THESIS ON INFORMATICS AND SYSTEM ENGINEERING C86

Fault Simulation and Code Coverage Analysis of RTL Designs Using High-Level Decision Diagrams

ULJANA REINSALU

TUT PRESS

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

Dissertation was accepted for the defense of the degree of Doctor of Philosophy in Computer and Systems Engineering on May 11, 2013

Supervisors: Prof. Peeter Ellervee
Prof. Jaan Raik
Dr. Aleksander Sudnitsõn

Opponents: Dr. Graziano Pravadelli, University of Verona, Italy
Dr. Juha Plosila, University of Turku, Finland

Defence of the thesis: June 12, 2013

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.

/Uljana Reinsalu/

European Union
European Social Fund

Investing in your future

ISSN 1406-4731
ISBN 978-9949-23-476-9 (publication)
ISBN 978-9949-23-477-6 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C86

Rikete simuleerimine ja koodikatte analüüs register-siirde tasemel kasutades kõrgtaseme otsustusdiagramme

ULJANA REINSALU

TTÜ
KIRJASTUS

To my great family

Abstract

This thesis addresses hardware testing issues as well as simulation-based hardware verification issues applied at register-transfer and behavioral levels of design abstraction. Particularly the main topics are Register-Transfer Level (RTL) fault simulation and structural coverage measurement exploiting advantages of High-Level Decision Diagrams (HLDD) design representation model.

First, a novel method for fault simulation at RTL based on the HLDD model is presented. The method is based on deductive fault simulation algorithm brought to higher level of abstraction and applied to the design represented by HLDDs. Efficient data structure was implemented into the algorithm in order to make fast bitwise operations with fault lists and this way to accelerate the fault simulation. Fault simulation is widely used in test stimuli generation for digital circuits. Other tasks as fault diagnosis, test stimuli compaction, built-in-self test optimization incorporate fault simulation as part of the process. Thus efficient fault simulation algorithm is very important for solving these tasks.

Second, a novel method for structural code coverage analysis based on the HLDD model is presented. Traditional code coverage metrics as statement coverage, branch coverage and toggle coverage are mapped onto HLDD constructs. With the help of fast HLDD-based simulation the measuring of these coverage is efficient. The method also implies manipulations with HLDDs for finding better HLDD model representation targeting different aspects in code coverage analysis. Moreover, observability coverage metric is implemented into HLDD simulation engine. This metric measures not only activation of the bugs but also evaluates the propagation of these bugs to the observable points. Observability coverage metric makes possible to better analyze the test stimuli and circuit's design.

All proposed methods rely on a HLDD-based simulation engine. Previous research works in TUT (Tallinn University of Technology) show that HLDDs are efficient models for digital circuits' simulation as well as convenient for diagnosis and debugging. The performed experiments confirm feasibility and efficiency of the proposed methods.

Kokkuvõte

Käesolev töö käsitleb nii digitaalriistvara testimise kui ka simuleerimisel põhineva verifitseerimise küsimusi register-siirde ja käitumuslikul tasemel. Töös pakutud lähenemised rikete simuleerimiseks register-siirde tasemel ning struktuurse katte mõõtmiseks kasutavad kõrgtaseme otsustusdiagrammide (KTOD) eeliseid skeemide esitamisel.

Kõigepealt on esitatud uudne meetod rikete simuleerimiseks register-siirde tasemel, mis põhineb KTOD mudelil. Meetod tugineb deduktiivsele rikete simuleerimisalgoritmile, mis on viidud kõrgemale abstraktsioonitasemele ning rakendatud KTOD-na esitatud digitaalriistvarale. Algoritmi on lisatud efektiivne andmestruktuur selleks, et kiirendada bitioperatsioone rikete nimekirjadega ning järelikult kiirendada rikete simulatsiooni tervikuna. Rikete simuleerimist kasutatakse laialt digitaalriistvara testi stiimulite genereerimisel. Sellised ülesanded nagu rikete diagnostika ja testi stiimulite kokkupakkimine isetestivate arhitektuuride projekteerimine vajavad oma töös rikete simuleerimist. Seega on efektiivne rikete simuleerimise algoritm väga tähtis nende ülesannete lahendamisel.

Teiseks on esitatud uudne meetod struktuurseks koodikatte analüüsiks, mis samuti põhineb KTOD mudelil. Traditsioonilised koodikatte mõõdud nagu lausete, harude ja andmevoo kated seoti KTOD struktuuriga. KTOD-põhine kiire simuleerimine võimaldab mõõta neid katteid efektiivselt. Samuti sisaldab pakutud meetod KTOD mudeli teisendusi, mis on suunatud koodikatte analüüsi erinevatele aspektidele. Lisaks on KTOD simulaatori jaoks realiseeritud jälgitavuse katte mõõt. See mõõt mõõdab mitte ainult vigade aktiveerimist vaid ka hindab nende levimist vaadeldavatesse punktidesse. Jälgitavuse katte mõõt võimaldab paremini analüüsida testi stiimuleid ning digitaalriistvara disaini.

Pakutud meetodid toetuvad KTOD-l põhinevale simulaatorile. Eelnev uurimistöö TTÜ-s on näidanud, et KTOD on efektiivne mudel simuleerimise läbiviimiseks ning sobilik digitaalsüsteemide diagnostikat ja silumist silmas pidades. Töös teostatud eksperimendid tõestavad pakutud lähenemiste rakendatavust ja efektiivsust.

Acknowledgements

I would like to express my sincere gratitude to everybody who have supported and advised me during my PhD studies.

First of all, I would like to thank my supervisors. I appreciate the support and advices of Prof. Peeter Ellervee. He is always open to discussions and helps in solving different problems. I am thankful to Prof. Jaan Raik for his guiding and consulting during the work on this thesis and also his joyful encouraging to continue working and to finish this thesis. His attitude to life gives me the energy in doing things. I would like to thank Aleksander Sudnitsõn for his remarks concerning this thesis. I would like to thank Prof. Raimund-Johannes Ubar for his help and wise advices.

Special thanks to the head of the Department of Computer Engineering Margus Kruus for creating outstanding environment for productive work and study.

I also would like to thank all my colleagues from Department of Computer Engineering for their interesting discussions and ideas. In particular I would express my appreciation to Sergei Devadze, Maksim Jenihhin, Artur Jutman, Marina Brik.

Moreover, I would like to acknowledge the organizations that have supported my PhD studies: Tallinn University of Technology, National Graduate School in Information and Communication Technologies (IKTDK), Estonian IT Foundation (EITSA), Centre of Integrated Electronic Systems and Biomedical Engineering (CEBE), FP7 STREP projects DIAMOND, FP6 STREP project VERTIGO.

Finally, I would like to thank my great family for the patience and support. In particular I would like to mention all my parents for their valuable support in my family nest, also my beloved husband Juri for his philosophical discussions and support and express my gratitude to my sweet children Artur and Timur for giving me chance for development.

Uljana Reinsalu,
Tallinn, May 2013

Table of Contents

List of Publications

Papers included in the thesis

1. Reinsalu, U.; Raik, J.; Ubar, R. (2010). Register-Transfer Level Deductive Fault Simulation Using Decision Diagrams. *In: Proceedings of the 12th Biennial Baltic Electronic Conference BEC2010,* Tallinn, Estonia, 2010, pp. 193 – 196
2. Reinsalu, Uljana; Raik, Jaan; Ubar, Raimund; Ellervee, Peeter (2011). Fast RTL Fault Simulation Using Decision Diagrams and Bitwise Set Operations. *Proceedings of 26th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. Vancouver, Canada, 2011, pp. 164-170
3. Raik, J.; Reinsalu, U.; Ubar, R.; Jenihhin, M.; Ellervee, P. (2008). Code Coverage Analysis using High-Level Decision Diagrams. *In: Proceedings of the 11th IEEE Workshop on Design and Diagnostics of Electronic Systems (DDECS), 16-18 April 2008,* Bratislava, Slovakia
4. Minakova, K.; Reinsalu, U.; Chepurov, A.; Raik, J.; Jenihhin, M.; Ubar, R.; Ellervee, P. (2008). High-Level Decision Diagram Manipulations for Code Coverage Analysis. *The 11th Biennial Baltic Electronics Conference (BEC'08)*, Tallinn, Estonia, Oct. 2008, pp. 207 – 210

System level modeling

5. Reinsalu, U.; Arhipov, A.; Ellervee, P. (2008). Architectural Exploration Tasks for On-Chip Embedded Systems. *The 11th Biennial Baltic Electronics Conference (BEC'08),* Tallinn, Estonia, Oct. 2008., pp. 171 – 174
6. Ellervee, P.; Reinsalu, U.; Arhipov, A. (2007). Translating Behavioral VHDL for Emulation. *25th NORCHIP Conference*, Aalborg, Denmark, Nov. 2007

7. Ellervee, P.; Arhipov, A.; Reinsalu, U. (2007). Using Emulation for System Model Analysis. *DATE'07 Friday Workshop on "Diagnostic Services in Network-on-Chips",* Nice, France, April 2007, pp. 280 – 282

Educational topics

8. Raik, J.; Jenihhin, M.; Chepurov, A.; Reinsalu, U.; Ubar, R. (2008). APRICOT: a Framework for Teaching Digital Systems Verification. *19th EAEEIE Annual Conference,* Tallinn, Estonia, 2008, pp. 1 - 6
9. Ellervee, P.; Reinsalu, U.; Arhipov, A.; Ivask, E.; Tammemäe, K.; Evartson, T.; Sudnitson, A. (2008). HDL-s and FPGA-s in Digital Design Education. *The 19th EAEEIE Annual Conference*, Tallinn, Estonia, June 2008, pp. 37 – 41
10. Reinsalu, U.; Arhipov, A.; Evartson, T.; Ellervee, P. (2007). HDL-s for Students with Different Background. *International Conference on Microelectronic Systems Education (MSE'07),* San Diego, CA, USA, June 2007, pp. 69 – 70
11. Ellervee, P.; Reinsalu, U.; Arhipov, A. (2006). Teaching HDL for IT-Students. *The 6th European Workshop on Microelectronics Education (EWME'2006*), Stockholm, Sweden, June 2006, pp. 112 – 115

Author’s Contribution to the Publications

Research paper I [64] Reinsalu, U.; Raik, J.; Ubar, R. (2010). Register Transfer Level Deductive Fault Simulation Using Decision Diagrams. *In: Proceedings of the 12th Biennial Baltic Electronic Conference BEC2010,* Tallinn, Estonia, 2010, pp. 193 – 196

The author modified the deductive fault simulation algorithm for RTL abstraction. The author implemented this algorithm using HLDDs. The author carried out the experiments with selected set of benchmarks and analyzed the results from the experiments. The author prepared the publication of the paper and presented this paper at the conference.

Research paper II [63] Reinsalu, Uljana; Raik, Jaan; Ubar, Raimund; Ellervee, Peeter (2011). “Fast RTL Fault Simulation Using Decision Diagrams and Bitwise Set Operations.”, *Proceedings of 26th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. Vancouver, Canada, 2011, pp. 164-170

The author proposed new data structure for storing the fault list required during RTL deductive fault simulation algorithm work. This data structure is efficient for making bitwise set operations. The author integrated this new data structure into the previously implemented RTL deductive fault simulation algorithm. The author carried out the experiments with selected set of benchmarks and analyzed the results from the experiments. The author prepared the publication of the paper and presented this paper at the conference.

Research paper III [59] Raik, J.; Reinsalu, U.; Ubar, R.; Jenihhin, M.; Ellervee, P. (2008). Code Coverage Analysis using High-Level Decision Diagrams. *In: Proceedings of the 11th IEEE Workshop on Design and Diagnostics of Electronic Systems (DDECS), 16-18 April 2008,* Bratislava, Slovakia

The author proposed the mapping of structural code coverage metrics onto high-level decision diagrams. The author integrated proposed metrics into existing HLDD simulation engine. The author carried out the experiments with selected set of benchmarks and analyzed the results from the experiments. The author helped preparing the publication of the paper and presented this paper at the conference.

Research paper IV [54] Minakova, K.; Reinsalu, U.; Chepurov, A.; Raik, J.; Jenihhin, M.; Ubar, R.; Ellervee, P. (2008). High-Level Decision Diagram Manipulations for Code Coverage Analysis. *The 11th Biennial Baltic Electronics Conference (BEC'08)*, Tallinn, Estonia, Oct. 2008, pp. 207 – 210

The author proposed the set of experiments with different modifications of HLDDs for code coverage analysis. The author supervised the experiments and helped in analysis the results. The author helped in preparation of the paper.

List of Abbreviations

AGM	Alternative Graph Model
APRICOT	Assertions, PRopertIes, Coverage and Test
ASIC	Application Specific Integrated Circuit
ATPG	Automatic Test Pattern Generator
BDD	Binary Decision Diagram
BIST	Built-In Self-Test
CAD	Computer Aided Design
CMOS	Complementary metal–oxide–semiconductor
CUT	Circuit Under Test
DD	Decision Diagram
DUV	Design Under Verification
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
FPGA	Field Programmable Gate Array
GCD	Greatest Common Devisor
GUI	Graphical User Interface
HDL	Hardware Description Language
HLDD	High-Level Decision Diagrams
IEEE	Institute of Electrical and Electronics Engineers
PC	Personal Computer
RTL	Register Transfer Level

SSF	Single Stuck-Fault
TLM	Transaction Level Modeling
TPG	Test Pattern Generation
TUT	Tallinn University of Technology
VHDL	VHSIC (Very-High-Speed Integrated Circuit) Hardware Description Language
VLSI	Very Large Scale Integration

Latin and English abbreviations:

aka - also known as
e.g. - for example
et al. - and other co-authors
etc. - and the rest
i.a. - among others
i.e. - that is
vs. - versus

Chapter 1
INTRODUCTION

This thesis presents hardware testing issues as well as simulation-based hardware verification issues. Particularly, the main topics are Register-Transfer Level (RTL) fault simulation and structural coverage measurement exploiting advantages of the High-Level Decision Diagrams (HLDD) design representation model.

This chapter begins with motivation to this work, followed by the problem formulation. Then, summary of the main contributions and an overview of the thesis structure are described.

1.1 Motivation

Different electronic devices have become a part of everyday's life. Nowadays electronic devices are developed not only for specific fields such as military, avionics, space, medical applications, etc., but also in quantity for general use such as mobile phones, tablet PCs, and many others. Although some devices have quite simple functionality, a huge amount of surrounded electronics has become more complex with wide range of functionality. This is thanks to the tremendous progress in the CMOS (Complementary metal–oxide–semiconductor) technology. According to the famous Moore's law [55][40], the number of transistors on integrated circuits doubles every two years. Thus complexity of integrated circuits grows, devices become smaller, density of transistors grows, which allow using a lot of functionality within one circuit. Despite the fact that such complex devices need more man power resources to implement, time-to-market imposes even shorter time than it demanded before for less complex devices. This fact raises the need for efficient EDA (Electronic

Design Automation) tools. The more automation included into the tools, the easier the process of finding the suitable solutions.

In this thesis, only the digital part of the systems is taken into account. During the last ten years digital electronic devices have become an important part of daily life. People have got dependent on surrounding electronics and its correct functionality. Strong reliability issues are a must for space, automotive applications, however reliability issues have become very significant for consumer electronics as well. Nobody wants a malfunctioning mobile phone during a very important talk or a tablet PC turning off while critical work is done on it. Also, consumers want to have manifold functionalities on their devices and new features can be desirable only if basic functionality does not fail. This obligates producers to spend more effort on reliability issues. In order to reach a certain level of reliability, considerable testing of electronic devices is required.

The cost of a hardware error is very high for the industry. It cannot be easily fixed by applying a patch as it is usually done for software products. A new device must be reproduced with errors fixed by withdrawing the previous version, which is extremely costly for the producer. There are many causes of errors: errors in specification, errors at any level of implementation, physical defect of manufacturing. Moreover, errors in hardware can appear during the lifetime of a device in consequence of a variety of reasons, such as high temperature or radiation for example. Therefore, it is strongly important to verify a design at every stage of implementation by fixing the bugs at any cost before manufacturing. Also, testing the devices for manufacturing defects is obligatory for every device [13].

Design cycles of the circuit are divided into several abstraction levels. Usually top-down design methodology is used. Thus a lot of decisions should be done at higher level of abstraction in order to shorten time-to-market. Therefore, many design tasks, which were used to be implemented at the gate level, are transferred to the register-transfer level (RTL) and higher levels. Sure, at this level one can not have exact data, however sufficient estimation can be done, which allows throwing out unsuitable solutions very early. Also, verification is mandatory after each step of design flow, detecting errors as early as possible and avoiding propagation of errors to lower levels of abstraction, thereby saving time and money.

One of the topics of the current thesis is improving fault simulation techniques, which is one of the most important issues in digital testing. Fault simulation is brought to the behavioral level of abstraction of the circuit design in order to speed up the design cycle by preparing valuable test suits for testing already at this level of abstraction. Fault simulation is heavily used by many test-oriented tasks. These are automatic test generation, fault diagnosis, test quality assessment, test suite compaction and other problems. If fault simulation is efficient, then accomplishment of all these tasks will gain in speed while keeping the quality.

Other topic of the current thesis is code coverage analysis at the register-transfer level as one of the simulation-based verification tasks. With the growth of digital devices' complexity a huge effort is required to verify the functionality of the device including finding the errors and localizing them. The exhaustive stimulus for today's designs is huge. It is mostly impossible to generate and exercise this stimulus due to the fact that it will take millions of years to execute [43]. Therefore, a coverage model is built to identify key stimulus values. These input values and their sequence allows sufficiently exercise design functionality. To measure the verification effort, different coverage metrics are employed to show whether the design is verified enough. Coverage metrics should be on one hand simple so that it would be quick to run the simulation using the metric and get an answer, on the other hand they should be sophisticated in order to thoroughly examine the design. Usage of the well-defined coverage metrics is widespread because they are integrated into simulation engines. The code coverage analysis process is fully automated.

Testing and fault simulation

The terms testing and verification differ. Testing does not refer to checking the correctness of design implementation, i.e., functional verification. Testing of electronic devices is a process of checking the manufacturing correctness [92]. Usually testing is done after each and every device is fabricated in silicon to ensure that the device is free of manufacturing defects that can appear during the manufacturing process. The types of physical defects depend on technology.

Usually during the testing phase ready-made stimuli is applied to the manufactured device and output responses are collected. Correct output values are also given with the set of input stimuli and compared to the real device outputs. If any mismatch in outputs has happened then a failure has occurred. The device can be sent to diagnosis for finding the defect. If it is possible then repair of the device is done. In the worst case the device is thrown away.

During preparation of the set of input stimuli, the device has not been manufactured yet, only the model of the device exists. Therefore, a model of defects is required to simulate actual physical defects. This model is called a *fault model*. It imitates the behavior of actual physical defects as close as possible. At the same time it is mathematically simple enough for fast computation. To determine the effectiveness of test vectors regarding detectability of faults, *fault simulation* is employed. During fault simulation test vectors are applied to implemented device one after other and for each test vector, detected faults are determined. One of the outputs of fault simulation is fault coverage, which is the ratio of faults detected by the test stimuli out of all possible faults in the device. The higher the fault coverage, the better the quality of test stimuli is. Obviously, the fault simulation process is computationally expensive and both memory and time consuming. Thus, it is very important to accelerate this process, which is used as a basic task in different test-oriented tasks mentioned above for improving their performance.

One possible approach is to use fault simulation at the RT (Register-Transfer) level that is to build RTL fault model, which is closely related to the gate-level fault model and to compute fault coverage already at this level of abstraction. The obtained set of test vectors can be reused at lower levels of abstraction, which makes the application of test-oriented tasks at lower levels of abstraction easier and faster. The current thesis is focused on fault simulation at the RTL using the RTL bit-coverage fault model, which is well correlating with the stuck-at fault model [29].

Verification and code coverage analysis

With the increase in complexity of modern integrated circuits, it has become imperative to address critical verification issues in the design cycle. The process of verifying correctness of designs takes roughly 70% of the design time [35]. For every designer the number of verification engineers can vary from 2 to 4 depending on the design complexity. The following aspects are the causes for the huge amount of resources spent on verification. First, design complexity increases. Second, historically more attention has been devoted to design process improvements that have produced significant progress in the design part, for example applying different automated tools for synthesis. However, verification process was not improved as much and has become a bottleneck.

For hardware verification, two types of methods are usually applied. These are formal methods and simulation-based verification. Formal methods use mathematical models to prove the correctness of the described model. Formal verification algorithmically and exhaustively explores all possible input values over time. However, formal verification can be performed only for limited design sizes due to the excessive time needed for proving the design correctness [66]. Thus, formal methods are applied only to some parts of the whole design implementation. Simulation-based verification relies on design simulation under a set of stimuli. Usually simulation-based verification assumes comparison of current implementation against the specification or against the implementation on another abstraction level [43]. In this thesis only simulation-based hardware verification is considered. Both, formal methods and simulation-based verification only detect the presence of an error in the design implementation, often providing description of the cases causing this incorrect behavior. Finding an error and fixing it is usually the manual work of verification engineer.

In order to verify the correctness of a design, different test cases are generated. Due to the fact that it is impossible to verify exhaustively all possible inputs and states of a design, the confidence level regarding the quality of the design must be quantified to control the verification effort. The fundamental question is: "How do I know if I have verified or simulated enough?" Verification coverage is a measure of confidence and it is expressed as a percentage of items verified out of all possible items. Different definitions of items give rise to different coverage measures or coverage metrics [43].

Various coverage metric classes exist such as code coverage, parameter coverage and functional coverage. Today, coverage-driven verification methodology is widespread, where verification progress is measured by achieving the coverage described by the coverage model. Coverage model consists of a set of various verification metrics and is built at the beginning of the design cycle. New methodology improves visibility into the verification process.

In this thesis, only code coverage would be used, which provides insight into how thoroughly the code of a design is exercised by a suite of simulations. The main disadvantage of code coverage metrics lies in the fact that they only measure the quality of the test case in stimulating the implementation and do not necessarily prove its correctness with respect to the specification. On the other hand, code coverage analysis is a well-defined, well-scalable procedure and, thus, applicable to large designs.

1.2 Problem formulation

Traditional design implementation is done using hardware description languages such as VHDL [90] or Verilog [89] for example. In this thesis, simulation-based verification issues and fault simulation at RTL using high-level decision diagrams (HLDDs) as the design representation model are addressed. Previous research works, including [84][85], have shown that

HLDDs are an efficient model for hardware design simulation and fault modeling since it provides a fast evaluation by graph traversal and easy identification of cause-effect relationships. Methods presented in this work are based on HLDD representation. This representation gives us opportunity to find some uncovered holes in verification by providing an alternative view compared to traditional methods.

Efficient fault simulation algorithms for combinational circuits are known for decades. However, sequential fault simulation which is frequently used in test and fault tolerance applications remains a very time-consuming task, in particular for larger circuits [23]. In order to contend the complexity, the research community has turned towards developing methods at higher design abstraction levels. In this work, a new approach, which is applicable directly at the RTL, is proposed. Three typical methods of fault simulation at the gate-level exist: parallel, deductive and concurrent fault simulation. Deductive fault simulation is faster than the parallel one and consumes less memory resources than the concurrent one. However, to the best of author's knowledge, it has never been used at higher level of abstraction than gate-level. In this work, deductive fault simulation algorithm is transformed for the register-transfer level and applied to HLDD-based designs, which allows accelerating the fault simulation.

Comprehensive verification coverage metrics help evaluating verification progress and managing verification effort [65]. In this thesis, a method and a tool for fast analysis of classical code coverage metrics, such as statement, branch and toggle coverage, are presented. All these metrics are built into a simulation tool working on HLDD design representation. Correspondingly, those classical coverage metrics are mapped to HLDD constructs. Also HLDDs can be seamlessly applied to observability coverage analysis. Commonly used code coverage metrics only point controllability of items in implemented design while ignoring their observability at outputs [4]. Taking into account the observability of a coverage item gives more information to the verification engineer. An observability coverage metric based on the toggle coverage metric is also built into the simulation tool based on HLDDs.

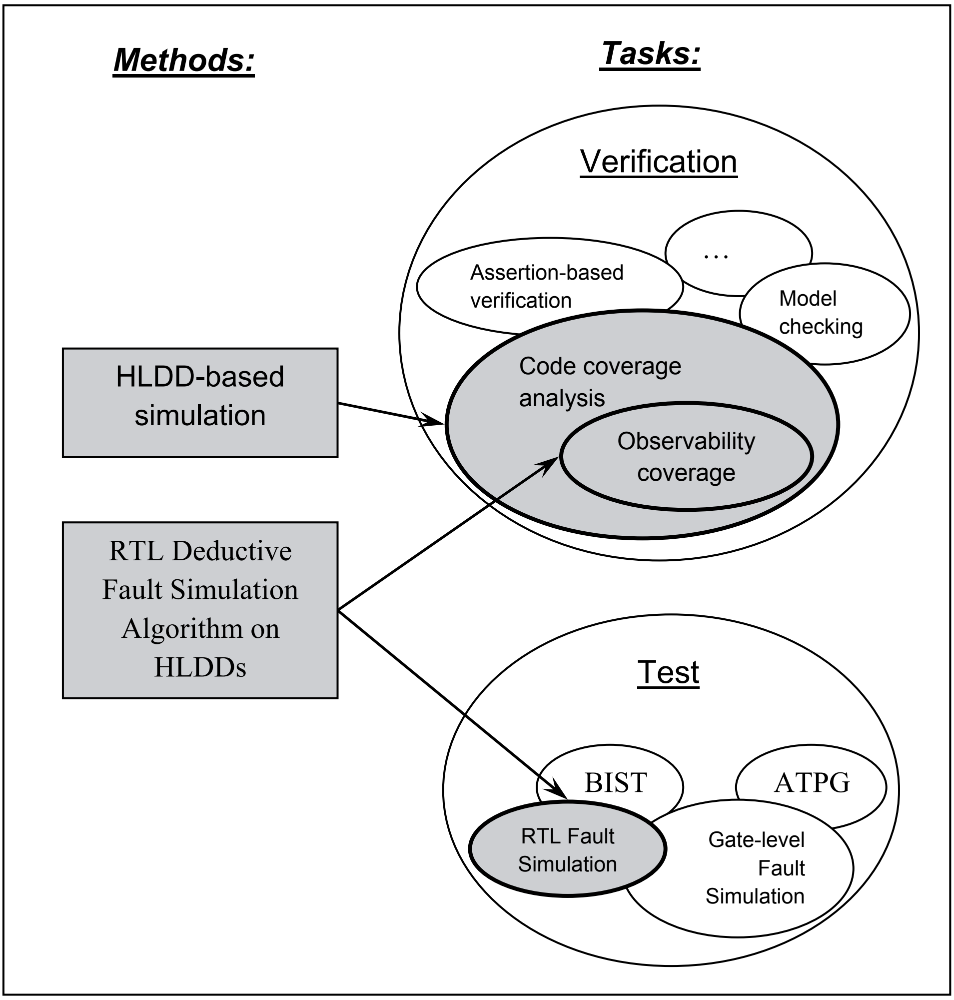


Figure1.1 Developed methods & tasks (grey background) in general verification and test steps

1.3 Contributions

The main contributions of this thesis are summarized below.

A new method for RTL fault simulation using High-Level Decision Diagrams was developed. This method was implemented using the deductive fault simulation algorithm. The initial deductive fault simulation algorithm, proposed by Armstrong for gate-level designs [3], was brought to RTL, where not only bits are taken into account when making decisions for fault lists propagation but word-level variables and arithmetic operations too. For fault simulation, RTL bit coverage fault model is used, which has proven to provide a good correspondence with gate-level structural faults [29].

An efficient data structure implementation to speed up set operations in deductive fault simulation algorithm at RTL was developed. The faults are coded with bits the way that it would be possible to make fast bitwise set operations with fault lists. Faulty data of the faults is stored in an array, which is closely related to faults IDs.

Fast HLDD-based simulation was extended to support code coverage analysis, such as node coverage, edge coverage, toggle coverage. A method of mapping traditional code coverage metrics to High-Level Decision Diagrams (HLDD) was described.

Manipulations on HLDDs to find the best representation for code coverage analysis were defined.

An observability coverage metric based on the bit-coverage fault model was presented, which takes into account not only the controllability of an internal point of the design, but also the observability at the outputs. The observability coverage metric gives more information to verification engineer and allows detection of testability problems at an early stage of a design cycle. This metric was implemented on HLDDs using toggle coverage as a basis for bugs insertions. The proposed deductive fault simulation algorithm on RTL is applied as the bugs propagation algorithm.

All above described methods were successfully integrated into a single tool, which is based on the HLDD simulation engine. General view of the tool methods and implemented tasks is depicted in Figure 1.1. Developed methods and tasks are colored in grey. In this figure one can see that code coverage analysis is one of the verification tasks, where observability coverage is part of a code coverage analysis. For implementation of these tasks in the thesis the HLDD-based simulation and the RTL deductive fault simulation algorithm were used. Also, the RTL deductive fault simulation algorithm was used for RTL fault simulation, which is one of the test tasks.

1.4 Thesis organization

This thesis consists of 6 chapters and 1 appendix.

Chapter 2 provides background information on related topics to this work. First, design representation by decision diagrams is presented including description of the High-Level Decision Diagram (HLDD) model. Second, introduction to fault simulation is described, where different fault models are shown and different levels of abstraction for circuit design are presented. Also, classic fault simulation algorithms for the gate-level are described. Third, introduction to verification is given, where necessary definitions are presented.

Chapter 3 starts with an overview of fault simulation. Then a new approach for fault simulation at the RTL using the HLDD design representation is presented in detail. The deductive fault simulation algorithm implemented at RTL is explained. An efficient implementation of algorithm's internal data structure for bitwise set operations is described. Then, results comparing RTL and gate-level fault simulations are presented.

Chapter 4 starts with an overview of code coverage for hardware designs. Code coverage metrics implemented on the HLDD simulation engine are presented. Then it is explained, how traditional code coverage metrics map to HLDDs and which representation of HLDDs better suits code coverage analysis. Experimental results for code coverage analysis are presented.

Chapter 5 starts with an overview of observability coverage analysis. Observability coverage metric presented in this thesis is defined. It is explained, how this observability coverage metric is built into the tool, implemented as the framework of this thesis. Experiments with measuring the observability coverage are shown and analysis of this metric is presented.

Chapter 6 concludes the thesis and discusses possible directions for future research.

The appendix presents research papers that form the basis for this thesis.

Chapter 2
BACKGROUND

This chapter presents background on the topics related to the current research. First, the High-Level Decision Diagram (HLDD) model is introduced. Register-transfer level fault simulation and HDL code coverage presented in this work take advantage of a design representation by High-Level Decision Diagrams developed at Tallinn University of Technology [61]. HLDDs themselves are not contributions of this thesis. However, most of the contributions in this research rely on these models. Second, digital test concepts are introduced. Classical fault simulation methods based on the stuck-at fault model are presented. The algorithm for RTL fault simulation proposed in this thesis is based on a classical deductive fault simulation algorithm. Third, verification concepts are introduced, where code coverage related topics are described in details.

2.1 Design representation by decision diagrams

The history of decision diagrams [61] based design representation goes back to seventies when the basic concept of *Binary Decision Diagrams* (BDD) was introduced. It was done by two authors, Raimund J. Ubar and Sheldon B. Akers, independently from each other in 1976 [81] and 1978 [2], respectively. In [81] decision diagrams were originally referred to as *alternative graphs*. During the following years, a number of works about using decision diagrams for test and simulation purposes were published, including [80] and [82]. However, it was not until the efficient Boolean manipulation method was presented by Randal E. Bryant in [11] when this type of representations became widely accepted by the research community.

Later, different special classes of binary decision diagrams have been proposed. They include Reduced Ordered BDDs (ROBDD) [11], multi-terminal

BDDs [17], edge-valued BDDs [42], binary moment diagrams [12], multi-valued decision diagrams [72], functional decision diagrams (FDD) [41] and others.

There is a number of word-level Decision Diagrams based models used for design representation at the Register-Transfer and higher levels. *High-Level Decision Diagrams* (HLDDs) were proposed by Raimund Ubar in [83] and further developed by Jaan Raik in [60] and [61] and Anton Karputkin and Mati Tombak in [38]. The other examples are multi-terminal DDs (MTDDs) [17] and Assignment DDs (ADDs) [16] are some of them. However, in MTDDs the non-terminal nodes hold Boolean variables only. The principal difference between HLDDs and ADDs lies in the fact that ADDs' edges are not labeled by activating values. They are rather used as connecting signals to represent structure. In HLDDs, the selection of a node activates a path through the diagram, which derives the needed value assignments for variables. Furthermore, ADD model includes four types of nodes (read, write, operator, assignment decision). In HLDD the nodes are divided into non-terminal (control) and terminal (data) ones.

The following two subsections provide an introduction to BDD and to HLDD models correspondingly.

2.1.1 Binary decision diagrams

This subsection presents the traditional BDDs, which are commonly used for representing Boolean functions. The general concept of BDD is explained and a widely used special class of BDDs, Reduced Ordered BDDs (ROBDD), is introduced.

A BDD is defined [61] as a directed acyclic graph with two terminal nodes, which are the *0-terminal* and *1-terminal* nodes. Each non-terminal node is labeled by an input variable of the Boolean function, and has two outgoing edges, called *0-edge* and *1-edge*.

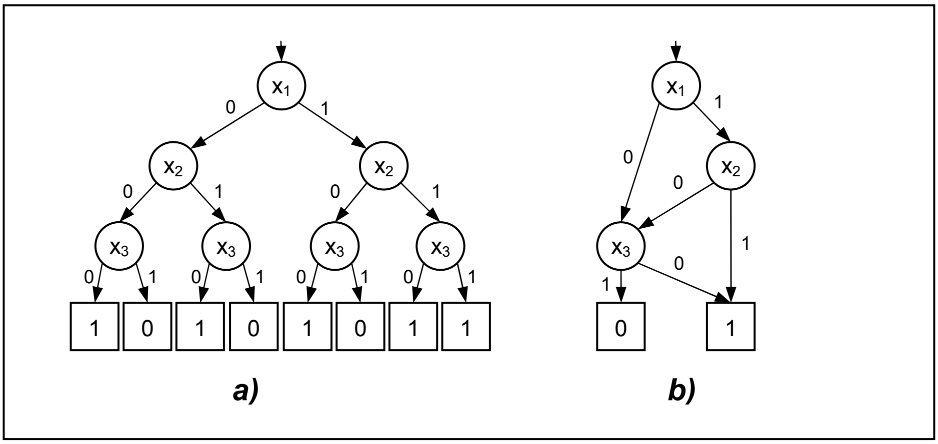


Figure 2.1 BDD representations for a Boolean expression $(x_1 \cdot x_2) \vee \neg x_3$;

a) full tree BDD; b) ordered BDD

Ordered BDD (OBDD) is a BDD, where the input variables appear in a fixed order on all the paths of the graph and no variable appears more than once in a path. Figure 2.1 shows an example of a full tree BDD (a) and ordered BDD (b) corresponding to a Boolean function $f = (x_1 \cdot x_2) \vee \neg x_3$. In the binary tree, 0- and 1-terminal nodes represent logic values 0 and 1, and each node represents the *Shannon's expansion* of the Boolean function:

$$f = \left(\overline{x_i}.f_0\right) \vee \left(x_i.f_1\right),$$

where i is the index of the variable and f_0 and f_1 are the functions of the nodes pointed to by 0- and 1-edges, respectively.

Reduced Ordered BDD (ROBDD) is created by applying the following reduction rules to OBDD [11]:

Reduction rule1: Eliminate all the redundant nodes where both edges point to the same node (Figure 2.2a).

Reduction rule2: Share all the equivalent sub-graphs (Figure 2.2b).

