THESIS ON INFORMATICS AND SYSTEM ENGINEERING C42

Simulation-Based Hardware Verification with High-Level Decision Diagrams

MAKSIM JENIHHIN

TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Engineering
Chair of Computer Engineering and Diagnostics

This dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer and Systems Engineering on November 5, 2008

Supervisors: Dr. Jaan Raik
Prof. Raimund Ubar

Opponents: Prof. Franco Fummi, University of Verona, Italy
Dr. Rainer Dorsch, IBM, Böblingen, Germany

Defence: December 8, 2008

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 before for any degree or examination.

/Maksim Jenihhin/

Copyright: Maksim Jenihhin, 2008

ISSN 1406-4731
ISBN 978-9985-59-863-4

INFORMAATIKA JA SÜSTEEMITEHNIKA C42

Simuleerimisel põhinev riistvara verifitseerimine kõrgtaseme otsustusdiagrammidel

MAKSIM JENIHHIN

To my parents Vyacheslav and Olga

Abstract

This thesis addresses the main simulation-based hardware verification issues that are speed and accuracy of the verification process. In particular we target aspects of assertion checking and coverage measurement by exploiting advantages of High-Level Decision Diagrams (HLDD) design representation model.

First, we present a novel method for assertion checking based on HLDD model. The presented approach proposes a temporal extension for the existing HLDD model aimed at supporting temporal properties expressed in Property Specification Language (PSL). Other contributions of this method are methodology for direct conversion of PSL properties to HLDD and HLDD-based simulator modification for assertions checking support.

Second, we present a novel method for verification structural coverage analysis based on HLDD model. The main contributions of this method include approaches for mapping traditional code coverage metrics such as statement, branch and data flow coverage to HLDD constructs. Another contribution is an approach for condition coverage metric analysis. It employs a hierarchical decision diagrams model consisting of HLDDs and BDD-based representations of the conditional statements. The method also implies HLDD model manipulations targeting different aspects of verification coverage analysis.

The proposed methods rely on homogeneous hardware verification flow based on HLDD model. Previous research works have shown that HLDDs are an efficient model for simulation and convenient for diagnosis and debug. The performed experiments demonstrate feasibility and efficiency of the proposed approaches.

Kokkuvõte

Antud töö on suunatud simuleerimisel-põhineva digitaalriistvara verifitseerimise kiiruse ja täpsuse tõstmisele. Töös on pakutud lähenemised väidete kontrolli ja verifitseerimise katte mõõtmise jaoks, mis rakendavad kõrgtaseme otsustusdiagrammide (KTOD) eeliseid skeemide esitamisel.

Esiteks on esitatud uudne meetod väidete kontrolliks, mis põhineb KTOD mudelil. Esitatud lähenemine pakub välja temporaalse laienduse olemasolevale KTOD mudelile, mis on mõeldud Property Specification Language (PSL) keeles esitatud omaduste toetamiseks. Lisaks on töös esitatud metodoloogia PSL omaduste vahetuks konverteerimiseks KTOD mudelisse ja KTOD simulaatori edasiarendus väidete kontrolli toetamiseks.

Teiseks on dissertatsioonis välja töötatud meetod verifitseerimise struktuurse katte KTOD mudelil põhinevaks analüüsiks. Meetodi peamiseks panuseks on traditsiooniliste kattemõõtude, nagu lausete, harude ja andmevoo katete sidumine KTOD struktuuriga. Lisatulemuseks on lähenemine tingimuste katte analüüsiks. Vimmane kasutab hierarhilist otsustusdiagrammide mudelit, mis koosneb KTOD-dest ja tingimuslike lausete binaarsetel otsustusdiagrammidel põhinevast esitusest. Samuti sisaldab pakutud meetod KTOD mudeli teisendusi, mis on suunatud verifitseerimise katte analüüsi erinevatele tasemetele.

Pakutud meetodid toetuvad homogeensele KTOD-l põhinevale riistvara verifitseerimise voole. Eelnev uurimistöö on näidanud, et KTOD on efektiivne mudel simuleerimise läbiviimiseks ning sobilik digitaalsüsteemide diagnostikat ja silumist silmas pidades. Dissertatsioonis teostatud katsed tõestavad pakutud lähenemiste rakendatavust ja efektiivsust.

Acknowledgements

I would like to express my sincere gratitude to everybody who has either directly or indirectly contributed to my PhD studies and this thesis.

In particular, I would like to thank my supervisors. I very much appreciate the support and advice of Dr. Jaan Raik. He has been guiding me through my PhD studies and has been not only an excellent teacher but also a friend. I am very much thankful to Prof. Raimund Ubar for bringing me to the exiting world of science and research. I was always getting a wise advice from him just at the time when it was needed.

I would also like to thank all my colleagues from Tallinn University of Technology and especially from Dept. of Computer Engineering who contributed to my work with discussions and ideas. In particular I would like to mention here in alphabetical order: Anton Chepurov, Sergei Devadze, Prof. Peeter Ellervee, Dr. Gert Jervan, Dr. Artur Jutman, Uljana Reinsalu and Assoc.Prof. Aleksander Sudnitsõn.

A special thank should go to the director of Dept. of Computer Engineering Dr. Margus Kruus for his support with many practical and administrative issues.

