation constraints

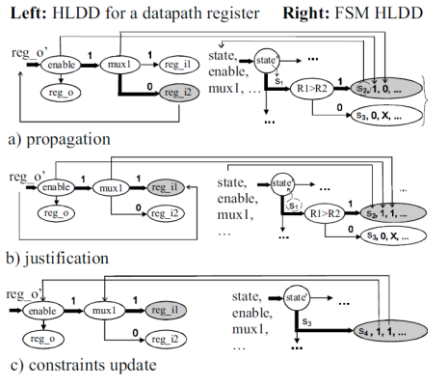


Fig. 3. RTL path activation on HLDDs

3.2 Test path activation and constraint extraction

Test generation for a circuit Module Under Test (MUT) starts with the fault manifestation procedure. During the fault manifestation phase, fault effect is set to the MUT output. In addition, symbolic transformation constraints are created in form of Di = xi, i = 1,...,n, where n is the number of inputs of current MUT. Di are symbolic values representing local test patterns applied to corresponding inputs of MUT and xi are the lines in the model representing these inputs.

Fault manifestation is followed by fault effect propagation. During the propagation stage we move forward in time (clock-cycles), fault effect is propagated towards primary outputs and path activation constraints are created whenever conditions in the control flow graph are traversed. Propagation is completed when we have obtained a state sequence transfering the fault effect to a primary output of the circuit.

Subsequent to propagation, constraint justification begins. Justification moves backwards in time, starting from the clock-cycle, where propagation ended. During this process existing constraints are updated and additional path activation constraints are created. The process will be terminated when all the variables in all the constraints are primary inputs. Finally, constraints solving procedure is applied to

the extracted constraints and MUT is fault simulated by constraint-driven, randomly generated local test data.

Fig. 3 explains the steps of propagation, justification and constraint update on HLDD models. Fig. 3a shows the implications we can perform on the model when propagating the fault effect from a datapath register reg_i2 to register reg_o. As a first step, we have to activate a path from the root node to the node labeled by reg_i2. From this path we obtain assignments enable := 1 and mux1 := 0. Then, we select the terminal node from the FSM HLDD, which is consistent with the above assignments. This node is marked with grey. We activate the path from the FSM root node to the chosen terminal node. This results to assignment: next state := s2. Since the activated path traverses a node with condition R1<R2 it has to be added to the set of path activation constraints. Similarly, assignments are derived during justification and constraint update steps.

IV. HLDD FAULT MODELS FOR FSM

As it was mentioned above the previous RTL ATPG test generation approaches focus on the datapath part of a circuit. The functional fault model for multiplexers proposed in [16] covers also the output logic of the FSM. However, fault models for targeting next state logic in a hierarchical test pattern generation context are missing. We propose two types of fault models that have to be addressed in HLDDs in order to set up tests for the next state logic. The cases are listed in the following: 1. Distinguishing the registers addressed by a pair of control words (i.e. FSM HLDD terminal nodes). 2. Distinguishing the mux inputs controlled by a pair of control words.

In the case, a wrong datapath register is addressed because of a fault in the next state logic, this will be revealed by distinguishing the values in the registers and performing propagation and justification through the circuit. Or, alternatively, if the two control words address the same set of registers, there can be a difference in some multiplexer address signals. In the latter case, the registers connected to inputs of the mux have to be distinguished.

There is another issue raising from conditional operations, whose outputs are connected to the FSM as status bits. There exists no path from these modules through the datapath to primary outputs. However, the internal structure of the comparison operation should be exhaustively covered by the fault model. In current approach, a functional fault model is selected to activate the fault effect at the status bit and a hierarchical fault model is implemented to test the module at the low-level. Figure 4 shows an example of the test for comparison operators.

Let current MUT be the conditional operation, whose output is connected to status bit ‘status’. Let us consider the test setup for the module with the output value being one (i.e. status = 1). (Note, that separate tests for the MUT have to be set up for output values 0 and 1, respectively). As a first step, we try to identify possible setup states for the fault manifestation stage. In order to do that we select a nonterminal node in the FSM DD labeled by variable ‘signal’. We activate the path to this node obtaining the following value assignments: reset := 0, state := Sj.

The next step is to select the datapath register for fault effect activation. The register is determined by comparing the vectors at the two terminal nodes of FSM DD, which lie at the end of the faulty and fault-free paths, respectively. Registers, which are selected by the faultfree terminal node and deselected by the faulty one are considered to be suitable for fault activation. (In our example, register Ri matched the requirements and was chosen). Subsequently, propagation, justification and constraint extraction

procedures are performed as explained in the previous section.

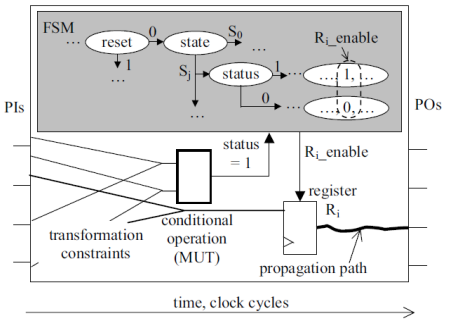


Fig. 4. FSM test covering conditional operations

V. EXPERIMENTAL RESULTS

In Table 1, comparison of test generation results of four ATPG tools on five hierarchical designs are presented. The benchmarks are mainly from HLSynth92 and HLSynth95 families. All the experiments were run on a 366 MHz SUN UltraSPARC 60 server with 512 MB RAM under SOLARIS 2.8 operating system. Five sequential ATPG tools were compared in the experiments. These were a gate-level deterministic ATPG HITEC [2] and a genetic algorithm based GATEST [3], hierarchical ATPG covering only datapath FUs, hierarcical test pattern generation for FUs and multiplexers, and finally, the approach proposed in current paper.