An important feature of ROBDDs is that they provide canonical forms for Boolean functions. This allows us to check the equivalence of two Boolean functions by merely checking isomorphism of their ROBDDs. This is a widely used technique in formal verification.

High-Level Decision Diagrams are derived from BDDs and used at a higher abstraction level of design representation, namely at word-level rather than Boolean-level. Below, explanation of this model is provided.

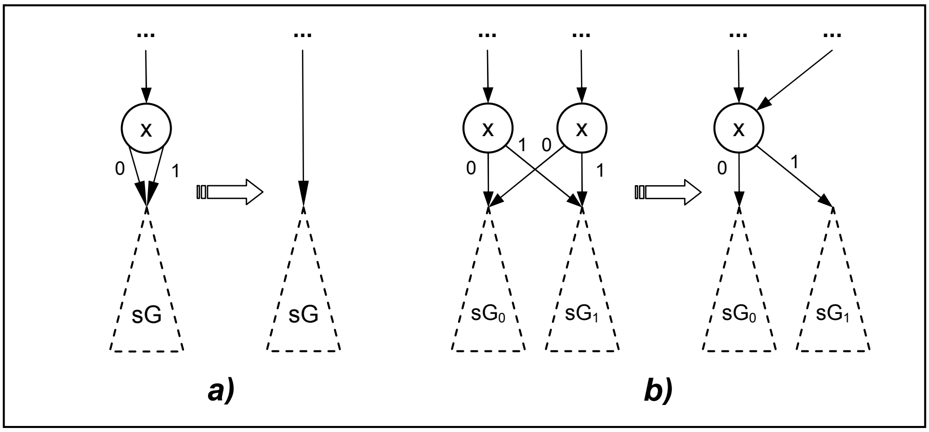


Figure 2.2 BDD reduction rules: a) reduction rule 1: eliminate all the redundant nodes, where both edges point to the same node; b) reduction rule 2: share all the equivalent sub-graphs

2.1.2 High-level decision diagrams

In this subsection, description of the HLDD model is provided. High-Level Decision Diagrams can be viewed as a generalization of BDDs. HLDDs can be used for representing different abstraction levels from RTL to behavioral.

2.1.2.1 HLDD model definition

Below the High-Level Decision Diagram (HLDD) data structure is defined based on [38]. Consider a digital system (Z, F) as a network of subsystems or components, where Z is the set of variables (Boolean, Boolean vectors or integers), which represent connections between components, primary inputs and primary outputs of the network, and F is a set of discrete functions. Let $Z = X \cup Y$, where X is the set of function arguments and Y is the set of function values, where $Q = X \cap Y$ is the set of state variables. $D(z)$ denotes the finite set of all possible values for $z \in Z$ and $D(Z')$ is the set of all possible vectors for all $Z' \subseteq Z$. Obviously, if $Z' = \{z_1, \ldots, z_n\}$ then $D(Z') = D(z_1) \times \ldots \times D(z_n)$. Let F be a set of discrete functions: $y_k = f_k(X_k)$, where $y_k \in Y$, $f_k \in F$, and $X_k \subseteq X$ (k iterates over all elements in F).

Definition 1. *The high-level decision diagram representing a function*

$f_k : D(X_k) \rightarrow D(y_k)$ *is a directed acyclic graph* $G = (V, E)$ *with one root node and a set of terminal nodes where:*

- *Each non-terminal node is labeled by some input or control variable $x \in X$.*[1] *We shall denote the variable of node v by x_v.*
- *Each terminal node w is labeled by some function $g_w : D(X_w) \to D(y_k)$ (possible a constant or single variable), where $X_w \subseteq X_k$.*
- *Each edge e from node v to u is labeled by a non-empty set of constants $C \in D(x_v)$. We denote such edge by (v, u, C).*
- *Each two edges $e_1 = (v, u_1, C_1)$ and $e_2 = (v, u_2, C_2)$ going from the same source node are labeled by different constants $C_1 \cap C_2 = \varnothing$.*
- *If the node v is labeled by x_v then the number of edges going from this node is $|D(x_v)|$.*

In simple words, HLDD is a data structure similar to BDD, but with many edges originating from a particular node and a number of functions at the end, instead of constants 0 and 1. We shall denote the set of terminal nodes by V^T, the set of non-terminal nodes by V^N and the set of all successors of the v by $\Gamma(v)$. For non-terminal nodes $v \in V^N$ an onto function exists between the values $c \in D(x_v)$ of labels x_v and the successors $v^c \in \Gamma(v)$ of v. By v^c we denote the successor of v for the value $x_v = c$.

The edge e, which connects nodes v and v^c, is called *activated* iff there exists an assignment $x_v = c$. Activated edges, which connect v_i and v_j, make up an *activated path* $l(v_i, v_j) \subseteq V$. An activated path $l(v_0, v^T)$ from the root node v_0 to a terminal node v^T is called the *main activated path* and v^T itself is referred to as the *activated terminal node*.

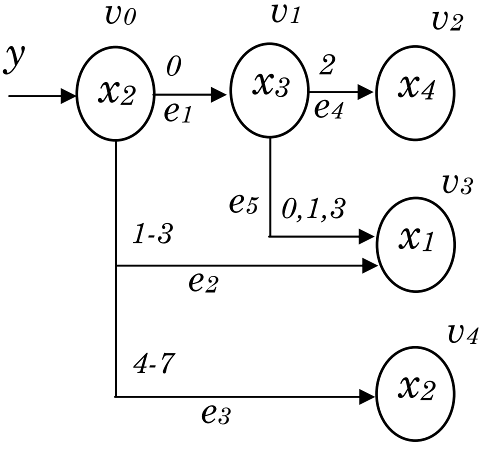
Remark 1. *Every BDD is an HLDD as well, with two terminal vertices labeled by constant functions 0 and 1, and $D(x) = \{0, 1\}$ for every variable x.*

Without loss of generality we assume further that each variable has at least two values, i.e. $\forall z \in Z$, $D(z) > 1$. Let D_i designate a subset of $D(x_v)$ labeling node v, such that assignments from it will activate its successor node v_i. $D(x_v)$ is partitioned into non-intersecting sets $D_1, \ldots, D_m$, where $m = |\Gamma(v)|$. More formally,

$$\bigcup_{i=1}^{m} D_i = D(x_v) \wedge \forall i, j, i \neq j \to D_i \cap D_j = \varnothing.$$

---

[1] Some of these variables are in fact atomic predicates but are treated as Boolean variables as there is no difference between a variable and a predicate in current context.



$G_y = (V,E)$;
$V = \{v_0, v_1, v_2, v_3, v_4\}$;
$E = \{e_1, e_2, e_3, e_4, e_5\}, e_1=(v_0, v_1, 0)$,
$e_2=(v_0, v_3, \{1\text{-}3\}), e_3=(v_0, v_4, \{4\text{-}7\})$,
$e_4=(v_1, v_2, 2),\ e_5=(v_1, v_3, \{0,1,3\})$;
$X=\{x_1=x_{v_3}, x_2=x_{v_0}=x_{v_4}, x_3=x_{v_1}, x_4=x_{v_2}\}$;
$D_1(x_{v0})=\{0\}, D_2(x_{v0})=\{1,2,3\}$,
$D_3(x_{v0})=\{4,5,6,7\}, D_1(x_{v1})=\{2\}$,
$D_2(x_{v1})=\{0,1,3\}$.

Figure 2.3 Graphical representation of a HLDD for function $y=f(x_1, x_2, x_3, x_4)$

In other words, with every value assignment to variable x_v one and only one successor node will be activated. In the following graphical examples the edges are merged according to their successor node v_i and labeled by the corresponding domain partition D_i.

Figure 2.3 depicts a HLDD G_y representing a discrete function $y=f(x_1,x_2,x_3,x_4)$. The diagram contains five nodes v_0, …, v_4. The root node v_0 is labeled by variable x_2 which is an integer with a range from 0 to 7. The node has three outgoing edges entering the nodes v_1, v_3 and v_4. The node v_1 is labeled by x_3 with a range from 0 to 3. It has two outgoing edges e_4 and e_5 entering terminal nodes v_2 and v_3, respectively. The edge e_4 is activated by x_3=2, while the edge e_5 is activated by x_3 having a value 0, 1 or 3.

Definition 2. A HLDD $G_k = (V, E)$ represents a function $y_k = f_k(X_k)$, iff for each assignment of variables in X_k, a main activated path exists, so that $y_k = z(v^T)$ is valid.

Each function $f_k \in F$ in the system (Z, F) is represented by a decision diagram G_k. Depending on the class of digital system (or level of its representation), we may have various DDs, in which nodes have different interpretations and relationships to the system structure. In RT level we usually decompose digital systems into control and data paths parts. State and output variables of the control part serve as addresses or control words, and the variables in the data paths part serve as data words. The functions of RTL components in the data paths are described by arithmetic operations on the word-level data variables. Non-terminal nodes in HLDDs correspond to control paths and they are labeled by control variables or logical conditions, whereas terminal nodes correspond to data paths, and they are labeled by the data or functions on data.

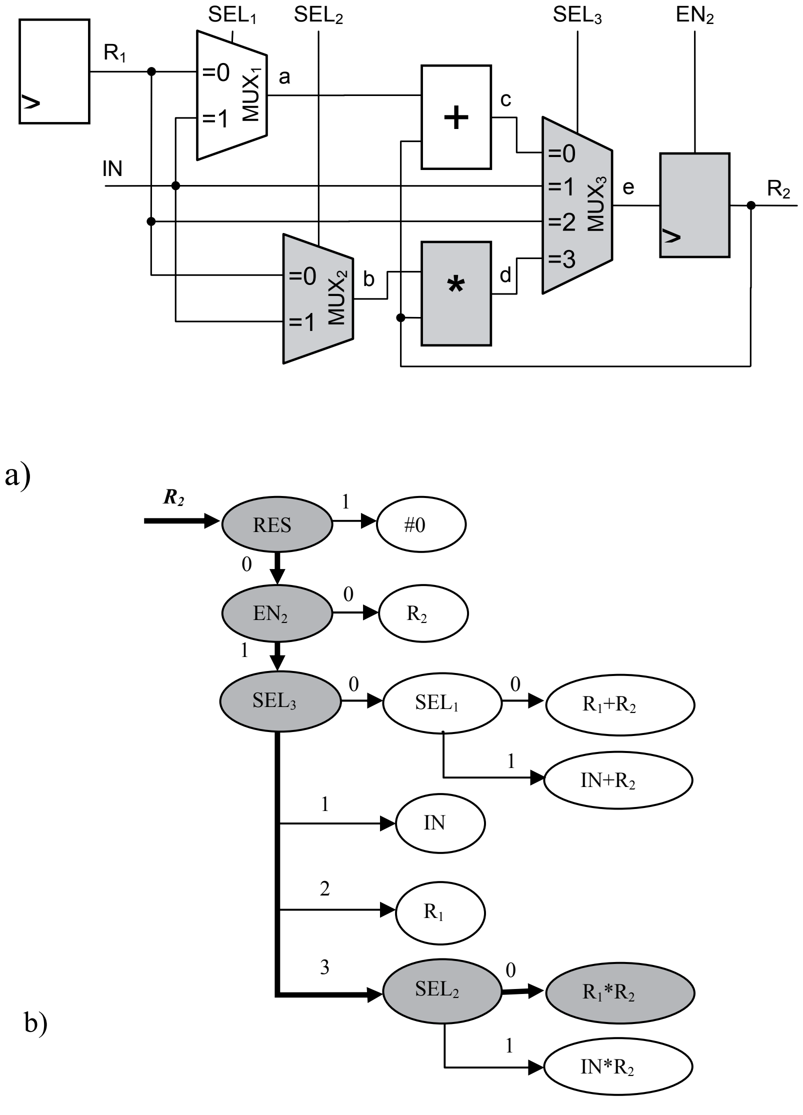


Figure 2.4 a) RTL schematic and b) its HLDD-based representation

2.1.2.2 Modeling RTL designs by HLDDs

In Fig. 2.4a the datapath is depicted and its corresponding HLDD representation shown in Fig. 2.4b. Here, *R1* and *R2* are registers (*R2* is also a primary output), *MUX1*, *MUX2* and *MUX3* are multiplexers, + and * denote addition and multiplication operations, *IN* is an input bus, *SEL1*, *SEL2*, *SEL3* represent multiplexer address signals, *EN2* serves as the signal for register *R2*, and *a*, *b*, *c*, *d* and *e* denote internal buses, respectively. In the HLDD, the control variables *RES*, *SEL1*, *SEL2*, *SEL3* and *EN2* are labeling internal decision nodes of the HLDD. The terminal nodes are labeled by a constant #0 (reset of *R2*), by word-level variables *R1* and *R2* (data transfers to *R2*), and by expressions related to the data manipulation operations of the network.

Consider, simulating HLDD with some values assigned to the variables. Let the value of *SEL2* be 0, the value of *SEL3* be 3, the value of *EN2* be 1 and the value of *RES* be 0 in the current simulation run. By bold lines and grey nodes, a main activated path in the HLDD is shown from *RES* to *R1*R2*, which corresponds to the pattern *RES=0*, *EN2*=1, *SEL3*=3, and *SEL2*=0. The activated part of the network at this pattern is denoted by grey boxes.

The main advantage and motivation of using HLDDs compared to the netlists of primitive functions is the increased efficiency of simulation and diagnostic modeling because of the direct and compact representation of cause-effect relationships. For example, instead of simulating the control word $SEL1=0, SEL_2=0, SEL3=3, EN2 = 1, RES=0$ by computing the functions $a = R1, b = R1, c = a + R2, d = b * R2, e = d$, and $R2 = e$, we only need to trace the nodes *RES, EN2, SEL3* and *SEL2* on the HLDD and compute a single operation $R2 = R1 * R2$. In case of detecting an error in *R2* the possible causes can be defined immediately along the simulated path through *RES*, *EN2*, *SEL3* and *SEL2* without complex diagnostic analysis inside the corresponding RTL netlist. The activated path provides the fault candidates, i.e. variables that are suspected to contain faults causing the error at *R2* during current simulation run. Further reasoning should be based on analyzing sources of these signals.

2.1.2.3 Basic simulation on HLDDs

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 $v_i \in V^N$ according to the value of the variable $x_v = c$ certain output edge $e = (v_i,v_j,C)$, $v_j \in \Gamma(v_i)$ will be chosen, which enters into its corresponding successor node v_j. 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 HLDD will be the value of the function result of constant labeling the terminal node of the main activated path.

Algorithm in figure 2.5 presents the HLDD based simulation engine for RTL, behavioral and mixed HDL description styles and has been proposed in [85].

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

Starting from the root node the outgoing successor is found according to the value of variable x_0 labeling a node v_0. While the terminal node is not reached value of variable $x_{currrent}$ activates edge e_{active} to the next successor node v_{next}. In the RTL style, the algorithm takes the previous time step value of variable $x_{current}$ labeling a node $v_{current}$ if $x_{current}$ represents a clocked variable in the corresponding HDL. Otherwise, the present value of $x_{current}$ will be used. In the case of behavioral HDL coding style HLDDs are generated and ranked in a specific order to ensure causality. For variables $x_{current}$ labeling HLDD nodes the

```
for each diagram G in the model
    $v_{current} = v_0$
    Let $x_{current}$ be the variable labeling $v_{current}$
    while $v_{current}$ is not a terminal node
        if $x_{current}$ is clocked or its DD is ranked after G then
            Value = previous time-step value of $x_{current}$
        else
            Value = present time-step value of $x_{current}$
        end if
        $v_{next} \in \Gamma(v_{current})$, where $e_{active} =( v_{current}, v_{next}) \wedge c$ = Value
        $v_{current} = v_{next}$
    end while
    if $x_{current}$ is a function then calculate a function;
    assign $x_{current}$ to the DD variable $x_G$
end for
```

Figure 2.5 Algorithm1. Simulation engine on HLDDs

previous time step value is used if the HLDD diagram calculating $x_{current}$ is ranked after current decision diagram. Otherwise, the present time step value will be used. Reaching the terminal node value of terminal variable is assigned to the graph variable y calculating first resulting value of a function if terminal node labels a function.

In Figure 2.6 example of simulation on the high-level decision diagram presented in Figure 2.3 is shown. Let assume that variable x_2 is equal to 0, variable $x_3 = 1$ and variable $x_1 = 3$. A path (marked by bold arrows) is activated from node v_0 (the root node) to a terminal node v_3 labeled by x_1. Thus, $y=x_1=3$. Note, that this type of simulation is event-driven since we have to simulate only those nodes that are traversed by the main activated path (marked by grey color in Figure 2.6).

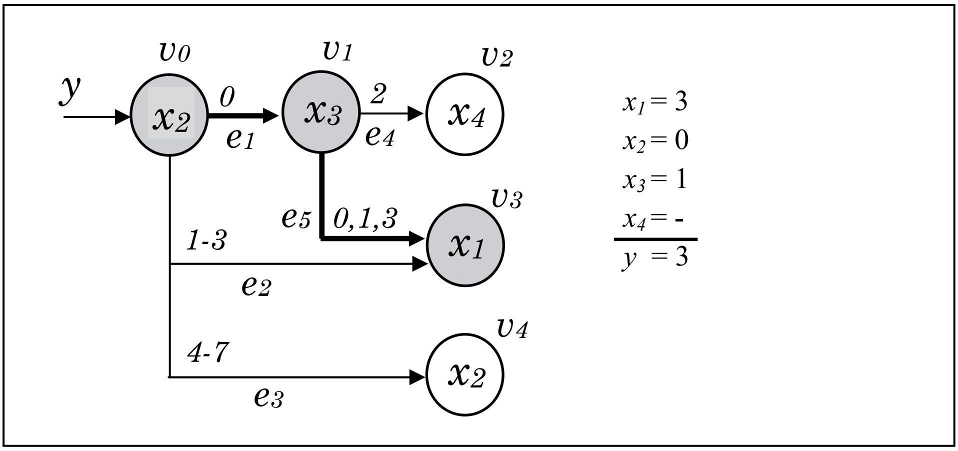


Figure 2.6 Example of design simulation on HLDD

2.2 Fault Simulation

In this subsection introduction to fault simulation is done. First, role of testing is described. Then, different fault models are presented, which mathematically describe defects of the circuit. Afterwards, the most widespread algorithms for fault simulation are presented. Also, applications of fault simulation are defined.

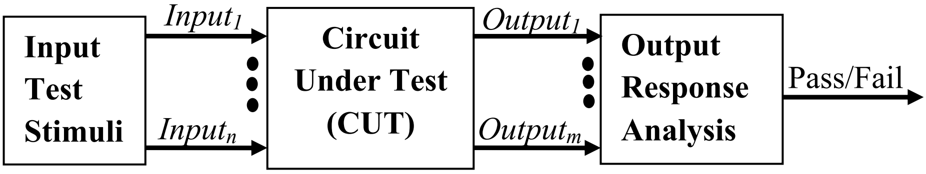


Figure 2.7 Basic testing approach [92]

2.2.1 The role of Testing

Reliability and testing techniques have become of increasing interest to different applications, including consumer electronics. A key requirement for obtaining reliable electronic systems is the ability to determine that the system is error-free [9]. A test is a procedure which allows one to distinguish between good and bad parts. In this work we concentrate only on digital testing, i.e. input and output signals of the circuit can only take on the value ‘logic 0’ or ‘logic 1’. Testing a circuit prior to its manufacturing is known as design verification. The question is not *whether one should verify* but *how well to verify* in order to have confidence that the device will comply with its specification [56]. This topic would be discussed in the next subchapter. The stress of this subchapter is testing a circuit after it is manufactured. Even though a circuit is designed error-free, manufactured circuit may not function correctly. Since the manufacturing process is not perfect, some defects such as short-circuits, open-circuits, open interconnections, pin shorts, etc., may be introduced. Davis [20] points out that the cost of detecting a faulty component increases ten times at each step between prepackage component test and system warranty repair. Therefore, testing has become a very important aspect of any VLSI manufacturing system.

Testing typically consists of applying a set of test stimuli to the inputs of the *circuit under test* (CUT) while analyzing the output responses, as illustrated in figure 2.7. 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. Testing is performed at various stages in the lifecycle of a VLSI device [92]. In addition, a diagnosis of the failing circuit can be performed in order to identify the location and type of the defect.

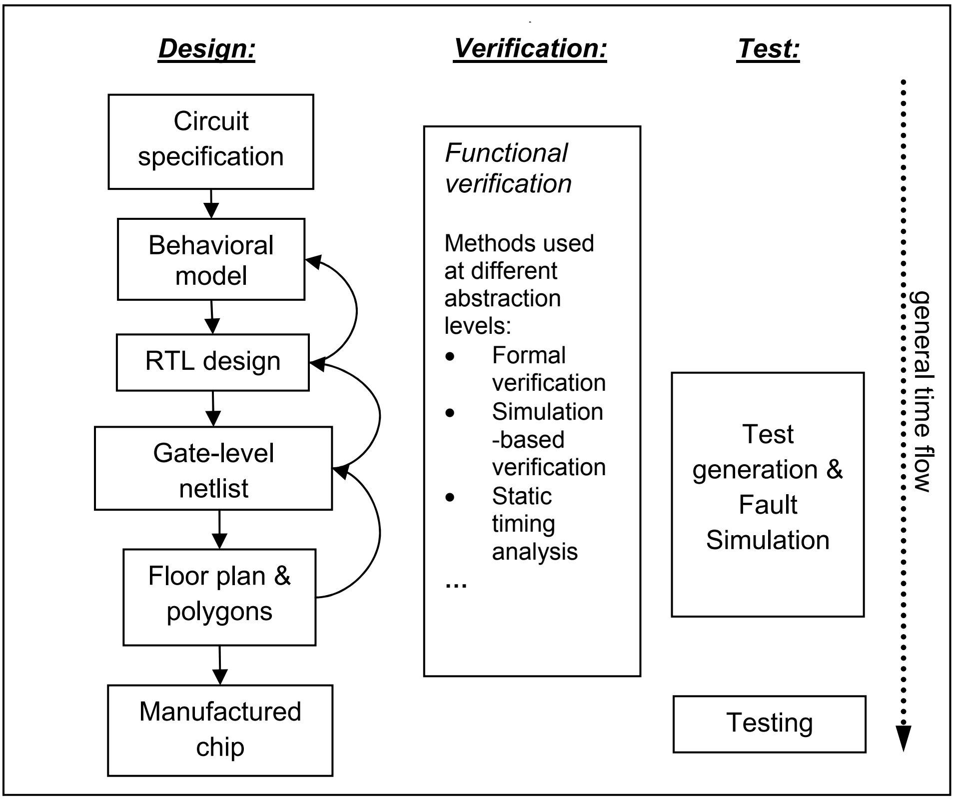


Figure 2.8 Top-Down design & test methodology

2.2.2 Top-Down Design and Test Methodology

A VLSI design can be described at different levels of abstraction. The design process is essentially a process transforming a higher level description of design to a lower level description either with a help of synthesis tools or by hand. One possible top-down design and test methodology is described in Figure 2.8. Starting from a circuit specification, a behavioral model of a circuit is developed in VHDL, Verilog, C or other language, program and simulated to determine if it is functionally equivalent to the specification. At the behavioral level functionality is modeled without regard to the hardware structure, electrical signals and detailed timing. Such models are useful as a proof-of-concept. The design is then described at the *register-transfer level* (RTL), which contains more structural information in terms of the sequential and combinational logic functions to be performed in the data paths and control circuits. The RTL modules are validated as stand-alone components before integrating them into a system. The RTL description must be verified with respect to the functionality of the behavioral description before proceeding with logic synthesis to the gate level. Logic synthesis transforms the RTL description into an optimized technology-specific hardware description, generally in the form of a gate-level netlist (connectivity description of Boolean gates). The gate level structure of the design stabilizes only after the synthesized circuit has been verified through logic simulation. Once the design is verified, gate-level SSF (Single Stuck-Fault) models are used for test generation and fault simulation using the technology-specific (gate-level) netlist. In addition, the gate-level netlist serves as a common database for various post-synthesis steps such as timing simulation, placement, routing, static timing analysis, etc., until a prototype is fabricated.

2.2.3 Fault modeling

Below, some definitions of basic terms are presented.

Definition 2.1. *A **defect** in an electronic system is the unintended difference between the implemented hardware and its intended design* [13].

Defects occur either during manufacturing process or during the use of the device. Some typical defects are [13]:

- Process defects (imperfection of manufacturing process: missing of contact windows, parasitic transistors, oxide break-down, etc.)
- Materials defects (surface impurities, bulk defects, etc.)
- Age defects (electromigration, dielectric breakdown, etc.)
- Package defects.

Definition 2.2. *A wrong output signal produced by a defective system is called an **error**. An error is an "effect" whose cause is some "defect"*[13].

Definition 2.3. *A representation of a "defect" at the abstracted function level is called a **fault**.[13]*

Definition 2.4. ***Test vector** is an input pattern applied to the circuit under test (CUT), and its responses are compared to the known good responses of a fault-free circuit.*

In order to completely test a circuit, a sequence of test vectors is required; however, it is difficult to know how many test vectors are needed and the order of test vectors in a sequence to guarantee a satisfactory reject rate. The effectiveness of the test sets is usually measured by the ***fault coverage*** and is defined as:

$$fault\ coverage = \frac{number\ of\ detected\ faults}{total\ number\ of\ faults}$$

The set of test vectors is complete if its fault coverage is 100%. This level of fault coverage is desirable but rarely attainable in most practical circuits. Moreover, 100% fault coverage does not guarantee that the circuit is fault-free. The fault coverage is calculated using a ***fault simulator***.

Definition 2.5 ***Fault simulator*** *is a logic simulator in which faults are injected at the appropriate nets of the circuit, usually one at a time. The responses of the circuit to test vectors are compared with the good responses of the circuit. The fault is considered detected if at least one of the test pattern has a response different from the good circuit response.*

Fault simulator typically works with a specific fault model. Because of the diversity of physical defects, it is impractical to work with real defects. ***Fault models*** were introduced to offer a simplified mathematical description of the erroneous behavior. Although most of the fault models do not provide exact description of the erroneous behavior of the circuit, they are very useful for generating and evaluating the quality of tests. A good fault model should satisfy two criteria: it should accurately reflect the behavior of defects, and it should be computationally efficient in terms of fault simulation and test pattern generation. Many fault models have been proposed, e.g. single stuck-at faults, transition faults, gate-delay faults, bridging faults [13], but, unfortunately, no single fault model accurately reflects the behavior of all possible defects that can occur. As a result, a combination of different fault models is often used.

Generally any fault model can be divided into two classes: single-fault model and multiple-fault model. For single-fault model it is assumed that only single fault can occur in the circuit at the time. A multiple-fault represents a condition caused by simultaneous presence of a group of single faults. For a given fault model let m be different type of faults that can occur at potential fault site and n be possible fault sites. Then for single-fault model number of possible faults equal to $m*n$ and for multiple-fault model number of possible faults equal to $(m+1)^n$ -1. Because latter amount of faults is too large even for small values of m and n, the single-fault assumption is usually considered in practice. Fortunately, tests for single stuck-at faults are known to cover a very high percentage (greater than 99.6%) of multiple stuck-at faults when the circuit is large and has several outputs [13].

Modeling of faults is closely related to the modeling of the circuit. Different levels of abstraction are used in the top-down design methodology. The *behavioral level* has fewer implementation details and fault models at this level may have no obvious correlation to manufacturing defects. Behavioral level fault models play greater role in the simulation-based design verification, than in testing. The *RTL level* faults usually imitate gate level stuck-at faults at the higher level of abstraction. Commonly this is not one to one correspondence. The *logic level* or *gate level* consists of a netlist of gates and the stuck-at faults at this level are the most popular fault models in digital testing. Other fault models at this level are bridging faults and delay faults. *Transistor* and other *lower levels* include technology-dependent faults, e.g. stuck-open faults.

Behavioral faults: Usually, the behavior of electronic system is described in a programming language, e.g. C, or in some hardware description language such as VHDL, Verilog. At the behavioral level the variables are not necessarily electrical, but correspond to a specific application domain. Behavioral faults refer to incorrect execution of the language constructs used in the program. Examples of behavioral faults are assertion faults, branch faults, and instruction faults. At the behavioral level, different coverage metrics (statement coverage, branch coverage, toggle coverage) can be used to measure efficiency of the test, although these do not conform to any specific fault model [13].

RTL faults: Straightforward extension to the stuck-at-fault model is to replace the concept of a signal line that is stuck with that of an internal RTL variable being stuck. Further it is possible to differentiate between *data faults* and *control faults*, depending on the type of the stuck variable [1].

- Typical *data faults* are register or memory bits that are stuck. Data faults are stuck-at-0 and stuck-at-1 faults: when the fault is present, the affected object (a signal or a variable representing memory element) loads the correct value, except for one bit that remains stuck to 0 or 1.
- *Control faults* are defined on variables that control conditional operations. These are stuck-at-then and stuck-at-else faults for *if* statements and selection faults for *case* statements. It is allowed having a stuck fault to the result of any expression that is part of a condition or the entire condition itself.

Logical faults represent the effect of physical faults on the behavior of the modeled system. Many physical faults can be modeled by the same logical fault. Logical faults affect the state of logical signals. Normally, the state is modeled as {0, 1, X, Z}, and a fault transform the correct value to any other value. Several types of faults can be modeled at this level. However, the term *logical fault* often implies stuck-at faults.

Lower level faults are not part of this work therefore their description is out of the scope of the thesis. As the Single Stuck-Fault (SSF) model is used as the basis for RTL level faults model, more detailed description of the SSF model is provided below.

SSF (Single Stuck-at Fault) model is a logical fault model that is most commonly used in digital testing [35]. A stuck-at fault is assumed to affect only the interconnections between gates. A single stuck-at fault (stuck-at-0 or stuck-at-1) represents a line in the circuit that is fixed to logic value 0 or 1, irrespective to the correct logic output of the gate driving it.

The SSF model is widely used, its usefulness results from the following attributes [1]:

- It represents many physical faults.
- It is independent on technology, as the concept of a signal line being stuck at a logic value can be applied to any structural model.
- Experience has shown that SSFs detect many non-classical faults as well.
- Compared to other fault models, the number of SSFs in the circuit is small. Moreover, the number of faults to be explicitly analyzed can be reduced by fault-collapsing techniques.
- SSFs can be used to model other types of faults.
- High SSFs coverage provides a high multiple stuck-at faults coverage.

In this work, two fault models: the SSF model and the RTL bit coverage fault model [29] - are used for fault simulations. A detailed description of fault simulation methods and experiments will be provided in the next chapter. Comparative table of the properties of the fault models is given below. It is important to mention that there is no one to one correspondence between SSF faults and RTL faults because for a given RTL description several gate-level implementations exist.

Table 1 Properties of fault models

Gate level SSF model	RTL bit coverage fault model [29]
Boolean components are assumed to be fault-free	Language operators are assumed to be fault-free
Signal lines contain faults: • Stuck-at-0 fault when the logic level is fixed at value 0 • Stuck-at-1 fault when the logic level is fixed at value 1	Variables contain faults: • Stuck-at-0 fault when the bit is fixed to value 0 • Stuck-at-1 fault when the bit is fixed to value 1
According to SSF assumption, only one fault is applied at a time when the test set is evaluated	Single fault assumption: only one fault is applied at a time when the test set is evaluated

2.2.4 Fault simulation

Simulation is the process of predicting the behavior of a circuit design before it is manufactured. For digital circuits, simulation serves dual purposes. First, during the design stage, logic (fault-free) simulation helps the designer verify that the design works according to the functional specifications. Verification with the help of simulation techniques will be part of the next subchapter. Second, during test development, fault simulation is applied to simulate faulty circuits. Definition of fault simulator is given in the previous subchapter. To summarize, a fault simulator must classify given modeled faults in a circuit as detected or undetected with given test vectors. Fault simulator determines the efficiency of test vectors in detecting the modeled faults of interest.

The section below presents the key fault simulation techniques based on the single stuck-at fault model. These techniques can be reused with modifications on other fault models as well as at other abstraction levels. Since these algorithms were developed for SSF model the explanations would be given on this model. Further, RTL fault simulation features would be shown.

2.2.4.1 Serial Fault Simulation

Serial fault simulation is the simplest fault simulation technique. First, the circuit is simulated in a fault-free mode for all test vectors and fault-free output values are stored. Then the fault simulator simulates faults one by one. For each fault, fault injection is performed, which modifies original circuit to mimic the circuit behavior in the presence of the fault. As simulation proceeds, the output values of fault simulation are compared with stored fault-free output values for all test vectors. All faults are simulated serially in this way. This kind of simulation is very time consuming. To improve the fault simulation performance, fault dropping is used. Halting the simulation as soon as comparison indicates detection of the target fault is called fault dropping.

The major advantage of serial fault simulation is its ease of implementation. It can simulate wide range of fault models, as long as the fault effect can be properly injected into the circuit. The major disadvantage of serial fault simulation is its low performance. There exists more intelligent algorithm to reduce the effort of fault simulation. These general algorithms are parallel [67], deductive [3] and concurrent [87] fault simulation techniques. They differ from the serial method in two fundamental aspects [1]:

- They determine the behavior of the circuit in the presence of faults without explicitly changing the model of the circuit.
- They are capable of simultaneously simulating a set of faults.

Two of these algorithms will be described in the following sections.

2.2.4.2 Parallel Fault Simulation

Parallel fault simulation benefits on the bitwise parallelism of logical operations in a digital computer in order to reduce computational time. For example, for a 32-bit machine word, logic operations such as AND, OR, XOR etc. can be performed on all 32 bits at once. The idea of parallel fault simulation belongs to [67]. It is assumed that signals work on logic 0 and/or 1. It is possible to expand 2-bit signal logic to wider coding such as X and Z values, but this method would require special encoding.

In parallel fault simulation $w-1$ faults would process in one pass, where w is the machine word size. One bit of w is used for fault-free value and other $w-1$ bits are values of signals in the faulty circuits for $w-1$ faults. Each bit of a word represents a signal value in a different circuit. The injection of a fault is done by changing the value of a bit corresponding to a signal value in a circuit. If the number of simulated faults is more than the machine word size, then more than one pass of fault simulation is required to simulate all faults. The technique of fault dropping can be applied in parallel fault simulator as well as in serial fault simulator to reduce computational time, however the simulation pass would

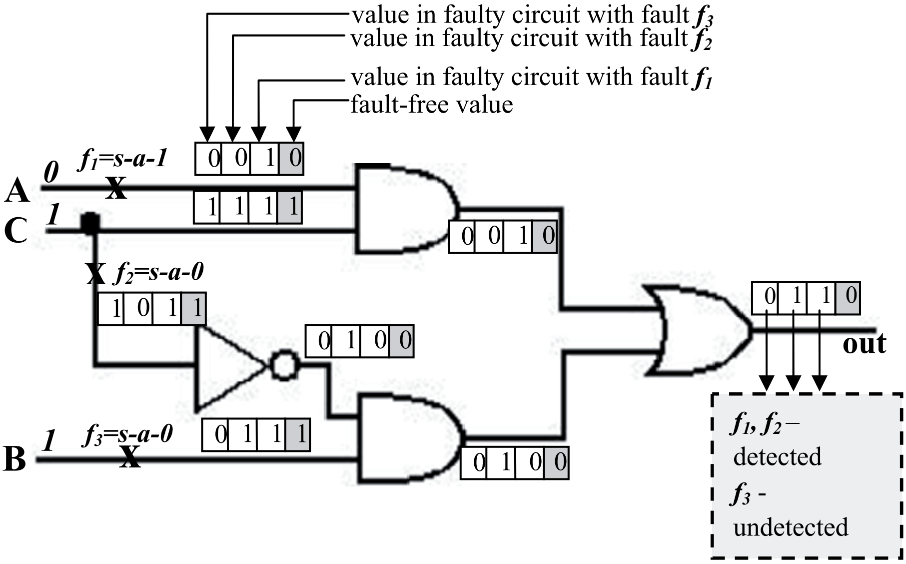


Figure 2.9 Parallel fault simulation

terminate only when all faults of the pass would be detected. Therefore, serial fault simulation gains more by fault dropping.