I would like to acknowledge several organizations that have supported my PhD studies, including the research presented in this thesis. They are Tallinn University of Technology, Enterprise Estonia funded ELIKO Development Centre, European Commission FP6 research project VERTIGO, National Graduate School in Information and Communication Technologies (IKTDK) and Estonian Information Technology Foundation (EITSA).

Finally, I would like to thank my family for all the patience and support. In particular I would like to mention my parents Vyacheslav and Olga, sister Anastasia and my beloved fiancée Anna. Thank you!

Maksim Jenihhin,
Tallinn, October 2008

Table of Contents

Chapter 1 INTRODUCTION 1
1.1 Motivation 1
1.2 Problem formulation 3
1.3 Contributions 4
1.4 Thesis organization 4
1.4.1 Formatting remarks 5
Chapter 2 BACKGROUND 7
2.1 Design representation by decision diagrams 7
2.1.1 Binary decision diagrams 8
2.1.2 High-level decision diagrams 10
2.1.2.1 HLDD model definition 10
2.1.2.2 Basic simulation on HLDDs 11
2.1.2.3 Pure RTL designs representation by HLDDs 12
2.1.2.4 Behavioural RTL designs representation by HLDDs 15
2.1.2.5 HLDD vs. ADD representations comparison 18
2.1.2.6 HLDD model advantages for debug in verification 20
2.2 Property specification language 21
2.2.1 PSL organization 22
2.2.1.1 Flavors 23
2.2.1.2 Modes 23
2.2.1.3 Layers 24
2.2.1.4 Styles 25
2.2.2 PSL properties 25
2.2.2.1 Operators 26
2.2.2.2 Strong vs. weak operators 28
2.2.2.3 Vacuous pass 28

2.2.2.4 PSL flexibility and common equivalences 28
2.2.3 PSL simple subset 30
2.3 Chapter summary 31
Chapter 3 ASSERTION-BASED VERIFICATION 33
3.1 Overview 34
3.1.1 Design flow 34
3.1.2 Design verification 35
3.1.3 Assertion-based verification 36
3.1.3.1 Diversity of assertion checking 38
3.1.4 State of the art 38
3.1.4.1 An experience with FoCs 40
3.1.5 APRICOT 42
3.2 Temporal extension for the HLDD model 44
3.2.1 THLDD model definition 45
3.2.2 THLDD interface 46
3.2.3 THLDD temporal relationships 47
3.3 PSL to THLDD conversion method 48
3.3.1 Primitive Property Graphs 48
3.3.1.1 PPG Library 49
3.3.2 Parser 50
3.3.3 Constructor 51
3.3.4 Representation types of THLDD properties 55
3.4 The method for assertions checking with HLDD*sim* 58
3.4.1 HLDD*sim* algorithm 58
3.4.2 HLDD*sim* modification for assertions checking 59
3.4.3 THLDD assertions checking timing issues 61
3.5 Experimental results 63
3.6 Verification assertions reuse for manufacturing testing 65
3.6.1 Assumption-based test generation 66
3.6.2 Assertion-based BIST 68
3.6.3 Assertion-based DfT by test points insertion 69
3.7 Chapter summary 70
Chapter 4 VERIFICATION COVERAGE ANALYSIS 71
4.1 Verification coverage overview 71
4.1.1 Verification coverage classification 73
4.1.2 Sufficiency of verification coverage 74
4.1.3 Assertion coverage 75
4.2 HLDD-based analysis of code coverage 76

4.2.1 Statement coverage mapping 77
4.2.2 Branch coverage mapping 78
4.2.3 Toggle coverage mapping 80
4.2.4 State coverage mapping 80
4.2.5 Data flow coverage mapping 81
4.2.6 Condition coverage mapping 81
4.3 A hierarchical approach for HLDD-based condition coverage analysis 82
4.4 HLDD model reduction manipulations for code coverage analysis 85
4.5 Experimental results 89
4.6 Chapter summary 91
Chapter 5 CONCLUSIONS AND FUTURE WORK 93
5.1 Conclusions 93
5.1.1 Contributions 93
5.1.2 Advantages 94
5.2 Future work 95
References 97
Appendix A PPG LIBRARY 107
Appendix B AGM FORMAT 123