The results show that the proposed method is very efficient for testing sequential designs. It achieves in average 2.5 % higher fault coverage than the genetic tool GATEST on the given benchmark set. For the sake of comparison, an experiment with complete tests for datapath functional units (FU) was performed. This resulted in a poor overall coverage of the design, thus, showing the need for FSM and multiplexer testing in RTL ATPG.

Table 1. Comparison of sequential circuit test generation tools

circuit	faults	HITEC [2]		GATEST [3]		FU only	FU+MUX [16]		FU+MUX+FSM	
		F.C., %	time, s	F.C., %	time, s	F.C., %	F.C., %	time, s	F.C., %	time, s
gcd16	1754	59.11	365	86.13	190.7	62.55	85.71	497.4	**90.95**	677.4
mult8x8	2036	65.9	1243	69.2	821.6	69.4	74.2	76.9	**74.7**	93.7
ellipf	5388	87.9	2090	94.7	6229	86.7	**95.04**	1258.9	**95.04**	1258.9
risc	6434	52.8	49,020	96.0	2459	83.9	**96.5**	150.5	**96.5**	150.5
diffeq	10,008	96.2	13,320	96.40	3000	96.44	96.52	441.7	**97.09**	453.7
average F.C.:		72.4		88.4		79.8	89.6		**90.9**	

VI. CONCLUSIONS

In the paper we defined high-level decision diagrams (HLDD) as an efficient model for RTL test pattern generation. For the first time, this model is compared to currently popular assignment decision diagrams. We point out the main benefits of the former and present a small example of main test path activation tasks.

The main novel contribution of the paper lies in introduction of new FSM fault models on HLDD representations. We show by experiments that these fault models allow us to reach higher fault coverages when compared to other approaches. In fact, our experimental results prove quite clearly that RTL test methods targeting datapath functional units only can not guarantee high fault coverages for the overall design.

ACKNOWLEDGEMENTS

The research has been supported by ETF grants G5637, G5910 and G5649 and by Enterprise Estonia funded ELIKO technology development center.

REFERENCES

[1] H.-K.T Ma, S. Devadas, A.R. Newton, A. Sangiovanni- Vincentelli, “Test generation for sequential circuits”, IEEE Trans. on CAD,Vol. 7, No. 10 pp. 1081-1093, Oct. 1988.

[2] T. M. Niermann, J. H. Patel, "HITEC: A test generation package for sequential circuits", Proc. European Conf. Design Automation (EDAC), pp.214-218, 1991.

[3] E. M. Rudnick, et al. "Sequential circuit test generation in a genetic algorithm framework", Proc. DAC, pp. 698-704, 1994.

[4] F. Corno, P. Prinetto, et al., "GATTO: A genetic algorithm for automatic test pattern generation

for large synchronous sequential circuits", IEEE Trans. CAD, pp.991-1000, Aug. 1996.

[5] M. S. Hiao, E. M. Rudnick, J. H. Patel, "Sequential circuit test generation using dynamic state traversal", Proc. European Design and Test Conf., pp. 22-28, 1997.

[6] A. Giani, et al., "Efficient Spectral Techniques for Sequential ATPG," Proc. IEEE DATE Conf., March 2001, pp. 204-208.

[7] D. Brahme, J. A. Abraham, "Functional Testing of Microprocessors", IEEE Trans. Comput., vol. C-33, 1984.

[8] A. Gupta, J. R. Armstrong, "Functional fault modeling", 30th ACM/IEEE DAC, pp. 720-726, 1985.

[9] F. Ferrandi, F. Fummi, D. Sciuto, "Implicit Test Generation for Behavioral VHDL Models," Int. Test Conf., pp. 587-596, 1998.

[10] B. T. Murray, J. P. Hayes, "Hierarchical test generation using precomputed tests for modules", Proc. ITC, pp.221-229, 1988.

[11] J. Lee, J.H. Patel, "Architectural level test generation for microprocessors", IEEE Trans. CAD, pp.1288-1300, Oct. 1994.

[12] V. Chayakul, D. D. Gajski, L. Ramachandran, "High-Level Transformations for Minimizing Syntactic Variances", Proc. Of ACM/IEEE DAC, pp. 413-418, June 1993.

[13] I. Ghosh, M. Fujita, "Automatic Test Pattern Generation for Functional RTL Circuits Using Assignment Decision Diagrams", Proc. of ACM/IEEE DAC, pp. 43-48, 2000.

[14] L. Zhang, I. Ghosh, M. Hsiao, "Efficient Sequential ATPG for Functional RTL Circuits", Int. Test Conf., pp.290-298, 2003.

[15] J. Raik, R. Ubar, "Fast Test Pattern Generation for Sequential Circuits Using Decision Diagram Representations.", JETTA, Kluwer, Vol. 16, No. 3, pp. 213-226, June, 2000.

[16] J.Raik, R.Ubar, "Enhancing Hierarchical ATPG with a Functional Fault Model for Multiplexers", Proc. IEEE DDECS, pp.219-222, April 2004.

[17] J.Raik, R.Ubar. "High-Level Path Activation Technique to Speed Up Sequential Circuit Test Generation", Proc. of the ETW, pp. 84-89, Konstanz, May 25-28, 1999.

[18] R. Ubar, "Test Generation for Digital Systems on the Vector Alternative Graph Model", Proc. of the 13th Annual Int. Symp. on Fault Tolerant Computing, Milano, Italy, 1983, pp.374-377.

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

PAPER III

Raik, J; Rannaste, A; Jenihhin, M; Viilukas, T; Fujiwara, H; Ubar, R (2011). Constraint-Based Hierarchical Untestability Identification for Synchronous Sequential Circuits. Proceedings of IEEE European Test Symposium, (1 - 6). IEEE Computer Society Press.