Consider an example of a multiplexer at the gate level (Figure 2.9). In this example, test vector *ACB=011* is applied to the inputs. Three faults f_1 (s-a-1), f_2 (s-a-0), f_3 (s-a-0) are injected into the circuit. Therefore, a packet of 4 bits is required for logic operations of parallel simulation, where 3 bits are used to encode values of the signals in faulty circuits and 1 bit is used for the fault-free circuit. To simulate in parallel, the signal of each line in a circuit is expressed as a word where 4 left-most bits are useful for simulation in this example. The bit, which represents faulty value of a signal in faulty circuit, is changed to the stuck-at value, other bits remain the same. During the simulation, the effect of the faulty value propagates towards the output. For example, fault f_1 is present only in the first circuit, thus second bit of signal *A* is changed to 1. Performing logic operations we obtain output *out=0110*. Faults f_1 and f_2 are detected because bits in the output signal representing faulty circuits with these faults differ from fault-free bit value; fault f_3 is not detected because its bit value is equal to the fault-free bit value.

Parallel fault simulation is approximately *w* times faster in comparison with serial fault simulation. However, it has limitations as well. It becomes impractical for multi-valued logic. And as it was mentioned above, the fault dropping technique is not as effective as for serial fault simulation.

2.2.4.3 Deductive Fault Simulation

In deductive fault simulation [3] only the fault-free circuit is simulated and the behavior of all faulty circuits is based on logic reasoning. All signal values of faulty circuits are deduced from fault-free circuit signal values and the structure of a circuit. All deductions are carried out simultaneously and only one fault-free simulation is performed. Thus deductive fault simulation can be very fast. It is possible theoretically to deduce all signal values, however practically this depends on the size of the available memory. Fault effects are represented by the fault list. A ***fault list*** L_i is associated with every signal i. L_i is a set of faults that cause the value of signal i to differ from its fault-free value. If signal i is a primary output then the fault list associated with i is the set of faults detected at this output. Thus the aim of the deductive fault simulator is eventually to construct a set of detected faults by uniting fault lists of all primary outputs. Based on logic reasoning, the process of deriving the fault list of a gate output from those of the gate inputs is called ***fault list propagation*** **[92]**.

Let us see the procedure of fault list propagation in general. In deductive fault simulation it is important to know either the gate input holds a controlling value or a non-controlling value. The ***value*** of an input is said to be ***controlling*** if it determines the value of the gate output regardless of the values of the other inputs [1]. For example, a controlling value for an AND gate is 0 (because appearance of 0 at least in one of the inputs of an AND gate will force the output of the gate into 0), for an OR gate is 1, etc. Let I be a set of inputs of a gate Z. Let C be a set of inputs with controlling value c, where $C \subseteq I$. z is an output signal of gate Z. The fault list L_z of gate Z is computed as follows where 2 cases are recognized:

1) **If** $C = \emptyset$ **then** $L_Z = \{\bigcup_{j\in I} L_j\} \cup \{fault\ z\}$

   This means, that if all inputs have non-controlling values, then all faults observed at the inputs propagate to the output of the gate adding a fault of the output signal line. In the example in Fig. 2.10 this is computation of L_{out}, because the OR gate has inputs 00, which both are non-controlling values for the OR gate.

2) **If** $C \neq \emptyset$ **then** $L_Z = \left\{\bigcap_{j\in C} L_j\right\} - \left\{\bigcup_{j\in I-C} L_j\right\} \cup \{fault\ z\}$

   This means, that if some of the inputs has the controlling value c, only faults of these inputs propagate to the output taking into account the self-masking effect - the appearance of the same fault at any of non-controlling inputs, which faults are excluded at the fault list of the gate output. As in the previous case, the fault of the output signal line is added. In the example in Fig. 2.10, L_h and L_f are calculated with this formula, because both AND gates have in one input the controlling value 0 and in the other input non-controlling value 1.

Consider the same multiplexer example (Figure 2.10). Let input vector to the schematic be the same ACB = 011. Also, the same faults f1 (s-a-1), f2 (s-a-0), f3 (s-a-0) are injected into the circuit. Faults are propagated by computation of fault lists at every signal line of a circuit:

$L_a = \{f_1\},\ L_c = \emptyset,\ L_g = \emptyset,\ L_h = L_a - L_g = \{f_1\},$

$L_b = \{f_3\},\ L_d = \{f_2\},\ L_e = L_d = \{f_2\},\ L_f = L_e - L_b = \{f_2\},$

$L_{out} = L_h \cup L_f = \{f_1, f_2\}.$

The following fault list is propagated to primary output *out* $L_{out} = \{f_1, f_2\}$, therefore detected faults for input vector *ACB =011* in this circuit are $\{f_1, f_2\}$, and f_3 is not detected.

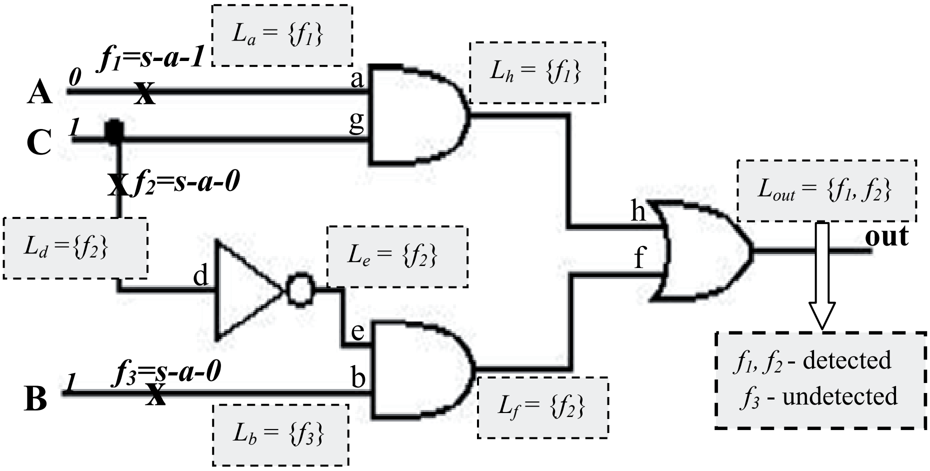


Figure 2.10 Deductive fault simulation

Deductive fault simulation is efficient and powerful technique, because it processes all faults in a single run without re-simulations of the same circuit. However, it has limitations as well. Unknown values are not easily handled. Algorithm spends a lot of CPU time for logic operations on sets. Also it has potential memory management problem, because the size of the fault lists cannot be predicted in advance.

2.2.4.4 RTL Fault Simulation

For RTL fault simulation, the input design is described at higher abstraction level - RT level. RTL constructs represent a subset of HDL (hardware description language) constructs in order to ensure the consistent synthesis by logic synthesis tool into gate-level. An RTL model represents micro-architecture of a circuit, where operations are synchronous transfers between functional units (e.g. arithmetic units) and registers. A fault model can be designed according needs, though a RT level fault model should guarantee fault coverage comparable to the gate-level fault coverage obtained for the same test sequence. In this work the RTL bit coverage fault model described in part 2.2.3 “Fault modeling” is considered, where memory bits stuck at value 0 or 1.

For RTL fault simulation any algorithm can be selected. Usually it is convenient to apply proven algorithms for SSF model in fault simulation at other levels of abstraction. Some of these standard algorithms were described in previous subsections. Certainly, different modifications for these algorithms are required in order to simulate the RTL bit coverage fault model. The RTL fault simulation method itself is analogous to the gate level approach, where fault-free and faulty circuits are created based on the SSF assumption and simulated with given test vectors. When the outputs of the fault-free circuit and the faulty circuit are different, the fault is considered to be detected. Simulation continues until all faults are evaluated with given test vectors. At the end of fault simulation a fault report is generated where RTL fault coverage of a circuit is provided.

In general RTL fault simulation has the following advantages with respect to the gate-level fault simulation:

- a performance gain compared to gate level approach;
- the possibility of improving tests prior to logic synthesis;
- early detection of testability problems, when design for testability is considered for the circuit.

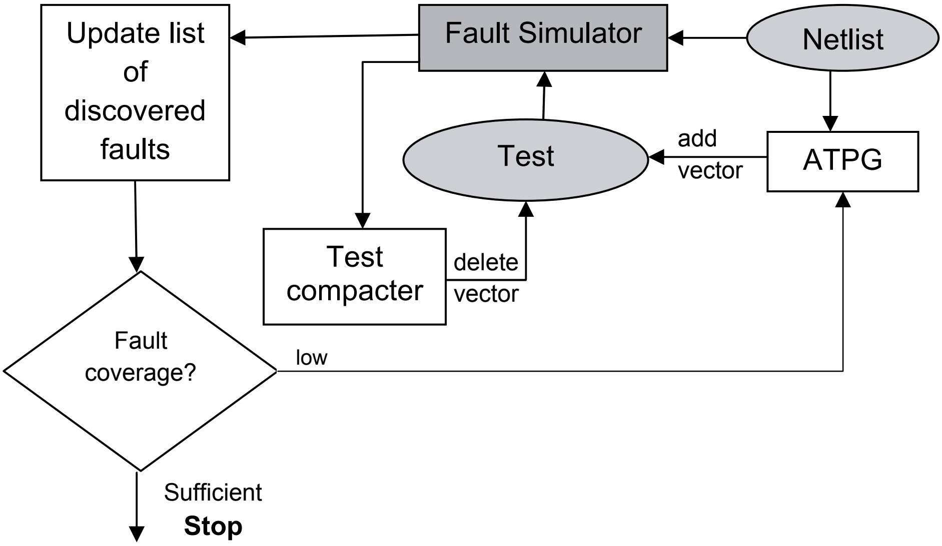


Figure 2.11 Fault simulation for test generation

2.2.5 Applications

In this subsection, main tasks, which require intensive use of fault simulation, are outlined. This helps to understand how important the performance of fault simulation is. The speed of simulation is very significant in a lot of tasks, because the process of simulation must perform many times in a cycle. General picture for test generation is presented at Fig 2.11, where other applications of fault simulation are also depicted.

First, fault simulation rates the effectiveness of a set of test vectors in detecting defects. The **test quality** is measured in terms of fault coverage with respect to the modeled faults of interest. Calculating the fault coverage is the primary task of any fault simulator.

Second, fault simulator helps to identify undetected faults. Fault simulator is often used in conjunction with an **Automatic Test Pattern Generator (ATPG)** in order to verify the generated test vectors. The generated test set is modified by adding new test vectors until the obtained fault coverage is considered satisfactory. These changes may be made by ATPG or by test designer in an interactive mode.

Third, it is possible to use the result of the fault simulator to remove test vectors from an already available test set without decreasing the fault coverage.

This process is called **test compaction** and often effectively used for random test generation on combinational ATPG. Test compaction is done in order to reduce time of test application and also to reduce the cost of storage for test vectors in the tester memory.

Fourth, fault simulation helps in **fault diagnosis**. If a device fails the tests, then diagnostic information assists to determine the type and location of a fault that best explains the faulty behavior of the circuit under test. One of the well-known methods to perform diagnosis is to use a **fault dictionary**. The fault dictionary stores the faulty output response to test vector of the faulty circuit corresponding to the simulated fault. Actually, the fault dictionary does not store all faulty output responses, but a certain function, called the signature of the fault. For the circuit under diagnosis, the signature is used for narrowing the suspected area of the fault and trying to identify the fault. This fault dictionary is constructed during fault simulation and must be done without the fault dropping technique.

Fifth, fault simulation can be used to find the optimal solution for **BIST** (Built-in Self-Test). BIST is a technique, which enables a circuit with the additional functionality of self-testing. A typical Logic BIST consists of a controller, a special pseudo-random test pattern generator and a response analyzer. Also, memory is sometimes required to store deterministic test vectors. To find a tradeoff between the size of additional memory and the quality of test many runs of fault simulation is required.

Another application of fault simulation is to **analyze the operation of a circuit in the presence of faults**. This is especially important in high-reliability systems. Examples [1]:

- a fault can induce races and hazards not present in the fault-free circuit;
- a faulty circuit can oscillate or enter a deadlock state;
- a fault can prohibit the proper initialization of a sequential circuit;
- a fault can transform a combinational circuit into a sequential one or a synchronous circuit into an asynchronous one.

2.3 Design Verification

Verification of digital circuits is a process of proving that digital circuit design meets the design specification and requirements before its manufacturing. It is important to distinguish between testing and verification, where testing ensures that the implemented device works correctly and verification is done during the implementation phase. Statistical data show that around 70% of the project development cycle is devoted to design verification [43]. A verification engineer must verify the design under all cases, not only cases represented in the specification. It is not possible during project development cycle to verify all cases thoroughly while meeting the time-to-market requirements. Therefore different methodologies are used to plan the verification process, in which critical parts of the design are proven to be bug-free. A verification methodology starts with a test plan that details specific functionality to be verified in order to satisfy specification. Different methods are used to track process against test plan. In reality it is impossible to verify a set of specifications completely [14]. Thus, a measure of verification quality is desirable. This measure is called coverage metrics. Usually used metrics are functional coverage and code coverage. Functional coverage shows the part of the functionality verified in percentage. Code coverage measures the percentage of code simulated.

One of the possible methodologies is borrowed from [14]. This methodology is based on automated metric-driven processes. Processes are important and improve the predictability of the project. The verification process model is shown in fig 2.12.

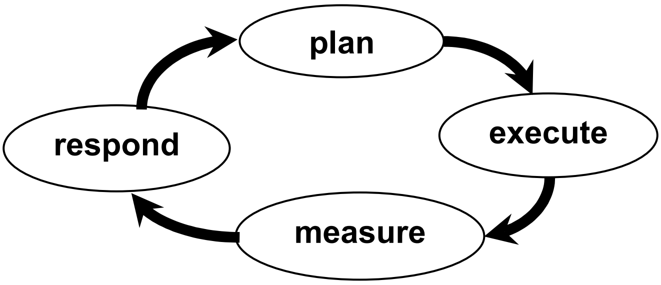


Figure 2.12 Verification Process Model [14]

In the first phase – *plan* - it is determined what requirements should be and how to measure the result. Then, the model of the design needs to be executed (*execute* phase). In the *measure* phase, effectiveness of the verification effort is measured with a help of different metrics such as code coverage, functional coverage, assertion coverage etc. In the *respond* phase, result data is analyzed and adjustments are worked out for the next iteration. This cyclic process repeats until all product requirements are satisfied.

Since the design process is usually hierarchical process, verification also must be made at all levels of design. Design verification is a reverse process of design. Design verification starts with implementation and confirms that the implementation meets the specification at every abstraction level. Also, equivalence checking between different levels is performed. A common intermediate form for determining equivalence between transistor-level and RTL is binary decision diagrams.

There exist two verification approaches: formal method-based verification and simulation-based verification. These methods can be distinguished as formal verification is output oriented (the output properties must be verified) and simulation-based verification is input oriented (input vectors must be supplied).

Formal verification mathematically proves that a protocol, assertion, or design rule holds true for all possible cases in the design [94]. Formally only a limited sized design can be verified, a formal verification engine consumes enormous amounts of computer resources, even for small designs. Formal methods work well on control logic blocks, where results can be returned in a reasonable time.

Simulation-based verification relies on the software model of a design, which runs on the simulation engine. Simulation reflects the behavior of the modeled device and generates corresponding outputs in response to specific inputs. During simulation-based verification the design is placed under testbench, input stimuli are applied to the testbench, and output from the design is compared with the reference output [43].

The most commonly used verification approach is simulation-based verification. Formal verification is a great complement to simulation and often applied to portions of the entire design. In this work only simulation-based verification for finding code coverage is used, thus this approach is described below in more details.

2.3.1 Simulation-based verification

Three interrelated problems in simulation-based verification are [1]:

- How does one generate the input stimuli?
- How does one know the results are correct?
- How "good" are the applied input stimuli, i.e. how "complete" is the testing experiment?

Usually input stimuli are organized as a sequence of test cases, which are extracted from design specification. Also pseudorandom input stimuli are generated in order to activate unusual bugs of which designers are unaware. The cost of creating pseudorandom inputs is much lower compared with directed tests.

The results are considered correct when they match expected results according the specification of the design. Goodness of applied input stimuli commonly measures with a help of different coverage metrics.

The typical flow of simulation-based design verification is shown in Fig 2.13 [43]. The components inside the dashed enclosure represent the components specific to the simulation-based methodology. During simulation-based verification, the design is placed under a test bench. A test bench consists of code that supports operations of the design, and generates input stimuli and compares the output with the reference output as well. Before a design is simulated, it runs through a linter program that checks static errors or potential errors and coding style guideline violations. Next, input vectors of the items in the test plan are generated. After the tests are created, simulators are chosen to carry out simulation. The coverage metrics measure how much the design is stimulated and verified. When a bug is found, it has to be communicated to the designer and fixed. This is usually done by logging the bug into a bug tracking system, which sends a notification to the owner of the design. When bugs are fixed, the regression is carried out, which goal is to make sure that none of previously existing functionality has broken. Design codes with newly added features and bug fixes must be made available to the team. Therefore, design codes are maintained using revision control software that arbitrates file access to multiple users and provides a mechanism for making the latest design code visible to all.

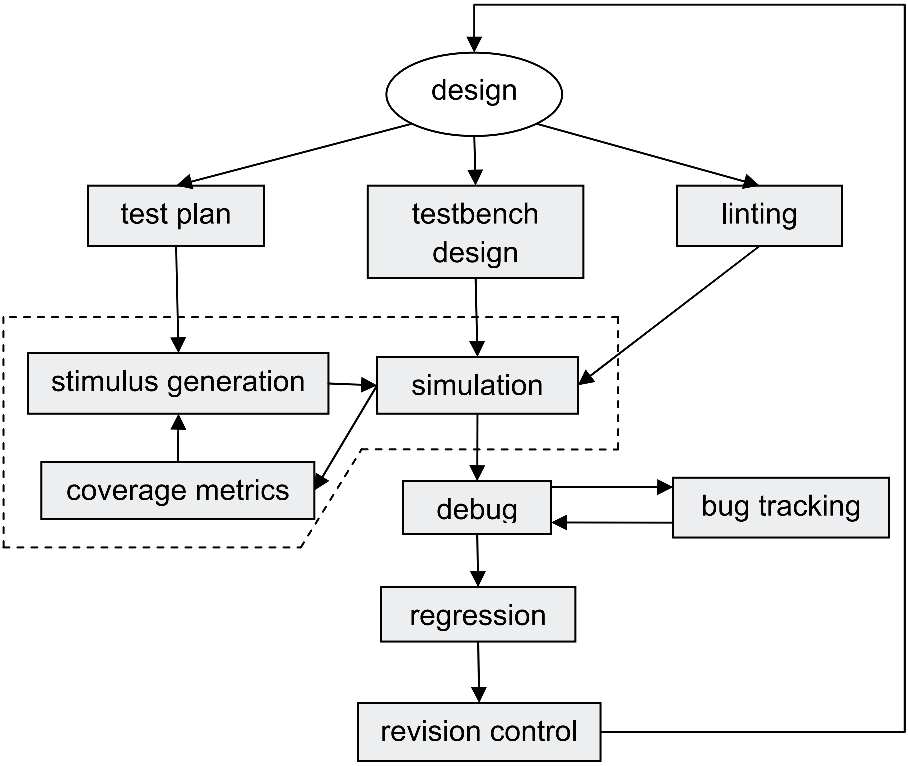


Figure 2.13 Simulation-based verification [43]

The simulation-based method has strengths and weaknesses [13]. Its strength is in details of the circuit behavior that can be simulated, such as timing details, logic details etc. Another advantage is use of hierarchy of the design. Circuit can be simulated at different levels of abstraction iteratively. Thus the speed of simulator is very important, because it is used very many times. The method weakness is its dependence of designer’s heuristics for generating input stimuli. Also these stimuli are non-exhaustive and therefore guarantee of conformance to specification is impossible. Such guarantee is possible with a formal verification method. But the high complexity of formal methods allows their use only for small designs and at the higher behavior levels. In case of the incompleteness, simulation provides a better check on the manufacturability of the design. Ideally both (simulation-based and formal) methods should be used in conjunction to verify the design.

2.3.2 Coverage metrics

To measure the quality of design verification, coverage metrics are widely used. Because it is impossible to verify exhaustively a design, the confidence level regarding quality of verification must be quantified. The fundamental question is “How do I know if I have simulated or verified enough?” Verification coverage is a measure of confidence and it is expressed as a percentage of items verified out of all possible items [43]. An item can be of various forms: a line of code, parameters, functionality in a specification etc. Thus coverage metrics are divided into code coverage, parameter coverage, functional coverage etc. Code coverage provides insight into how well the code of an implemented design is verified by the stimuli. Functional coverage computes the amount of features and functions that are exercised in a design. Code coverage is also called implementation coverage, because it is based on the implementation of the design, whereas functional coverage is based on specification only. Therefore code coverage does not necessary provide implementation’s correctness with respect to its specification, because it assumes only current implementation of a design. However, using code coverage metrics designer sees the parts of a design that have not been exercised yet and can create input stimuli for these parts. To guarantee a good level of confidence different class of metrics must be used, so usually code and functional metrics are used in conjunction.

Carter et al. in [14] propagate metric-driven design verification, where they state that metric-driven approach allows improving the predictability, productivity and quality of both implementation and engine execution. First, metrics are able to point out holes of uncovered areas in the design. Second, with automatically captured metrics it is possible to fully automate some processes and remove the human element.

In this thesis, code coverage metrics were implemented based on HLDDs. Three metrics were implemented: statement, branch and toggle. Only these metrics are described in details below.

Statement coverage calculates how many statements are executed during simulation among all possible statements in a code. Statement is syntactical structure of HDL specification. The statement coverage gives the knowledge that statement has been executed.

Branch coverage reports the count of control flow transfers for HDL control statements. This means keeping track on which conditions the simulation encounters and which it does not. The limitation of this coverage is that decisions can be implemented not only with a help of conditional constructs. Thus some decisions are not taken into account by branch coverage.

Toggle coverage measures transitions between values of bits in registers, wires. Toggle coverage is the ratio of bits toggled from 0 to 1 and from 1 to 0 to the total number of bits in registers and wires. Toggle coverage is simple to implement and easy-to-understand. It is a general activity indicator, but gives a very coarse view of signal activity and associates no semantic meaning to recorded results.

Code coverage can clearly improve the verification quality. They are easy to implement and to use, and give information of uncovered parts of the design. Also, these metrics are usually embedded into EDA (Electronic Design Automation) tools and straightforward to use during simulation. All these advantages have made code coverage an important feature of HDL simulation engine.

2.4 Chapter summary

The goal of this chapter was to provide background information required to understand the contributions of this thesis. The first part of this chapter gives an overview of HLDDs, proposed and developed at TUT (Tallinn University of Technology). Advantages of this model for representing digital designs for fault simulation purposes as well as for simulation-based verification purposes were described.

The second part of this chapter described the role of testing of digital circuits. Then, basic concepts were given such as fault and defect. Also different fault models at different abstraction levels and their relation to real defects were described. Then, fault simulation principles were presented and classic fault simulation algorithms based on the widespread stuck-at fault model were explained.

The third part of this chapter provided an overview of design verification issues. The main emphasis of this part is given to simulation-based verification. Then, basic knowledge of code coverage metrics was presented and the advantages of simulation-based verification were explained.

Chapter 3
HLDD-BASED FAULT SIMULATION

In this chapter, a new approach to fault simulation based on High-Level Decision Diagrams (HLDDs) is proposed, which is applicable directly at the Register-Transfer Level (RTL). The fault simulation algorithm is built on the RTL bit coverage fault model, which has proven to yield good correspondence with gate-level structural faults [29]. A new deductive fault simulation algorithm is described. An efficient data structure based on bitwise set operations is introduced in order to achieve a high speed of simulation. Experiments on RTL benchmark circuits are presented.

3.1 Overview

While several efficient algorithms for fault simulation of combinational circuits exist, the task of analyzing structural faults in sequential circuits remains a highly difficult issue. In order to contend the complexity the research community has turned towards developing methods at higher design abstraction levels.

Existing fault simulation tools typically rely on gate-level algorithms. One of the earliest sequential fault simulators, PROOFS [57] combines the advantages of differential fault simulation and parallel fault simulation. HOPE [44], a parallel fault simulator, simulates 32 faults at a time. Faults with short propagation paths are excluded from parallel simulation, since most of the time the faulty circuit response would be identical to the correct one during the simulation of such faults. Also PARIS [31] is based on a parallel fault simulation model. Heuristics are used to minimize the number of events that must be tracked. LIFTING [8] is an open-source simulator able to perform logic and fault simulations for single/multiple stuck-at faults and single event upset

(SEU) on digital circuits described in Verilog. Despite of a wide range of methods, fault simulation for sequential circuits at the gate-level is slow for larger designs, in particular when long test sequences are considered.

Functional fault simulation of VHDL designs has been proposed in [28][93]. This approach is fast but it lacks accuracy since there is no strict correlation between the functional fault model and actual structural faults in the circuit. In [45] an architectural-level fault simulation tool ARSIM is presented. The tool uses symbolic data to simultaneously process the fault effects for groups of faults in the module under simulation. However, ARSIM is capable of reporting only pessimistic fault simulation results because of the limitations of the symbolic algebra applied in fault propagation. This shortcoming has been contended in an improved symbolic approach by Sinanoglu and Orailoglu [70] by utilizing the rightmost faulty bit location information to enhance the method's ability of propagating symbolic data. In [39], Kassab, Rajski and Tyszer, propose hierarchical functional fault simulation that provides high speed-up but relies on building blocks that have regular structures. Shen introduced a concurrent Register-Transfer Level (RTL) fault simulator VFSim [69], which is capable of simulating Verilog designs. However, comparison to the gate-level fault simulator HOPE [44] showed no speed-up in most cases.

Deductive fault simulation algorithm was first introduced by Armstrong [3]. Some improvements of this algorithm for 2-value bit logic were worked out in [52][74]. This algorithm is developed for 2-value bit logic and different gate-level fault simulators [15], [71], [91] were built based on this technique. In this thesis deductive fault simulation algorithm is reused for register-transfer level of design representation, where integer values are used to pass signal values.

In [21] fast RTL fault simulator is presented. This simulator is based on Reduced Ordered Ternary Decision Diagrams ROTDD. For fault propagation it combines advantages of both bit-parallel and deductive techniques. In comparison with HOPE [44] proposed in [21] the method is faster, but there is no strict connection between the RTL fault model, applied in [21], and the gate-level fault model.

The paper [19] focuses on fault simulation at the RTL, and aims at exploiting the capabilities of VHDL simulators to compute faulty responses. Authors established rules for RTL code, which allow the RTL fault coverage to become more and more correlated to the gate-level fault coverage. The feasibility of this method has been confirmed by experiments with ITC'99 benchmarks. However, these RTL faults are modeled only on all assignment targets of the executed statements that respect a defined set of rules.

In [73] fault simulation at RT level for digital circuits is proposed. The method is based on Verilog hardware description language, where fault-free circuit description is changed to faulty description. Commercial simulator is used to compare results. The fault model is based on an assumption that only interconnections are fault affected, thus these map to operators and variables in RTL. The RTL Fault Coverage obtained by the proposed fault modeling methodology has a close match to the Gate-Level Fault Coverage for the tested digital circuits. However, for each of the faults, a new faulty circuit is created in Verilog. This is a time-consuming task and authors of the paper do not provide time of fault simulation neither at RTL nor at gate level.

Moreover, there exists acceleration of fault simulation even at TLM (Transaction-Level Modeling) level [7]. It is clear that many details (time, structure) are dropped, which allows fast design exploration. When TLM exploration is performed a RTL fault simulator is still required for more detailed design evaluation.

A concept of hierarchical fault simulation using a deductive algorithm on High-Level Decision Diagram (HLDD) [83] models was introduced in [86]. The method assumed that gate-level descriptions of all the modules exist and faults were modeled in the circuits hierarchically at the register-transfer and logic levels. Another, RTL algorithm was proposed in [64], which is the first version of the algorithm presented in this thesis. However, experimental results showed that the method becomes prohibitively slow when circuits with large number of arithmetic operations are considered. An efficient data structure based on bitwise set operations is introduced in [63] in order to increase the speed of fault simulation.

3.2 Deductive Fault Simulation on HLDDs

Deductive fault simulation at RT-level based on HLDD models will be described in this subsection. This simulation algorithm and its implementation is the contribution of [63][64]. In Figure 2.4 an example of a small digital circuit and corresponding HLDD-based representation is depicted. This example circuit would be used to explain the deductive fault simulation algorithm. The main advantage and motivation of using HLDDs compared to the netlists of primitive functions is the increased efficiency of simulation and diagnostic modeling because of the direct and compact representation of cause-effect relationships.

In this work we rely on the RTL bit coverage fault model, described in section 2.2.3. According to this fault model the faults are injected to every bit of every register in the RTL circuit. Single fault assumption is used, i.e. a fault is expected to be present at one of the register bits at the time. The bit is assumed to be permanently stuck to the value 0 or 1. This RTL bit coverage fault model has been proven to have a good correspondence with gate-level structural faults [29].

The central datastructure of the deductive fault simulation algorithm is a *fault propagation record.* A fault propagation record T_y is generated for all the variables $y \in Y$ that represent registers.

$$T_y=\{p_0, (p_1, M_1), \ldots, (p_k, M_k)\},$$

where p_0 is the fault free value of the register variable y and p_j is the faulty value corresponding to the faults $m_{j,i} \in M_j$ propagated to variable y. $M_1 \cup \ldots \cup M_k = M'$ and $M_1 \cap \ldots \cap M_k = \varnothing$, where $M' \subseteq M$, and M is the set of all faults and M' is the set of faults propagated to the register variable y. If two faults produce the same faulty value p_j then they should be merged to the same fault group M_j.

3.2.1 Algorithm Structure

The algorithm processes test vectors one by one. The design under test is translated automatically into internal representation at High Level Decision Diagrams (HLDDs). Usually there is a set of diagrams rather than one diagram representing the circuit. Simulation and fault propagation of the DUT (Design Under Test) is done diagram by diagram, where diagrams are ordered according dependences between them. The fault insertion is done for all the bits in variables, which represent registers. The RTL bit coverage fault model (data bit is stuck-at-0 or stuck-at-1) is implemented by flipping the bits to the stuck-at-value from correct one for every test vector. Then, propagation of the fault sets is done from the root of the diagram to the terminal nodes. For saving the fault information, the fault propagation record is used for every node of the diagrams.

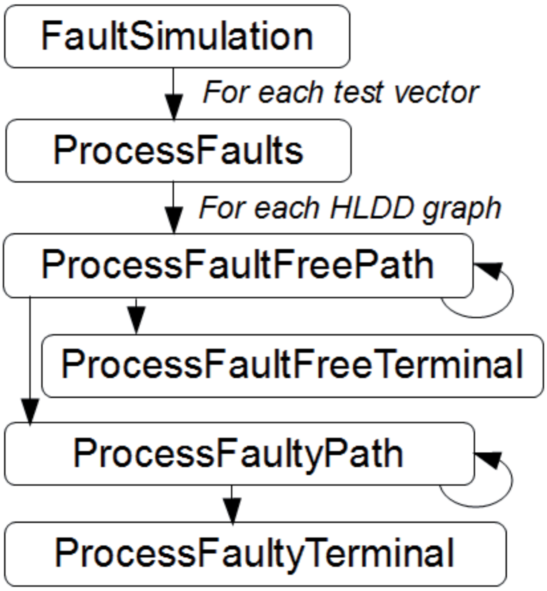


Figure 3.1 Call graph of the deductive fault simulation algorithm

General function call graph of the deductive fault simulation algorithm is presented in figure 3.1, pseudo-code of the algorithm is depicted in figure 3.2. As it has already been mentioned, the fault simulation on the HLDDs is performed vector by vector (function *FaultSimulation*). For each vector, all HLDD graphs are traversed one after another (function *ProcessFaults*). In every graph (HLDD), first, the fault free path is followed by processing the nodes along the main activated path of the HLDD recursively (function *ProcessFaultFreePath*) until the terminal node v^T is reached (function *ProcessFaultFreeTerminal*). While following the main activated path for every node the set of possible faults out of fault free path is calculated. If this set is not empty the faulty path is processed (function *ProcessFaultyPath*) propagating the faults to the terminal nodes. This propagation is done recursively node by node until terminal node of the graph is reached. All the possible branches are taken to process the faulty paths. While processing the terminal nodes, new fault propagation record T_y for the HLDD g_y is calculated (function *ProcessFaultyTerminal*).

At the beginning of a new cycle (new test vector) new faults are injected according to the RTL bit coverage fault model. Thus, the set of faults to propagate to the next clock cycle consists of propagated faults and newly injected faults.

During the fault simulation, two sets of faults M_{excl} and M_{incl} are collected for each graph. M_{excl} (*excluded faults*) is the set of faults that can't be propagated to the output of the fault-free path. M_{incl} (*included faults*) is the set of faults that are included into the fault propagation path. As there are usually many faulty paths M_{incl} is calculated separately for every faulty path.

The set of functions depicted in the call graph of the fault simulation of the HLDD g_y (fig. 3.1) is described in more details below. The preudo-code of the algorithm (fig. 3.2) is the reference to the following description.

ProcessFaultFreePath. The fault-free path is simulated in accordance to the input test vector. For current HLDD g_y the fault-free value of y=D(x_{v}^{T}) is calculated, where v^T is the terminal node of the main activated path. In order to follow the main activated path the call of *processFaultFreePath* function is done recursively. In this function current node is processed and successor node is found for recursive call.

Following the main activated path for current node $M_{excl(v)}$ is calculated:

$M_{excl(v)} = M_{excl(v)}' \cup M_v$, where

- M_v is a set of faults in a current node v and
- $M_{excl(v)}'$ is a set of excluded faults collected along main activated path until current node v.

Initially, M_{excl} is empty set. The condition of reaching the node v in the fault-free path during fault simulation is the absence of all the faults from $M_{excl(v)}'$.

Denote by $M_{incl(v)}$ the set of faults consistent to the current faulty path from the initial node v_0 up to the node v. For the node v for fault-free path we have

$M_{incl(v)}=M_v - M_{excl(v)}'$

When processing the node in function *ProcessFaultFreePath()* first we call recursively the function itself with newly calculated $M_{excl(v)}$ while reaching the terminal node. Then function *ProcessFaultyPath()* is called if $M_{incl(v)}$ is not empty.

When reaching the terminal node of the fault-free path the terminal node is processed by *ProcessFaultFreeTerminal()*.

ProcessFaultFreeTerminal. Fault simulation of a terminal node v^T of the fault-free path lies in finding all the combinations of (p_j, M_j), of the terminal node's fault propagation record Tx_{v}^{T}. The set of propagated faults $M_{excl(vT)}$ is excluded from the Tx_{v}^{T}. This fault propagation record is assigned to the graph variable y (T_y= Tx_{v}^{T}). Also fault-free value of the graph is calculated.