List of Publications

HLDD-based functional verification flow

- Maksim Jenihhin, Jaan Raik, Anton Chepurov, Raimund Ubar "Temporally Extended High-Level Decision Diagrams for PSL Assertions Simulation", *Proc. of 13th IEEE European Test Symposium (ETS'08),* Verbania, Italy, May 25-29, 2008, pp. 61-68
- Maksim Jenihhin, Jaan Raik, Anton Chepurov, Raimund Ubar, "PSL Assertion Checking with Temporally Extended High-Level Decision Diagrams", *Proc. of 9th IEEE Latin American Test Workshop (LATW'08)*, Puebla, Mexico, February 17-20, 2008, pp. 49-54
- Maksim Jenihhin, Jaan Raik, Anton Chepurov, Raimund Ubar, "Assertion Checking with PSL and High-Level Decision Diagrams", *Digest of the IEEE 8th Workshop on RTL and High Level Testing (WRTLT'07)*, Beijing, China October 12-13, 2007, pp. 105-110
- Maksim Jenihhin, Jaan Raik, Anton Chepurov, Raimund Ubar, "PSL Assertion Checking Using Temporally Extended High-Level Decision Diagrams", *Journal of Electronic Testing: Theory and Applications (JETTA) [submitted on September 30, 2008]*
- Jaan Raik, Maksim Jenihhin, Anton Chepurov, Uljana Reinsalu, Raimund Ubar, "APRICOT: a Framework for Teaching Digital Systems Verification", *Proc. of 19th EAEEIE Annual Conference, IEEE,* Tallinn, Estonia, June 29 - July 2, 2008, pp. 1-6
- Jaan Raik, Uljana Reinsalu, Raimund Ubar, Maksim Jenihhin, Peeter Ellervee, "Code Coverage Analysis using High-Level Decision Diagrams", *Proc. of the 11th IEEE Workshop on Design and Diagnostics of Electronic Systems (DDECS'08)*, April, 2008, pp. 201-206
- Karina Minakova, Uljana Reinsalu, Anton Chepurov, Jaan Raik, Maksim Jenihhin, Raimund Ubar, Peeter Ellervee, "High-Level Decision Diagram

Manipulations for Code Coverage Analysis", *Proc of the 11th IEEE Biennial Baltic Electronics Conference (BEC'08),* Tallinn, Estonia, October 2008, pp. 207 - 210

- Jenihhin, Maksim, "PSL Assertions based Verification with HLDD Tools", *Proc. of the 2nd IKTDK Conference*, Viinistu, Estonia, May 11-12, 2007, pp. 17 - 20

- Maksim Jenihhin, "Assertion-based verification and testing with Decision Diagrams", *PhD Forum (abstract + poster), Design, Automation and Test in Europe (DATE'08)*, Munich, Germany, March 10-14, 2008

Assertion-based manufacturing test

- Maksim Jenihhin, Jaan Raik, Raimund Ubar, Anton Chepurov, "On reusability of verification assertions for testing", *Proc. of 11th IEEE Biennial Baltic Electronics Conference (BEC'08),* Tallinn, Estonia, October 2008, pp. 151 - 154

- Knut Hermann, Jaan Raik, Maksim Jenihhin "TTBist: a DfT Tool for Enhancing Functional Test for SoC", *Proc. of the Baltic Electronics Conference*, Laulasmaa, Estonia, 2006, pp. 191-194

- Jaan Raik, Maksim Jenihhin, Rain Adelbert, "Sequential Circuits BIST Synthesis from Signal Specifications", *Proc. of IEEE Norchip Conference*, Oulu, Finland, November 21-22, 2005, pp.196 - 199

- Jenihhin, Maksim, "On reusability of verification assertions for testing", *Proc. of the 3rd IKTDK Conference*, Voore, Estonia, April 25-26, 2008, pp. 43-46

High-level and hierarchical manufacturing test

- Jaan Raik, Raimund Ubar, Taavi Viilukas, Maksim Jenihhin, "Mixed Hierarchical-Functional Fault Models for Targeting Sequential Cores", *Journal of Systems Architecture, 54(3-4), Elsevier,* 2008, pp. 465 - 477

- Giuseppe Di Guglielmo, Franco Fummi, Maksim Jenihhin, Graziano Pravadelli, Jaan Raik, Raimund Ubar, "On the Combined Use of HLDDs and EFSMs for Functional ATPG", *Proc. of IEEE East-West Design and Test Symposium,*Yerevan, Armenia, September 7-10, 2007, pp. 503 - 508

- Raimund Ubar, Jaan Raik, Artur Jutman, Maksim Jenihhin, Marina Brik, Martin Instenberg, Heinz-Dieter Wuttke, "Diagnostic Modeling of Microprocessors with High-Level Decision Diagrams", *Proc. of 11th IEEE Biennial Baltic Electronics Conference (BEC'08),* Tallinn, Estonia, October 2008, pp. 147 - 150

- Raimund Ubar, Jaan Raik, Artur Jutman, Maksim Jenihhin, Martin Instenberg, Heinz-Dieter Wuttke, "Modeling Microprocessor Faults on High-Level Decision Diagrams", *Proc. of 2nd Workshop on Dependable and Secure Nanocomputing (WDSN'08),* Anchorage Hilton Hotel, AK, USA, June 2008, pp. 1 - 6

- Raimund Ubar, Sergei Devadze, Maksim Jenihhin, Jaan Raik, Gert Jervan, Peeter Ellervee, "Hierarchical Calculation of Malicious Faults for Evaluating the Fault-Tolerance", *Proc. of IEEE International Symposium on Electronic Design, Test and Applications (DELTA'08)*, Hong Kong, January 23 - 25, 2008, pp. 222-227

- Raimund Ubar, Gert Jervan, Jaan Raik, Maksim Jenihhin, Peeter Ellervee, "Dependability Evaluation in Fault-Tolerant Systems with High-Level Decision Diagrams", *Proc. of the Computer Science meets Automation: 52. IWK - Internationales Wissenschaftliches Kolloquium,* Ilmenau, Germany, 10-13 September, 2007, pp. 147 - 152