Constraint-Based Hierarchical Untestability Identification for Synchronous Sequential Circuits

Jaan Raik, Anna Rannaste, Maksim Jenihhin, Taavi Viilukas, Raimund Ubar
Department of Computer Engineering
Tallinn University of Technology, Estonia
E–mail: jaan@pld.ttu.ee

Hideo Fujiwara
Graduate School of Information Science, Nara Institute of Science and Technology
Kansai Science City, 630–0192, Japan
E–mail: fujiwara@is.naist.jp

***Abstract*— The paper proposes a new hierarchical untestable stuck-at fault identification method for non-scan sequential circuits containing feedback loops. The method is based on deriving, minimizing and solving test path activation constraints for modules embedded into Register-Transfer Level (RTL) designs. First, an RTL test pattern generator is applied in order to extract the set of all possible test path activation constraints for a module under test. Then, the constraints are minimized and a constraint-driven deterministic test pattern generator is run providing hierarchical test generation and untestability proof in sequential circuits. We show by experiments that the tool is capable of quickly proving a large number of untestable faults obtaining high fault efficiency. As a side effect, our study shows that traditional bottom-up test generation based on symbolic test environment generation at RTL is too optimistic due to the fact that propagation constraints are ignored.**

I. INTRODUCTION

Test generation for sequential synchronous designs is a time-consuming task. Automated Test Pattern Generation (ATPG) tools spend a lot of effort not only for deriving test vectors for testable faults but also for proving that there exist no tests for the untestable faults. Because of this reason, the identification of untestable faults has been an important aspect in speeding up the sequential ATPG.

For combinational circuits, untestable faults occur due to the redundant logic in the circuit, while for sequential circuits, untestable faults (i.e. *sequentially untestable faults*) may also result due to unreachable states or due to impossible state transitions. A number of works have been proposed in order to tackle the problem of identifying sequentially untestable faults. The first methods [1] were fault-oriented and based on applying combinational ATPG to the expanded time-frame model of the sequential circuit. However, such approach does not scale because of the size-explosion of the unrolled sequential models. Thus, the fault independent method was introduced by Iyer et al. in [2]. The new algorithm was called FIRES and it implemented illegal state information to complement redundancy analysis. This was followed by a number of fault independent methods including MUST [3], FUNI [4], FILL [4] and others. Liang [5] proposed a simulation based approach for sequential untestable fault identification. However, it was shown in [4] that this method may result in ‘false positives’, i.e. a fault may be declared untestable when there actually exists a test for it. The common

limitation of the above methods is that they operate at the logic-level representation of the design. Thus a considerable amount of effort is put on the implication process carried out at the level of logic netlists.

In their previous work [6], the authors introduced a specific subclass of sequentially untestable faults, called register enable stuck-on faults and a method for proving them untestable using a model checker. In this paper we propose a hierarchical untestability identification method. The new method allows detecting sequential untestability in combinational modules (functional units, multiplexers) embedded into a hierarchical circuit and is based on path activation constraints extracted by a Register-Transfer Level (RTL) ATPG.

In hierarchical RTL test generation, top-down and bottom-up strategies are known. In the bottom-up approach, tests generated at the low-level will be later assembled at the higher abstraction level. Such algorithms yield short run-times but ignore the incompleteness problem: constraints imposed by other modules and/or the network structure may prevent test vectors from being assembled. In the top-down approach, constraints are extracted at the higher level as a goal to be considered when deriving tests for modules at the lower level. This approach allows testing modules embedded deep into the RTL structure. However, as modules are often tested through highly complex constraints, their fault coverage may be compromised.

Early methods on bottom-up RTL testing relied on the assembly of module tests and were applicable of the simplest systems only [7]. A more solid basis for the bottom-up paradigm was laid by Ghosh et al. in [8]. In their work, test environments are generated for each functional module of a given functional RTL circuit described in an Assignment Decision Diagram (ADD) [9] using symbolic justification/propagation rules using a nine-valued algebra. In this method, a test sequence is then formed by substituting the corresponding test patterns into the test environment. However, regardless of the existence of some test environments, the proposed nine-valued algebra cannot always generate the test environments. To overcome this drawback, Zhang et al. upgraded the nine-valued algebra to a ten-valued algebra by taking the signal line value range into consideration. This algebra is able to generate much more test environments [10]. In [11], Zhang's approach has been further improved by introducing additional propagation rules.

Lee and Patel introduced top-down constraint-based test pattern generation for microprocessors in [12]. Several constraint-based top-down approaches followed, including [13, 14]. [15] proposed a bottom-up approach based on a High-Level Decision Diagram (HLDD) engine and a commercial SICStus constraint solver. As experiments show, the tool achieves lower fault coverage in comparison to a commercial logic-level Automated Test Pattern Generator (ATPG). In [16], a top-down approach including a constraint solving package ECLiPSe [17] has been proposed.

None of the previous methods apply RTL constraints in order to prove logic-level untestable faults. Thus, the fault efficiency reported by the approaches [12-16] is often low, which decreases the test engineer's confidence in the test. (Fault efficiency refers to the ratio of the number of tested faults to the number of testable faults). In addition, as we will show in this paper, in many cases, fault coverage obtained for the modules in RTL test generation may considerably decrease if path activation constraints are being ignored.

In this paper we propose a new hierarchical untestability identification method for non-scan sequential circuits

containing feedback loops. To the best of our knowledge this is the first method that can prove sequentially untestable stuck-at faults starting from the RTL. The method is based on deriving, minimizing and solving test path constraints for modules embedded into RTL designs. First, an RTL test pattern generator is applied in order to extract the set of all possible test path activation constraints for a module under test within a certain clock cycle limit. Then, the constraints are minimized and a constraint-driven deterministic test pattern generator is run providing a time-limit-bounded hierarchical test generation and untestability proof for sequential circuits. We show by experiments that the tool is capable of quickly proving a large number of untestable faults obtaining high fault efficiency. As a side effect, our study shows that traditional bottom-up test generation based on symbolic test environment generation at RTL is too optimistic due to the fact that propagation constraints are ignored.