Beginning…

```
FaultSimulation(){
  for each vector vec
      ProcessFaults(vec)
  end for
}

ProcessFaults(vec){
   $M_{excl}=\varnothing$
   for each graph $g_y$
      ProcessFaultFreePath($g_y$, $v_0$, $M_{excl}$)
      In $T_y$ assign the value of $y$ to $p_0$
      All single bit flip faults of $p_0$ are added to $T_y$
   end for
}
ProcessFaultFreePath(g, v, $M_{excl}$){
    $M_{incl}$={faults in $Tx_v$} – $M_{excl}$
    $M_{excl}=M_{excl} \cup$ {faults in $Tx_v$}
    if $v \notin v^T$
        ProcessFaultFreePath(g, $v^c$, $M_{excl}$)
        //$v^c$ - successor
        if $M_{incl} \neq \varnothing$
            ProcessFaultyPath(g, v, $M_{incl}$, True)
        end if
    else
        ProcessFaultFreeTerminal(g, v, $M_{incl}$)
    end if
}
ProcessFaultyPath(g, v, $M_{incl}$, FlagFromFaultFree){
   If $v \notin v^T$      // if non-terminal
       if(FlagFromFaultFree==False)
          $M_{param}=M_{incl}$ – {faults in $Tx_v$}
          ProcessFaultyPath(g, $v^{xv}$, $M_{param}$, False)
       end if
       For each faulty response $e=px_{v,j}$ of $Tx_v$
          $M_{incl}= M_{incl} \cap Mx_{v,j}$
          ProcessFaultyPath(g, $v^e$, $M_{incl}$, False)
       End for
   Else   // if terminal
       ProcessFaultyTerminal(g, v, $M_{incl}$)
   End if
}
….
```

Continuation…

```
ProcessFaultFreeTerminal($g_z$, v, $M_{incl}$){
    $T_y = Tx_v$
    Remove faults that are not in $M_{incl}$ from $T_z$
}

ProcessFaultyTerminal (g, v, $M_{incl}$){
    $M_{diff} = M_{incl}$ – {faults in $Tx_v$}
    $M_{insec} = M_{incl} \cap$ {faults in $Tx_v$}
    If ($M_{diff} \neq \varnothing$)
          add new pair (faulty value, $M_{diff}$) to $T_y$
    End if
    If($M_{insec} \neq \varnothing$)
          add pairs with faults $M_{insec}$ from $Tx_v$ to $T_y$
    End if
}
```

Figure 3.2 Pseudo-code of fault simulation algorithm

ProcessFaultyPath. Fault simulation along the faulty path of a non-terminal node v labeled by the variable x_v with fault propagation record $Tx_v = \{p_0, (p_1, M_1), \ldots, (p_k, M_k)\}$ consists of the following:

- For v, faulty responses are processed as follows. For each faulty response $e=p_i$ of Tx_v, $M_{incl(v)}$ is calculated as follows

  $\mathrm{M}_{\mathrm{incl(v)}} = \mathrm{M}_{\mathrm{incl(v)}}' \cap \mathrm{M}_{\mathrm{i}}$, where

    - $M_{incl(v)}'$ is a set of included faults collected along the faulty path until the current node v.
    - M_i is a set of faults which corresponds to the faulty value p_i

  According to the faulty response value p_i the new successor node is found and the function *ProcessFaultyPath()* is called recursively with $M_{incl(v)}$ and the successor node of v with edge label equal to p_i.

- If v does not belong to the fault-free path then the non-faulty response p_0 of Tx_v is found and $M_{incl(v)}$ is calculated as follows

  $\mathrm{M}_{\mathrm{incl(v)}} = \mathrm{M}_{\mathrm{incl(v)}}' - \mathrm{M}_{\mathrm{i}}$

  The function *ProcessFaultyPath()* is called recursively with $M_{incl(v)}$ and the successor node of v with edge label equal to p_0.

When reaching the terminal node of the faulty path the terminal node is processed by *ProcessFaultyTerminal()*.

ProcessFaultyTerminal. When reaching the terminal node v of a faulty path the difference of faults M_{diff} is calculated by subtracting all the faults in Tx_v (propagation record of terminal node v) from $M_{incl(v)}'$. Then, if M_{diff} is not empty, a new pair (value p_0 of terminal node v, M_{diff}) is added to the fault propagation record T_y (y is a variable holding the result of graph g_y).

Also, an intersection of faults M_{insec} is calculated.

$M_{insec} = M_{incl(v)}' \cap$ all faults in Tx_v,

Only the pairs (faulty value p_i of the terminal node v, $\mathrm{M}_{\mathrm{insec,i}}$) corresponding to faults M_{insec} ($M_{insec,i} \subseteq M_{insec}$) are added from fault propagation record Tx_v to the fault propagation record T_y.

If a terminal node v is labeled by a function then faulty values of function parameters are taken in order to calculate function result. Parameters' faulty values correspond to a fault from the set of propagated faults to the variables corresponding to the parameters. If parameter's fault propagation record does not contain the current propagated fault, then the fault-free value of this parameter is taken. Function result is a new faulty value and pair (function faulty value, current propagated fault) is added to fault propagation record T_y (y is a variable holding the result of graph g_y).

As a result of the fault simulation we create a fault propagation record for the variable y: $T_y = \{p_0, (p_1, M_1), \ldots, (p_k, M_k)\}$. All the pairs (p_j,M_j) where $p_j = p_0$ are eliminated since the faults M_j are self-masked at this point. All the groups of pairs $\{(p_i,M_i), (p_j,M_j)\}$ where $p_i = p_j$ are merged into a single pair (p_i,M_i), so that $M_i = M_i \cup M_j$.

3.2.2 Example of deductive fault simulation on HLDD

Figure 2.4 (see part 2.1.2) presented a HLDD g_{R2} . Consider a fragment of this HLDD in Figure 3.3 with a set of fault propagation records:

$T_{SEL3} = \{3, 0\ (f_3,f_4), 1\ (f_1,f_2,f_5)\}$,

$T_{SEL2} = \{0, 1\ (f_3,f_5)\}$,

$T_{SEL1} = \{1, 0\ (f_4,f_6)\}$,

$T_{R2} = \{7, 3\ (f_4,f_5,f_7), 4\ (f_1,f_3,f_9)\}$,

$T_{R1} = \{2, 4\ (f_3,f_7), 6\ (f_2,f_8), 9\ (f_4,f_5)\}$,

$T_{IN} = \{4, 5\ (f_6,f_7), 8\ (f_2), 10\ (f_1,f_3,f_4)\}$.

All the paths traced during the fault simulation are highlighted and marked by details of simulation in Fig.3.3. The fault-free paths are shown by bold lines in the Figure. The edges on paths in Fig.3.3 are labeled by pairs $\{e,(M)\}$, where e is the value of the node variable when leaving the node at this direction, and M is a subset of faults: $M_{excl}(v)$ for the successor of node v on the fault-free path, and $M_{incl}(v)$ for the successor of node v on the faulty paths. $M_{excl}(v)$ is marked with line above the fault IDs. The result of function *processTerminal* (either faulty-free of faulty) is depicted in the grey boxes attached to the terminal node.

Since $M_{excl}(SEL_2) = \{f_1,f_2,f_3,f_4,f_5\}$ includes both faults of SEL_2, the faults f_3 and f_5, no faulty paths are simulated from the node SEL_2:

- for the value $SEL_2 = 1$: $M_{incl}(SEL2) = (M_{SEL2} - M_{excl}'(SEL2)) =(\{f_3,f_5\} - \{f_1,f_2,f_3,f_4,f_5\}) = \varnothing$.

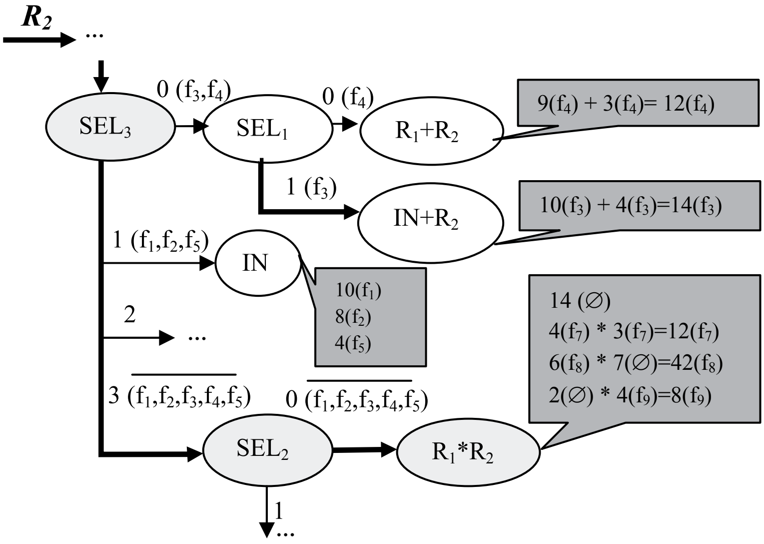


Figure 3.3 Example of deductive fault simulation on HLDD

From all the faults propagated to R_2, only the faults f_7, f_8 and f_9 are simulated at the node R_1*R_2. Such as node R_1*R_2 is a function, then for fault f_7 faulty values of $R_1 = 4$ and $R_2 = 3$ are taken and result of the function is calculated $\{12(f_7)\}$. The same calculations are done for other propagated faults f_8 and f_9.

At the terminal node *IN,* only the faults f_1, f_2, f_5 are simulated, since only they are consistent to the condition of leaving the node SEL_3 towards this direction. For terminal node *IN* on the faulty path we have the following calculations:

- $M_{insec} = M_{incl(IN)}$' ∩ all faults in $T_{IN} = \{f_1,f_2,f_5\} \cap \{f_1,f_2,f_3,f_4,f_6,f_7\} = \{f_1,f_2\}$. Then from T_{IN}, faulty values consistent to fault f_1 and f_2 are found $\{10(f_1), 8(f_2)\}$ and are added to the fault propagation record T_{R2}.
- $M_{diff} = M_{incl(IN)}$' - all the faults in $T_{IN} = \{f_1,f_2,f_5\} - \{f_1,f_2,f_3,f_4,f_6,f_7\} = \{f_5\}$. Then fault-free value of terminal node *IN* is found and pair $\{4(f_5)\}$ is added to the fault propagation record T_{R2}.

After fault simulation of all the faults that reached terminal nodes, we compose the final result of T_{R2} as follows: the fault f_3 propagated to the node IN+R_2 is self-masked because the value $IN+R_2 = 14$ calculated for the fault f_3 is equal to the fault-free value calculated at the node R_1*R_2. The faults f_2 and f_9 propagated to different terminal nodes are merged into the same group because they produce the same new value 8 for R_2. Also, the faults f_4 and f_7 are merged into the same group. The final value of the fault propagation record for R_2 is:

- $T_{R2} = \{14, 4(f_5), 8(f_2,f_9), 10(f_1), 12(f_4,f_7), 42(f_8)\}$.

3.2.3 Internal Data Representation

The core part of any deductive fault simulation algorithm is the set operations on faults. In order to perform calculations with fault sets during propagation more efficiently, a dedicated representation of the fault propagation record was developed. The following operations with fault sets were carried out: difference, intersection, union. To achieve high speed, bitwise set operations were implemented. For that, fault data were stored in a specific way. In particular, for every variable two arrays exist: *FaultOccurences* array and *FaultyValues* array (Figure 3.4).

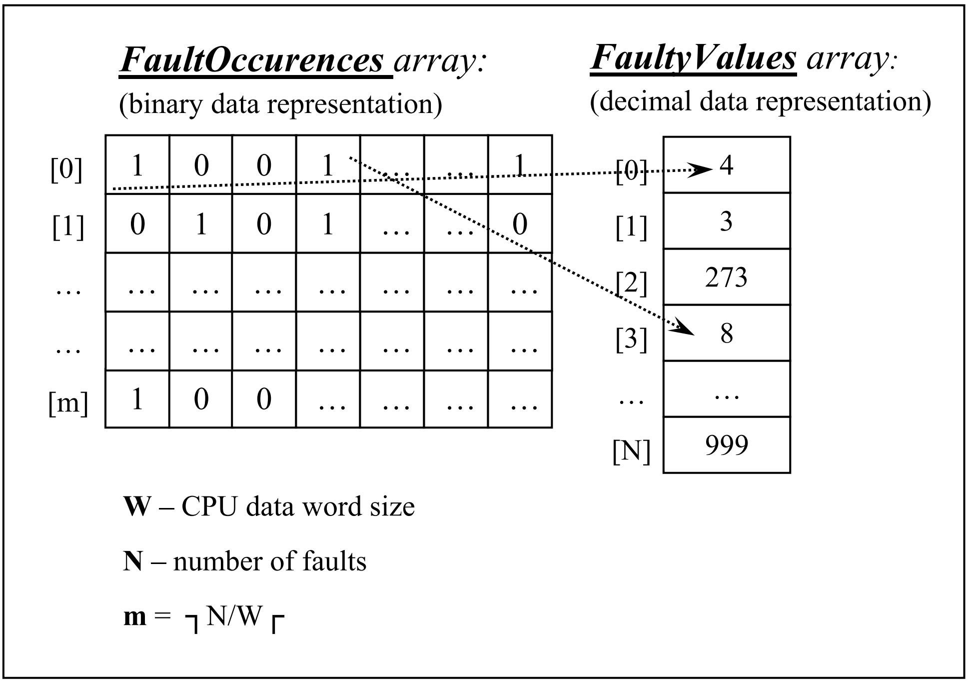


Figure 3.4 Data structures for storing propagated faults

For every variable in the circuit, all propagated fault IDs are stored in an array *FaultOccurences* (binary representation of data). In addition, faulty values corresponding to these fault IDs are stored in an array *FaultyValues* (decimal representation of data) to the respective positions.

The array *FaultOccurences* represents the presence of faults in a variable. In order to reduce simulation time the width *W* of processor data is fully utilized. If computer has a 32-bit architecture (W=32) then set operation of 32 faults is performed simultaneously. Every bit of array *FaultOccurences* denotes whether the fault is propagated to variable *v* or not (‘1’ denotes propagated, ‘0’ – not propagated). Index *i* represents the bit index in an array and denotes the fault ID. If the number of faults is more than W then more than one word is used in the array *FaultOccurences*. If *N* is the number of faults in a fault list then *m* is the number of words in *FaultOccurences* array.

m= ┐ N/W ┌

Another array *FaultyValues* is used to store the faulty values corresponding to the propagated faults for a variable *v*. This array is linked to array *FaultOccurences* and has length *N* (number of faults in the fault list). Word in *FaultyValues* array store faulty values propagated to the variable *v*. Word's index *i* represent the faulty value of the fault with ID = *i*. If the *i-th* bit in the *FaultOccurences* array is '1' then *i-th* word in the *FaultyValues* array stores the corresponding faulty value of the fault with ID = *i*. Otherwise, if the *i-th* bit in the *FaultOccurences* array is '0' then value of *i-th* element of *FaultyValues* array is out of our interest. For example, in Fig. 3.4 for some variable *v* there exist faults with ID=0, 3, ..., 33, 35,… and the corresponding faulty values are 4, 8, ... Note that value 273 in array FaultyValues at position *i=2* is not used. Thus, this structure usually requires more storage than is needed, but it is of fixed size.

The proposed data structure for representing fault lists proved extremely efficient. Complex set operations with large fault lists could be performed in very short run times. For example, the set intersection operator is reduced to just performing bitwise AND operations on the *FaultOccurences* array, while the union operator is just represented by bitwise OR, etc. Experiments performed on a set of sequential benchmarks prove the efficiency of this approach.

3.2.4 Analysis of the algorithm

To evaluate the algorithm's complexity, it should be noted that every test vector is analyzed independently. The number of test vectors depends on the test strategy that is not covered in this section. In the deductive fault simulation algorithm every node is traversed once per test vector.

Let *n* be the number of nodes in the graphs (High-Level Decision Diagrams). Let *N* be the number of faults in the circuit and *m* be the number of words representing the fault list *m*= ┐ *N/W* ┌ (see also Fig. 3.4).

For every node it is required to perform operations with all the faults in the fault list, because a fault in any other variable (node) may affect the node under analysis. As a rule, the number of required operations is *m*n*, because it is possible to analyze *W* faults at once using bitwise set operations.

In the worst case, $N_{max}=n*W$ because every variable (node) requires an entire processor word, i.e., W faults must be inserted to every node. Thus, in the worst case, $m=n$. Then the algorithm's *time complexity* is $O(n^2)$.

In general, to store the information about faults

Memory size = $(1+m+N)*n+2m$ [processor words] is required, where

- *1* processor word is to store fault free value for one node;
- m processor words is to store fault occurrences for one node;
- N processor words is to store faulty values for one node;
- n is number of nodes, for every node we need to have storage of the size $(1+m+N)$ processor words;
- $2m$ is to store propagated faults during the work of the algorithm one m is for M_{excl} and further m is for M_{incl};

In the worst case, when $m=n$, then

Memory size = $(1+n+W*n)*n + 2n = (3+n+W*n)*n = (W+1)\, n^2 + 3n$.

Thus, the overall *memory complexity* is $O(n^2)$.

3.3 Experimental results

Comparative experiments between high-level fault simulation based on the deductive algorithm, which was presented in this chapter, and a gate-level fault simulation tool from Turbo Tester [79] were carried out. Four circuits: *sosq*, *gcd16*, *diffeq*, *mult8x8* and one processor circuit *risc* were chosen for experiments. The experiments were run on Intel Core 2 CPU, 1.83 GHz, 2 GB of RAM, Windows XP.

In Table 2 the results of experiments are shown. The column 'vectors' reports the number of test vectors simulated. Column 'fault coverage' shows the fault coverage achieved according to the RTL bit coverage fault model. The following two columns document the run times of the proposed deductive algorithm and the Turbo Tester fault simulator, respectively. The last column presents the ratio indicating the speed-up of the proposed RTL fault simulator with respect to the gate-level approach. As it can be seen from the table, the speed-up tends to increase steadily with the run times reaching to about two orders of magnitude with the *diffeq* example.

Table 2 Fault simulation results

circuit	Vectors	fault coverage [%]	time [s]		time ratio (gate/RTL)
			RTL	gate-level	
gcd16	4000	100	3.28	29.23	8.91
mult8x8	4000	71.80	11.38	66.14	5.81
sosq	4970	78.15	12.58	66.36	5.28
risc	4000	100	18.19	366.3	20.14
diffeq	10000	100	37.02	3339.9	90.23

The method is based on the RTL bit coverage fault model, which has been proven to have a good correspondence with gate-level structural faults (see Table 2). Experiments on chosen benchmark circuits were carried out showing the feasibility of the new method for RTL fault simulation on the system model of high-level decision diagrams. Efficient data structures were implemented to speed up set operations in the deductive fault simulation algorithm. Experiments on RTL benchmark circuits show that up to two orders of magnitude shorter run-times are achieved with the method in comparison to gate-level fault simulation.

3.4 Chapter summary

The goal of this chapter was to present one of the contributions of this thesis, namely the RTL-based deductive fault simulation algorithm on HLDDs. The first part of this chapter gives an overview of fault simulation.

The second part of this chapter gives an overview of the deductive fault simulation algorithm on HLDDs. The proposed fault simulation algorithm uses the RTL bit coverage fault model. At the beginning, the general structure of the algorithm was presented, where a dedicated data structure *fault propagation record* is used throughout the algorithm. Furthermore, pseudo-code with step-by-step description was provided. Propagation of faults with the help of the proposed algorithm was shown on a small example of HLDD. Efficient internal data representation, which allows using bitwise set operations in order to achieve a high speed of simulation, was introduced. Also, analysis of the algorithm complexity was provided.

The third part of this chapter provided the experimental results, which were carried out on RTL benchmarks circuits. Experiments show that up to two orders of magnitude shorter run-times were achieved with the method in comparison to state-of-the-art gate-level simulation.

Chapter 4
HLDD-BASED CODE COVERAGE

In this chapter a method and a tool [59] for fast analysis of classical code coverage metrics, such as statement, branch and toggle coverage, are presented. High-Level Decision Diagrams (HLDD) model for efficient code coverage analysis is introduced and it is shown, how those classical coverage metrics map to HLDD constructs. Also, a set of HLDD manipulations [54] is proposed in order to generate diagrams that would allow more stringent code coverage measurement without sacrificing performance, i.e., computation time and memory requirements. The manipulation techniques include generation of HLDD-trees from HDL descriptions and two types of HLDD collapsing methods. Experiments on a set of ITC'99 benchmarks are presented.

4.1 Overview

With the increase in size and complexity of modern integrated circuits, it has become imperative to address critical verification issues in the design cycle. The process of verifying correctness of designs consumes between 60% and 80% of design effort [35]. Ensuring functional correctness is the most difficult part of designing a hardware system [76]. One possible way to verify the correctness of a design is by generating different test cases. Due to the fact that it is impractical to verify exhaustively all possible inputs and states of a design, the confidence level regarding the quality of the design must be quantified to control the verification effort. The fundamental question is: How do I know if I have verified or simulated enough? Verification coverage is a measure of confidence and it is expressed as a percentage of items verified out of all possible items. Different definitions of items give rise to different coverage measures or coverage metrics.

Various coverage metrics exist such as code coverage, parameter coverage, and functional coverage. In this work, only code coverage will be used, which provides insight into how thoroughly the code of a design is exercised by a suite of simulations. The main disadvantage of code coverage metrics lies in the fact that they only measure the quality of the test case in stimulating the implementation and do not necessarily prove its correctness with respect to the specification. On the other hand, code coverage analysis is a well-defined, well-scalable procedure and, thus, applicable to large designs.

Following Miller and Maloney [53], a large variety of code coverage metrics have been proposed, including statement coverage, block coverage, path coverage, branch coverage, expression coverage, transition coverage, sequence coverage, toggle coverage, etc. [43][76]. The *statement coverage* metric measures the percentage of code instructions exercised with respect to total instructions contained in the code by the program stimuli. *Toggle coverage* shows the percentage of bits toggling in the nodes in the design, i.e., how many bits change their state from 0 to 1 or vice versa. In the case of *branch coverage*, we measure the ratio of branches in the control flow graph of the code that are traversed under the set of stimuli. *Path coverage* measures the percentage of paths in the control flow graph is exercised by the stimuli. A potential goal of software testing is to have 100 % path coverage that implies branch and statement coverage. However, full path coverage is a very stringent requirement as the number of paths in a program is exponentially related to program size.

Current work is motivated by our previous encouraging research results obtained on HLDD based simulation [85] and test pattern generation [60].

4.2 Coverage metrics on HLDD

In order to analyze the quality of verification of hardware designs translated to HLDDs, three traditional coverage metrics were chosen and built in to the HLDD based simulation tool. These include statement coverage, branch coverage and toggle coverage. As it was mentioned above, the statement coverage measures the number of times every instruction is exercised by the program stimuli. Toggle coverage shows whether and how many times nodes in the design toggle, i.e. how many bits change their state from 0 to 1 or vice versa. In the case of branch coverage, we measure the number of times each branch in the control flow graph of the code is taken or not taken under the set of program stimuli.

4.2.1 Simulation algorithm

The basis for code coverage analysis in this work is a simulator engine relying on HLDD models. An algorithm has been implemented supporting both Register-Transfer Level and behavioral design abstraction levels. The description of the simulation algorithm is presented in subsection 2.1.2.3.

Some metrics were chosen and built into simulation engine in order to analyze the quality of verification of simulated design [59]. As the simulation is based on HLDDs, the possible structures to measure are nodes and edges. Therefore, we have built the node coverage, the edge coverage and the toggle coverage mechanism into the simulation engine. Node coverage measures, how many nodes are traversed during simulation against all nodes in the HLDDs. Edge coverage shows how many edges are traversed during simulation against all possible edges in the HLDDs. Toggle coverage presents how many bits are toggled from 0 to 1 and backwards in the variables labeling the nodes. The

```
for each diagram $g_y$ in the model
    $v_{current} = v_0$
    Let $x_{current}$ be the variable labeling $v_{current}$
    while $v_{current}$ is not a terminal node
        $v_{current}$.setFlag(Traversed); //set flag for node coverage
        if  is $x_{current}$ clocked or its DD is ranked after $g_y$ then
            Value = previous time-step value of $x_{current}$
        else
            Value = present time-step value of $x_{current}$
        end if
        $v_{next} \in \Gamma(v_{current})$, where $e_{active} = (v_{current}, v_{next}) \wedge c = Value$
        $e_{active}$.setFlag(Traversed);//set flag for edge coverage
        $v_{current} = v_{next}$
    end while
    if $x_{current}$ is a function then calculate a function;
    Assign Value $x_{current}$ to the DD variable $y$
    y.calculateCurrentToggleCoverage();//calculate current toggle coverage
                                       //for variable y
end for
```

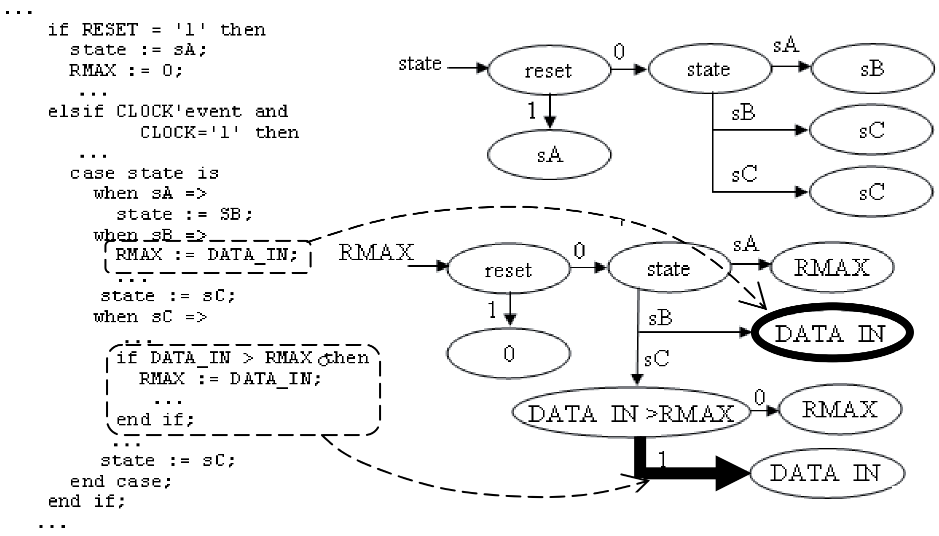
Figure 4.1 Pseudo-code of simulation algorithm for code coverage analysis

amount of code, which allows measuring this coverage is quite small. This code is highlighted with gray colors at the figure 4.1. Flag “traversed” is put to node or edge separately if it is traversed. A simple method *calculateCurrentToggleCoverage()* is used to calculate the toggle coverage at every iteration. Actual calculation of coverage is done after simulation process while generating report. While simulating the design, it is possible to mark with a program flag whether code coverage would be included into the simulation or not. As the overhead in data processing during simulation with code coverage analysis is small, the time overhead for simulation is also small (see experimental result part 4.4).

4.2.2 Mapping standard coverage metrics on HLDDs

Standard coverage metrics such as statement coverage and branch coverage can be mapped into metrics used in HLDDs: node coverage and edge coverage. Consider the example in figure 4.2. This is a small part of b04 from ITC99 benchmarks [34]. The part of the code is represented in VHDL and corresponding generated HLDDs are shown for variables *state* and *RMAX* (data register).

The *statement coverage* maps directly to the *node coverage* in HLDD representation, i.e. ratio of nodes $v_{current}$ traversed during the HLDD simulation



```
...
    if RESET = '1' then
      state := sA;
      RMAX := 0;
       ...
    elsif CLOCK'event and
            CLOCK='1' then
       ...
      case state is
        when sA =>
          state := SB;
        when sB =>
         RMAX := DATA_IN;
          ...
         state := sC;
        when sC =>
          ...
         if DATA_IN > RMAX then
           RMAX := DATA_IN;
             ...
         end if;
          ...
         state := sC;
      end case;
    end if;
   ...
```

Figure 4.2 b04 example: VHDL vs. HLDDs for variables ***“state”*** and “***RMAX***” to the total number of nodes in HLDDs presenting a circuit. For example, the statement *"RMAX := DATA_IN"* is represented by the terminal node *"DATA_IN"* surrounded by bold circle in the corresponding HLDD (see fig. 4.2). Covering all nodes in the HLDD model corresponds to covering all statements in the respective HDL. However, it can be noticed that usually total number of nodes is bigger than total number of statements. This is due to the fact that in HLDDs diagrams are generated to each data variable separately. Thus in the example (fig. 4.2) statement *"case state is"* is represented twice by the node *"state"* in diagram for variable *state* and in diagram for variable *RMAX*. The same holds for statement *"if RESET ='1' then"* is represented by node *"reset"*.

Similar to the statement coverage, branch coverage has also very clear representation in HLDD simulation. The ratio of every edge e_{active} activated in the simulation algorithm to total number of edges in HLDDs, i.e. edge coverage, constitutes to the HLDD branch coverage. For example, the branch coverage item corresponding to *"DATA_IN > RMAX = true"* in the VHDL code of the b04 design maps to the edge *e = (DATA_IN>RMAX, DATA_IN)* denoted by a bold arrow in the HLDD representing variable *RMAX* in Figure 4.2. Covering all edges in the HLDD model corresponds to covering all branches in the respective HDL. However, it can be noticed that usually total number of edges is bigger than total number of branches. This is due to the same fact that in HLDDs, diagrams are generated to each data variable separately.

HLDD toggle coverage is calculated similarly to traditional HDL toggle coverage. The information about toggling the bits for every variable in the HLDD model is collected.

4.3 HLDD manipulations for code coverage

The main contribution of this part is HLDD manipulation technique allowing efficient code coverage analysis [54]. In fact, if HLDD is generated for each output variable and the generation process is terminated at the primary input signals then code coverage analysis for the diagram will be equivalent to the path coverage metric. However, enumerating all the paths through a design is infeasible and it is easy to see that the corresponding HLDD may be of exponential size.

Therefore, another approach is adopted that differs from the traditional one of generating a diagram for each primary output. When representing systems by

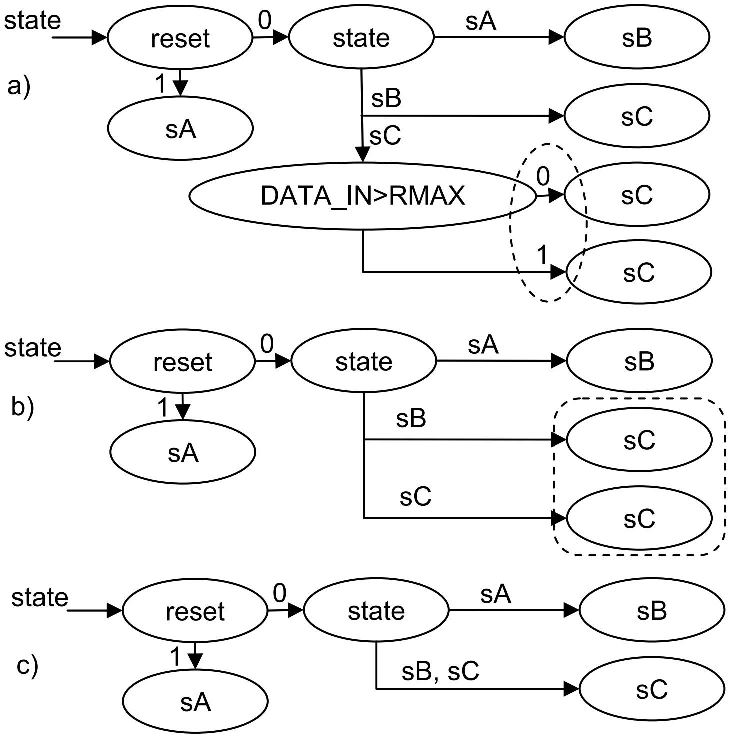


Figure 4.3 a) HLDD tree, b) reduced HLDD and c) minimized reduced HLDD

decision diagram models, a network of HLDD-s is implemented where each internal HDL variable has its corresponding HLDD. During the simulation in HLDD systems, the values of some variables labeling the nodes of a HLDD are calculated by other HLDD-s of the system. Such partitioning helps avoiding the node explosion problem of DD-s and keeps the size requirements for resulting HLDD systems acceptable.

The method proposed for generating reduced HLDDs and minimized reduced HLDDs suitable for code coverage analysis is similar to BDD reduction rules [61] and it consists of the following steps:

1. Generate a HLDD tree for each system variable
2. Follow the reduction rules:
    2.1. Eliminate all redundant nodes whose all edges point to the equivalent sub-graphs
    2.2. Share all equivalent sub-graphs

The above steps are explained by an example presented in Fig. 4.3, which depicts HLDD manipulations for the '*state'* variable of the b04 design presented in Fig. 4.2. We distinguish three types of diagrams: *HLDD tree*, *reduced HLDD* and *minimized reduced HLDD*. All these types are depicted in Fig. 4.3 and described below.

As the first step, a *HLDD tree* for variable *v* is generated by traversing the full control flow graph of the design and collecting the values assigned to *v* at each control step. If the value of *v* does not change at current control step then a terminal node with the present value of the variable will be created. Fig. 4.3a shows the *HLDD tree* generated for the variable *state* in b04.

Then, the first reduction rule is applied to eliminate nodes for which all successor nodes (in general case, succeeding sub-graphs) are identical. As a result a *reduced HLDD* is obtained (Fig. 4.3b).

Finally, we create a *minimized reduced HLDD* by uniting identical terminal nodes (Fig. 4.3c). In general case, uniting identical sub-graphs, which is application of the second reduction rule.

HLDD generation experiments on a set of ITC99 benchmarks [34] show that around 45-80% of nodes are removed by the reduction step from the initial HLDD tree. Further 40-60% of nodes will be eliminated by the minimization step.

According to experimental results presented below, we propose *reduced HLDD*-s as a suitable model for code coverage analysis because it provides for more stringent coverage metrics than *minimized HLDD*-s. At the same time it is a more compact representation than *full HLDD tree*s. Furthermore, in terms of speed of simulation reduced HLDD offers equal performance when compared to the minimized model. This is because of the fact that by both models the number of edges to be traversed is exactly the same. However, in full trees the number of diagram edges would be considerably higher.

4.4 Experimental results

In this section, experimental results [54],[59] for code coverage analysis on HLDDs are presented. First, comparative experiments between the HLDD-based code coverage analysis tool implemented in this thesis and a popular HDL commercial simulation tool were carried out. Experiments were run on a set of circuits from the ITC99 benchmark family [34] and the Greatest Common Divisor (GCD) example.

Table 3 shows the comparison between traditional code coverage assessment (statement, branch and toggle coverage) carried out by a state-of-the-art commercial HDL simulator and by the HLDD-based simulator (implementing node, edge, toggle coverage, respectively). The code has been exercised using random set of stimuli of different length. The experiments were run on AMD Athlon 64 Processor 3000+, 1.80 GHz, 2.00GB of RAM, Windows XP.

Table 3 coverage measure experiments on different simulators

Design (1)	**Test length (2)**	**Commercial HDL simulator**			**HLDD simulator**		
		simulation time, s		***ratio (4)/(3) (5)***	***simulation time, s***		***ratio (7)/(6) (8)***
		w/o coverage (3)	***w coverage (4)***		***w/o coverage (6)***	***w coverage (7)***	
b00	5000	0.0137	0.0173	1.263	0.046	0.048	***1.043***
	10000	0.0243	0.0311	1.280	0.099	0.100	***1.010***
b04	5000	0.0131	0.0166	1.267	0.051	0.053	***1.039***
	10000	0.0227	0.0300	1.322	0.106	0.107	***1.009***
b09	5000	0.0151	0.0262	1.735	0.010	0.012	***1.200***
	10000	0.0270	0.0483	1.789	0.023	0.024	***1.043***
GCD	5000	0.0135	0.0178	1.319	0.015	0.016	***1.067***
	10000	0.0240	0.0316	1.317	0.031	0.032	***1.032***

While there is no definite advantage of the speed of basic logic simulation of benchmarks to either of the tools it should be noted that the overhead of coverage checking in Modelsim is much higher than in the case of HLDDs (See columns (5) and (8)). When HLDDs have coverage calculation overhead for 10000 patterns in a 1 to 4 % range, the commercial simulator uses 28 up to 78 % extra time.

Second, experiments with different HLDD manipulations were carried out. Table 4 presents the characteristics of the different HLDD representations introduced in previous sub-chapter. The columns *tree, red.* and *min* show the number of nodes/edges in HLDD tree, reduced HLDD and minimized reduced HLDD models, respectively. From the Table 4, it can be seen that around 45-80 % of nodes are removed by the reduction step from the initial HLDD tree. Further 40-60 % of nodes will be eliminated by the minimization step.