Manufacturing test targeting physical defects

- Maksim Jenihhin, Jaan Raik, Raimund Ubar, Witold Pleskacz, Michal Rakowski, "Layout to Logic Defect Analysis for Hierarchical Test Generation", *Proc. of 10th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems(DDECS'07)*, Kraków, Poland, April 11-13, 2007, pp. 35-40

- Witold Pleskacz, Maksim Jenihhin, Jaan Raik, Michal Rakowski, Raimund Ubar, Wieslaw Kuzmicz, "Hierarchical Analysis of Short Defects between Metal Lines in CMOS IC", *Proc. of the 11th Euromicro Conference on Digital System Design (DSD) Architectures, Methods and Tools*, Parma, Italy, September 2008, pp 729 - 734

- Jenihhin, Maksim, "Case Study: Defect-Oriented Testing of a Combinational Circuit", *Proc. of the 1st IKTDK Conference*, Jäneda, Estonia, May 12-13, 2006, pp. 78 - 81

Built-in self test optimization

- Gert Jervan, Petru Eles, Zebo Peng, Raimund Ubar, Maksim Jenihhin "Test Time Minimization for Hybrid BIST of Core-Based Systems", *Journal of Computer Science and Technology, 21(6)*, 2006, pp. 907 - 912

- Raimund Ubar, Maksim Jenihhin, Gert Jervan, Zebo Peng "Hybrid BIST Optimization for Core-based Systems with Test Pattern Broadcasting", *Proc. of the IEEE International Workshop on Electronic Design, Test and Applications (DELTA 2004)*, Perth, Australia, January 28-30, 2004, pp. 3 - 8

- Raimund Ubar, Maksim Jenihhin, Gert Jervan, Zebo Peng "An Iterative Approach to Test Time Minimization for Parallel Hybrid BIST Architecture" *Proc. of 5th IEEE Latin-American Test Workshop (LATW'04)*, Cartagena, Colombia, March 8-10, 2004, pp. 98 - 103

- Gert Jervan, Petru Eles, Zebo Peng, Raimund Ubar, Maksim Jenihhin "Hybrid BIST time minimization for core-based systems with STUMPS architecture", *Proc. of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2003),* Boston, MA, USA, November 3-5, 2003, pp. 225 - 232

- Gert Jervan, Petru Eles, Zebo Peng, Raimund Ubar, Maksim Jenihhin "Test Time Minimization for Hybrid BIST of Core-Based Systems", *Proc. of the 12th IEEE Asian Test Symposium (ATS03)*, Xian, China, November 17-19, 2003, pp. 318 - 323

- Raimund Ubar, Maksim Jenihhin, Gert Jervan, Zebo Peng "Test Time Minimization for Hybrid BIST with Test Pattern Broadcasting", *Proc. of the 21st NORCHIP Conference*, Riga, Latvia, November 10-11, 2003, pp. 112 - 116

- Maksim Jenihhin, "Test Time Minimization for Parallel Hybrid BIST Architectures", *Master thesis, Tallinn University of Technology,* Tallinn, June 2004

- Maksim Jenihhin, "Test Time Minimization for Hybrid BIST of Systems-on-Chip", *Bachelor thesis, Tallinn University of Technology,* Linköping, June 2003

List of Abbreviations

ABV	Assertion-Based Verification
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
CTL	Computation Tree Logic
DD	Decision Diagram
DfT	Design for Testability
DfV	Design for Verifiability
DUV	Design Under Verification
EDIF	Electronic Design Interchange Format
FL	Foundation Language
FoCs	Formal Checkers
FPGA	Field Programmable Gate Array
GCD	Greatest Common Devisor
GDL	General Description Language
GUI	Graphical User Interface
HDL	Hardware Description Language

HIF	HDL Intermediate Format
HLDD	High-Level Decision Diagrams
IEEE	Institute of Electrical and Electronics Engineers
LTL	Linear Time temporal Logic
OBE	Optional Branching Extension
PC	Personal Computer
PPG	Primitive Property Graph
PSL	Property Specification Language
RTL	Register Transfer Level
SERE	Sequential Extended Regular Expressions
SSBDD	Structurally Synthesized Binary Decision Diagrams
THLDD	Temporally extended High-Level Decision Diagrams
TLM	Transaction Level Modelling
TPG	Test Pattern Generation
TUT	Tallinn University of Technology
VHDL	VHSIC (Very-High-Speed Integrated Circuit) Hardware Description Language
WG	Working Group

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 addresses several simulation-based hardware verification issues. The main emphasis is put on assertion checking and structural coverage measurement exploiting advantages of High-Level Decision Diagrams (HLDD) design representation model.

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

1.1 Motivation

Nowadays, it is not easy to realize that mobile phones, so ordinary today, have got a wide spread only ten years ago as well as consumer digital cameras just five years ago. Not to mention the times (slightly more than 15 years ago) when usual people lived without the Internet. The technology advances very rapidly and today we are surrounded by complex electronic devices and embedded systems that have become a common part of our lives. We rely on them and accept their correct behaviour as granted. At the same time we are becoming more and more dependent on them. Minor failures may annoy us while a major one may have a serious catastrophic effect and even cost human lives.