The paper is organized as follows. Section 2 presents the definition of ADD models, which is used as the basis of presenting the untestability identification method. In Section 3, the new untestability indentification setup is presented and a motivating example showing the limitations of existing bottom-up approaches is presented. Section 4 explains the process of obtaining the set of RTL constraints for proving sequential untestability. Section 5 discusses the core benefits of the proposed method. Section 6 provides experimental results. The paper ends with Conclusions.

II. TEST ENVIRONMENT GENERATION WITH ADDS

Assignment decision diagram (ADD) [9] is an acyclic graph that consists of a set of nodes that can be categorized into four types: read node, write node, operation node and assignment decision node (ADN), and a set of edges which contain the connectivity information between two nodes (Figure 1). A read node represents a primary input port, a storage unit or a constant while a write node represents a primary output port or a storage unit. An operation node expresses an arithmetic operation unit or a logic operation unit while an ADN selects a value from a set of values that are provided to it based on the conditions computed by the logic operation units. If one of the condition inputs becomes true, the value of the corresponding data input will be selected.

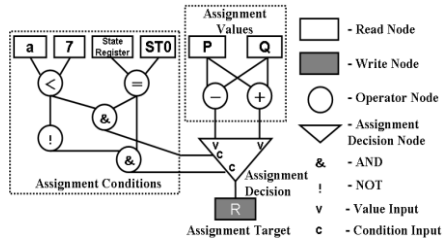


Figure 1. Assignment Decision Diagram (ADD)

When a node *N* is under test, the testability of the node is guaranteed if (a) any value can propagate from a read node corresponding to a primary input port to the input of *N*, and (b) the value at the output of *N* can propagate to a write node corresponding to a primary output port. The paths which allow (a) and (b) to occur are called *justification path* and *propagation path*, respectively. Justification and propagation can be done through symbolic processing that utilizes nine-valued algebra. The series of symbols obtained from the symbolic processing that activates justification and propagation paths is known as the *test environment* for the node under test.

For a given node under test, its test sequence is generated by first extracting a test pattern from the *test set library* and by substituting the test pattern for the test environment. The test set library is obtained beforehand by first simply taking a logic-level circuit of the node

under test, then generating the test patterns for all faults in the circuit using a combinational ATPG algorithm. In the case where the node is synthesized into a circuit which is different, fault simulation must be performed to check the fault efficiency of the test patterns.

The symbols of Ghosh's nine-valued algebra [10], each of which can be assigned true or false, are as follows:

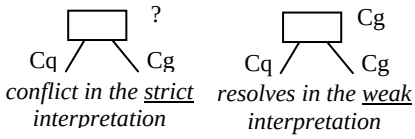
• *Cg*(*v*): variable *v* can be set to any value.
• *C0*(*v*): variable *v* can be set to 0.
• *C1*(*v*): variable *v* can be set to 1.
• *Ca1*(*v*): all bits of variable *v* can be set to 1's.
• *Cq*(*v*): variable *v* can be set to a constant.
• *Cz*(*v*): variable *v* can be set to high impedance *Z*.
• *Cs*(*v*): state variable *v* can be set to a specific state.
• *O*(*v*): any fault effect at variable *v* can be observed.
• *O'*(*v*): fault effect of *D'* can be observed for a single bit variable v.

To generate a test environment, first an objective has to be set. In order to achieve the test environment objective, the test sequence for each ADD can be generated through the following two phases using justification/propagation rules [10]:

Phase 1: Generate the test environment of the node under test.

Phase 2: Generate the test sequence of the node under test by substituting the test patterns of the logic-level circuit corresponding to the node under test for the test environment.

Without going into details of the symbol propagating rules, consider Figure 2 presenting backward propagation (justification) of two symbols Cq and Cg that converge in a fanout read node. In the *strict interpretation* of the propagation rules of [10] the two symbols when converging in the fanout result in a conflict. In the weak interpretation the symbols will resolve in assigning Cg to the read node.



a) b)

Figure 2. Handling of fanouts during justification

Thus, the strict interpretation of Ghosh's algebra [10] lead to overly pessimistic results because tests for some Modules Under Test (MUTs) are aborted due to justification conflicts. On the other hand, the weak interpretation is too optimistic and can also lead to loss of fault coverage because some of the test patterns that are expected to cover faults in the MUT do not propagate. Experiments in this paper show that this loss may be as high as 8-14 percent of the stuck-at fault coverage.

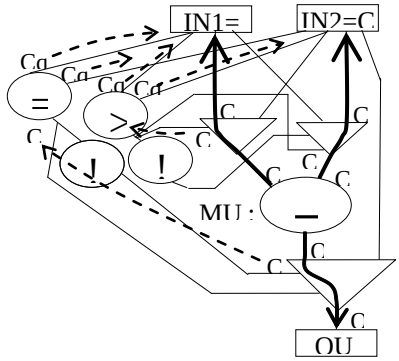


Figure 3. Test environment generation example. An unrolled view.

Consider as an example, a simplification of the ADD for the Greatest Common Division (GCD) benchmark presented in Figure 3. Without loss of generality in this ADD the control state

information and the data registers have been removed and the circuit has been unrolled by applying time-frame expansion in order to improve the readability of the diagram. (Note, that the original GCD benchmark still contains a data dependent loop, which has been unrolled in Figure 3).

Assume that our task is generating a test environment for the subtraction module (MUT) in Figure 3. The output value of MUT will be propagated to the primary output OUT only if the first value input of the corresponding assignment decision is 1. Therefore we set the corresponding condition input of the ADN to C1. When we justify this particular condition input and the symbols at the MUT inputs according to the propagation rules presented in [10], then the strict interpretation of these rules would lead into a contradiction (See Figure 2a). However, the weak interpretation (also used in [11]) would still allow the following test environment: IN1=Cg and IN2=Cg. Note, that in current situation the weak rules are preferable since they at least allow testing part of the MUT while the strict rules would not generate any test environment at all.