Table 4 Characteristics of different HLDD manipulations

Design	**Number of nodes**			**Number of edges**	
	tree	***red.***	***min***	***red.***	***min***
b01	267	57	30	52	52
b02	48	26	16	24	24
b06	440	116	47	83	83
b09	125	69	44	62	62

Table 5 presents code coverage analysis comparing statement coverage and branch coverage assessment results on *reduced HLDD-s* (*red.*), on *minimized reduced HLDD-s* (*min*) and on a well-known commercial tool using the same set of input stimuli for all three models. As it can be seen from the experiments, the *reduced HLDD* model always achieves the best (i.e. most stringent results) of all three. The *minimized reduced HLDD* has the poorest outcome for node coverage and traditional HDL simulator is the weakest for measuring branch coverage in most cases.

Table 5 Comparison of code coverage analysis results

Design	**Stimuli,** (vectors)	**Node coverage,** (%)			**Edge coverage,** (%)		
		red. HLDD	***min. HLDD***	***VHDL***	***red. HLDD***	***min. HLDD***	***VHDL***
b01	14	86.0	100	93.8	74.2	84.6	88.9
	23	96.5	100	100	90.3	100	100
b02	10	92.3	100	96.3	91.7	91.7	93.8
	14	100	100	100	100	100	100
b06	11	80.2	100	85.5	79.3	89.2	87.5
	52	98.3	100	100	98.2	100	100
b09	23	87.0	100	100	85.9	87.1	100
	33	100	100	100	100	100	100

4.5 Chapter summary

The goal of this chapter was to present one of the contributions of this thesis, namely code coverage analysis based on HLDDs. First, a set of code coverage metrics were built into the HLDD-based simulator engine. Then, manipulations on HLDDs were done in order to find the best way to represent the circuit for code coverage analysis.

The first part of this chapter presented a new approach to analyzing validation code coverage metrics using High-Level Decision Diagrams. A technique was proposed, where HLDD-based simulation was extended to support code coverage analysis. It was shown how classical code coverage metrics can be mapped to HLDD constructs. Experiments on ITC99 benchmark circuits indicated the feasibility of the proposed approach.

The second part of this chapter presented a set of straight-forward manipulations on High-Level Decision Diagrams to support code coverage analysis. Experiments on ITC99 benchmark circuits showed that the *reduced HLDD* model proposed in this thesis offers higher accuracy in statement and branch coverage analysis than traditional models. The gain in accuracy is achieved only with a slight increase in memory requirements. The simulation times for all three models are nearly identical.

Chapter 5
HLDD-BASED OBSERVABILITY COVERAGE

In this chapter, the observability coverage metric based on the toggle coverage metric is presented. The description of this metric is provided. Propagation of bugs to the outputs for this observability coverage metric is based on the deductive RTL fault simulation algorithm. The integration of this metric with developed methods in the framework of this thesis is provided. Results of experiments on chosen set of benchmarks are also presented.

5.1 Overview

Verification of the HDL designs is not a trivial task. Usually it can be stated that this process is somewhat subjective. In order to make it more objective, an automated process for test vector generation and test vector analysis must be created. The end of the verification process is seen, when a certain degree of confidence is achieved with the help of coverage metrics and it is proved that the product is free of significant errors.

The most common RTL coverage metrics are adopted from the software testing metrics, such as statement coverage, branch coverage, and condition coverage [5]. However, in hardware description languages these coverage metrics have got a little bit different content. This is because software is different from hardware. First, hardware is mostly concurrent in comparison with software that dramatically degrades the path coverage metric widely used in software. Also, at description levels higher than gate-level there usually exists combination of behavioral and structural description styles that prevents the use of techniques suitable for control and data-oriented circuits only. Timing is also not considered in software. Moreover, software metrics only consider reachability of conditions that corresponds to fault controllability. They do not explicitly check weather erroneous effects propagate to the observation points. Bugs may remain undetected even if they are activated during the simulation. Therefore, an observability measure is required and the above mentioned metrics may overestimate the validation extent.

Keeping track of covered lines of code does not generally reflect if the respective items influence the primary outputs of the system. The quality of validation is low when only code coverage items corresponding to the internal lines of the system are exercised but not propagated to the system outputs. Furthermore, while the general function of the system is specified at the outputs, the internal signals may be difficult for the designer or verification engineer to comprehend and verify.

In testability arena at the gate-level, traditional observability analysis is based on the well-known stuck-at-fault model [10],[32],[68]. At RTL Fallah et al. [26] propose observation coverage in their method called OCCOM, where simplified fault grading is carried out in order to assess, which code items have been covered and propagated to an observable output. They show that 100 % statement coverage corresponds to 60-80 % observable coverage in the worst case. However, the OCCOM method [26] and its recent improvements [25],[27] are based on representing the effect of an error by a tag that can propagate through the circuit according to a set of rules similar to the D calculus. The main problem of the method is that it over-simplifies the fault-effect propagation. In this work, we propose an alternative approach, where calculation on HLDDs is used for observability analysis.

The OCCOM method [26] is the first code coverage metric that considers the essential observability issue in RTL designs. Based on this approach different applications were worked out. One of the most widespread is ATPG (Automatic Test Pattern Generator) [18][95]. In [18] observability coverage metric is used as the basis for a fitness function of ATPG. Special tags are attached to internal signals as in OCCOM, simulation is done and propagation of tags is observed in the outputs. The tags used in [18] have coarse granularity. In addition, a commercial simulator is used to watch the effect of propagation. Moreover, different modifications of OCCOM were worked out and compared with fault coverage at gate-level. In [95], the OCCOM method is improved by adding U-sign to more accurately tag the propagation effect. Based on the observability coverage metric test vectors are generated and it is shown that tags propagation is reasonable for ATPG, because tags coverage helps to generate efficient test vectors set. These works prove that observability coverage metric is very useful and essential in verification.

The OCCOM method was commercialized by Synopsys Inc. [75] and became Observability-Based Coverage (OBC), a part of the VCS™ simulator. The official definition of OBC is "stimulation of a line whose effect can be subsequently observed at a user-specified point." In essence, OBC will report that a line is not covered if it could be removed from the source code without impacting the simulation. Tensilica [77] reports: “Because OBC provides a more accurate measurement of the completeness of verification stimulus on the design code, it allows us to produce higher quality designs compared to traditional coverage tools”[65]. OBC appears to have been discontinued as of VCS™ simulator 2006.06 release, just at about the time when Synopsys Inc. received a patent for the technique [88].

Dynamic analysis, like the techniques proposed in [36],[50],[96] can determine how thoroughly the test stimuli examine design code at RTL and subsequently propagate potential design errors to the outputs. Probabilistic observability measure and its efficient computation algorithm were developed in [37]. For every variable in the design it is possible to find statistically probabilistic value of observability calculated by special rules. The threshold value is set in order to report variables with low propagation probability. The method takes time for calculations in comparison with easier metrics.

In [48] observability-enhanced code coverage metric is developed which shows covered only when there is some degree of confidence that any error were at least propagated to the checker and ideally detected. This work summarizes previous works [46][47][49]. In [46] tags are injected into compiled C code, which represent high-level circuit description, in order to observe the propagation of the tags on the outputs. A mutation coverage tool [49] was developed for C++/SystemC designs, which supports previous solutions. Also analytical technique, Coverage Discounting, was developed that uses fault insertion to add checker sensitivity to existing functional coverpoints [47]. This is good complex solution for high-level design descriptions.

The RTL fault grading presented in [51] injects faults only into input and internal variables. The injection is done by modification of the original code and then fault simulation is performed. Only single stuck-at-fault is injected per RTL variable and simulation is done twice for optimistic (without taking into account fan-outs) and pessimistic (with taking into account fan-out branch of RTL variable), then average is calculated. Similar approach is supported also by the method proposed in this thesis as when constructing HLDDs, one variable could be represented by many nodes, depending on its logical connection with other data. Thus, every node would be processed separately, which is similar to injection of a fault to each fan-out of a variable.

Validation Vector Grade (VVG) described in [78] is an observability-based validation metric, which can be used for early testability analysis at the RT level. The method is very well examined and proven based on Verilog code modification for fault injection. Fault simulation is done using *Verifault-XL*.

Our approach is based on HLDDs. The code coverage analysis described in previous chapter is done on the same design representation observability coverage that is going to be presented here is based on the same structure.

5.2 HLDD-based observability coverage

As it has already been stated above, commonly used code coverage metrics only point controllability of certain items of the RTL design from primary inputs while ignoring their observability at outputs or observation points. A metric, which takes into account observability, gives more information to verification engineer and allows detection of testability problems at early stage of a design cycle.

In previous chapter of this thesis common code coverage metrics were built into our simulation engine. In this chapter new observability-based metric is presented, this metric is built into the same simulation system working on HLDD design representation. The metric extends possibilities of code coverage analysis presented in chapter 4. The algorithm for observability coverage metric is based on the RTL fault simulation algorithm presented in chapter 3. Thus, approach presented in this part unites ideas presented in two previous chapters and extends them to a new observability coverage metric.

HLDD-based observability coverage metric shows the percentage of bugs inserted into a RTL design represented by HLDD and observed at the outputs/observation points with respect to total number of inserted bugs in the design simulated by stimuli. The bug can have different meanings. In our case, we have examined bugs, which is inverted bit of a variable. This is very similar to our fault simulation at RTL. Actually, the same algorithm of fault simulation at RTL proposed in chapter 3 is used for our observability coverage metric. This way effectiveness of verification and proposed/generated test vectors can be evaluated directly at RTL. Analysis of not observed bugs can help to improve test vectors set and/or even circuit design.

5.2.1 Integration into a tool

The observability coverage metric is built into the HLDD-based framework tool created in this thesis, which is based on HLDD fast simulation. This tool supports the following applications: RTL fault simulation (described in chapter 3), code coverage analysis at RTL (described in chapter 4) and observability coverage metric (topic of current chapter). Last application actually combines both algorithms presented in chapters 3 and 4. The general view of the algorithms/methods used in the tool is presented in the figure 5.1. Developed methods and tasks are colored in grey. In this figure one can see that code coverage analysis is one of the verification tasks, where observability coverage is part of a code coverage analysis. As can be seen from the picture 5.1 only 2 algorithms are core of the tool, both based on HLDD design representation. These algorithms are HLDD-based simulation and RTL deductive fault simulation. HLDD-based simulation initially is applied for code coverage analysis. RTL deductive fault simulation algorithm is used for RTL fault simulation. Adding observability coverage metric application requires to process toggle coverage data received from code coverage analysis results. This toggle coverage data is inverted into bugs and RTL deductive fault simulation algorithm is used for the propagation of inserted bugs and for observing them on the outputs/observation points. Thus, new application "observability coverage metric" unites all the work presented above.

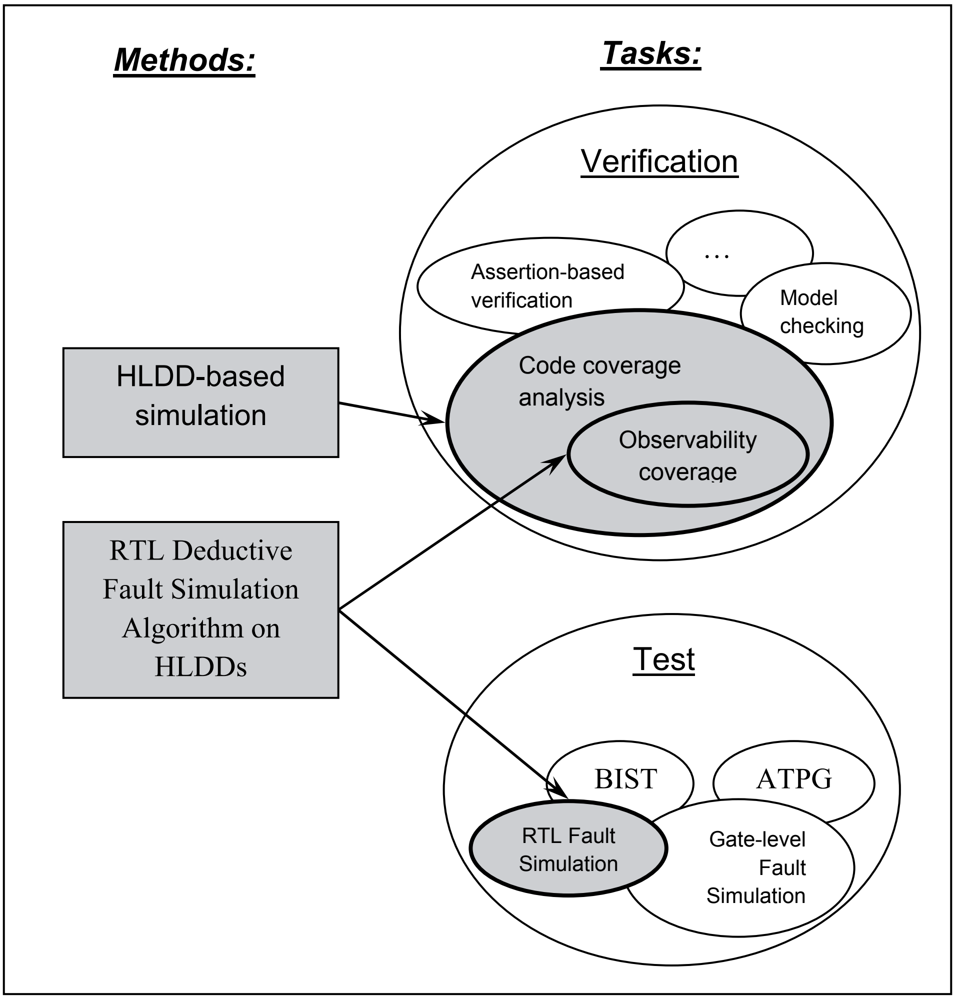


Figure 5.1 Developed methods & tasks in general verification and test steps

Instead of inserting the tags as in OCCOM method [26] for observability, bugs represented by inverted bits, are inserted. First, our bugs have finer granularity in comparison with tags, which represent difference of variable value. Also, algorithm used for bug propagation is the same as for fault simulation without simplifications, and this algorithm is fast due to HLDD design representation. In general observability coverage is compared with statement and branch coverage (based on most papers presented in the overview). From our point of view, to achieve 100% of statement coverage is considerably easy in comparison with other code coverage metrics. To change one statement is possible by many ways, further propagating the change (tag/bug). Thus, this is one-to-many relation and comparison of statement coverage and observability coverage is not exact. Toggle coverage actually includes statement coverage, because if at least one bit of a variable is toggled then the statement containing this variable is covered as well. But when toggle coverage metric and observability coverage metric are taken then we have one-to-one mapping of items to cover. Therefore, it was decided to compare observability coverage metric with toggle coverage metric trying to reach the condition when statement and branch coverage reach 100%. Taking toggle coverage results after HLDD-based design simulation under stimuli, this information is given to deductive fault simulation algorithm in order to find observability at outputs of toggled bits. As statements consist of variables the observability coverage metric built into our tool can also be analyzed to show statement observability coverage.

The main steps for observability coverage analysis are:

- Simulate design under stimuli to obtain code coverage results;
- Translate toggle coverage results into bugs;
- Insert bugs (faults) into design and run deductive fault simulation algorithm under the same stimuli;
- Analyze propagated bugs to the outputs, calculate observability coverage in percentage;
- Show unobservable bugs;

As the deductive fault simulation algorithm allows analysis of all faults in parallel, all observability coverage items are also analyzed simultaneously.

5.2.2 Simulation results

In this section experimental results for observability coverage analysis on HLDDs are presented. Comparative experiments between the code coverage analysis (in particular node coverage, edge coverage and toggle coverage) based on the HLDD simulation algorithm, presented in chapter 4, and observability coverage analysis based on HLDD deductive fault simulation algorithm, presented in this chapter, are carried out. For experiments the following circuits were chosen: *sosq, gcd16, diffeq, risc*. The Greatest Common Divisor (*gcd16*) and the Differential Equation (*diffeq*) are examples from the HLSynth92 academic benchmarks suite [33]. The design *risc* is a processor example from a FUTEG research project. The test stimuli for the academic benchmarks were generated by a hierarchical test pattern generator Decider [60]. The experiments were run on Intel Core 2 CPU, 1.83 GHz, 2 GB of RAM, Windows XP.

In Table 6 the results of experiments are shown. All coverage metrics for the benchmark were measured under the same set of test stimuli. Columns show node coverage, edge coverage, toggle coverage and observability coverage in percentage.

Table 6 Comparative experimental results for observability coverage analysis

Circuit	Node coverage	Edge coverage	Toggle coverage	Observability coverage
Sosq	100%	100%	56.3 %	**54.3 %**
Gcd16	100%	100%	100 %	**100 %**
diffeq	100%	100%	100 %	**90.4 %**
RISC	100%	100%	100 %	**100 %**

As it can be seen from the table, observability coverage is less than toggle coverage for 2 circuits (*sosq* and *diffeq*), i. e. some bugs are non-observable at the outputs. For *sosq* circuit there are 2% of non-observable bugs and for *diffeq* the percentage is almost 10. Other circuits are fully covered. These results show that with observability coverage metric it is possible to get more information about the quality of the stimuli and draw attention to the places in circuits where bugs are non-observable.

5.3 Observability coverage metric discussion

The observability coverage metric can be used for testability property analysis at RTL stage of design cycle. If observability coverage is less than 100% it is due to at least one of the following reasons:

- Insufficient simulation test vectors set;
- Poor observability of internal nodes.

In the first case better stimuli test suite should be worked out. In the second case the designer should modify the circuit design for better testability. Therefore, the observability coverage metric can be a good indicator for RTL testability as well as a good predicator for gate-level fault coverage (because observability coverage item is closely related to stuck-at-fault model).

Even though a block of code is fully exercised by usual code coverage metrics, the verification effort may be meaningless unless results of the activated items propagate to the outputs. The observability coverage metric allows observing changes of all internal RTL variables at the outputs. Thus problem of propagation is solved.

Assume that a certain observability coverage is obtained for RTL circuit design. Assume that some items are not propagated to the outputs. By controlling non-propagated items one by one with the same suite of stimuli it is possible to determine blocking statement(s) (or statement(s) which mask the bug). The values of the variables in the blocking statement(s) should be changed in order to propagate bug further. The designer should find the legal test suite to change blocking statement(s) variable values. If designer is unable to come up with such test stimuli, then the circuit design may have redundancy or there is a design error.

Designs become bigger and more complicated, thus self-checking verification environments are very welcome. Verification suites consist of input stimuli and expected outputs to these stimuli. Self-checking verification environment allows monitoring outputs/observation points during the simulation and report any difference from expected results. The observability coverage metric provides self-checking verification environment to detect design bugs.

Observability coverage metric allows giving indication of whether certain logic is functionally used during simulation. For example, enable signal of a counter should be reported covered only if it is properly exercised in a functional way. If counter is in a reset state and the enable signal is toggled, it would not be observed at the counter outputs, thus it would not be covered by the observability coverage metric.

It is also possible to build a diagnosis table, which is filled with results of the observability coverage metric. This is similar to building diagnosis tables at the gate-level. Observability coverage metric item has good correspondence to the stuck-at-fault model, but it is applied to the designs at RTL, thus diagnosis at this level can be done for searching errors in the design.

For a good coverage metric it has to be easy to write, easy to understand, cheap to implement and have a good ability to track the types of errors most likely to happen. The stuck-at-fault model conforms to all these requirements and that’s why is suitable for fault modeling. Observability coverage metric is very similar to the stuck-at-fault model and conforms to all the above listed requirements, thus we can derive that it is a suitable coverage metric. Every time a design is changed, all test cases must be rerun and coverage metric must be recalculated.

It would be beneficial to combine functional coverage and observability coverage. If some bugs of a design are not covered according to observability coverage, i.e. erroneous value of a bug is not propagated to the outputs, specific functional test suite can be specified for this bug. Also, if functional coverage gives positive feedback and observability coverage negative, then total report: “is not covered”. The bug is not propagated to the outputs and better test suite must be worked out.

5.4 Chapter summary

In this chapter observability coverage metric was presented. The objects for this metric are RTL designs represented by HLDDs. The metric allows measuring not only activated bugs of the design but also the propagation of these bugs to the observable outputs. The observability coverage metric allows making better analysis of the test stimuli and design itself.

The observability coverage metric was built into the HLDD-based simulation engine using for propagation RTL deductive fault simulation algorithm presented in chapter 3 and for bugs insertion the toggle coverage analysis presented in chapter 4. Experiments were carried out on RTL benchmark circuits showing that the observability coverage metric gives information of non-propagated bugs in the design under the test stimuli. An analysis of observability coverage metric was presented as well its usefulness was discussed.

Chapter 6

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the thesis and discusses possible directions for future research.

6.1 Conclusions

This thesis presents several approaches targeting RTL fault simulation and RTL code coverage analysis. All approaches exploit advantages of high-level decision diagrams representation model.

A deductive fault simulation algorithm based on HLDDs for RTL designs was proposed in this thesis. The initial deductive fault simulation algorithm, proposed by Armstrong for gate-level designs, was brought to RTL where not only bits are taken into account when making a decision for fault lists propagation, but also variables and arithmetic operations. The algorithm was successfully implemented into a tool using HLDD-based simulation engine. Analysis of algorithm characteristics is provided, which shows its quite good properties. The algorithm complexity depending on a number of graphs nodes is $O(n^2)$. For fault simulation, the RTL bit coverage fault model is applied, which has proven by the set of experiments to provide a good correspondence with gate-level structural faults [29].

An efficient data structure was proposed and implemented to speed up set operations in the deductive fault simulation algorithm at RTL. The faults are coded with bits the way that it would be possible to make fast bitwise set operations with fault lists. As a result, the speed of RTL fault simulation was increased.

Experiments for fault simulation algorithm were carried out on RTL benchmark circuits and show feasibility of proposed approaches. Up to two orders of magnitude shorter run-times were achieved with the method in comparison to state-of-the-art gate-level fault simulation.

Fast HLDD-based simulation was extended to support code coverage analysis, such as node coverage, edge coverage, toggle coverage. A method of mapping traditional code coverage metrics to High-Level Decision Diagrams was described. Experiments on ITC99 benchmark circuits indicate the feasibility of the proposed approach. The time overhead of coverage checking in a commercial simulator was much higher than in the case of HLDDs. When HLDDs had coverage calculation overhead in a 1 to 4 % range, the commercial simulator used 28 up to 78 % extra time.

Manipulations on High-Level Decision Diagrams were developed to find the best representation of HLDDs for code coverage analysis. Experiments on ITC99 benchmark circuits showed that the *reduced HLDD* model proposed in this thesis offered higher accuracy in statement and branch coverage analysis than traditional models. The gain in accuracy was achieved only with a slight increase in memory requirements. The simulation times for all three models were nearly identical.

An observability coverage metric based on the bit coverage fault model was proposed, which take into account not only controllability of the internal point of the design, but also observability at the outputs. Observability coverage metric was built into the tool, based on HLDD-based simulation engine, where for bugs propagation, RTL deductive fault simulation algorithm developed in this thesis is used. Bugs were inserted relying on the toggle coverage metric, i.e. toggled bit was inverted into a bug and the bug was propagated to the output. Experiments were carried out on RTL benchmark circuits and shown that the observability coverage metric allows finding bugs which are activated but not propagated to the outputs under test stimuli. For circuits *sosq* and *diffeq*, the percentage of unobservable bugs was 2 and 10, respectively.

6.2 Future work

Experiments presented in this thesis are carried out on RTL benchmark circuits of relatively small size. It would be very beneficial to make experiments on the circuits of bigger sizes, especially on some industrial benchmarks. This would need to investigate the scalability of the proposed techniques.

Deductive fault simulation algorithm implementation uses a lot of recursive functions. The benchmark circuits have been examined by this algorithm did not demand additional management of resources. However, for bigger circuits the memory requirements certainly need investigation.

The tool presented in the framework of this thesis could be further developed.

It would be beneficial to include other coverage metrics into the presented tool. One of the comprehensive metrics could be a metric that takes into account not only controllability and observability of the item, but also insertion of checkers to the observable points. This technique provides a measure of the quality of verification allowing detection of functional errors. This would be one of the possibilities to combine structural and functional verification.

References

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

[2] Akers, S. B., “Binary Decision Diagrams”, *IEEE Trans. on Computers*, Vol. 27, 1978, pp.509-516

[3] Armstrong, D.B., “A deductive method for simulating faults in logic circuits”, *IEEE Trans. On Computers*, Vol. 11, No. 2, 1972, pp. 198-207

[4] Bailey, Brian, “The Great EDA Cover-up”, *EE Times*, 2007

[5] Beizer, B., “Software Testing Techniques, 2nd ed.”, *New York, Van Nostrand Rheinold*, 1990

[6] Bhatnagar, H.; “Advanced ASIC Chip Synthesis”, *Kluwer Academic Publishers*, Boston, MA., 1999, pp. 2-4

[7] Bombieri, N.; Fummi, F.; Guarnieri, V., “Accelerating RTL Fault Simulation through RTL-to-TLM Abstraction”, *European Test Symposium ETS’2011*, May 2011, pp. 117-122.

[8] Bosio, A.; Di Natale; G., “LIFTING: A Flexible Open-Source Fault Simulator”, *ATS’08*, 2008, pp. 35-40

[9] Breuer, M. A.; Friedman A. D., “Diagnosis and Reliable Design of Digital Systems”, *Computer Science Press, New York*, 1976

[10] Brglez, F., “In Testability of Combinational Networks”, *IEEE International Symposium on Circuits and Systems*, 1984

[11] Bryant, R. E., “Graph-based algorithms for Boolean function manipulation”, *IEEE Trans. on Computers*, Vol. C-35, No. 8, August 1986, pp. 677-691

[12] Bryant, R. E.; Chen Y.-A., "Verification of arithmetic functions with binary moment diagrams", *Proc. 32nd ACM/IEEE DAC*, June 1995

[13] Bushnell, Michael L.; Agrawal, Vishwani, D., "Essentials of Electronic Testing: for Digital, Memory & Mixed-Signal VLSI Circuits", *Spinger*, 2000

[14] Carter, Hamilton B.; Shankar, G. Hemmady, "Metric Driven Design Verification: An Engineer's and Executive's Guide to First Pass Success", *Springer, New York*, 2007

[15] Chang, H. Y.; Chappell, S.G., "Deductive Techniques for Simulating Logic Circuits", *Computer, Vol. 8, Issue 3*, 1975, pp. 52-59

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

[17] Clarke, E. M.; McMillan, K. L.; Zhao, X.; Fujita, M.; Yang, J., "Spectral transforms for large Boolean functions with applications to technology mapping", *Proc. of the 30th ACM/IEEE DAC*, June 1993, pp. 54-60

[18] Corno, F.; Reorda, M. S.; Squillero, G., "High-level observability for effective high-level ATPG", *Proc. in VLSI Test Symposium, 2000*, pp. 411-416

[19] Corno, F.; Cumani, G.; Reorda, M. S.; Squillero, G., "An RT-level Fault Model with High Gate Level Correlation", *High-Level Design Validation and Test Workshop, 2000. Proceedings*, pp. 3-8

[20] Davis, B., "The Economics of Automatic Testing", *McGraw-Hill, London*, 1982

[21] Deniziak, S.; Sapiecha, K., "Fast High-level Fault Simulator", *ICECS 2004, proceedings, 2004*, pp. 583-586.

[22] Devadas S.; Ghosh, A.; Keutzer, K., "An Observability-based code coverage metric for functional simulation", *in Proc. International Conference Computer Aided Design*, 1996, pp. 418-425

[23] Devadze, Sergei, "Fault Simulation of Digital Systems", *PhD thesis, TUT press*, 2009

[24] EU's 6th Framework Programme research project VERTIGO web page, [*http://www.vertigo-project.eu*]

[25] Fallah, F., "Coverage directed validation of hardware models", *Ph.D. disseratation, M.I.T., Cambridge, MA*, 1999

[26] Fallah, F.; Devadas, S.; Keutzer, K., „OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification." *Proc. Design Automation Conference*, pp.152-157, 1998

[27] Fallah, F.; Devadas, S.; Keutzer K., „OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification.”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 2001, pp.1003-1015

[28] Federici, D.; Bisgambiglia, P.; Santucci, J.-F., "High level fault simulation: experiments and results on ITC'99 benchmarks," pp.118, *Fifth IEEE International High-Level Design Validation and Test Workshop (HLDVT'00)*, 2000

[29] Fummi, F.; Marconcini, C. and Pravadelli, G., “Logic-level mapping of high-level faults”, *Integration, the VLSI Journal*, Volume 38, Issue 3, January 2005, pp. 467-490

[30] FUTEG research project webpage, [*http://www.inf.mit.bme.hu/en/research/projects/functional-test-generation-and-diagnosis-futeg*]

[31] Goders, N. and Kaibel, R., “PARIS: A parallel pattern fault simulator for synchronous sequential circuits”, in *ICCAD*, pp. 542-545,1991

[32] Goldstein, L. H., “Controllability/observability analysis of digital circuits”, *IEEE Transaction on Circuits Systems*, 1979

[33] HLSynth92 benchmarks family webpage, Collaborative Benchmarking and Experimental Algorithmics Lab, [*http://www.cbl.ncsu.edu:16080/benchmarks/HLSynth92/*]

[34] ITC'99 Benchmarks webpage, CAD Group, Politecnico di Torino, [*http://www.cad.polito.it/tools/itc99.html*]

[35] ITRS, “International Technology Roadmap for Semiconductors report”, 2007 Edition, Design section, [*www.itrs.net*]

[36] Jiang, T.-Y.; Liu, C.-N. J.; Jou J.-Y., “An observability measure to enchance statement coverage metric for proper evaluation of verification completeness”, *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2005

[37] Jiang, T.-Y.; Liu, C.-N. J.; Jou, J.-Y., “Observability analysis on HDL Descriptions for Effective Functional Validation”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2007, pp. 1509-1521

[38] Karputkin, Anton, “Formal Verification and Error Correction on High-Level Decision Diagrams”, *PhD thesis, TUT press*, 2012

[39] Kassab, Mark; Rajski, Janusz; Tyszer, Jerzy, “Hierarchical Functional-Fault Simulation for High-Level Synthesis”, *ITC 1995*, pp. 596-605

[40] Keyes, R.W., “The Impact of Moore’s Law”, *IEEE Solid-State Circuits Issue*, Sept 2006

[41] Kuebschull, U.; Schubert, E.; Rosenstiel, W., “Multilevel logic synthesis based on functional decision diagrams”, *Proc of the IEEE EDAC*, March 1992, pp. 43-47

[42] Lai, Y.-T.; Pedram, M.; Vrudhula, S. B., “FGILP: An integer linear program solver based on function graphs”, *Proc. of the IEEE/ACM ICCAD*, November 1993, pp. 685-689

[43] Lam, William K., “Hardware Design Verification: Simulation and Formal Method-Based Approaches”, *Prentice Hall, Pearson*, 2005

[44] Lee, H.K. and Ha, D. S., “HOPE: An efficient parallel fault simulator for synchronous sequential circuits”, in *DAC*, pp. 336340,1992

[45] Lee, J.; Rudnick, E. M. and Patel, J. H., “Architectural level fault simulation using symbolic data”, in *European Conference on Design Automation*, pp. 437-442, 1993

[46] Lisherness, P.; Cheng, K.-T., “An Instrumented Observability Coverage Method for System Validation”, *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2009, pp. 88-93

[47] Lisherness, P.; Cheng, K.-T., “Coverage Discounting: A Generalized Approach for Testbench Qualification”, *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2011

[48] Lisherness, P.; Cheng, K.-T., “Improving Validation Coverage Metrics to Account for Limited Observability”, *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012

[49] Lisherness, P.; Cheng, K.-T., “SCEMIT: A SystemC Error and Mutation Injection Tool”, *Design Automation Conference (DAC)*, 2010

[50] Ly, T.; Liu, L. yi; Zhao, Y.; H. wei Li; X. wei Li, “An observability branch coverage metric based on dynamic factored use-define chains”, *Asian Test Symposium (ATS)*, 2006

[51] Mao, W.; Gulati, R., “Improving Gate Level Fault Coverage by RTL Fault Grading”, in *Proc. International Test Conference*, 1996, pp. 596-605

[52] Menon, Premachandran R.; Chappell, Stephen G., “Deductive Fault Simulation with Functional Blocks”, IEEE Trans. On Computers, Vol. C-27, No. 8, August 1978, pp. 689-695

[53] Miller, J. C.; Maloney, C. J., “Systematic Mistake Analysis of Digital Computer Programs”, *Comm. ACM*, 1963, pp. 58-63

[54] Minakova, K.; Reinsalu, U.; Chepurov, A.; Raik J.; Jenihhin M.; Ubar, R.; Ellervee, P., “High-Level Decision Diagram Manipulations for Code Coverage Analysis”, *The 11th Biennial Baltic Electronics Conference (BEC’08)*, 2008, pp. 207-208

[55] Moore, G., “Cramming More Components onto Integrated Circuits”, *reprint from IEEE proceedings on Electronics*, Vol. 38, No.8, 1965

[56] Mourad, Samina; Zorian, Yervant, “Principles of Testing Electronic Systems”, *A Wiley-Interscience publication*, 2000

[57] Niermann, T.M.; Cheng, W. T. and Patel, J. H., “PROOFS: A fast, memory efficient sequential fault simulator”, *IEEE TCAD, vol. 11, n. 2*, pp. 198-207, February 1992

[58] OpenCores design repository webpage, [*http://www.opencores.org*]

[59] Raik, J.; Reinsalu, U.; Ubar, R.; Jenihhin, M.; Ellervee, P. “Code Coverage Analysis using High-Level Decision Diagrams”, *Proceedings of the 11th IEEE Workshop on Design and Diagnostics of Electronic Systems (DDECS)*, 2008

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

[61] Raik, Jaan, "Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams", *PhD thesis, TTU press*, 2001

[62] Raik, J.; Jenihhin, M.; Chepurov, A.; Reinsalu, U.; Ubar, R., “APRICOT: a Framework for Teaching Digital Systems Verification”, *19th EAEEIE Annual Conference,* pp. 1 - 6

[63] Reinsalu, U.; Raik, J.; Ubar, R.; Ellervee, P., “Fast RTL Fault Simulation Using Decision Diagrams and Bitwise Set Operations”, *Proceedings of 26th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems* pp.164 - 170, 2011

[64] Reinsalu, U.; Raik, J.; Ubar, R., “Register-Transfer Level Deductive Fault Simulation Using Decision Diagrams”, *Proceedings of the 12th Biennial Baltic Electronic Conference BEC2010*, Oct. 2010, Tallinn, Estonia, pp. 193 - 196.