There are known famous cases of a fault occurrence in electronic devices. One of them is the flight tragedy of the European Space Agency's first Ariane 5 launcher on June4, 1996. Its first flight, known as Flight 501, has failed with the rocket self-destructing 37 seconds after launch. This case is sometimes called one of the most expensive "computer bugs" in history. The tragedy was caused purely by the system design error. Further, the official investigation report on this case [87] has concluded: "*The extensive reviews and tests carried out during the Ariane 5 Development Programme did not include adequate analysis and testing ..., which could have detected the potential failure*".

Another well known case is the “Pentium FDIV bug” in several Intel's original Pentium processors’ families [88]. Because of this fault, some particular floating point division operations performed with these processors could produce incorrect results. This fault was also one of the most expensive “computer bugs” and cost Intel Corp. $475 million.

The rapid development of the digital systems requires a huge increase of efforts to verify the functionality, i.e. ascertain that the implemented design meets the intended specification. It is obvious that, an exhaustive functional verification of an average digital hardware design requires exercising an extremely large amount of possible input combinations. For example (the example is from [5]), let us consider a digital hardware design that has 10 inputs and 100 flip-flops (i.e. a bit more than three 32-bit registers). This design would require in the worst case $(2^{10})^{100}$, i.e. 2^{1000}, test vectors to try. If we simulate 1000 test vectors per second, it would take us: $(2^{1000}/(60^2*24*365.25))/1000$ = 339,540,588,380,062,907,492,466,172,668,391, 072,376,037,725,208,993,588,689,808,600,264,389,893,757,743,339,953,988,382, 771,724,040,525,133,303,203,524,078,771,892,395,266,335,942,544,299,458,056, 845,215,567,848,460,205,301,551,163,124,606,262,994,092,425,972,759,467,835, 103,001,336,717,048,865,167,147,297,613,428,902,897,465,679,093,821,978,784, 398,755,534,655,038,141,450,059,156,501 years to execute. Therefore, methods to overcome the complexity, yet provide acceptable results, are vital.

The cause of an electronic system’s failure observable by its final user can hide behind a wide set of the system’s aspects. They include correctness of its software part, physical implementation, analog hardware part, the system timing issues etc. In this thesis we address digital hardware design verification against incorrect implementation of its intended functionality. This is type of design verification is usually referred to as hardware design functional verification. Several other design verification types as well as classification of hardware design functional verification are discussed in more detail in the introduction to Chapter 3.

The growing complexity of the state-of-the-art hardware designs has made their verification a very important phase in the complete development process. As it was estimated in the last International Technology Roadmap for Semiconductors report [102], verification takes roughly 70% of design time, and therefore demands a huge amount of expensive resources such as man- or CPU-hours. This part of complete system development is often the most expensive phase. According to [102], the problem is caused by a pair of recent processes. They are, first, rapid design complexity increase and, second, the historically greater emphasis on other aspects of the design process that has produced significant progress in this area (e.g. automated tools for logic synthesis) leaving verification as the bottleneck.

Hardware verification is usually divided into two types. They are, first, formal, which assumes theorem proving and other formal methods of mathematics and, second, simulation-based, which relies on design simulation with the provided set of test vectors (aka stimuli). In this thesis we focus on the second type. Simulation-

based hardware verification usually assumes comparison of one implementation against specification or another implementation (alternative or simplified, e.g. at a higher abstraction level). The other way is not to compare against a reference but to narrowly aim at specific design properties. The second approach is usually referred as Assertion-Based Verification (ABV). This thesis considers both of the possibilities but in different order.

ABV can be considered as one of the Design-for-Verifiability (DfV) techniques. They assume application of complementary parts of Hardware Description Language (HDL) code introduced especially for design verification assistance. In case of ABV this complementary code is assertions. ABV allows discovering design's misbehaviour (causing assertions violation) earlier and more effective.

On the other hand comprehensive verification coverage metrics help to estimate the verification progress and more effectively manage verification efforts. Coverage measurement addresses an important question of "when the design is verified enough".

1.2 Problem formulation

Traditional design representation models are based on HDLs (e.g. VHDL or Verilog). However, there are known a number of drawbacks related to application of HDLs-based models in verification.

The awkwardness and usually even inability of HDLs to represent complex temporal assertions has caused introduction of languages especially dedicated for this purpose such as Property Specification Language (PSL). The latter one in turn is not always supported by design simulation tools or this support may be expensive. The attempts to unify design implementation and its properties' representations normally result in creation of large hardware checkers that assume significant restrictions on the initial assertion functionality. At the same time a comprehensive verification coverage measurement based on HDL model may require complicated HDL code manipulations resulting in inefficient resource consumption.

In this thesis we address the main simulation-based hardware verification issues that are speed and accuracy of the verification process. In particular we target aspects of assertion checking and coverage measurement by exploiting decision diagrams based model advantages. The proposed approaches use a homogeneous hardware verification flow based on high-level decision diagrams design representation model. Previous research works, including [44],[45], have shown that HLDDs are an efficient model for design simulation and convenient for diagnosis and debug.