III. CONSTRAINT-BASED UNTESTABILITY PROOF FLOW

As opposed to the bottom-up test environment generation presented in Section 2, the constraint-driven deterministic untestability identification method proposed in current paper is based on the top-down approach. The method contains three main phases. During the first phase, the full set of constraints for setting up a test path to test an RTL module are extracted at the high-level. During the second phase this set of constraints is minimized. The third phase generates deterministic tests to the low-level module taking into account the path constraints.

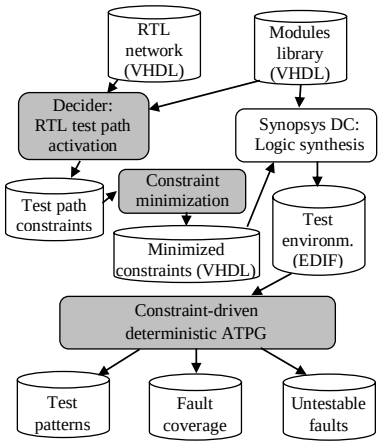


Figure 4. Constraint-based untestability proof flow

Figure 4 presents the corresponding test flow. We apply RTL ATPG Decider [16] in order to extract the constraints for accessing the MUT. Decider activates as many sets of constraints as there are test paths for that module in a bounded limit of clock-cycles. In [16], test constraints were utilized to propagate test patterns to and from the MUT. However, in this paper the purpose is to process the set of constraints in order to derive conditions for a dedicated logic-level ATPG in proving untestability. The constraints are minimized as shown in the next Section, translated into VHDL and synthesized to logic-level using Synopsys Design Compiler. Subsequently, the constraint-driven logic-level ATPG is run. As a result we obtain the list of sequentially untestable faults in the MUT as well as test patterns for the entire design.

IV. CONSTRAINT-BASED TEST ENVIRONMENT

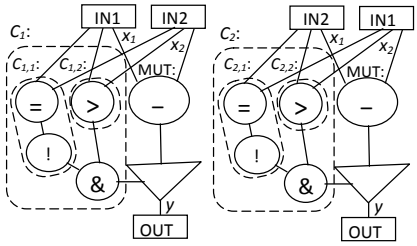
In this Section we explain minimization of the test path constraints for a MUT. We show how to compute the constraint-based test environment from the set of test constraints. For the sake of completeness we briefly summarize the concept of test generation constraints below.

In order to extract the RTL constraints for a MUT, an RTL ATPG tool Decider [16] is applied. The high-level test generation constraints considered by Decider are divided into three categories. These are path activation constraints, transformation constraints and propagation constraints. *Path activation constraints* correspond to the logic conditions in the control flow graph that have to be satisfied in order to perform propagation and value justification through the circuit. *Transformation constraints*, in turn, reflect the value changes along the paths from the inputs of the high-level MUT to the primary inputs of the whole circuit. These constraints are needed in order to derive the local test patterns for the module under test. *Propagation constraints* show how the value propagated from the output of the MUT to a primary output is depending on the values of the signals in the system. The main idea here is to guarantee that fault effect will not be masked when propagated.

All the above categories of constraints are represented by common data structures and manipulated by common procedures for creation, update, modeling and simulation.

Note, that the extracted constraints consist of operations on primary inputs and constants. Furthermore, the exponential size complexity of the constraints is avoided by uniting multiple occurrences of the same variable (i.e. the literals) in the constraints at each time step into one single fanout variable. The size requirements for the constraints are linear with respect to justification time-frames and they represent a subset of the expanded time-frame model of the circuit.

Consider Figure 5, which gives the ADD of the full set of constraints for the MUT from the example of Figure 3. In other words, the MUT can only be tested using one of the two test paths presented in Figure 5a and 5b. The two paths are identical except for the fact that the primary inputs IN1, IN2 are swapped in them.



a) b)

Figure 5. Full set of test path constraints for MUT

Note, that from the point of view of accessing the MUT these two environments are equivalent. It is irrelevant which primary input is used in applying the test patterns when representing the constraint-based test environment for proving untestability. Therefore, we denote the value justified from the i-th input of the MUT by x_k and the value propagated from the MUT output by y.

The constraints C_1 and C_2 both consist of two sub-constraints $C_{1,1}$, $C_{1,2}$ and $C_{2,1}$, $C_{2,2}$, respectively. $C_{1,1}$ (which is equivalent to $C_{2,1}$) states that x_1 must not be equal to x_2. $C_{1,2}$ (equivalent to $C_{2,2}$) states that x_1 must be greater than x_2. Since all the sub-constraints within a constraint should hold simultaneously they be combined using the conjunction operator. In turn, all the constraints are combined using the disjunction operation because any one of the test paths may be used for accessing the MUT. In general case for constraints C_i each consisting of sub-constraints $C_{i,j}$ the constraint-environment for proving sequential untestability is calculated using the following formula:

$$\bigvee_i \bigwedge_j C_{i,j}. \quad (1)$$

Subsequent to combining the test path constraints constraint minimization is performed. For the example in Figure 5 we obtain:

$$(x_1 \neq x_2) \wedge (x_1 > x_2) \vee (x_1 \neq x_2) \wedge (x_1 > x_2) = x_1 > x_2.$$

Figure 6 shows the ADD for the minimized constraint-based environment resulting for testing the MUT of the example presented in Figure 3. The constraint shows that the MUT (a subtractor) may only be accessed when the first input of it, i.e. x_1 is greater than the second one, x_2.