[65] Reynaud, “Code Coverage techniques – a hands-on view”, *EE Times* 09/12/2002

[66] Sanghavi, Alok, “What is formal verification?”, *EE Times-Asia*, [*http://www.eetasia.com*]

[67] Sesuh, S. and Freeman, D. N., On improved diagnosis program, *IEEE Trans.Electron. Comput.*, EC-14(1), 76–79, 1965

[68] Seth, S. C., Pan, L., Agrawal, V. D., “PREDICT: Probabilistic estimation of digital circuit testability”, *International Fault-Tolerant Computer Symposium*, June 1985

[69] Shen, Li, “VFSim: concurrent fault simulation at register transfer level”, *Journal of Computer Science and Technology, Volume 20, Issue 2*, March 2005

[70] Sinanoglu, Ozgur; Orailoglu, Alex, “RT-level Fault Simulation Based on Symbolic Propagation”, *VTS 2001*, pp. 240-245

[71] Smith, S.P.; Mercer, M. R.; Underwood, B., “D^3FS: a Demand Driven Deductive Fault Simulator”, Test Conference, 1988. *Proceedings. New Frontier in Testing, International,* pp. 582-592

[72] Srinivasan, A.; Kam, T.; Malik, S.; Brayton, R., “Algorithms for discrete function manipulation”, *Proc. IEEE/ACM ICCAD*, November 1990, pp. 92-95

[73] Suma, M.S.; Gurumurthy, K.S., “Fault Simulation of Digital Circuits at Register Transfer Level”, *International Journal of Computer Applications, Vol. 30, No. 7*, 2011, pp. 1-5

[74] Sureshkumar, P.R.; Jacob, James; Srinivas, M.K.; Agrawal, Vishwani D., “An Improved Deductive Fault Simulator”, Proceedings on the 7th International Conference on VLSI Design, 1994, pp. 307-310

[75] Synopsys Inc. Homepage [*http://www.synopsys.com/mhome.aspx*]

[76] Tasiran, S.; Keutzer, K., “Coverage metrics for functional validation of hardware designs.” *Design & Test of Computers*, IEEE, Volume 18, Issue 4, Jul-Aug. 2001, pp. 36-45

[77] Tensilica Homepage [*http://www.tensilica.com/*]

[78] Thaker, P. A.; Agrawal V. D.; Zaghloul M. E., “Validation Vector Grade (VVG): A New Coverage Metric for Validation and Test”, *Proc. 17th IEEE VLSI Test Symposium*, April 1999, pp. 182-188

[79] Turbo Tester homepage [*http://www.pld.ttu.ee/tt*]

[80] Ubar, R., “Alternative Graphs and Test Generation for Digital Systems”, *Proc. of 2nd Conf. On Fault Tolerant Systems and Diagnostics*, Brno, Czechoslovakia, 1979, pp. 177-184

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

[82] Ubar, R., “Test Pattern Generation for Digital Systems on the Vector Alternative Graph model”, *Proc. of 13-th International Symposium on Fault Tolerant Computing*, Milano, Italy, 1983, pp. 347-351

[83] Ubar, R., “Test Synthesis with Alternative Graphs”, *IEEE Design and Test of Computers, Spring*, 1996, pp.48-59

[84] Ubar, R.; Morawiec, A.; Raik J., “Cycle-based Simulation with Decision Diagrams”, *Proc. of the DATE Conference*, Munich, Germany, March 9-12, 1999, pp. 454-458

[85] Ubar, R.; Raik, J.; Morawiec, A., “Back-tracing and Event-driven Techniques in High-level Simulation with Decision Diagrams”, *Proc. of ISCAS 2000*, Vol. 1, pp. 208-211

[86] Ubar, R.; Raik, J.; Ivask, E.; Brik, M.; “Multi-level fault simulation of digital systems on decision diagrams”, *IEEE International Workshop on Electronic Design, Test and Applications, DELTA 2002. Proceedings*, pp. 86 – 91, 2002

[87] Ulrich, E.G.; Baker, T., “The Concurrent Simulation of Nearly Identical Digital Networks”, *Proc. of 10th Design Automation Workshop*, 1973, pp. 145-150

[88] US Patent No. US 6990438 B1, “Method and Apparatus for Observability-based Code Coverage”, *by Synopsys Inc.*, Jan. 2006

[89] Verilog [*www.verilog.com*]

[90] VHDL [*www.vhdl.org*]

[91] Wang, Fuh-lin; Lowe, E.; Angeli F., “Design of a Functional Test Generator with a Functional Deductive Simulator for Digital Systems”, *AUTOTESTCON’78*, 1978, pp.134-142

[92] Wang, Laung-Terng; Wu, Cheng-Wen; Wen, Xiaoqing, “VLSI Test Principles and Architectures”, *Elsevier Inc., San Francisco*, 2006

[93] Ward, P. C.; Armstrong, J. R., "Behavioral fault simulation in VHDL," pp.587-593, *27th ACM/IEEE Design Automation Conference*, 1990

[94] Wile, Bruce; Goss, John C., Roesner, Wolfgang, “Comprehensive Functional Verification”, *Elsevier Inc., San Francisco,* 2005

[95] Zhang, Liang; Ghosh, I.; Hsiao, M. S., “A Framework for Automatic Design Validation of RTL Circuits Using ATPG and Observability-Enhanced Tag Coverage”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2006, pp. 2526-2538

[96] Zhang, Q.; Harris, I. G., “ A data flow fault coverage metric for validation of behavioral HDL descriptions”, *International Conference on Computer Aided Design (ICCAD)*, 2000

Appendix

RESEARCH PAPERS

Research paper I

Reinsalu, U.; Raik, J.; Ubar, R., “Register-Transfer Level Deductive Fault Simulation Using Decision Diagrams”, *Proceedings of the 12th Biennial Baltic Electronic Conference BEC2010*, Oct. 2010, Tallinn, Estonia, pp. 193 – 196

Register-Transfer Level Deductive Fault Simulation Using Decision Diagrams

Uljana Reinsalu, Jaan Raik, Raimund Ubar

Department of Computer Engineering, Tallinn University of Technology, E-mail: uljana@pld.ttu.ee

ABSTRACT: The paper presents a deductive method for register-transfer level fault simulation on the system model of high-level decision diagrams. The method is based on the bit coverage fault model, which has been proven to have a good correspondence with gate-level structural faults. Experiments on ITC99 benchmark circuits have been carried out showing the feasibility of the proposed approach.

1 Introduction

While several efficient algorithms for fault simulation of combinational circuits exist, the task of analysing structural faults in sequential circuits remains a highly complex issue. In order to contend this complexity the research community has turned towards developing methods on higher design abstraction levels.

Functional fault simulation of VHDL designs has been proposed in [1, 2]. This approach is fast but lacks accuracy since there is no strong relation between the functional fault model and actual structural faults in the circuit. In [3] an architectural-level fault simulation tool ARSIM is presented. The tool uses symbolic data to simultaneously process the fault effects for groups of faults in the module under simulation. However, ARSIM is capable of reporting only pessimistic fault simulation results because of the limitations of the symbolic algebra applied in fault propagation. This shortcoming has been partly removed in an improved symbolic approach by Sinanoglu and Orailoglu [4]. In [5] Kassab, Rajski and Tyszer, propose hierarchical functional fault simulation that provides high speed-up but relies on building blocks that have regular structures only.

The authors of this paper proposed a concept of hierarchical fault simulation [6] using a deductive algorithm on High-Level Decision Diagram (HLDD) models [7]. The method assumed that gate-level descriptions of all the modules exist and modeled faults in the circuits hierarchically at the register-transfer and logic levels. In this paper, we propose a new approach which is applicable at the Register-Transfer Level (RTL). We build on the bit-coverage fault model, which has proven to have good correspondence with gate-level structural faults [8].

The paper is organized as follows. Section 2 explains HLDD representation for RTL circuits. Section 3 presents the bit-coverage fault model and introduces the deductive algorithm for RTL fault simulation. In Section 4 experimental results are provided. Finally conclusions are drawn.

2 Representing RTL designs using HLDDs

Consider a digital system as a network $N = (Z, F)$ of components where Z is the set of all variables (Boolean, Boolean vectors, integers) which represent the connections between components, inputs and outputs of the network. Denote by $X \subset Z$ and $Y \subset Z$, correspondingly, the subsets of input and output variables. $V(z)$ denotes the possible values for $z \in Z$, which are finite. Let F be the set of digital functions on Z: $z_k = f_k(z_{k,1}, z_{k,2}, \dots, z_{k,p}) = f_k(Z_k)$ where $z_k \in Z$, $f_k \in F$, and $Z_k \subset Z$. Some of the functions $f_k \in F$, for the state variables $z \in Z_{STATE} \subset Z$, are next state functions.

Definition 1. A *High-Level Decision Diagram* (HLDD) is a directed acyclic graph $G = (M, \Gamma, Z)$ where M is a set of nodes, Γ is a relation in M, and $\Gamma(m) \subset M$ denotes the set of successor nodes of $m \in M$. The nodes $m \in M$ are marked by labels $z(m)$. The labels can be either variables $z \in Z$, or algebraic expressions of $z \in Z$, or constants.

For non-terminal nodes m, where $\Gamma(m) \neq \varnothing$, an onto function exists between the values of $z(m)$ and the successors $m^e \in \Gamma(m)$ of m. By m^e we denote the successor of m for the value $z(m) = e$. The edge (m, m^e) which connects nodes m and m^e is called *activated* iff there exists an assignment $z(m) = e$. Activated edges, which connect m_i and m_j make up an *activated path* $l(m_i, m_j)$. An activated path $l(m^0, m^T)$ from the initial node m^0 to a terminal node m^T is called the *full activated path*.

Definition 2. A decision diagram $G_k = (M, \Gamma, z)$ represents a function $z_k = f_k(z_{k,1}, z_{k,2}, \dots, z_{k,p}) = f_k(Z_k)$ iff for each value $v(Z_k) = v(z_{k,1}) \times v(z_{k,2}) \times \dots \times v(z_{k,p})$, a full path in G_k to a terminal node m^T is activated, where $z(m^T) = z_k$.

Each function $f_k \in F$ in the system network $N = (Z, F)$ is represented by a decision diagram $z_k = G_k(Z_k)$. Depending on the class of digital system (or level of its representation), we may have various DDs, in which nodes have different interpretations and relationships to the system structure. In RT level descriptions, we usually decomposition digital systems into control and datapath parts. State and output variables of the control part serve as addresses or control words, and the variables in the datapath part serve as data words. The functions of RTL components in the datapath are described by arithmetic operations on the word-level data variables .

BDDs [9] represent a special class of HLDDs where all the variables in Z are binary, i.e. for all $z \in Z$, $V(z) = 2$. An example of a DD $G'_A(q', A, B, C)$ for the register A with the following behaviour

$$A = \begin{cases} B' + C', & \text{if } q' = 0, \\ \neg A' + 1, & \text{if } q' = 1 \,\&\, x_A = 0, \\ \neg C' + B', & \text{if } q' = 3 \,\&\, x_C = 1, \\ A' + B' + C', & \text{if } q' = 4 \,\&\, x_A = 0 \,\&\, x_C = 0, \\ A' & \text{in other cases,} \end{cases}$$

is represented in Fig.1. The trailing quote character after variable names denotes that values are to be taken from the preceding time step (i.e. preceding clock-cycle).

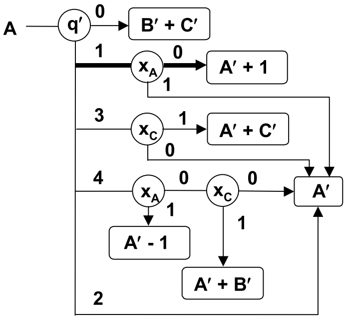


Fig.1. High-Level Decision Diagram

3 Deductive fault simulation on HLDDs

In this paper we rely on the bit coverage fault model, which has been proven to have a good correspondence with gate-level structural faults [8]. In the bit coverage model, faults are injected to every bit of every register in the RTL circuit. Single fault assumption is used, i.e. a fault is expected to be present at one of the register bits at the time. The bit is assumed to be permanently stuck to the value 0 or 1.

A *complex pattern* T is generated for all the registers. $T=\{P_0, (P_1, D_1), \ldots, (P_k, D_k)\}$, where P_0 is fault free value of the register, P_1 is faulty value corresponding to the fault D_1. If two faults produce the same faulty value then they should be merged. If $P_1=P_2$, then $(P_1, D_1 \cup D_2)$. If fault-free value P_0 and faulty value P_j are the same, then all faults of P_j are self-masked, and the component (P_j, D_j) should be removed from the complex pattern T.

Fault simulation on HLDDs is carried out by tracing the activated paths on HLDDs in accordance to the given values of variables as specified in Definition 1. For example, at the given input (state) pattern $P = \{q'=1, x_A =0\}$ of the block A we reach the terminal node m^T of the graph G_A with label $A' + 1$ (see the highlighted path in Fig.1). The new value of A will be $A = A' + 1$.

In high-level fault propagation in the digital system $S=(Z,F)$ through a block with function $z = f(z_1, z_2, \ldots, z_n)= f(Z'), Z' \subseteq Z$, which is represented by a DD G_z, we proceed from the fact that the defects may have been propagated to all of the variables $z_i \in Z'$ used in labels of nodes in the graph. To each node m of the HLDD with the label $z(m)$, a complex pattern

$$T_{z(m)} = \{P_{z(m),0}, (P_{z(m),1}, D_{z(m),1}), \ldots, (P_{z(m),kz}, D_{z(m),kz})\}$$

corresponds. From this pattern, it results that a set of defects $D_{z(m)}= D_{z(m),1} \cup \ldots \cup D_{z(m),k}$ has been propagated to the node m. Let D be the set of all faults currently activated and listed in $T_{z(m)}$.

Consider the fault simulation on the HLDD G_z as the following set of procedures.

Procedure 1. The fault-free path is simulated in accordance to the fault-free input pattern $P_{z(m),0}$,, and the fault-free value of $z=z(m^{T,0})$ is calculated, where $m^{T,0}$ is the terminal node of the fault-free activated path.

Following the fault-free path D_{exl} is calculated. D_{exl} is the set of faults, which can't be at current node in the path. Therefore following the fault-free path we collect $D_{exl(m)}=D_{exl} \cup T_m$. The condition of reaching the node m in the fault-free path during fault simulation is the absence of all the faults $D_{exl(m)}$. Denote by $D_{incl(m)}$ the set of faults consistent to the current faulty path from the initial node m_0 up to the node m. For the node m for fault-free path we have $D_{incl(m)}=T_m - D_{exl(m)}$. It should be noted that reaching the node m at fault-free path $D_{incl(m)}$ must be calculated first, then $D_{exl(m)}$ as shown in function ProcessFaultyFreePath() (Fig. 3).

First we call recursively the function ProcessFaulty-FreePath() with newly calculated D_{exl} while reaching the terminal node. Then function ProcessFaultyPath() is called if D_{incl} is not empty. Function ProcessFaultyPath() is called also many times recursively according to the faults in D_{incl}.

When reaching the terminal node of fault-free path the terminal node is processed (see procedure 2).

Procedure 2. Fault simulation of a terminal node m^T of fault-free path is finding all combination of (P_i, D_i), of the terminal complex pattern $T_{z(m^T)}$, which exclude set of propagated faults $D_{exl(mT)}$ and assigning this complex pattern to the graph root variable z. Also fault-free value of the graph is calculated and according to the model of faults generation new faults are generated for the next cycle. If the same fault is preserved for all cycles (for example flip of the first bit to 0 in certain register), and fault corresponding to this situation is already propagated, then generation of faulty value is calculated according to the propagated faulty value, not according to fault-free value (see function ProcessFaults()).

Procedure 3. Fault simulation of a nonterminal node m with the variable z(m) for complex pattern $T_{z(m)} = \{P_{z(m),0}, (P_{z(m),1}, D_{z(m),1}), \ldots, (P_{z(m),k}, D_{z(m),k})\}$ consists of the following:

- If m belongs to fault-free path then only faulty responses are processed. For each faulty response $e=P_{z(m),i}$ of $T_{z(m)}$ $D_{incl(m)}$ is calculated as following $D_{incl(m)}= D_{incl} \cap D_{z(m),i}$ and function processFaultyPath() is called recursively.
- If m does not belong to the fault-free path then non-faulty response $P_{z(m),0}$ of $T_{z(m)}$ of the m is found and $D_{incl(m)}$ is calculated as following $D_{incl(m)}=D_{incl} - D_{z(m),I}$ and function processFaultyPath() is called recursively with $D_{incl(m).}$ Also all faulty responses of m are processed.

Procedure 4. When reaching the terminal node of faulty path the difference of faults if calculated $D_{diff}=D_{incl} - z(m),i$, if this difference is not empty pair (value of terminal node, Ddiff) is added to the complex pattern Tz. Also intersection of faults is calculated $D_{insec}== D_{incl} \cap D_{z(m),i}$, and only faults D_{insec} are added from complex pattern $T_{z(m)}$ to complex pattern T_z.

As the result of the fault simulation by Procedures 2 and 4 we create a complex pattern for the variable z: $T_z = \{P_{z,0}, (P_{z,1},D_{z,1}), \ldots, (P_{z,kz},D_{z,kz})\}$. All the pairs $(P_{z,i},D_{z,i})$ where $P_{z,i} = P_{z,0}$ are eliminated since the defects $D_{z,i}$ are self-masked at this point. All the groups of pairs $\{(P_{z,i},D_{z,i}), (P_{z,j},D_{z,j})\}$ where $P_{z,i} = P_{z,j}$ are merged into a single pair $(P_{z,i},D_{z,i})$, so that $D_{z,i} = D_{z,i} \cup D_{z,j}$.

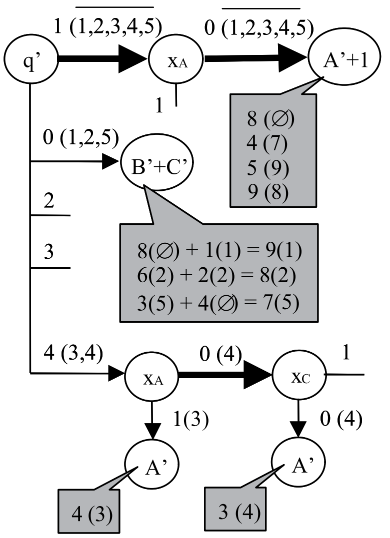


Fig.2. Fault simulation on the HLDD of Fig.1

Example Consider the HLDD G_A in Fig.2 with a set of complex patterns:

$T_q = \{1, 0\ (1,2,5), 4\ (3,4)\}$,

$T_{xA} = \{0, 1\ (3,5)\}$,

$T_{xC} = \{1, 0\ (4,6)\}$,

$T_A = \{7, 3\ (4,5,7), 4\ (1,3,9), 8\ (2,8)\}$,

$T_B = \{8, 3\ (4,5), 4\ (3,7), 6\ (2,8)\}$,

$T_C = \{4, 1\ (1,3,4), 2\ (2,6), 5\ (6,7)\}$.

All the paths traced during the fault simulation are highlighted and marked by details of simulation in Fig.2. The fault free paths are shown by bold lines both, in Fig.1 and Fig.2. The edges on paths in Fig.2 are labelled by pairs *e*,(*D*), where *e* is the value of the node variable when leaving the node at this direction, and *D* is a subset of defects: $D_{exl}(m)$ for the next node *m* on the fault-free path, and $D_{incl}(m)$ for the next node *m* on the faulty paths. Since $D_{exl}(x_A) = \{1,2,3,4,5\}$ includes both of the defects 3 and 5 propagated to x_A, no faulty paths are simulated from the node x_A: for the value $x_A = 1$: $D'_{xA} = (D_{xA} - D_{exl}\ (x_A)) = \varnothing$. From all the defects propagated to *A'*, only the defects 7, 8 and 9 are simulated at the node *A'*+1. At the terminal node *B*'+*C*' only the defects 1,2,5 are simulated, since only they are consistent to the condition of leaving the node *q'* at this direction.

After fault simulation of all 3 reached terminal nodes we compose the final result as follows: the defect 2 propagated to the node *B'+C'* is self-masked because the value *B'+C'* = 8 calculated for the defect 2 is equal to the fault-free value calculated at the node *A*'+1. The defects 3 and 7 propagated to different terminal nodes are merged into the same group because they produce the same new value 4 for *A*. Also, the defects 1 and 8 are merged into the same group. The final value of the new complex pattern is: $T_A = \{8, 3(4), 4(3,7), 5(9), 7(5), 9(1,8)\}$.

4 Experimental results

In Table 1 the results of the deductive fault simulation algorithm on ITC99 benchmark circuits [10] are shown. The first column shows the circuit. The second and third columns present the number of primary inputs and output signals, respectively. The fourth column reports the number of HLDD nodes. The fifth column presents the test length in clock-cycles. The 6th column reports the achieved bit fault coverage and the final column reports run times.

Table 1 Fault simulation results

circuit	PI	PO	HLDD nodes	test length	fault cover, %	time, s
b00	3	2	35	50	69.0	0.77
b01	3	2	26	50	90.0	0.17
b02	2	1	16	50	87.5	0.094
b03	5	1	137	200	83.3	9.75
b04	5	1	58	100	97.7	5.05
b06	3	4	44	100	94.4	0.56
b09	2	1	44	100	98.2	22.0
b10	9	3	157	200	82.5	1248
b11	3	1	57	100	83.3	9.06
b13	4	7	178	200	76.4	14.7

```
FaultSimulation(){
For each vector v
        ProcessFauls(v)
  End for
}

ProcessFaults(v){
    $D_{exl}=\varnothing$
    For each graph
            //$m_0$ is the root node of g
        ProcessFaultyFreePath(g, $m_0$, v, $D_{exl}$)
        // assign faultfree value
        $P_{v,z(m)}=z(m)$
    // inject bit faults
       Build $T_{v,z(m)}$ from all single bit flips of $P_{v,z(m)}$
            (if fault corresponding to current bit is propagated
            from previous cycle, then faulty value is flipped
            otherwise fault free  value is flipped)
    End for
}

ProcessFaultyFreePath(g, m, v, $D_{exl}$){
    $D_{incl}$={faults in $T_{v,z(m)}$} – $D_{exl}$;
    $D_{exl}=D_{exl} \cup$ {faults in $T_{v,z(m)}$};
    If $m \notin m^T$
            ProcessFaultyFreePath(g,m,v, $D_{exl}$)
            If $D_{incl} \neq \varnothing$
                ProcessFaultyPath(g, m, v, $D_{incl}$, True)
            End if

  Else
            processTerminalFaultFreePath(g, m, v, $D_{incl}$)
  End if
}

ProcessFaultyPath(g, m, v, $D_{incl}$, FlagFromFaultyFree){
    If $m \notin m^T$      // if non-terminal
      If(!FlagFromFaultyFree)
          nonFaulty response $P_{z(m),0}$ of $T_{v,z(m)}$
              $D_{incl}=D_{incl}-D_{z(m),i}$
        // process the nonFaulty response
        ProcessFaultyPath(g, $m^{z(m)}$, v, $D_{incl}$, false)
     End if    //flagFromFaultyFree
    For each faulty response $e=P_{z(m),i}$ of $T_{v,z(m)}$
        $D_{incl}= D_{incl} \cap D_{z(m),i}$
        ProcessFaultyPath(g, $m^e$, v, $D_{incl}$, false)
    End for
Else        // if terminal
        ProcessTerminalFaultyPath(g, m, v, $D_{incl}$)
End if
}

ProcessTerminalFaultFreePath(g, m, v, $D_{incl}$){
            $T_{v,z}$= only faults $D_{incl}$ from $T_{v,z(m)}$
}

ProcessTerminalFaulyPath(g, m, v, $D_{incl}$){
    $D_{diff}= D_{incl}-D_{z(m),i}$
    $D_{insec}== D_{incl} \cap D_{z(m),i}$
  If ($D_{diff}$!=empty)
            $T_{v,z}=T_{v,z}$ add $D_{diff}$
  End if
 If($D_{insec}$!=empty)
            $T_{v,z}= T_{v,z}$ add only faults $D_{insec}$ from $T_{v,z(m)}$
 End if
}
```

Fig. 3. Deductive simulation algorithm for bit faults

5 Conclusions

The paper proposed a deductive method for register-transfer level fault simulation on the system model of high-level decision diagrams. The method is based on the bit coverage fault model, which has been proven to have a good correspondence with gate-level structural faults. Experiments on ITC99 benchmark circuits were carried out showing the feasibility of the proposed approach.

Acknowledgements

The work has been supported by Estonian SF grants 7068, 7483, EC FP7 ICT STREP project DIAMOND, FP7 REGPOT project CREDES, and Research Centre CEBE funded by EU Structural Funds.

References

[1] P. C. Ward, J. R. Armstrong, "Behavioral fault simulation in VHDL," pp.587-593, *27th ACM/IEEE Design Automation Conference (DAC '90)*, 1990.

[2] D. Federici, P. Bisgambiglia, J.-F. Santucci, "High level fault simulation: experiments and results on ITC'99 benchmarks," pp.118, *Fifth IEEE International High-Level Design Validation and Test Workshop (HLDVT'00)*, 2000

[3] J. Lee, E. M. Rudnick and J. H. Patel, "Architectural level fault simulation using symbolic data", *in European Conference on Design Automation*, pp. 437-442, 1993..

[4] Ozgur Sinanoglu, Alex Orailoglu, "RT-level Fault Simulation Based on Symbolic Propagation", *VTS 2001*, pp. 240-245.

[5] Mark Kassab, Janusz Rajski, Jerzy Tyszer, "Hierarchical Functional-Fault Simulation for High-Level Synthesis", *ITC 1995*, pp. 596-605.

[6] Ubar, R.; Raik, J.; Ivask, E.; Brik, M.; "Multi-level fault simulation of digital systems on decision diagrams", *IEEE International Workshop on Electronic Design, Test and Applications, DELTA 2002. Proceedings*, pp. 86 – 91, 2002.

[7] R.Ubar, "Test Synthesis with Alternative Graphs", *IEEE Design and Test of Computers*, Spring, 1996, pp.48-59.

[8] F. Fummi, C. Marconcini and G. Pravadelli, "Logic-level mapping of high-level faults", *Integration, the VLSI Journal*, Volume 38, Issue 3, January 2005, Pages 467-490

[9] Randal E. Bryant, "Graph-based algorithms for Boolean function manipulation", *IEEE Transactions on Computers*, v.35 n.8, p.677-691, Aug. 1986.

[10] www.cerc.utexas.edu/itc99-benchmarks/bench.html

Research paper II

U. Reinsalu, J. Raik, R. Ubar, P. Ellervee, “Fast RTL Fault Simulation Using Decision Diagrams and Bitwise Set Operations”, *Proceedings of 26th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems* pp.164 - 170, 2011

Fast RTL Fault Simulation Using Decision Diagrams and Bitwise Set Operations

Uljana Reinsalu, Jaan Raik, Raimund Ubar, Peeter Ellervee
Department of Computer Engineering
Tallinn University of Technology
Tallinn, Estonia
e-mail: {uljana, jaan, raiub, lrv}@pld.ttu.ee

***Abstract*—Efficient fault simulation algorithms for combinational circuits are known for decades. However, sequential fault simulation which is frequently used in test and fault tolerance applications remains a very time-consuming task, in particular for larger circuits. Current paper proposes a new deductive method for Register-Transfer Level (RTL) fault simulation on the system model of high-level decision diagrams. We apply the bit coverage fault model which has proven to provide a good correspondence with gate-level structural faults. Simulation speed-up is achieved due to efficient data structures implemented to perform set operations in the deductive fault simulation algorithm. Experiments on RTL benchmark circuits show that up to two orders of magnitude shorter run-times are achieved with the method in comparison to gate-level fault simulation.**

Keywords - register-transfer level; fault simulation; high-level decision diagrams

I. INTRODUCTION

While several efficient algorithms for fault simulation of combinational circuits exist, the task of analysing structural faults in sequential circuits remains a highly difficult issue. In order to contend the complexity the research community has turned towards developing methods at higher design abstraction levels.

Existing fault simulation tools typically rely on gate-level algorithms. One of the earliest sequential fault simulators, PROOFS [1] combines the advantages of differential fault simulation and parallel fault simulation. HOPE [2], a parallel fault simulator, simulates 32 faults at a time. Faults with short propagation paths are excluded from parallel simulation, since most of the time the faulty circuit response would be identical to the correct one during the simulation of such faults. Also PARIS [3] is based on a parallel fault simulation model. Heuristics are used to minimize the number of events that must be tracked. Despite of a wide range of methods, fault simulation for sequential circuits at the gate-level is slow for larger designs, in particular when long test sequences are considered.

Functional fault simulation of VHDL designs has been proposed in [4, 5]. This approach is fast but it lacks accuracy since there is no strict correlation between the functional fault model and actual structural faults in the circuit. In [6] an architectural-level fault simulation tool ARSIM is presented. The tool uses symbolic data to simultaneously process the fault effects for groups of faults in the module under simulation. However, ARSIM is capable of reporting only pessimistic fault simulation results because of the limitations of the symbolic algebra applied in fault propagation. This shortcoming has been contended in an improved symbolic approach by Sinanoglu and Orailoglu [7] by utilizing the rightmost faulty bit location information to enhance the method's ability of propagating symbolic data. In [8], Kassab, Rajski and Tyszer, propose hierarchical functional fault simulation that provides high speed-up but relies on building blocks that have regular structures. Shen introduced a concurrent Register-Transfer Level (RTL) fault simulator VFSim [9], which is capable of simulating Verilog designs. However, comparison to the gate-level fault simulator HOPE [2] showed no speed-up in most cases.

A concept of hierarchical fault simulation using a deductive algorithm on High-Level Decision Diagram (HLDD) [10] models was introduced in [11]. The method assumed that gate-level descriptions of all the modules exist and faults were modeled in the circuits hierarchically at the register-transfer and logic levels. Another, RTL algorithm was proposed in [12]. However, experimental results showed that the method becomes prohibitively slow when circuits with large number of arithmetic operations are considered.

In this paper, we propose a new approach which is applicable directly at the Register-Transfer Level (RTL). We build on the bit-coverage fault model, which has proven to yield good correspondence with gate-level structural faults [13]. We introduce efficient data structures based on bitwise set operations in order to achieve a high speed of simulation. Experiments on RTL benchmark circuits show that up to two orders of magnitude shorter run-times are achieved with the method in comparison to state-of-the-art gate-level simulation.

The paper is organized as follows. Section 2 defines High-Level Decision Diagrams (HLDD). Section 3 explains HLDD representation of RTL circuits. Section 4 presents the deductive algorithm for RTL fault simulation. Section 5 introduces the data structures to perform set operations on the lists of propagated faults. Finally, experimental results and conclusions are provided.

II. REPRESENTING RTL DESIGNS USING HLDDS

In this Section we define the High-Level Decision Diagram (HLDD) data structure. Consider a digital system (Z, F) as a network of subsystems or components, where Z is the set of variables (Boolean, Boolean vectors or integers), which represent connections between components, primary inputs and primary outputs of the network. Let $Z = X \cup Y$, where X is the set of function arguments and Y is the set of function values where $Q = X \cap Y$ is the set of state variables. $D(z)$ denotes the finite set of all possible values for $z \in Z$ and $D(Z')$ is the set of all possible vectors in some variable set $Z' \subseteq Z$. Obviously, if $Z' = \{z_1, \ldots, z_n\}$ then $D(Z') = D(z_1) \times \ldots \times D(z_n)$. Let F be the set of discrete functions: $y_k = f_k(X_k)$, where $y_k \in Y, f_k \in F$, and $X_k \subseteq X$ (k iterates over all elements in F).

Definition 1. High-level decision diagram representing a function $f_k : D(X_k) \rightarrow D(y_k)$ is a directed acyclic multigraph $G = (V, E)$ with a single root node and a set of terminal nodes where:

- Each non-terminal node is labeled by some input or control variable $x \in X$. We shall denote the variable of node v by x_v.
- Each terminal node w is labeled by some function $g_w : D(X_w) \rightarrow D(y_k)$, where $X_w \subseteq X_k$.
- Each edge $e = (v, u)$ is labeled by a constant $c_e \in D(x_v)$.
- Each two edges $e_1 = (v, u_1)$ and $e_2 = (v, u_2)$ starting from the same source node are labeled by different constants $c_{e_1} \neq c_{e_2}$.
- If the node v is labeled by x_v then the number of edges starting from this node is $|D(x_v)|$.

Remark 1. Every BDD is an HLDD too, with two terminal vertices labeled by constant functions 0 and 1, and $D(x) = \{0, 1\}$ for every variable x.

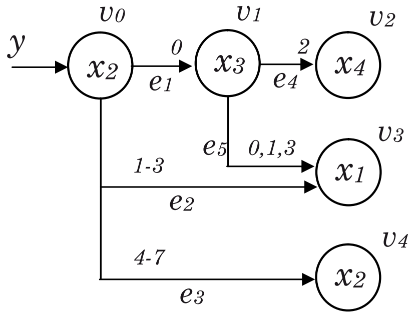
In other words, HLDD is a data structure similar to BDD, but with many edges originating from a particular node and a number of functions at the end, instead of constants 0 and 1. We shall denote the set of terminal nodes by V^T and the set of non-terminal nodes by V^N and the set of all successors of the node v by $\Gamma(v)$. For non-terminal nodes $v \in V^T$ an onto function exists between the values $c \in D(x_v)$ of labels x_v and the successors $v^c \text{-} \in \Gamma(v)$ of v. By v^c we denote the successor of v for the value $x_v = c$.

The edge (v, v^c), which connects nodes v and v^c, is called *activated* iff $x_v = c$. Activated edges, which connect v_i and v_j, form an *activated path* $l(v_i, v_j) \subseteq V$. An activated path $l(v_0, v^T)$ from the root node v_0 to a terminal node v^T is called the *main activated path* and v^T itself is refered to as the activated terminal node.

Without loss of generality we assume further that each variable has at least two values, i.e. $\forall z \in Z, D(z) > 1$. Let D_i designate a subset of $D(x_v)$ labeling node v, such that assignments from it will activate its successor node v_i. $D(x_v)$ is partitioned into non-intersecting sets $D_1, \ldots, D_m$, where $m = |\Gamma(v)|$. More formally,

$$\bigcup_{i=1}^{m} D_i = D(x_v) \wedge \forall i, j, i \neq j \rightarrow D_i \cap D_j = \varnothing.$$

In other words, with every value assignment to variable x_v one and only one successor node will be activated. In the following graphical examples we merge the edges according to their successor node v_i and label them by the corresponding domain partition D_i.



$G_y = (V,E,X)$,
$V = \{v_0, v_1, v_2, v_3, v_4\}$;
$E = \{e_1, e_2, e_3, e_4, e_5\}$, $e_1=(v_0, v_1)$, $e_2=(v_0, v_3)$, $e_3=(v_0, v_4)$, $e_4=(v_1, v_2)$, $e_5=(v_1, v_3)$;
$X=\{x_1=x_{v_3}, x_2=x_{v_0}=x_{v_4}, x_3=x_{v_1}, x_4=x_{v_2}\}$;
$D_1(x_{v0})=\{0\}$, $D_2(x_{v0})=\{1,2,3\}$, $D_3(x_{v0})=\{4,5,6,7\}$, $D_1(x_{v1})=\{2\}$, $D_2(x_{v1})=\{0,1,3\}$.

Figure 1 Graphical representation of a HLDD for a function $y=f(x_1,x_2,x_3,x_4)$

Fig. 1 presents a HLDD G_y representing a discrete function $y=f(x_1,x_2,x_3,x_4)$. The diagram contains five nodes $v_0, \ldots, v_4$. The root node v_0 is labeled by variable x_2 which is an integer with a range from 0 to 7. The node has three outgoing edges entering the nodes v_1, v_3 and v_4. The node v_1 is labeled by x_3 with a range from 0 to 3. It has two outgoing edges e_4 and e_5 entering terminal nodes v_2 and v_3, respectively. The edge e_4 is activated by $x_3=2$, while the edge e_5 is activated by x_3 having a value 0, 1 or 3.

III. Modeling RTL designs by HLDDs

Consider the datapath depicted in Fig. 2a and its corresponding HLDD representation shown in Fig. 2b. Here, R_1 and R_2 are registers (R_2 is also a primary output), MUX_1, MUX_2 and MUX_3 are multiplexers, + and * denote addition and multiplication operations, *IN* is an input bus, SEL_1, SEL_2, SEL_3 represent multiplexer address signals, EN_2 serves as the signal for register R_2, and *a, b, c, d* and *e* denote internal buses, respectively. In the HLDD, the control variables *RES*, SEL_1, SEL_2, SEL_3 and EN_2 are labeling internal decision nodes of the HLDD. The terminal nodes are labeled by a constant *#0* (reset of R_2), by word-level variables R_1 and R_2 (data transfers to R_2), and by expressions related to the data manipulation operations of the network.