1.3 Contributions

The main contributions of this thesis are summarised as follows:

A new approach for HLDD-based assertions checking

- A *temporal extension for the existing HLDD model*. The new extended model is aimed at temporal properties expression and named *Temporally extended High-Level Decision Diagrams* (THLDD). The extension supports a set of commonly used temporal constructs that can be used to express a wide set of possible complex temporal relationships.
- A *methodology for direct conversion of assertions* expressed in Property Specification Language (PSL) to THLDD. The proposed hierarchical approach introduces an extendable library of Primitive Property Graphs (*PPG Library*). The components of this library serve as building blocks for a complex THLDD property construction.
- An *approach for HLDD-based assertion checking*. A *modification* of the existing HLDD-based *simulator* (HLDD*sim*) is proposed to support THLDDs and assertion checking. This part is supported by explanations of temporal issues and different varieties of THLDD properties.

  A minor contribution includes discussions of verification assertions reuse for manufacturing testing.

A new approach for HLDD-based coverage analysis

- An approach for *mapping* traditional verification structural coverage metrics to HLDD-based coverage. In addition to the base code coverage metrics such as statement and branch coverage, the approach considers also more sophisticated ones, including FSM and data flow coverage metrics.
- An approach for *condition coverage* analysis. The approach employs a *hierarchical decision diagrams* model consisting of HLDDs and BDD-based representations of the conditional statements.
- An approach for *HLDD model manipulations* targeted to different aspects of verification coverage analysis.

1.4 Thesis organization

This thesis consists of 5 main chapters. The rest of it is organized as follows.

Chapter 2 provides background information required for discussion of the further proposed approaches. First, design representation by decision diagrams is presented. It includes a brief introduction to Binary Decision Diagrams (BDD) and

description of High-Level Decision Diagrams (HLDD) model. Further Property Specification Language (PSL) is discussed with respect to its application for the proposed approaches.

Chapter 3 starts with an overview of hardware functional verification, focusing on assertion-based verification. This section also includes the discussion of the related works and a brief presentation of Tallinn University of Technology verification framework APRICOT. Further, the approach for HLDD-based assertion checking is presented in the following sections. The sections are temporal extension to HLDD model, PSL to HLDD conversion method and the method for assertion checking with HLDDsim simulator for HLDD. The following section presents the experimental results proving the feasibility and efficiency of the proposed approach. A discussion of verification assertions reuse ideas is provided at the end of the chapter.

Chapter 4 starts with discussion of verification coverage metrics basic classification and the main aspects related to their measurement while keeping the main focus on structural coverage. It is followed by proposal of an approach for mapping traditional verification coverage metrics to HLDD coverage. Further, an approach employing a hierarchical decision diagrams' model for the condition coverage measurement is presented. Finally, HLDD model manipulations for the verification coverage analysis are discussed. The chapter is concluded with experimental results which demonstrate the feasibility and efficiency of HLDD-based coverage analysis approach.

Chapter 5 draws conclusions for this thesis and discusses possible directions for future work.

Two appendix sections are also included at the end of the thesis. The first one presents library of Primitive Properties' Graphs (PPG library) as one of the HLDD-based assertion checking approach contributions. The library is used for THLDD properties construction. The second appendix provides syntax for an internal file format AGM used for HLDD and THLDD models representation.

1.4.1 Formatting remarks

The text of the thesis has the following hierarchy of division:

- 1 Chapter
- 1.1 Section
- 1.1.1 Subsection
- 1.1.1.1 Clause

All co-authored references are emphasized in the work by a superscript suffix as follows: [*ref.*]$^{\text{co-auth.}}$.

Chapter 2
BACKGROUND

The approaches for hardware verification presented in this work take the advantage of design representation by High-Level Decision Diagrams (HLDDs) developed in Tallinn University of Technology. The purpose of this chapter is to introduce this model. Traditional Binary Decision Diagrams (BDDs) are also described in this chapter. BDD and HLDD themselves are not contributions of this thesis. However, most of the contributions rely on these models or are their extensions.

This chapter introduces also IEEE standard Property Specification Language (PSL) applied for expressing assertions. Within this work PSL is not just a choice among available properties' expression languages, but also serves in frames of this work as a reference for the supported set of assertions and their classification.

2.1 Design representation by decision diagrams

The history [9] of decision diagrams based design representation model development 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 [36] and 1978 [37] respectively. In [36] 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 [38] and [39]. However, it was not until the efficient Boolean manipulation method was presented by Randal E. Bryant in [40] when this type of representations became widely accepted by the research community.

Further, a number of special classes of binary decision diagrams have been proposed. They include popular Reduced Ordered BDDs (ROBDD) [40], multi-terminal BDDs [49], edge-valued BDDs [50], binary moment diagrams [51], multi-

valued decision diagrams [52], zero-suppressed BDDs [53], functional decision diagrams (FDD) [54], Kronecker FDDs [55] and others.

Structurally Synthesized BDDs (SSBDDs), formerly structural alternative graphs, are a class of BDD that have been proposed by Raimund Ubar in [36], [41]. This model is used for design representation at gate level and supported by a set of testing tools developed in Tallinn University of Technology and known as Turbo Tester ([58] and [90]).