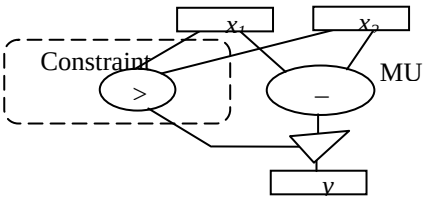


Figure 6. Constraint-based test environment for MUT

V. DISCUSSION ON THE EFFECT OF THE TOP-DOWN PROOF

Existing high-level ATPG methods do not allow proof of sequentially untestable stuck-at faults. An exception is a previous work by the authors where a specific class of untestable register control faults were proven untestable by applying model-checking at the RTL [6]. The current work considers the general case of sequentially untestable stuck-at faults within RTL modules.

As a side-effect of our study, we show that the top-down test environment generation is more accurate than the bottom-up one. In particular, the strict interpretation of Ghosh's algebra leads to overly pessimistic results because tests for some MUTs are aborted due to justification conflicts. On the other hand, the weak interpretation is too optimistic and can also lead to loss of fault coverage because some of the test patterns that are expected to cover faults in the MUT do not propagate.

Consider the case where in a bottom-up scenario we have a deterministic test T_q generated for the MUT reaching the maximum fault coverage W_q for the module. Then, we generate the test environment for the module and substitute T_q into the test environment. Due to the test path constraints the actual fault coverage that can be achieved for the MUT inside the network is W_a, which is generally lower than the fault coverage W_q. However, when we fault simulate T_q substituted into the test environment we obtain a fault coverage W_r, where $W_r \leq W_a \leq W_q$.

In other words, the bottom-up approach may lose some fault coverage with respect to the top-down one because the set of the tests to choose from is restricted to T_q. If the local test generation algorithm for the MUT had had knowledge about the test path constraints it would have generated a different test T_a, whose fault coverage would have been equal to W_a. Furthermore, the remaining faults inside the MUT would have been proven untestable. Thus, a deterministic ATPG taking into account the test path constraints is necessary in order to achieve maximum fault coverage and also to prove untestability within sequential circuits. Experiments with the constraint-driven deterministic ATPG presented in Section 6 show that the difference between the coverages W_r and W_a may be even as high as 8-14 per cent of stuck-at coverage.

VI. EXPERIMENTAL RESULTS

In order to evaluate the hierarchical untestability identification and test generation method, experiments on HLSynth92 and HLSynth95 benchmarks were run. In addition, to compare the solution with the traditional bottom-up approach (e.g. [10]) and assess its fault efficiency, a detailed case-study was carried out.

Table 1 presents the characteristics of the example circuits used in test pattern generation experiments in this paper. The following benchmarks were included to the test experiment: a Greatest Common Divisor (GCD), an 8-bit sequential multiplier (MULT8x8), and a Differential Equation (DIFFEQ). In the Table, the number of single stuck-at faults, the number of primary input and primary output bits, and the number of registers, multiplexers and functional units in the RTL code are reported, respectively.

TABLE I. BENCHMARK CHARACTERISTICS

circuit	# faults	PI bits	PO bits	# reg.	# mux	# FU
gcd	472	33	16	3	4	3
mult8x8	2356	17	16	7	4	9
diffeq	10326	81	48	7	9	5

In Table 2, comparison of test generation results of three ATPG tools on the hierarchical benchmark designs are presented. This comparison was carried out in order to show the time needed for extracting the constraint-based environment as explained in Section 4. The tools include a logic-level deterministic ATPG Hitec [18], a genetic algorithm based Gatest [19], hierarchical ATPG Decider applied in current paper. Columns 'F.C., %' give the single stuck-at fault coverages of the test patterns generated. Columns 'time, s' stand for test generation run-times in seconds. As it can be seen the three sequential designs analyzed introduce a serious challenge to the deterministic and genetic algorithm-based ATPG tools. For the former, the search space becomes too large and many faults have to be dropped after a time-out value has been encountered. For the latter, the genetically engineered vectors are unable to create tests for faults that require specific sequences for activation and propagation.

TABLE II. COMPARISON OF SEQUENTIAL ATPG

circuit	HITEC		GATEST		Decider	
	F.C., %	time, s	F.C., %	time, s	F.C., %	time, s
gcd	59.11	365	86.13	190.7	**90.95**	677.4
mult8x8	65.9	1243	69.2	821.6	**74.7**	93.7
diffeq	96.2	13,320	96.40	3000	**97.09**	453.7

Table 3 shows experiments of the constraint-driven ATPG developed in this paper. The experiments present comparison of the proposed method to the bottom-up paradigm [10]. For creating the test library for the bottom-up approach, the modules were first tested by the ATPG in a stand-alone mode. As a result a test sequence T_q yielding 100 % stuck-at fault coverage W_q was obtained. The proposed top-down constraint-driven ATPG reached fault coverage W_a which was less than W_q because of the constraints when accessing the module under test that was embedded into the network. However, the fault efficiency of the proposed approach was always 100 % for all the modules.

When test T_q was substituted to the test environment in a bottom-up manner then fault coverage W_r was reached, which was always lower than W_a because some of the tests were invalidated by sequential dependencies. In fact, W_r was considerably lower (by 8-14 %) for all the four modules analyzed. Thus, the proposed top-down method was capable of reaching maximum fault coverage for the analyzed module and proving all of the sequentially untestable faults in them.

The test environment synthesis from VHDL to logic-level using Synopsys Design Compiler remained almost constant and was around 5 to 10 s per module while the deterministic constraint-based ATPG spent less than 0.02 s per module under test. The synthesis and test experiments were carried out on a Sun-

Fire-V250 station with 1.28 GHz sparcv9 processor under Solaris 2.9 OS.