Consider, simulating HLDD with some values assigned to the variables. Let the value of SEL_2 be 0, the value of SEL_3 be 3, the value of EN_2 be 1 and the value of *RES* be 0 in the current simulation run. By bold lines and grey nodes, a main activated path in the HLDD is shown from *RES* to R_1*R_2, which corresponds to the pattern $RES=0$, $EN_2=1$, $SEL_3=3$, and $SEL_2=0$. The activated part of the network at this pattern is denoted by grey boxes.

The main advantage and motivation of using HLDDs compared to the netlists of primitive functions is the increased efficiency of simulation and diagnostic modeling because of the direct and compact representation of cause-effect relationships. For example, instead of simulating the control word $SEL_1=0$, $SEL_2=0$, $SEL_3=3$, $EN_2 = 1$, $RES=0$ by computing the functions $a = R_1$, $b = R_1$, $c = a + R_2$, $d = b * R_2$, e = d, and $R_2 = e$, we only need to trace the nodes *RES*, EN_2, SEL_3 and SEL_2 on the HLDD and compute a single operation $R_2 = R_1 * R_2$. In case of detecting an error in R_2 the possible causes can be defined immediately along the simulated path through *RES*, EN_2, SEL_3 and SEL_2 without complex diagnostic analysis inside the corresponding RTL netlist. The activated path provides the *fault candidates*, i.e. variables that are suspected to contain faults causing the error at R_2 during current simulation run. Further reasoning should be based on analyzing sources of these signals.

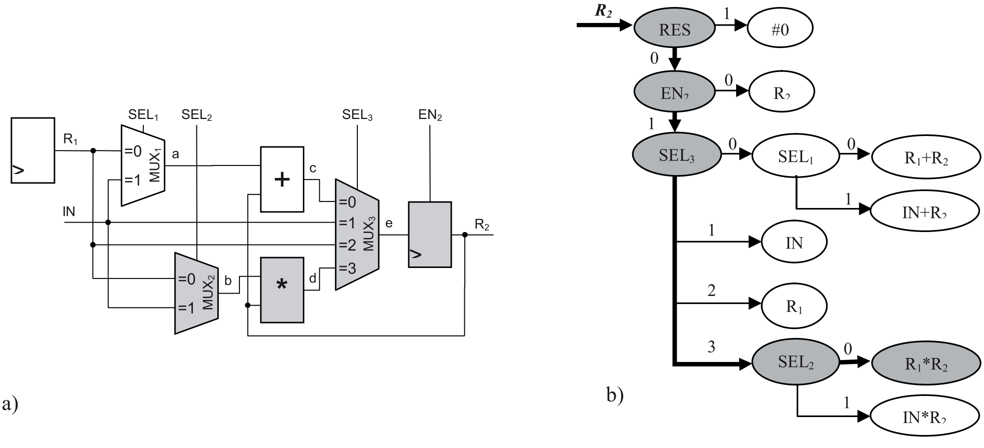


Figure 2 a) RTL schematic and b) its HLDD-based representation

IV. DEDUCTIVE FAULT SIMULATION ON HLDDs

In this paper we rely on the bit coverage fault model, where the faults are injected to every bit of every register in the RTL circuit. Single fault assumption is used, i.e. a fault is expected to be present at one of the register bits at the time. The bit is assumed to be permanently stuck to the value 0 or 1. The bit coverage model has been proven to have a good correspondence with gate-level structural faults [13].

The central datastructure of the deductive fault simulation algorithm is a *fault propagation record.* A fault propagation record T_y is generated for all the variables $y \in Y$ that represent registers. $T_y=\{p_0, (p_1, M_1), \ldots, (p_k, M_k)\}$, where p_0 is the fault free value of the register variable y and p_j is the faulty value corresponding to the faults $m_{j,i} \in M_1$. $M_1 \cup \ldots \cup M_k = M'$ and $M_1 \cap \ldots \cap M_k = \varnothing$, where $M' \subseteq M$, and M is the set of all faults and M' is the set of faults propagated to the register variable y. If two faults produce the same faulty value then they should be merged to the same fault group M_j.

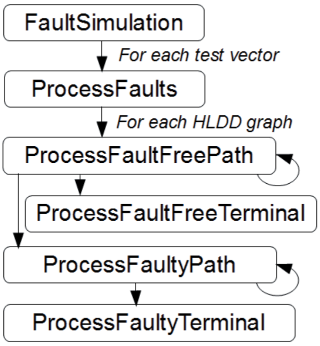


Figure 3 Call graph of the fault simulation algorithm

Below, we present the structure of the deductive fault simulation algorithm.

General structure. The fault simulation on the HLDDs is performed test vector by test vector (See Figures 3 and 4, function *FaultSimulation*). For each vector, all HLDD graphs are traversed one after another. First, the fault free path is followed by processing nodes along the main activated path of the HLDD recursively (function *ProcessFaultFreePath*) until the terminal node v^T is reached (function *ProcessFaultFreeTerminal*). Subsequent to collecting the information about possible faults out of the fault free path, all the possible branches are taken to process the faulty paths (function *ProcessFaultyPath*). Faulty paths are also processed recursively node by node until terminal nodes of the graph are reached. While processing the terminal nodes, new fault propagation record T_y for the HLDD g_y is calculated (function *ProcessFaultyTerminal*). At the beginning of a new cycle new faults are generated according to the bit coverage fault model. Thus, the set of faults to propagate to the next clock cycle consists of propagated faults and newly generated faults. During the fault simulation, two

```
FaultSimulation(){
  For each vector vec
      ProcessFaults(vec)
  End for
}

ProcessFaults(vec){
    $M_{excl}=\varnothing$
    For each graph $g_y$
        ProcessFaultFreePath($g_y$, $v_0$, $M_{excl}$)
        In $T_y$ assign the value of $y$ to $p_0$
        All single bit flip faults of $p_0$ are added to $T_y$
    End for
}

ProcessFaultFreePath(g, v, $M_{excl}$){
    $M_{incl}$={faults in $T_{x_v}$} – $M_{excl}$
    $M_{excl}$=$M_{excl} \cup$ {faults in $T_{x_v}$}
    If $v \notin v^T$
        ProcessFaultFreePath(g, $v^{x_v}$, $M_{excl}$)
        If $M_{incl} \neq \varnothing$
            ProcessFaultyPath(g, $v^{x_v}$, $M_{incl}$, True)
        End if
    Else
        ProcessFaultFreeTerminal(g, v, $M_{incl}$)
    End if
}

ProcessFaultyPath(g, v, $M_{incl}$, FlagFromFaultFree){
    If $v \notin v^T$     // if non-terminal
        If(FlagFromFaultFree=False)
            $M_{param}$=$M_{incl}$ – {faults in $T_{x_v}$}
            ProcessFaultyPath(g, $v^{x_v}$, $M_{param}$, False)
        End if
        For each faulty response e=$p_{x_{v,j}}$ of $T_{x_v}$
            $M_{incl}$= $M_{incl} \cap M_{x_{v,j}}$
            ProcessFaultyPath(g, $v^e$, $M_{incl}$, False)
        End for
    Else    // if terminal
        ProcessFaultyTerminal(g, v, $M_{incl}$)
    End if
}

ProcessFaultFreeTerminal($g_y$, v, $M_{incl}$){
    $T_y = T_{x_v}$
    Remove faults that are not in $M_{incl}$ from $T_y$
}

ProcessFaultyTerminal (g, v, $M_{incl}$){
    $M_{diff} = M_{incl}$ – {faults in $T_{x_v}$}
    $M_{insec} = M_{incl} \cap$ {faults in $T_{x_v}$}
    If ($M_{diff} \neq \varnothing$)
        add new pair (faulty value, $M_{diff}$) to $T_y$
    End if
    If($M_{insec} \neq \varnothing$)
        add pairs with faults $M_{insec}$ from $T_{x_v}$ to $T_y$
    End if
}
```

Figure 4 Pseudocode for the fault simulation algorithm sets of faults M_{excl} and M_{incl} are collected for each graph. M_{excl} is the set of faults that can't be propagated to the output of fault-free path. M_{incl} is the set of faults that are included into the propagation path.

The set of functions for the fault simulation of the HLDD G_y is listed below.

ProcessFaultFreePath. The fault-free path is simulated in accordance to the fault-free input pattern. The fault-free value of $y=x(v^T)$ is calculated, where v^T is the terminal node of the main activated path.

Following the fault-free path M_{excl} is calculated by $M_{excl(v)}=M_{excl} \cup M_v$ (M_v is a set of faults of current node). The condition of reaching the node v in the fault-free path during fault simulation is the absence of all the faults $M_{excl(v)}$.

Denote by $M_{incl(v)}$ the set of faults consistent to the current faulty path from the initial node v_0 up to the node v. For the node v for fault-free path we have $M_{incl(v)}=M_v - M_{excl(v)}$.

First we call recursively the function ProcessFaultFreePath() with newly calculated M_{excl} while reaching the terminal node. Then function ProcessFaultyPath() is called if M_{incl} is not empty.

When reaching the terminal node of the fault-free path the terminal node is processed by ProcessFaultFreeTerminal().

ProcessFaultFreeTerminal. Fault simulation of a terminal node v^T of the fault-free path lies in finding all the combinations of (p_j, M_j), of the terminal node's fault propagation record $Tx_v{}^T$, which exclude set of propagated faults $M_{excl(vT)}$ and assigning this fault propagation record to the graph variable y. Also fault-free value of the graph is calculated and according to the model of fault insertion (bit coverage fault model) new faults are generated for the next cycle.

ProcessFaultyPath. Fault simulation of a nonterminal node v with the variable x_v with fault propagation record $Tx_v = \{p_0, (p_1, M_1), \ldots, (p_k, M_k)\}$ consists of the following:

- For v faulty responses are processed as follows. For each faulty response $e=p_i$ of Tx_v, $M_{incl(v)}$ is calculated as follows $M_{incl(v)}= M_{incl} \cap M_i$ and the function ProcessFaultyPath() is called recursively with $M_{incl(v)}$.
- If v does not belong to the fault-free path then non-faulty response p_0 of Tx_v is found and $M_{incl(v)}$ is calculated as follows $M_{incl(v)}=M_{incl} - M_i$ and the function ProcessFaultyPath() is called recursively with $M_{incl(v)}$.

ProcessFaultyTerminal. When reaching the terminal node of a faulty path the difference of faults M_{diff} is calculated by subtracting from M_{incl} all the faults in Tx_v and new pair (value of terminal node, M_{diff}) is added to the fault propagation record T_y. Also intersection of faults is calculated, M_{insec} is assigned M_{incl} intersection with faults in Tx_v, and only pairs (faulty value, $M_{insec,j}$) corresponding to faults M_{insec} are added from fault propagation record Tx_v to fault propagation record T_y.

As the result of the fault simulation we create a fault propagation record for the variable y: $T_y = \{p_0, (p_1, M_1), \ldots, (p_k, M_k)\}$. All the pairs (p_j,M_j) where $p_j = p_0$ are eliminated since the faults M_j are self-masked at this point. All the groups of pairs $\{(p_i,M_i), (p_j,M_j)\}$ where $p_i = p_j$ are merged into a single pair (p_i,M_i), so that $M_i = M_i \cup M_j$.

Example Fig. 2 presented a HLDD g_{R2} . Consider a fragment of this HLDD in Fig.5 with a set of fault propagation records:

$T_{SEL3} = \{1, 0\ (3,4), 1\ (1,2,5)\}$,

$T_{SEL2} = \{0, 1\ (3,5)\}$,

$T_{SEL1} = \{1, 0\ (4,6)\}$,

$T_{R2} = \{7, 3\ (4,5,7), 4\ (1,3,9)\}$,

$T_{R1} = \{2, 4\ (3,7), 6\ (2,8), 9\ (4,5)\}$,

$T_{IN} = \{4, 5\ (6,7), 8\ (2), 10\ (1,3,4)\}$.

All the paths traced during the fault simulation are highlighted and marked by details of simulation in Fig.5. The fault-free paths are shown by bold lines in the Figure. The edges on paths in Fig.5 are labelled by pairs $e,(D)$, where e is the value of the node variable when leaving the node at this direction, and M is a subset of faults: $M_{excl}(v)$ for the next node v on the fault-free path, and $M_{incl}(v)$ for the next node v on the faulty paths. Since $M_{excl}(\mathrm{SEL}_2) = \{1,2,3,4,5\}$ includes both, the faults 3 and 5 propagated to SEL_2, no faulty paths are simulated from the node SEL_2: for the value $SEL_2 = 1$: $M'_{SEL2} = (M_{SEL2} - M_{excl}\ (SEL_2)) = \varnothing$. From all the faults propagated to R_2, only the faults 7, 8 and 9 are simulated at the node $R_1{*}R_2$. At the terminal node IN, only the faults 1,2,5 are simulated, since only they are consistent to the condition of leaving the node SEL_3 towards this direction.

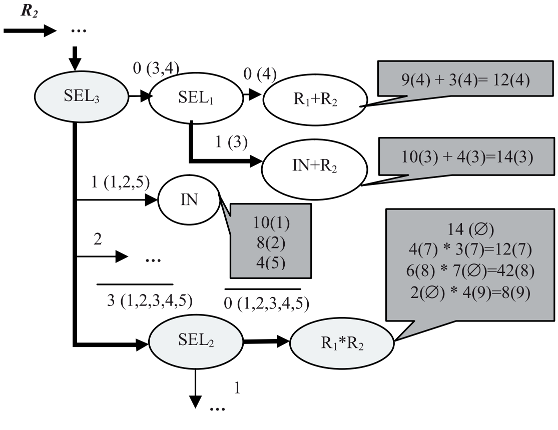


Figure 5 Fault simulation on the HLDD

After fault simulation of all the faults that reached terminal nodes we compose the final result as follows: the fault 3 propagated to the node $IN+R_2$ is self-masked because the value $IN+R_2 = 14$ calculated for the fault 3 is equal to the fault-free value calculated at the node R_1*R_2. The faults 2 and 9 propagated to different terminal nodes are merged into the same group because they produce the same new value 8 for R_2. Also, the faults 4 and 7 are merged into the same group. The final value of the new fault propagation record is: $T_{R2} = \{14, 4(5), 8(2,9), 10(1), 12(4,7), 42(8)\}$.

V. DATA STRUCTURES FOR SET OPERATIONS ON THE LISTS OF PROPAGATED FAULTS

The core part of any deductive fault simulation algorithm is the set operations on faults. In order to perform calculations with faults sets during propagation more efficiently a dedicated representation of the fault propagation record was worked out. The following operations with fault sets were carried out: difference, intersection, union. To achieve high speed, bitwise set operations were implemented.

All fault IDs for every variable in the circuit information are stored in a bit-array *FaultOccurences*. In addition, faulty values corresponding to these fault IDs are stored in an array of integers *FaultyValues*. The bit-array *FaultOccurences* represents faults presence in a variable and it is stored in the integer bit representation array of length m (m= ┐ n/<number of bits in integer> ┌), where n is the number of faults in the fault list. Every bit of array *FaultOccurences* shows whether there is fault for this variable with index i, where index shows the ID of the fault, or not. At the same time there exists another array *FaultyValues* of length n (number of faults), which is linked to array *FaultOccurences*. Every integer field of *FaultyValues* array stores a faulty value for a variable. If the i-th bit in the *FaultOccurences* array is 1 then at the same index i in the *FaultyValues* array the corresponding faulty value of the fault with ID=i is stored. Otherwise, if the i-th bit in the *FaultOccurences* array is 0 then value of i-th element of *FaultyValues* array is out of our interest. For example, in Fig. 6 for some variable *v* there exist faults with ID=0, 3, ... and the corresponding faulty values are 4, 8, ...

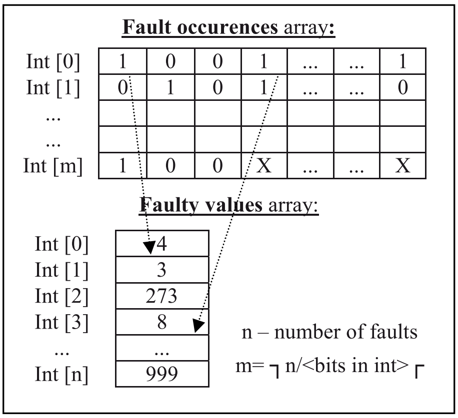


Figure 6 Data structure for propagated faults

The proposed data structure for representing fault lists proved extremely efficient. Complex set operations with large fault lists could be performed in very short run times. For example, the set intersection operator is reduced to just performing bitwise and operations on the *FaultOccurences* array, while the union operator is just represented by bitwise or, etc. Experiments performed on a set of sequential benchmarks prove the efficiency of this approach.

VI. EXPERIMENTAL RESULTS

Comparative experiments between high-level fault simulation based on deductive algorithm, which was presented in this paper, and a gate-level fault simulation tool from Turbo Tester [14] were carried out. Four circuits: sosq, gcd16, diffeq, mult8x8 and one processor circuit risc were chosen for experiments. The experiments were run on Intel Core 2 CPU, 1.83 GHz, 2 GB of RAM, Windows XP.

In Table 1 the results of experiments are shown. The column ‘vectors’ reports the number of test vectors simulated. Column ‘fault coverage’ shows the fault coverage achieved according to the bit-coverage fault model. The following two columns document the run times of the proposed deductive algorithm and the Turbo Tester fault simulator, respectively. The last column presents the ratio indicating the speed-up of the proposed RTL fault simulator with respect to the gate-level approach. As it can be seen from the table, the speed-up tends to increase steadily with the run times reaching to about two orders of magnitude with the *diffeq* example.

Table 1 Fault simulation results

circuit	vectors	faults number	fault coverage [%]	time [s]		time ratio (gate/RTL)
				RTL	gate-level	
gcd16	4000	102	100	3.28	29.23	8.91
mult8x8	4000	188	71.80	11.38	66.14	5.81
sosq	4970	206	78.15	12.58	66.36	5.28
risc	4000	260	100	18.19	366.3	20.14
diffeq	10000	230	100	37.02	3339.9	90.23

VII. CONCLUSIONS

The paper proposed a new method for register-transfer level fault simulation on the system model of high-level decision diagrams. The method is based on the bit coverage fault model, and implements efficient data structures to speed up set operations in the deductive fault simulation algorithm. Experiments on RTL benchmark circuits show that up to two orders of magnitude shorter run-times are achieved with the method in comparison to gate-level fault simulation.

ACKNOWLEDGEMENTS

The work has been supported by Estonian SF grants 7483, EC FP7 ICT STREP project DIAMOND, FP7 REGPOT project CREDES, and Research Centre CEBE funded by EU Structural Funds.

REFERENCES

[1] T. M. Niermann, W. T. Cheng and J. H. Patel, “PROOFS: A fast, memory efficient sequential fault simulator”, *IEEE TCAD,* vol. 11, n. 2, pp. 198-207, February 1992.

[2] H. K. Lee and D. S. Ha, “HOPE: An efficient parallel fault simulator for synchronous sequential circuits”, *in DAC*, pp. 336340,1992.

[3] N. Goders and R. Kaibel, “PARIS: A parallel pattern fault simulator for synchronous sequential circuits”, *in ICCAD,* pp. 542-545,1991.

[4] P. C. Ward, J. R. Armstrong, "Behavioral fault simulation in VHDL," pp.587-593, *27th ACM/IEEE Design Automation Conference*, 1990.

[5] D. Federici, P. Bisgambiglia, J.-F. Santucci, "High level fault simulation: experiments and results on ITC'99 benchmarks," pp.118, *Fifth IEEE International High-Level Design Validation and Test Workshop (HLDVT'00)*, 2000

[6] J. Lee, E. M. Rudnick and J. H. Patel, “Architectural level fault simulation using symbolic data”, *in European Conference on Design Automation*, pp. 437-442, 1993..

[7] Ozgur Sinanoglu, Alex Orailoglu, “RT-level Fault Simulation Based on Symbolic Propagation”, *VTS 2001*, pp. 240-245.

[8] Mark Kassab, Janusz Rajski, Jerzy Tyszer, “Hierarchical Functional-Fault Simulation for High-Level Synthesis”, *ITC 1995*, pp. 596-605.

[9] Li Shen, “VFSim: concurrent fault simulation at register transfer level”, *Journal of Computer Science and Technology*,Volume 20, Issue 2, March 2005.

[10] R.Ubar, “Test Synthesis with Alternative Graphs”, *IEEE Design and Test of Computers*, Spring, 1996, pp.48-59.

[11] Ubar, R.; Raik, J.; Ivask, E.; Brik, M.; “Multi-level fault simulation of digital systems on decision diagrams”, *IEEE International Workshop on Electronic Design, Test and Applications, DELTA 2002. Proceedings*, pp. 86 – 91, 2002.

[12] U. Reinsalu, J. Raik, R. Ubar, “Register-Transfer Level Deductive Fault Simulation Using Decision Diagrams”, *Baltic Electronics Conference, BEC 2010. Proceedings*, 2010.

[13] F. Fummi, C. Marconcini and G. Pravadelli, “Logic-level mapping of high-level faults”, *Integration, the VLSI Journal*, Volume 38, Issue 3, January 2005, Pages 467-490

[14] http://www.pld.ttu.ee/tt

Research paper III

J. Raik, U. Reinsalu, R. Ubar, M. Jenihhin, P. Ellervee, “Code Coverage Analysis using High-Level Decision Diagrams”, *Proceedings of the 11th IEEE Workshop on Design and Diagnostics of Electronic Systems (DDECS)*, 2008

Code Coverage Analysis using High-Level Decision Diagrams

Jaan Raik, Uljana Reinsalu, Raimund Ubar, Maksim Jenihhin, Peeter Ellervee

Tallinn University of Technology, Department of Computer Engineering
{jaan | uljana | raiub | maksim | lrv }@pld.ttu.ee

Abstract

The paper proposes a novel method of analyzing code coverage metrics on a system representation called High-Level Decision Diagrams (HLDD). Previous works have shown that HLDDs are an efficient model for simulation and test pattern generation. Current paper presents a technique, where fast HLDD based simulation is extended to support seamless code coverage analysis. We show how classical code coverage metrics can be directly mapped to HLDD constructs. In addition, we introduce an observability coverage calculation method using HLDD models. Experiments on ITC99 benchmark circuits indicate the feasibility of the proposed approach.

1 Introduction

With the increase in size and complexity of modern integrated circuits, it has become imperative to address critical verification issues in the design cycle. The process of verifying correctness of designs consumes between 60% and 80% of design effort [1]. For every designer the number of verification engineers can vary from 2 to 4 depending on the design complexity. Moreover, validation is so complex that, even though it consumes the most computational resources and time, it is still the weakest link in the design process. Ensuring functional correctness is the most difficult part of designing a hardware system [2].

In order to verify the correctness of a design, different test cases are generated. Due to the fact that it is impractical to verify exhaustively all possible inputs and states of a design, the confidence level regarding the quality of the design must be quantified to control the verification effort. The fundamental question is: How do I know if I have verified or simulated enough? Verification coverage is a measure of confidence and it is expressed as a percentage of items verified out of all possible items. Different definitions of items give rise to different coverage measures or coverage metrics.

Various coverage metrics exist such as code coverage, parameter coverage and functional coverage. In this paper, only code coverage would be used, which provides insight into how thoroughly the code of a design is exercised by a suite of simulations. The main disadvantage of code coverage metrics lies in the fact that they only measure the quality of the test case in stimulating the implementation and do not necessarily prove its correctness with respect to the specification. On the other hand, code coverage analysis is a well-defined, well-scalable procedure and, thus, applicable to large designs. The goal of current work is to propose a method for speeding up the analysis by implementing new models for simulating coverage items.

The first published reference about code coverage was as early as in 1963 by Miller and Maloney in [3]. Over the following years a large variety of code coverage metrics have been proposed, including statement coverage, block coverage, path coverage, branch coverage, expression coverage, transition coverage, sequence coverage, toggle coverage etc [2][4]. The *statement coverage* metric measures the number of times every instruction is exercised by the program stimuli. *Toggle coverage* shows whether and how many times nodes in the design toggle, i.e. how many bits change their state from 0 to 1 or vice versa. In the case of *branch coverage*, we measure the number of times each branch in the control flow graph of the code is taken or not taken under the set of program stimuli. *Path coverage* measures the number of times every path in the control flowgraph is exercised by the set of program stimuli. A potential goal of software testing is to have 100% path coverage, which implies branch and line coverage. However, full path coverage is a very stringent requirement as the number of paths in a program may be exponentially related to program size.

In this paper, we present a method and a tool for fast analysis of classical code coverage metrics, such as statement, branch and toggle coverage. We introduce High-Level Decision Diagrams (HLDD) model for efficient code coverage analysis and show how those classical coverage metrics map to HLDD constructs. We show that HLDDs can be seamlessly applied to observability coverage analysis, thus, replacing the classical D-calculus based methods (e.g. [13]). Current work is motivated by our previous encouraging research results obtained on HLDD based simulation [5, 6] and test pattern generation [7]. This is the first attempt to use HLDD models in validation and code coverage analysis.

The paper is organized as follows. Section 2 defines the HLDD based graph model for which different coverage metrics were built in. Section 3 shows how HLDDs can be used for measuring code coverage including observability coverage. Section 4 presents comparison with a popular simulation tool for hardware description languages. Finally, Section 5 concludes the paper.

G_y=(M,E,X,D),
M={m_1, m_2, m_3, m_4, m_5};
E={e_1, e_2, e_3, e_4, e_5}, $e_1=(m_1, m_2)$, $e_2=(m_1, m_4)$, $e_3=(m_1, m_5)$, $e_4=(m_2, m_3)$, $e_5=(m_2, m_4)$;
X(m_1)=X(m_5)=(x_2,{0,1,2,…,7}), X(m_2)=(x_3,{0,1,2,3}), X(m_3)=(x_4,…), X(m_4)=(x_1,…);
D(e_1)={0}, D(e_2)={1,2,3}, D(e_3)={4,5,6,7}, D(e_4)={2}, D(e_5)={0,1,3}.

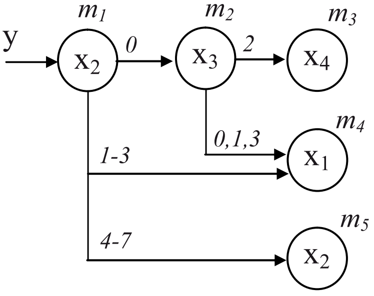


Fig. 1 . A HLDD for a function y=f(x1,x2,x3,x4)

2 High-Level Decision Diagrams

Decision Diagrams (DD) have been used in verification for about two decades. Reduced Ordered Binary Decision Diagrams (BDD) [8] as canonical forms of Boolean functions have their application in equivalence checking and in symbolic model checking. Recently, a higher abstraction level DD representation, called Assignment Decision Diagrams (ADD) [9], have been successfully applied to, both, register-transfer level (RTL) verification and test [10, 11].

The main issue with the BDDs and assignment decision diagrams is the fact that they allow logic or RTL modeling, respectively. In this paper we consider a different decision diagram representation, High-Level Decision Diagrams (HLDD) that, unlike ADDs can be viewed as a generalization of BDDs. HLDDs can be used for representing different abstraction levels from RTL to behavioral. It has proven to be an efficient model for simulation and fault modeling since it provides for a fast evaluation by graph traversal and for easy identification of cause-effect relationships [5, 6].

2.1 Basic definitions

Definition: A HLDD representing a discrete function $y=f(x)$ is a directed acyclic 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 $X(m_i)$ returns is the variable letter x_i, which is labeling node m_i. Each node of a HLDD is labeled by a variable. In special cases, nodes can be labeled by constants or algebraic expressions. An edge $e \in E$ of a HLDD is an ordered pair $e=(m_1,m_2) \in E^2$, where E^2 is the set of all the possible ordered pairs in set E. 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 of the variable x_i denoted by X_i, where $e=(m_i,m_j)$. It is required that $Pm_i=\{D(e) \mid e=(m_i,m_j) \in E\}$ is a partition of the set X_i. HLDD has only one starting node (*root node*), for which there are no preceding nodes. The nodes, for which successor nodes are missing, are referred to as *terminal nodes*.

Fig. 1 presents an example of a graphical interpretation of a HLDD.

2.2 Modeling digital systems by HLDDs

In HLDD models representing digital systems, the non-terminal nodes correspond to conditions or to control signals, and the terminal nodes represent operations (functional units). Register transfers and constant assignments are treated as special cases of operations. When representing systems by decision diagram models, in general case, a network of HLDDs rather than a single HLDD is required. During the simulation in HLDD systems, the values of some variables labeling the nodes of a HLDD are calculated by other HLDDs of the system. Fig. 2 presents an example of an HLDD for two variables, state and RMAX in the ITC99 benchmark b04.

3. Code Coverage Analysis on HLDDs

3.1 Simulation at RTL and behavioral levels

The basis for code coverage analysis in this paper is a simulator engine relying on HLDD models. We have implemented an algorithm supporting, both, Register- Transfer Level (RTL) and behavioral design abstraction levels. In the RTL style, the algorithm takes the previous time step value of variable x_j labeling a node m_i if x_j represents a clocked variable in the corresponding HDL. Otherwise, the present value of x_j will be used.

In the case of behavioral HDL coding style HLDDs are generated and ranked in a specific order to ensure causality. For variables x_j labeling HLDD nodes the previous time step value is used if the HLDD diagram calculating x_j is ranked after current decision diagram. Otherwise, the present time step value will be used.

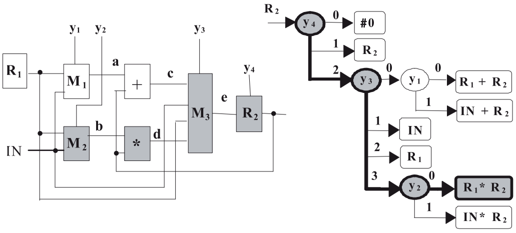
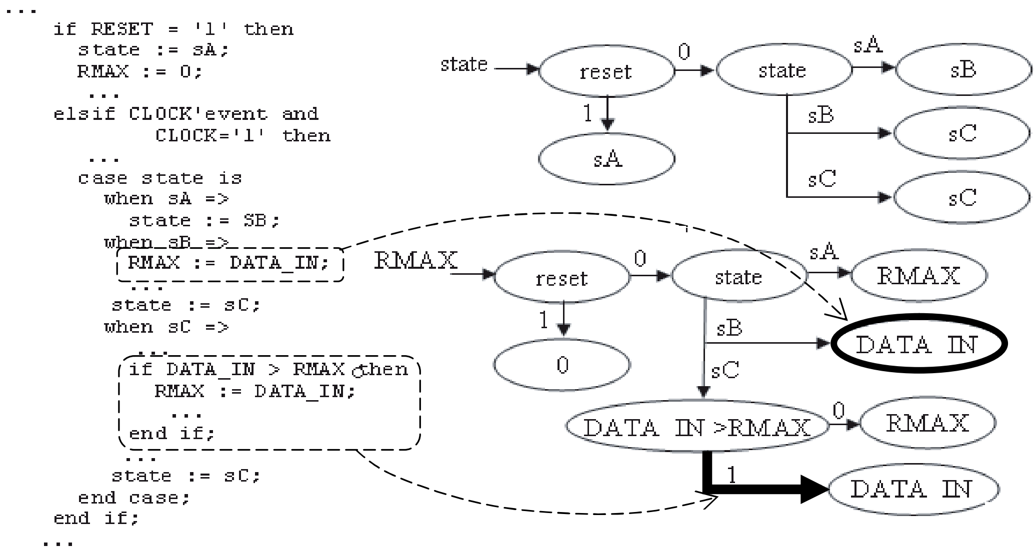


Figure 3. HLDD for an RTL datapath



```
...
    if RESET = '1' then
      state := sA;
      RMAX := 0;
       ...
    elsif CLOCK'event and
            CLOCK='1' then
       ...
      case state is
        when sA =>
          state := SB;
        when sB =>
        RMAX := DATA_IN;
         ...
        state := sC;
        when sC =>
          ...
        if DATA_IN > RMAX then
          RMAX := DATA_IN;
           ...
        end if;
         ...
        state := sC;
      end case;
    end if;
  ...
```

Fig. 2. b04 example: HLDDs for variables *state* and *RMAX*

Algorithm 1 presents the HLDD based simulation engine for RTL, behavioral and mixed HDL description styles.

Algorithm 1. RTL/behavioral simulation on HLDDs

```
For each diagram G in the model
    $m_{Current} = m_0$
    Let $x_{Current}$ be the variable labeling $m_{Current}$
    While $m_{Current}$ is not a terminal node
        If   is $x_{Current}$ clocked or its DD is ranked after G then
            Value = previous time-step value of $x_{Current}$
        Else
            Value = present time-step value of $x_{Current}$
        End if
        If Value $\in$ D($e_{active}$), $e_{active}$ =( $m_{Current}$, $m_{Next}$) then
            $m_{Current} = m_{Next}$
        End if
    End while
    Assign $x_{Current}$ to the DD variable $x_G$
End for
```

3.2 Advantages of HLDD modeling

As an example of modeling systems by HLDDs, consider a subnetwork of a digital system and its HLDD depicted in Figure 3. Here, R_1 and R_2 are registers (R_2 is also a primary output), M_1, M_2 and M_3 are multiplexers, + and * denote adder and multiplier, *IN* is an input bus, y_1, y_2, y_3 and y_4 serve as input control variables, and *a, b, c, d, e* denote internal buses. In the HLDD, the control variables y_1, y_2, y_3 and y_4 are labeling internal decision nodes of the HLDD with their values shown at edges. The terminal nodes are labeled by constant *#0* (reset of R_2), by word variables R_1 and R_2 (data transfers to R_2), and by expressions related to data manipulation operations of the network. By bold lines and colored nodes, a full activated path in the HLDD is shown from $X(m^0)=y_4$ to a terminal node $X(m^T)=R_1*R_2$, which corresponds to the pattern $y_4=2$, $y_3=3$, and $y_2=0$. The part of the network that is activated by this pattern is denoted by colored boxes.

The main advantage and motivation of using high-level DDs compared to the netlists of primitive functions are the increased efficiency of simulation and diagnostic modeling because of direct and compact presentation of cause-effect relationships. For example, instead of simulating the control word y_1, y_2, y_3, y_4 = {0, 0, 3, 2} by computing the functions
$a = R_1$, $b = R_1$, $c = a + R_1$, $d = b * R_1$, $e = d$, and $R_2 = e$,
we need only to trace the nodes y_4, y_3 and y_2 on the HLDD and compute a single operation $R_2 = R_1 * R_2$.

3.3 Mapping coverage metrics to HLDD

In order to analyze quality of verification of hardware designs translated to HLDDs three traditional coverage metrics were chosen and built in to the HLDD based simulation tool. These include statement coverage, branch coverage and toggle coverage. As it was mentioned above, the statement coverage measures the number of times every instruction is exercised by the program stimuli. Toggle coverage shows whether and how many times nodes in the design toggle, i.e. how many bits change their state from 0 to 1 or vice versa. In the case of branch coverage, we measure the number of times each branch in the control flow graph of the code is taken or not taken under the set of program stimuli.

The statement coverage maps directly to the ratio of nodes $m_{Current}$ traversed during the HLDD simulation presented in Algorithm 1. For example, see Fig. 2 for HLDD representations of state and data register variables in a VHDL design. Covering all nodes in the HLDD model corresponds to covering all statements in the respective HDL. However, the opposite is not true. HLDD node coverage is slightly more stringent that HDL statement coverage. This is due to the fact that in HLDDs diagrams are generated to each data variable separately. Such partition on variables includes an additional context to statement coverage.

Similar to the statement coverage, branch coverage has also very clear representation in HLDD simulation. The ratio of every edge e_{active} activated in the simulation process of Algorithm 1 constitutes to HLDD branch coverage.

HLDD toggle coverage is calculated similarly to traditional HDL toggle coverage. However, in this paper a more stringent approach has been selected, where, both, rising and falling front toggling are counted separately. Furthermore, toggling is measured in HLDD nodes, which outnumber the HDL variables.