There is a number of word-level Decision Diagrams based models used for design representation at Register-Transfer and higher levels. *High-Level Decision Diagrams* (HLDDs) were proposed by Raimund Ubar in [41] and further developed by Jaan Raik in [9] and [42], [19][co-auth.]. The other examples are multiterminal DDs (MTDDs) [49], K*BMDs [56] and Assignment DDs (ADDs) [57] are some of them. However, in MTDDs the nonterminal nodes hold Boolean variables only. K*BMDs, where additive and multiplicative weights label the edges are useful for compact canonical representation of functions on integers (especially wide integers). However, the main goal of HLDD representations is not canonicity but simulation and implications. The principal difference between HLDDs and ADDs lies in the fact that ADDs' edges are not labelled 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. There is a comparison example of HLDD vs. ADD representation of the same design provided in Clause 2.1.2.5.

The following two subsections provide a brief introduction to BDD and a more comprehensive one to HLDD models correspondingly.

2.1.1 Binary decision diagrams

BDD is a common representation for Boolean functions. A BDD is defined [9] as a directed acyclic graph with two terminal nodes, which are the *0-terminal* and *1-terminal* nodes. Each non-terminal node is labelled by an input variable of the Boolean function, and has two outgoing edges, called *0-edge* and *1-edge*.

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 3 different representations for a BDD corresponding to a Boolean function $f = (x_1 \cdot x_2) \vee \neg x_3$. Figure 2.1a shows a full tree BDD, Figure 2.1b shows an OBDD and Figure 2.1c shows the same OBDD from Figure 2.1b but in alternative description style.

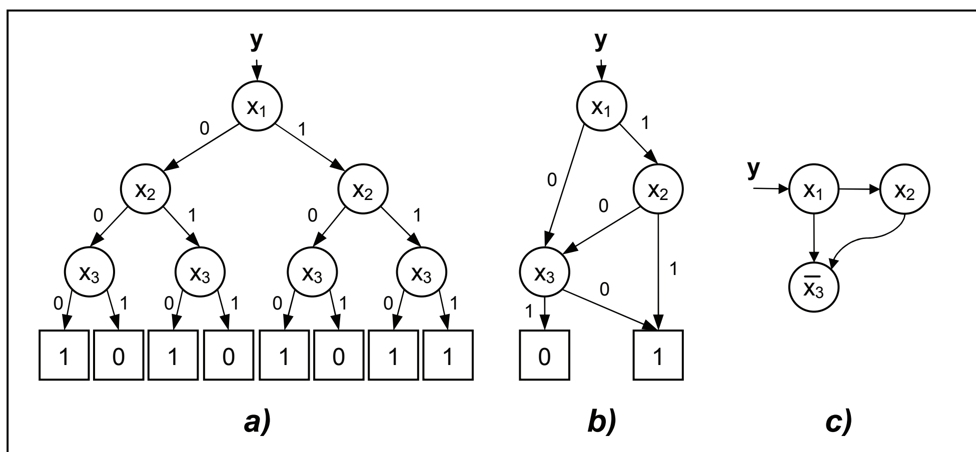


Figure 2.1. Different BDD representations for a Boolean function $y = (x_1 \cdot x_2) \vee \neg x_3$

Alternative description style differs from the traditional BDD description style by the following. Logic *'1'* and *'0'* constants holding terminals are omitted. Instead of them, a convention exists, that the right-hand edge of a node corresponds to *1-edge* and the lower-hand edge to *0-edge*. Exiting the BDD rightwards corresponds to the solution y = *'1'*, while exiting downwards corresponds to y = *'0'*. In this type of description style the nodes can be labelled by both, variables and their inversions (see $\neg x_3$ in Figure 2.1c).

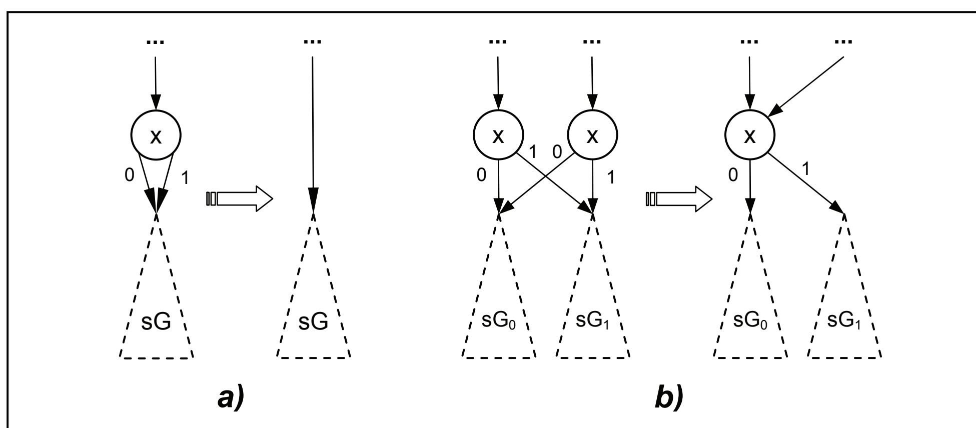


Figure 2.2. BDD reduction rules

Reduced Ordered BDD (ROBDD) is created by applying the following *reduction rules* to OBDD [40]:

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).

The important feature of ROBDDs is that they provide for canonical forms of 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.