TABLE III. CONSTRAINT-DRIVEN TOP-DOWN ATPG VERSUS BOTTOM-UP ATPG RESULTS FOR CIRCUIT MODULES

circuit:	gcd	mult8x8			diffeq	
module:	SUB	ADD2	ADD3	SUB2	MUX3	MUX4
W_g, %	100	100	100	100	100	100
W_a, %	**95.74**	**86.64**	**55.88**	**85.33**	**75.00**	**75.00**
W_e, %	85.11	72.49	47.06	74.07	64.71	64.71
ATPG, s	0.01	0.01	< 0.01	0.02	< 0.01	< 0.01
synthesis, s	5.38	5.33	9.52	5.25	5.10	5.10

Table 4 presents detailed statistics of the circuits analyzed. The Table lists the total number of stuck-at faults in the whole circuit, the number of tested faults, number of unobservable/uncontrollable faults, the number of faults proven sequentially untestable by the proposed constraint-based approach and finally the number of all the remaining faults. The experiments show the efficiency of the constraint-driven engine in untestability identification. Though the method quickly classifies untestable faults caused by sequential untestability in the considered modules with 100 % fault efficiency, there remains a number of faults which are still neither tested nor proven untestable. Some of these remaining faults can be tested or proven untestable by traditional approaches at the logic-level.

TABLE IV. DISTRIBUTION OF FAULTS

	gcd	mult8x8	diffeq
# total faults	472	2356	10326
# tested faults	439	1737	9867
# unobs./uncontr.	28	195	252
# sequentially untestable faults	**4**	**156**	**68**
# remaining faults	1	268	139

VII. CONCLUSIONS

The paper introduces a new method and tool for hierarchical untestable stuck-at fault analysis of non-scan sequential circuits. The method is based on extracting and minimizing RTL test path activation constraints that drive a dedicated logic-level deterministic ATPG. Experiments show that the tool is capable of generating tests yielding maximum fault efficiency for the embedded modules under test. To the best of the authors' knowledge this is the first method that can prove sequential untestability starting from the RTL.

In addition, our study shows that traditional test generation at RTL based on symbolic test environment generation is too optimistic due to the fact that constraints in accessing the modules under test have been ignored. Experiments presented in this paper showed that bottom-up strategies caused a decrease of stuck-at fault coverage up to the range of 8-14 % in the modules tested when compared to the proposed approach. This short-coming is now overcome by the proposed constraint-based method which obtains 100 per cent stuck-at fault efficiency for all the modules considered.

ACKNOWLEDGMENTS

The work has been supported by European Commission Framework Program 7 project FP7-ICT-2009-4-248613 DIAMOND, by Research Centre CEBE funded by European Union through the European Structural Funds and by Estonian Science Foundation grants 7068 and 7483.

REFERENCES

[1] V. D. Agrawal and S. T. Chakradhar, "Combinational ATPG theorems for identifying untestable faults in sequential circuits," *IEEE Trans Comput.-Aided Des.*, vol. 14, no. 9, pp. 1155–1160, Sep. 1995.

[2] M. A. Iyer, D. E. Long, and M. Abramovici, "Identifying sequential redundancies without search," *in Proc. 33rd Annu. Conf. DAC*, LasVegas, NV, Jun. 1996, pp. 457–462.

[3] Q. Peng, M. Abramovici, and J. Savir, "MUST:Multiple stem analysis for identifying sequential untestable faults," *in Proc. Int. Test*

Conf., Atlantic City, NJ, Oct. 2000, pp. 839–846.

[4] D. E. Long, M. A. Iyer, M. Abramovici, "FILL and FUNI: Algorithms to identify illegal states and sequentially untestable faults," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 5, no. 3, pp. 631–657, Jul. 2000.

[5] H.-C. Liang, C. L. Lee, and E. J. Chen, "Identifying untestable faults in sequential circuits," *IEEE Des. Test. Comput*., vol. 12, no. 3, pp. 14–23, Sep. 1995.

[6] J. Raik, H. Fujiwara, R. Ubar, A. Krivenko. "Untestable fault identification in sequential circuits using model-checking". *ATS*, pp. 667-672, 2008.

[7] B. T. Murray, J. P. Hayes, "Hierarchical test generation using precomputed tests for modules", *Proc. ITC*, pp.221-229, 1988.

[8] I. Ghosh, M. Fujita, "Automatic test pattern generation for functional RTL circuits using assignment decision diagrams", *Proc. DAC*, pp. 43-48, 2000.

[9] V. Chayakul, D. D. Gajski, L. Ramachandran, "High-Level Transformations for Minimizing Syntactic Variances", *DAC*, pp. 413-418, 1993.

[10] L. Zhang, I. Ghosh, M. Hsiao, "Efficient Sequential ATPG for Functional RTL Circuits", *Int. Test Conf*., pp.290-298, 2003.

[11] H. Fujiwara, C. Y. Ooi, Y Shimizu, "Enhancement of Test Environment Generation for Assignment Decision Diagrams", *9th IEEE Workshop on RTL and High Level Testing*, Nov. 27-28, 2008.

[12] J. Lee, J.H. Patel, "Architectural level test generation for microprocessors", *IEEE Trans. CAD*, pp. 1288-1300, Oct. 1994.

[13] J. Raik, R. Ubar. Sequential Circuit Test Generation Using Decision Diagram Models, *Proceedings of the DATE Conference*, pp. 736-740, 1999.

[14] V. Vedula, J. Abraham, "FACTOR: A Hierarchical Methodology for Functional Test Generation and Testability Analysis," *DATE Conf.*, 2002.

[15] G. Jervan et al., "High-Level and Hierarchical Test Sequence Generation", *IEEE HLDVT*, Cannes, 2002.

[16] T. Viilukas, J. Raik, M. Jenihhin, R. Ubar, A. Krivenko, "Constraint-based test pattern generation at the register-transfer level", *13th IEEE DDECS Symposium*, 2010, pp. 352-357.

[17] The ECLiPSe Constraint Programming System http://eclipse-clp.org/

[18] T. M. Niermann, J. H. Patel, "HITEC: A test generation package for sequential circuits", *Proc. EDAC*, pp. 214-218, 1991.