For example, the branch coverage item corresponding to DATA_IN > RMAX = true in the VHDL code of the b04 design maps to the edge denoted by a bold arrow in the HLDD in Figure 2. The statement RMAX := DATA_IN is represented by the terminal node surrounded by bold circle in the corresponding HLDD.

3.4 Observation coverage on HLDDs

Keeping track of covered lines of code does not generally reflect if the respective items influence the primary outputs of the system. The quality of validation is low when only code coverage items corresponding to the internal lines of the system are exercised but not propagated to the system outputs. Furthermore, while the general function of the system is specified at the outputs the internal signals may be difficult for the designer or verification engineer to comprehend and verify. Fallah et al. [13] propose observation coverage in their method called OCCOM, where simplified fault grading is carried out in order to asses, which code items have been covered and propagated to an observable output. They show that 100 % code coverage corresponds to 60-80 % observable coverage in the worst case.

However, the OCCOM method [13] and its recent improvements are based on representing the effect of an error by a tag that can propagate through the circuit according to a set of rules similar to the D calculus. The main problem of the method is that it over-simplifies the fault-effect propagation. In this paper, we propose an alternative approach, where a straightforward analysis on HLDDs is used for observability analysis.

Observability analysis on HLDDs is carried out as follows. First, the HLDD G, where the code coverage item has been detected is set to a faulty value by inverting the real simulated value. Then, concurrent fault simulation on HLDDs is carried out. This high-level fault simulation allows analysis of several code coverage items in parallel. The procedure is explained below.

In fault propagation for the digital system $S=(X,F)$ through a high-level block with a function $x = f(x_1, x_2, ..., x_n) = f(X')$, $X' \subseteq X$, which is represented by a HLDD G_x, we proceed from the fact that the errors may have been propagated to all of the variables $x_i \in X'$ used in labels of nodes in the graph. To each node m of the HLDD with the label $X(m)$, a complex pattern $T_{X(m)} = \{P_{X(m),0}, (P_{X(m),1}, D_{X(m),1}), \ldots, (P_{X(m),k}, D_{X(m),k})\}$ corresponds. From this pattern, it results that a set of error values $D_{X(m)} = D_{X(m),1} \cup \ldots \cup D_{X(m),k}$ has been propagated to the node m. Let D be the set of all errors currently activated and listed in $T_{X(m)}$.

Consider the fault simulation on the HLDD G_z as the following set of procedures.

Procedure 1. The fault-free path is simulated in accordance to the fault-free input pattern $P_{X(m),0}$, and the fault-free value of $x=X(m^{T,0})$ is calculated, where $m^{T,0}$ is the terminal node of the fault-free activated path.

Let us denote the set of all nodes traced in the fault-free path up to the node m (m itself not included) by $M_{FF}(m)$. Let $D_{FF}(m)$ be the set of all faults propagated to the nodes $m \in M_{FF}(m)$. The condition of reaching the node m in the fault-free path during fault simulation is the absence of all the faults in $D_{FF}(m)$. Denote by $D_{CF}(m)$ the set of faults consistent to the current faulty path from the initial node m_0 up to the node m. For the nodes m on the fault-free path we have $D_{CF}(m) = D - D_{FF}(m)$.

Let us denote by L the list of all nodes of the HLDD to be fault simulated. All the nodes met on the fault-free path are included into dynamic list L. For carrying out fault simulation of the nodes in L, either Procedure 2 (for terminal nodes) or Procedure 3 (for nonterminals) will be used. As the result of the procedures the list L will be updated. Fault simulation is terminated when the list L gets empty.

Procedure 2. Fault simulation of a terminal node $m^{T,0} \in L$ with the function $x = X(m^{T,0}) = f(x_1, \ldots, x_p)$ for the set of complex input patterns $T = (T_1, \ldots, T_p)$, $T_i = \{P_{i,0}, (P_{i,1}, D'_{i,1}), \ldots, (P_{i,ki}, D'_{i,ki})\}$, $i = 1,2, \ldots, p$, where $\forall i,j$: $D'_{i,j} = (D_{i,j} - D_{FF}(m^{T,0})) \cap D_{CF}(m)$.

Procedure 3. Fault simulation of a nonterminal node $m \in L$ with the variable $X(m)$ for the complex pattern $T_{X(m)} = \{P_{X(m),0}, (P_{X(m),1}, D'_{X(m),1}), \ldots, (P_{X(m),km}, D'_{X(m),km})\}$ where $\forall j$: $D'_{X(m),j} = (D_{X(m),j} - D_{FF}(m)) \cap D_{CF}(m)$, consists of the following:

- If m belongs to the fault-free path, and if $D'_{X(m)} = D'_{X(m),1} \cup \ldots \cup D'_{X(m),k} = \varnothing$ then no nodes will be included into L;
- If m does not belong to the fault-free path, and if $D'_{X(m)} = \varnothing$, the successor node m^e which will be selected by the value $X(m) = e = P_{X(m),0}$, will be included into L; for the new node m^e in L we calculate: $D_{FF}(m^e) = D_{FF}(m) \cup D_{z(m^e)}$, and $D_{CF}(m^e) = D_{CF}(m)$,
- If $D'_{X(m)} \neq \varnothing$, all the nodes m^e, where $e = P_{X(m),i}$, i: $D'_{X(m),i} \neq \varnothing$, will be included into L; for all these nodes we calculate $D_{CF}(m^e) = D_{CF}(m) \cap D'_{X(m),i}$, $D_{FF}(m^e) = D_{FF}(m)$.

As a result of the fault simulation by Procedures 2 and 3 we create a complex pattern for the variable x: $T_x = \{P_{x,0}, (P_{x,1}, D_{x,1}),$

…, $(P_{x,k},D_{x,k})\}$. All the pairs $(P_{x,i},D_{x,i})$ where $P_{x,i} = P_{x,0}$ are eliminated since the faults $D_{z,i}$ are self-masked at this point. All the groups of pairs $\{(P_{x,i},D_{x,i}), (P_{x,j},D_{x,j})\}$ where $P_{x,i} = P_{x,j}$ are merged into a single pair $(P_{x,i},D_{x,i})$, so that $D_{x,i} = D_{x,i} \cup D_{x,j}$.

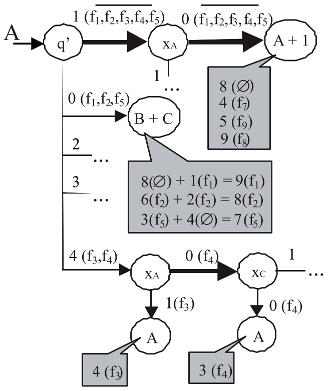


Figure 4. Observability analysis on the HLDD

Example: Consider the HLDD G_A in Fig.3 with a set of complex patterns:

$T_q = \{1, 0\ (f_1,f_2,f_5), 4\ (f_3,f_4)\}$,
$T_{xA} = \{0, 1\ (f_3,f_5)\}$,
$T_{xC} = \{1, 0\ (f_4,f_6)\}$,
$T_A = \{7, 3\ (f_4,f_5,f_7), 4\ (f_1,f_3,f_9), 8\ (f_2,f_8)\}$,
$T_B = \{8, 3\ (f_4,f_5), 4\ (f_3,f_7), 6\ (f_2,f_8)\}$,
$T_C = \{4, 1\ (f_1,f_3,f_4), 2\ (f_2), 5\ (f_7)\}$.

All the paths traced during the fault simulation are highlighted and marked by details of simulation in Fig.4. The fault free paths are shown by bold lines. The edges on paths in Fig.4 are labeled by pairs $e,(D)$, where e is the value of the node variable when leaving the node at this direction, and D is a subset of faults: $D_{FF}(m)$ for the next node m on the fault-free path, and $D_{CF}(m)$ for the next node m on the faulty paths. Since $D_{FF}(x_A) = \{f1,f2,f3,f4,f5\}$ includes both of the faults f3 and f5 propagated to x_A, no faulty paths are simulated from the node x_A: for the value $x_A = 1$: $D'_{xA} = (D_{xA} - D_{FF}(x_A)) = \varnothing$. From all the faults propagated to A, only the faults f7, f8 and f9 are simulated at the node A+1. At the terminal node B+C only the faults f1, f2, f5 are simulated, since only they are consistent to the condition of leaving the node q in this direction.

After fault simulation of all 3 reached terminal nodes we compose the final result as follows: the fault 2 propagated to the node B+C is self-masked because the value B+C = 8 calculated for the fault f2 is equal to the fault-free value calculated at the node A+1. The faults f3 and f7 propagated to different terminal nodes are merged into the same group because they produce the same new value 4 for A. Also, the faults f1 and f8 are merged into the same group. The final value of the new complex pattern is: $T_A = \{8, 3(f4), 4(f3,f7), 5(f9), 7(f5), 9(f1,f8)\}$.

4 Experimental results

Comparative experiments between the HLDD-based code coverage analysis tool implemented in this paper and a popular HDL commercial simulation tool were carried out. Table 1 presents the circuits from the ITC99 benchmark family [12] and the Greatest Common Divisor (GCD) example to run the experiments.

Table 2 shows the comparison between traditional code coverage assessment (statement, branch and toggle coverage) carried out by a state-of-the-art commercial HDL simulator and by the HLDD-based simulator (implementing node, edge, toggle coverage, respectively). The code has been exercised using random set of stimuli of different length. The experiments were run on AMD Athlon 64 Processor 3000+, 1.80 GHz, 2.00GB of RAM, Windows XP.

While there is no definite advantage of the speed of basic logic simulation of benchmarks to either of the tools it should be noted that the overhead of coverage checking in Modelsim is much higher than in the case of HLDDs (See columns (5) and (8)). **When HLDDs have coverage calculation overhead for 10000 patterns in a 1 to 4 % range, the commercial simulator uses 28 up to 78 % extra time.** This result proves HLDD based code coverage analysis especially promising. As a future work we plan to incorporate the advanced features from event-driven cycle-based HLDD simulation algorithms [5] into our coverage analysis tool.

5 Conclusions

The paper presented a new approach to analyzing validation code coverage metrics using High-Level Decision Diagrams. A technique was proposed, where fast HLDD based simulation was extended to support code coverage analysis. We showed how classical code coverage metrics can be mapped to HLDD constructs. Moreover, we introduced an approach for observability coverage analysis on HLDD models. Experiments on ITC99 benchmark circuits indicated the feasibility of the proposed approach.

Acknowledgements

The work has been supported partly by EC FP 6 research project VERTIGO FP6-2005-IST-5-033709, Enterprise Estonia funded ELIKO Development Center and Estonian SF grants 6717, 7068 and 7483.

Table 1. ITC99 benchmark circuits

design	# of lines	# of inputs	# of outputs	# of signals	# of HLDD nodes
b00	76	4	2	7	37
b04	84	6	1	14	58
b09	102	4	1	9	44
b10	169	10	3	14	116

Table 2. Comparison of traditional and HLDD-based code coverage measurement execution times

design (1)	test length (2)	Commercial HDL simulator			HLDD simulator		
		simulation time, s		ratio (4)/(3) (5)	simulation time, s		ratio (7)/(6) (8)
		w/o coverage (3)	w coverage (4)		w/o coverage (6)	w coverage (7)	
b00	1000	0.0053	0.0061	1.151	0.016	0.020	1.25
	5000	0.0137	0.0173	1.263	0.046	0.048	1.043
	10000	0.0243	0.0311	1.280	0.099	0.100	1.010
b04	1000	0.0051	0.0060	1.176	0.019	0.022	1.158
	5000	0.0131	0.0166	1.267	0.051	0.053	1.039
	10000	0.0227	0.0300	1.322	0.106	0.107	1.009
b09	1000	0.0056	0.0077	1.375	N/A*	0.007	N/A*
	5000	0.0151	0.0262	1.735	0.010	0.012	1.200
	10000	0.0270	0.0483	1.789	0.023	0.024	1.043
GCD	1000	0.0052	0.0061	1.173	N/A*	0.003	N/A*
	5000	0.0135	0.0178	1.319	0.015	0.016	1.067
	10000	0.0240	0.0316	1.317	0.031	0.032	1.032

* - N/A : the run time was too short to measure.

References

[1] International Technology Roadmap for Semiconductors 2006 report, [URL] www.itrs.net, 2007

[2] S. Tasiran, K. Keutzer, *Coverage metrics for functional validation of hardware designs*. Design & Test of Computers, IEEE, Volume 18, Issue 4, Jul-Aug. 2001, Pages 36-45.

[3] J. C. Miller, C. J. Maloney, *Systematic Mistake Analysis of Digital Computer Programs*. Communications of the ACM, 1963, Pages 58-63.

[4] William K. Lam, "*Hardware Design Verification: Simulation and Formal Method-Based Approaches*", Pearson Education Inc., 2005, 585 pages.

[5] R. Ubar, J. Raik, A. Morawiec, *Back-tracing and Event-driven Techniques in High-level Simulation with Decision Diagrams*. ISCAS 2000, Vol. 1, pp. 208-211.

[6] Raimund Ubar, Adam Morawiec, Jaan Raik. *Cycle-based Simulation with Decision Diagrams,* Proceedings of the DATE Conference, pp. 454-458, 1999.

[7] J. Raik, R. Ubar, *Fast Test Generation for Sequential Circuits Using Decision Diagrams Representations*. Journal of Electronic Testing: Theory and Applications 16, Kluwer Academic Publisher, 2000, pp. 213-226.

[8] R. Bryant. *Graph-based algorithms for boolean function manipulation.* IEEE Transactions on Computers, C-35, 8:677-691, 1986

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

[10] 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.

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

[12] ITC'99 benchmark homepage, [www] http://www.cerc.utexas.edu/itc99-benchmarks/bench.html

[13] F. Fallah, S. Devadas, K. Keutzer. OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification. *Proc. Design Automation Conference*, pp.152-157, 1998.

Research paper IV

Minakova, K., Reinsalu, U., Chepurov, A., Raik J., Jenihhin M., Ubar, R., Ellervee, P., “High-Level Decision Diagram Manipulations for Code Coverage Analysis”, *The 11th Biennial Baltic Electronics Conference (BEC’08)*, 2008, pp. 207-208

High-Level Decision Diagram Manipulations for Code Coverage Analysis

Karina Minakova, Uljana Reinsalu, Anton Chepurov, Jaan Raik,
Maksim Jenihhin, Raimund Ubar, Peeter Ellervee

Department of Computer Engineering, Tallinn University of Technology, Estonia
{ uljana | anchep | jaan | maksim | raiub }@pld.ttu.ee, LRV@cc.ttu.ee

ABSTRACT: ***Previous works have shown that High-Level Decision Diagrams (HLDD-s) are suitable for system representation for analyzing code coverage metrics. This is due to the fact that HLDD models implicitly represent classical code coverage items, such as statement and branch coverage. However, research on the properties of HLDD-s, which contribute to the accuracy of coverage assessment, is missing. Current paper proposes a set of HLDD manipulations in order to generate diagrams that would allow more stringent code coverage measurement without sacrificing performance, i.e., computation time and memory requirements. The techniques include generation of HLDD-trees from Hardware Description Language (HDL) descriptions and two types of HLDD collapsing methods, which are a generalization of the BDD reduction rules. Experiments on ITC99 benchmark circuits show that the code coverage assessment based on the proposed HLDD manipulation is more stringent than what can be achieved with classical methods. At the same time, the model is well scalable because HLDD generation is terminated in the HDL variables.***

1 Introduction

With the increase in size and complexity of modern integrated circuits, it has become imperative to address critical verification issues in the design cycle. The process of verifying correctness of designs consumes between 60% and 80% of design effort [1]. Ensuring functional correctness is the most difficult part of designing a hardware system [2]. One possible way to verify the correctness of a design is by generating different test cases. Due to the fact that it is impractical to verify exhaustively all possible inputs and states of a design, the confidence level regarding the quality of the design must be quantified to control the verification effort. The fundamental question is: How do I know if I have verified or simulated enough? Verification coverage is a measure of confidence and it is expressed as a percentage of items verified out of all possible items. Different definitions of items give rise to different coverage measures or coverage metrics.

Various coverage metrics exist such as code coverage, parameter coverage, and functional coverage. In this paper, only code coverage would be used, which provides insight into how thoroughly the code of a design is exercised by a suite of simulations. The main disadvantage of code coverage metrics lies in the fact that they only measure the quality of the test case in stimulating the implementation and do not necessarily prove its correctness with respect to the specification. On the other hand, code coverage analysis is a well-defined, well-scalable procedure and, thus, applicable to large designs.

Following Miller and Maloney [3], a large variety of code coverage metrics have been proposed, including statement coverage, block coverage, path coverage, branch coverage, expression coverage, transition coverage, sequence coverage, toggle coverage, etc. [2][4]. The *statement coverage* metric measures the percentage of code instructions exercised with respect to total instructions contained in the code by the program stimuli. *Toggle coverage* shows the percentage of bits toggling in the nodes in the design, i.e., how many bits change their state from 0 to 1 or vice versa. In the case of *branch coverage*, we measure the ratio of branches in the control flow graph of the code that are traversed under the set of stimuli. *Path coverage* measures the percentage of paths in the control flow graph is exercised by the stimuli. A potential goal of software testing is to have 100 % path coverage that implies branch and statement coverage. However, full path coverage is a very stringent requirement as the number of paths in a program may be exponentially related to program size.

Current work is motivated by our previous encouraging research results obtained on HLDD based simulation [5] and test pattern generation [6]. The authors' work in [7] was the first attempt to use HLDD models in validation and code coverage analysis. In [7] we also introduced HLDD model for efficient code coverage analysis and showed how classical coverage metrics map to HLDD constructs. Additionally, we envisioned an algorithm that applied HLDD-s in observability coverage analysis, thus, replacing the classical D-calculus based methods (see, e.g., [8]).

However, research on the properties of HLDD-s, which contribute to the accuracy of coverage assessment, is missing. The paper proposes a set of HLDD manipulations in order to generate diagrams that would allow more stringent code coverage measurement without sacrificing performance, i.e., computation time and memory requirements. The manipulation techniques include generation of HLDD-trees from HDL descriptions and two types of HLDD collapsing methods. Experiments presented in Section 5 show that the code coverage assessment based on the proposed HLDD manipulation is more stringent than what can be achieved with classical methods. The model is well scalable because HLDD generation is terminated in the HDL variables.

2 High-Level Decision Diagrams

Definition: A HLDD representing a discrete function $y=f(x)$ is a directed acyclic 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 $X(m_i)$ returns is the variable letter x_i, which is labeling node m_i. Each node of a HLDD is labeled by a variable. In special cases, nodes can be labeled by constants or algebraic expressions. An edge $e \in E$ of a HLDD is an ordered pair $e=(m_1,m_2) \in E^2$, where E^2 is the set of all the possible ordered pairs in set E. 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 of the variable x_i denoted by X_i, where $e=(m_i,m_j)$. It is required that $Pm_i=\{D(e) \mid e=(m_i,m_j) \in E\}$ is a partition of the set X_i. HLDD has only one starting node (*root node*), for which there are no preceding nodes. The nodes, for which successor nodes are missing, are referred to as *terminal nodes*.

Modeling digital systems by HLDD-s: In HLDD models representing digital systems, the non-terminal nodes correspond to conditions or to control signals and the terminal nodes represent operations (functional units). Register transfers and constant assignments are treated as special cases of operations. When representing systems by decision diagram models, in general case, a network of HLDDs rather than a single HLDD is required. During the simulation in HLDD systems, the values of some variables labeling the nodes of a HLDD are calculated by other HLDDs of the system. Fig. 1 presents an example of an HLDD for two variables, state and RMAX in the ITC99 benchmark b04.

3 Code Coverage Analysis on HLDD-s

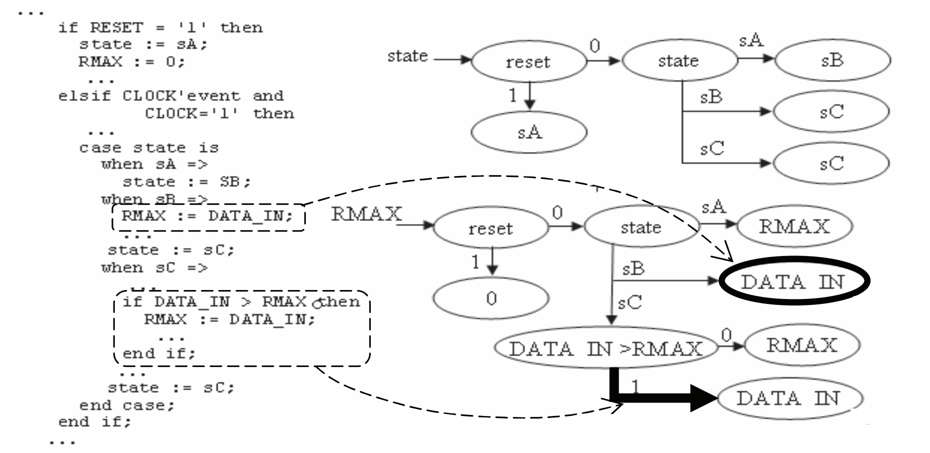
Simulation at RTL and behavioral levels. The basis for code coverage analysis in this paper is a simulator engine relying on HLDD models. We have implemented an algorithm supporting both Register- Transfer Level (RTL) and behavioral design abstraction levels. In the RTL style, the algorithm takes the previous time step value of variable x_j labeling a node m_i if x_j represents a clocked variable in the corresponding HDL. Otherwise, the present value of x_j will be used.

In the case of behavioral HDL coding style, HLDD-s are generated and ranked in a specific order to ensure causality. For variables x_j labeling HLDD nodes the previous time step value is used if the HLDD diagram calculating x_j is ranked after current decision diagram. Otherwise, the present time step value will be used.

Algorithm 1 presents the HLDD based simulation engine for RTL, behavioral, and mixed HDL description styles.

Algorithm 1. RTL/behavioral simulation on HLDDs

```
For each diagram G in the model
    $m_{Current} = m_0$
    Let $x_{Current}$ be the variable labeling $m_{Current}$
    While $m_{Current}$ is not a terminal node
        If   is $x_{Current}$ clocked or its DD is ranked after G then
            Value = previous time-step value of $x_{Current}$
        Else
            Value = present time-step value of $x_{Current}$
        End if
        If Value $\in$ D($e_{active}$), $e_{active}$ =( $m_{Current}$, $m_{Next}$) then
            $m_{Current} = m_{Next}$
        End if
    End while
    Assign $x_{Current}$ to the DD variable $x_G$
End for
```



```
...
    if RESET = '1' then
      state := sA;
      RMAX := 0;
       ...
    elsif CLOCK'event and
            CLOCK='1' then
       ...
      case state is
        when sA =>
          state := SB;
        when sB =>
          RMAX := DATA_IN;
          ...
         state := sC;
        when sC =>
          ...
          if DATA_IN > RMAX then
            RMAX := DATA_IN;
              ...
          end if;
          ...
         state := sC;
      end case;
    end if;
  ...
```

Fig. 1. b04 example: HLDDs for variables *state* and *RMAX*

Mapping coverage metrics to HLDD. In order to analyze quality of verification of hardware designs translated to HLDD-s, three traditional coverage metrics were chosen and built in to the HLDD based simulation tool. These include statement coverage, branch coverage, and toggle coverage. As it was mentioned above, the statement coverage measures the number of times every instruction is exercised by the program stimuli. Toggle coverage shows whether and how many times nodes in the design toggle, i.e., how many bits change their state from 0 to 1 or vice versa. In the case of branch coverage, we measure the number of times each branch in the control flow graph of the code is taken or not taken under the set of program stimuli.

The statement coverage maps directly to the ratio of nodes $m_{Current}$ traversed during the HLDD simulation presented in Algorithm 1. As an example, Fig. 1 depicts HLDD representations of state and data register variables of a VHDL design. Covering all nodes in the HLDD model corresponds to covering all statements in the respective HDL. However, the opposite is not true. HLDD node coverage is slightly more stringent that HDL statement coverage. This is due to the fact that in HLDD-s diagrams are generated to each data variable separately. Such partition on variables includes an additional context to statement coverage.

Similar to the statement coverage, branch coverage has also very clear representation in HLDD simulation. The ratio of every edge e_{active} activated in the simulation process of Algorithm 1 constitutes to HLDD branch coverage. For example, the branch coverage item corresponding to DATA_IN > RMAX = true in the VHDL code of the b04 design maps to the edge denoted by a bold arrow in the HLDD in Fig. 1. The statement RMAX := DATA_IN is represented by the terminal node surrounded by bold circle in the corresponding HLDD.

4 HLDD manipulations for code coverage

The main contribution of this paper is the new HLDD manipulation technique allowing efficient code coverage analysis. In fact, if HLDD is generated for each output variable and the generation process is terminated at the primary input signals then code coverage analysis for the diagram will be equivalent to the path coverage metric. However, as it was mentioned above, enumerating all the paths through a design is infeasible and it is easy to see that the corresponding HLDD may be of exponential size.

Therefore, another approach is adopted in this paper that differs from the traditional one of generating a diagram for each primary output. When representing systems by decision diagram models, a network of HLDD-s is implemented where each internal HDL variable has its corresponding HLDD. During the simulation in HLDD systems, the values of some variables labeling the nodes of a HLDD are calculated by other HLDD-s of the system. Such partitioning helps avoiding the node explosion problem of DD-s and keeps the size requirements for resulting HLDD systems acceptable.

The method proposed for generating HLDDs suitable for code coverage analysis is similar to BDD reduction rules [9] and it consists of the following steps:

1. Generate a HLDD tree for each system variable
2. Reduce nodes with identical succeeding subgraphs
3. Unite identical terminal nodes

The above steps are explained by an example presented in Fig. 2, which depicts HLDD manipulations for the '*state'* variable of the b04 design presented in Fig. 1. As the first step, a HLDD tree for variable *v* is generated by traversing the full control flow graph of the design and collecting the values assigned to *v* at each control step. If the value of *v* does not change at current control step then terminal node with the present value of variable will be created. Fig. 2a shows the HLDD generated for the variable *state* in b04.

Then, reduction rules are applied to eliminate nodes for which all successor nodes (in general case, succeeding subgraphs) are identical. As a result a reduced HLDD is obtained (Fig. 2b). Finally, we create a minimized reduced HLDD by uniting identical terminal nodes (Fig. 2b). HLDD generation experiments on a set of ITC99 benchmarks show that around 45-80% of nodes are removed by the reduction step from the initial HLDD tree. Further 40-60% of nodes will be eliminated by the minimization step.

In this paper, we propose reduced HLDD-s as a suitable model for code coverage analysis because it provides for more stringent coverage metrics than minimized HLDD-s. At the same time it is a more compact representation than full HLDD trees. Furthermore, in terms of speed of simulation reduced HLDD offers equal per-

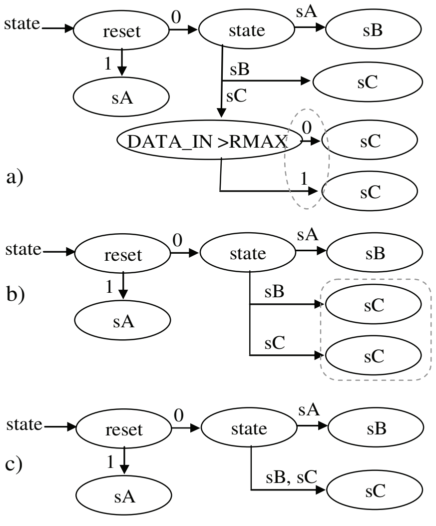


Fig. 2. a) HLDD tree, b) reduced HLDD and c) minimized reduced HLDD

formance when compared to the minimized model. This is because of the fact that by both models the number of edges to be traversed is exactly the same. However, in full trees the number of diagram edges would be considerably higher.

5 Experimental results

Comparative experiments between the HLDD-based code coverage analysis tool implemented in this paper and a popular HDL commercial simulation tool were carried out on circuits belonging to the ITC99 benchmark family. While there was no definite advantage of the speed of basic logic simulation of benchmarks to either of the tools, the overhead of coverage checking in the popular commercial tool is much higher than in the case of HLDD-s. When HLDD-s have penalty for coverage calculation time in a 1% to 4% range, the commercial simulator uses from 28% up to 78% extra time for coverage assessment (see Table 1).

Table 2 presents the characteristics of the different HLDD representations introduced in Section 4. The columns *min*, *red.* and *tree* show the number of nodes/edges in minimized HLDD, reduced HLDD and HLDD tree models, respectively. From the Table, it can be seen that around 45-80 % of nodes are removed by the reduction step from the initial HLDD tree. Further 40-60 % of nodes will be eliminated by the minimization step.

Table 3 compares code coverage analysis comparing statement coverage and branch coverage assessment results on reduced HLDD-s (*red.*), on minimized HLDD-s (*min*) and on a well-known commercial tool using the same set of input stimuli for all three models. As it can be seen from the experiments, the reduced HLDD model always achieves the best (i.e. most stringent results) of all three. The minimized HLDD has the poorest outcome for statement coverage and traditional HDL simulator is the weakest for measuring branch coverage in most cases.

6 Conclusions

The paper presented a set of straight-forward manipulations on High-Level Decision Diagrams to support validation code coverage analysis. Experiments on ITC99 benchmark circuits showed that the reduced HLDD model proposed in this work offers higher accuracy in statement and branch coverage analysis than traditional models. The gain in accuracy is achieved only with a slight increase in memory requirements. The simulation times for all three models are nearly identical.

Acknowledgements

The work has been supported partly by EC FP 6 research project VERTIGO FP6-2005-IST-5-033709, Enterprise Estonia funded ELIKO Development Center and Estonian SF grants 6717, 7068 and 7483.

References

[1] ITRS 2006 report, [URL] www.itrs.net, 2006

[2] S. Tasiran, K. Keutzer, Coverage metrics for functional validation of hardware designs. *Design & Test of Computers*, IEEE, Volume 18, Issue 4, Jul-Aug. 2001, pp. 36-45.

[3] J. C. Miller, C. J. Maloney, Systematic Mistake Analysis of Digital Computer Programs. *Comm. ACM*, 1963, pp. 58-63.

[4] William K. Lam, “Hardware Design Verification”, Pearson, 2005, 585 p.

[5] R. Ubar, J. Raik, A. Morawiec, Back-tracing and Event-driven Techniques in High-level Simulation with Decision Diagrams. *ISCAS 2000*, Vol. 1, pp. 208-211.

[6] J. Raik, R. Ubar, Fast Test Generation for Sequential Circuits Using Decision Diagrams Representations. *JETTA*, 16, Kluwer, 2000, pp. 213-226.

[7] J. Raik, et al. Fast Code Coverage Analysis using High-Level Decision Diagrams. *DDECS*, 2008.

[8] F. Fallah, S. Devadas, K. Keutzer. OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification. *Proc. DAC*, pp.152-157, 1998.

[9] R. E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Trans. on Computers*, August, 1986.

Table 1. Coverage analysis penalty: traditional vs HLDD [7]

design	Overhead time of analysis, %	
	Commercial HDL simulator	HLDD simulator
b00	28.0	1.0
b04	32.2	0.9
b09	78.9	4.3
GCD	31.7	3.2

Table 2. Characteristics of different HLDD manipulations

Design	# of nodes			# of edges	
	min	red.	Tree	min	red.
b01	30	57	267	52	62
b02	16	26	48	24	24
b06	47	116	440	83	111
b09	44	69	125	62	64

Table 3. Comparison of code coverage analysis results

Design	Test len.	Statement coverage, %			Branch coverage, %		
		red.	min	HDL	red.	min	HDL
b01	14	**86.0**	100	93.8	**74.2**	84.6	88.9
	23	**96.5**	100	100	**90.3**	100	100
b02	10	**92.3**	100	96.3	**91.7**	**91.7**	93.8
	14	**100**	**100**	**100**	**100**	**100**	**100**
b06	11	**80.2**	100	85.5	**79.3**	89.2	87.5
	52	**98.3**	100	100	**98.2**	100	100
b09	23	**87.0**	100	100	**85.9**	87.1	100
	33	**100**	**100**	**100**	**100**	**100**	**100**

Curriculum Vitae in English

Personal data

Name	Uljana Reinsalu
Date and place of birth	07.06.1981, ESTONIA
Citizenship	Estonian

Contact data

Address	Akadeemia tee 15A, Tallinn 12618, ESTONIA
E-mail	uljana@ati.ttu.ee

Education

2005 - ...	Ph.D. student in Computer Engineering, Tallinn University of Technology
2003 - 2005	Tallinn University of Technology, Department of Computer and Systems Engineering (MSc)
2002 - 2003	Alari University, “Master of Engineering in Embedded Systems Design”
1999 - 2002	University of Tartu, “Physical Information Technology” at the Faculty of Physics and Chemistry
1997 - 1999	Tallinna Tõnismäe Reaalkool

Career

2012 - ...	Tallinn University of Technology, Faculty of Infotechnology, Dept. of Computer
Engineering	
	Chair of Computer Engineering and Diagnostics, teaching assistant
2008 - 2012	Maternity leave

2005 - 2008	Tallinn University of Technology, department of Computer Engineering, chair of Digital Systems, teaching assistant
2004 - 2005	Schneider Electric Eesti AS, Java developer

Honours & Awards

2012-2013 -- IT Akadeemia scholarship

2008 - Anita Borg Europe scholarship

2005-2008 – "Tiger University" grant for ICT PhD students, Estonian Information Technology Foundation (EITSA)

2002 - medal from Ministry of Education given by Estonian president A. Rüütel in honour of the best graduating students

2002 - scholarship from Rotalia fund

2002 - ALaRI scholarship for covering tuition fees and accommodation costs

Curriculum Vitae eesti keeles

Isikuandmed

Nimi	Uljana Reinsalu
Sünniaeg ja -koht	07.06.1981, EESTI
Kodakondsus	Eesti

Kontaktandmed

Aadress	Akadeemia tee 15A, Tallinn 12618, EESTI
E-post	uljana@ati.ttu.ee

Hariduskäik

2005 - ...	doktorant, Tallinna Tehnikaülikool, Arvutitehnika Instituut
2003 - 2005	Tallinna Tehnikaülikool, Arvuti- ja süsteemitehnika eriala (MSc)
2002 - 2003	Alari Ülikool, “Master of Engineering in Embedded Systems Design” (Šveits)
1999 - 2002	Tartu Ülikool, füüsika-keemia teaduskond, “Füüsikaline infotehnoloogia” eriala
1997 - 1999	Tallinna Tõnismäe Reaalkool

Teenistuskäik

2012 - ...	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Arvutitehnika Instituut, assistent
2008 – 2012	lapsehoolduspuhkus
2005 - 2008	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Arvutitehnika Instituut, assistent
2004 - 2005	Schneider Electric Eesti AS, Java arendaja

Teaduspreemiad ja -tunnustused

2012 - 2013 -- IT Akadeemia stipendium

2008 - Anita Borg Europe stipendium

2005-2008 - "Tiigriülikooli" stipendium IKT doktorantidele
(EITSA)

2002 - Haridusministeeriumi medal parimate kõrgkooli lõpetajate vastuvõtult president A. Rüütli poolt

2002 - Rotalia fondi stipendium

2002 - ALaRI stipendium AlaRI ülikooli õpinguteks ja elamiseks Šveitsis

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 Bio-impedance. 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 Identification 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.

78. **Taavi Viilukas**. Constraints Solving Based Hierarchical Test Generation for Synchronous Sequential Circuits. 2012.

79. **Marko Kääramees**. A Symbolic Approach to Model-based Online Testing. 2012.

80. **Enar Reilent**. Whiteboard Architecture for the Multi-agent Sensor Systems. 2012.

81. **Jaan Ojarand**. Wideband Excitation Signals for Fast Impedance Spectroscopy of Biological Objects. 2012.

82. **Igor Aleksejev**. FPGA-based Embedded Virtual Instrumentation. 2013.