[19] E. M. Rudnick, et al. "Sequential circuit test generation in a genetic algorithm framework", *Proc. DAC*, pp. 698-704, 1994.

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON INFORMATICS AND SYSTEM ENGINEERING

1. Lea Elmik. Informational Modelling of a Communication Office. 1992.

2. Kalle Tammemäe. Control Intensive Digital System Synthesis. 1997.

3. Eerik Lossmann. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. Kaido Kikkas. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. Nazmun Nahar. Global Electronic Commerce Process: Business-to-Business. 1999.

6. Jevgeni Riipulk. Microwave Radiometry for Medical Applications. 2000.

7. Alar Kuusik. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. Jaan Raik. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. Andri Riid. Transparent Fuzzy Systems: Model and Control. 2002.

10. Marina Brik. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. Raul Land. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. Ants Ronk. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. Toivo Paavle. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. Irina Astrova. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. Kuldar Taveter. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. Taivo Kangilaski. Eesti Energia käiduhaldussüsteem. 2004.

17. Artur Jutman. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

18. Ander Tenno. Simulation and Estimation of Electro-Chemical Processes in Maintenance-Free Batteries with Fixed Electrolyte. 2004.

19. Oleg Korolkov. Formation of Diffusion Welded Al Contacts to Semiconductor Silicon. 2004.

20. Risto Vaarandi. Tools and Techniques for Event Log Analysis. 2005.

21. Marko Koort. Transmitter Power Control in Wireless Communication Systems. 2005.

22. Raul Savimaa. Modelling Emergent Behaviour of Organizations. Time-Aware, UML and Agent Based Approach. 2005.

23. Raido Kurel. Investigation of Electrical Characteristics of SiC Based Complementary JBS Structures. 2005.

24. Rainer Taniloo. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. Pauli Lallo. Adaptive Secure Data Transmission Method for OSI Level I. 2005.

26. Deniss Kumlander. Some Practical Algorithms to Solve the Maximum Clique Problem. 2005.

27. Tarmo Veskioja. Stable Marriage Problem and College Admission. 2005.

28. Elena Fomina. Low Power Finite State Machine Synthesis. 2005.

29. Eero Ivask. Digital Test in WEB-Based Environment 2006.

30. Виктор Войтович. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. Tanel Alumäe. Methods for Estonian Large Vocabulary Speech Recognition. 2006.

32. Erki Eessaar. Relational and Object-Relational Database Management Systems as Platforms for Managing Softwareengineering Artefacts. 2006.

33. Rauno Gordon. Modelling of Cardiac Dynamics and Intracardiac Bioimpedance. 2007.

34. Madis Listak. A Task-Oriented Design of a Biologically Inspired Underwater Robot. 2007.

35. Elmet Orasson. Hybrid Built-in Self-Test. Methods and Tools for Analysis and Optimization of BIST. 2007.

36. Eduard Petlenkov. Neural Networks Based Identification and Control of Nonlinear Systems: ANARX Model Based Approach. 2007.

37. Toomas Kirt. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. Juhan-Peep Ernits. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. Innar Liiv. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. Andrei Pokatilov. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. Karin Lindroos. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. Maksim Jenihhin. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. Ando Saabas. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. Ilja Tšahhirov. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. Toomas Ruuben. Wideband Digital Beamforming in Sonar Systems. 2009.

46. Sergei Devadze. Fault Simulation of Digital Systems. 2009.

47. Andrei Krivošei. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components. 2009.

48. Vineeth Govind. DfT-Based External Test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. Andres Kull. Model-Based Testing of Reactive Systems. 2009.

50. Ants Torim. Formal Concepts in the Theory of Monotone Systems. 2009.

51. Erika Matsak. Discovering Logical Constructs from Estonian Children Language. 2009.

52. Paul Annus. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. Maris Tõnso. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. Aivo Jürgenson. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. Erkki Joasoon. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. Jürgo-Sören Preden. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.

57. Pavel Grigorenko. Higher-Order Attribute Semantics of Flat Languages. 2010.

58. Anna Rannaste. Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.

59. Sergei Strik. Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.

60. Rain Ottis. A Systematic Approach to Offensive Volunteer Cyber Militia.2011.

61. Natalja Sleptšuk. Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.

62. Martin Jaanus. The Interactive Learning Environment for Mobile Laboratories. 2011.

63. Argo Kasemaa. Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.

64. Kenneth Geers. Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

65. Riina Maigre. Composition of Web Services on Large Service Models. 2011.

66. Helena Kruus. Optimization of Built-in Self-Test in Digital Systems. 2011.

67. Gunnar Piho. Archetypes Based Techniques for Development of Domains, Requirements and Sofware. 2011.

68. Juri Gavšin. Intrinsic Robot Safety Through Reversibility of Actions. 2011.

69. Dmitri Mihhailov. Hardware Implementation of Recursive Sorting Algorithms Using Tree-like Structures and HFSM Models. 2012.

70. Anton Tšertov. System Modeling for Processor-Centric Test Automation. 2012.

71. Sergei Kostin. Self-Diagnosis in Digital Systems. 2012.

72. Mihkel Tagel. System-Level Design of Timing-Sensitive Network-on-Chip Based Dependable Systems. 2012.

73. Juri Belikov. Polynomial Methods for Nonlinear Control Systems. 2012.

74. Kristina Vassiljeva. Restricted Connectivity Neural Networks based IdentiVcation for Control. 2012.

75. Tarmo Robal. Towards Adaptive Web – Analysing and Recommending Web Users‘ Behaviour. 2012.

76. Anton Karputkin. Formal Verification and Error Correction on High-Level Decision Diagrams. 2012

77. Vadim Kimlaychuk. Simulations in Multi-Agent Communication System. 2012.