The mentioned above *Structurally Synthesized BDDs* (SSBDDs) model is not directly used in frames of this thesis and therefore it is not discussed in detail. However, for the approaches where a hierarchical design representation is convenient, SSBDD model can complement HLDDs by representing the design's modules at the gate level. As it will be shown in the next subsection HLDDs are applied for design representation at RTL and higher abstraction levels.

2.1.2 High-level decision diagrams

The HLDD model description provided in this subsection is mostly based on the description provided in [9] and considers minor refinements made in ([19] and [13]) [co-auth.].

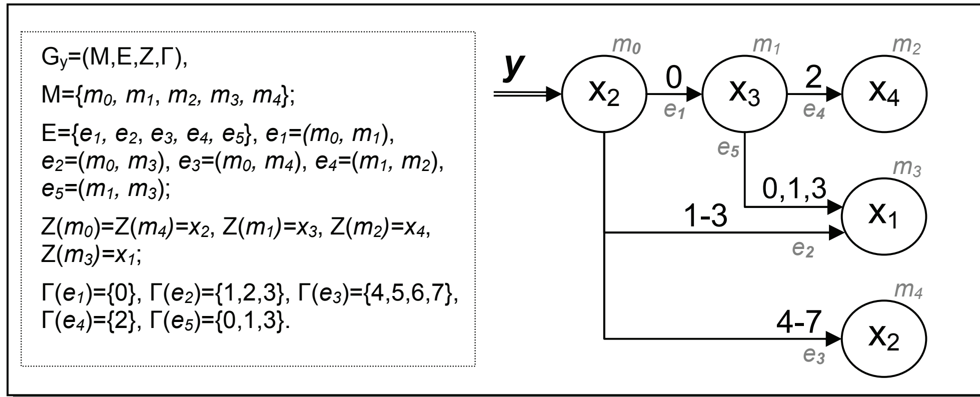


Figure 2.3. A high-level decision diagram representing a function $y=f(x_1,x_2,x_3,x_4)$

2.1.2.1 HLDD model definition

A *High-Level Decision Diagram* (HLDD) is a graph representation of a discrete function. A discrete function $y = f(x)$, where $y = (y_1, \ldots, y_n)$ and $x = (x_1, \ldots, x_m)$ are vectors is defined on $X = X_1 \times \ldots \times X_m$ with values $y \in Y = Y_1 \times \ldots \times Y_n$, and both, the domain X and the range Y are finite sets of values. The values of variables may be Boolean, Boolean vectors, integers. Figure 2.3 presents an example of a graphical interpretation of a HLDD.

Definition 1**:** A high-level decision diagram is a directed non-cyclic labelled graph that can be defined as a quadruple $G=(M,E,Z,\Gamma)$, where M is a finite set of vertices (referred to as *nodes*), E is a finite set of *edges*, Z is a function which defines the *variables labelling the nodes*, and Γ is a function on E.

The function $Z(m_i)$ returns the variable x_k, which is labelling node m_i. Each node of a HLDD is labelled by a variable. In special cases, nodes can be labelled by constants or algebraic expressions. An edge $e \in E$ of a HLDD is an ordered pair $e=(m_{pc},m_{sc}) \in E^2$, where E^2 is the set of all the possible ordered pairs in set E. Graphical interpretation of e is an edge leading from node m_{pc} to node m_{sc}. It is said that m_{pc} is a *predecessor node* of m_{sc}, and m_{sc} is a *successor node* of the node m_{pc}, respectively. Γ is a function on E representing the activating conditions of the edges for the simulating procedures. The value of $\Gamma(e)$ is a subset of the domain X_k of the variable x_k, where $e=(m_i,m_j)$ and $Z(m_i)=x_k$. It is required that $Pm_i = \{ \Gamma(e) \mid e = (m_i,m_j) \in E\}$ is a partition of the set X_k.

Figure 2.3 presents a HLDD for a discrete function $y=f(x_1,x_2,x_3,x_4)$. HLDD has only one starting node (*root node*) m_0, for which there are no preceding nodes. The nodes that have no successor nodes are referred to as *terminal nodes* $M^{term} \in M$ (nodes m_2, m_3 and m_4 in Figure 2.3). Design representation by high-level decision diagrams, in general case, is a system of HLDDs rather than a single HLDD. During the simulation in HLDD systems, the values of some variables labelling the nodes of a HLDD are calculated by other HLDDs of the system.

In this thesis we propose to emphasize the connection between HLDD label (i.e. the graph name, bold *y* for the example in Figure 2.3) and the root node of the graph by a *double arrow* to distinguish it from the edges connecting the HLDD's nodes. In case of design representation by a system of HLDDs the notations of variables labelling the terminal nodes M^{term} are proposed to be *underlined* or remain *normal* for the explicit variables (i.e. input signals) and set off in *italics* for implicit variables (the ones referring to another HLDD graph in a system of HLDDs). An example of a system of HLDDs can be found in Figure 2.7.

2.1.2.2 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 $m_i \notin M^{term}$ according to the value v_k of the variable $x_k=Z(m_i)$ certain output edge $e = (m_i,m_j)$, $v_k \in \Gamma(e)$ will be chosen, which enters into its corresponding successor node m_j. Let us call such connections *activated edges* under the given values and denote them by $m_i^{v_k}$. 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 variable labelling the terminal node of the main activated path.

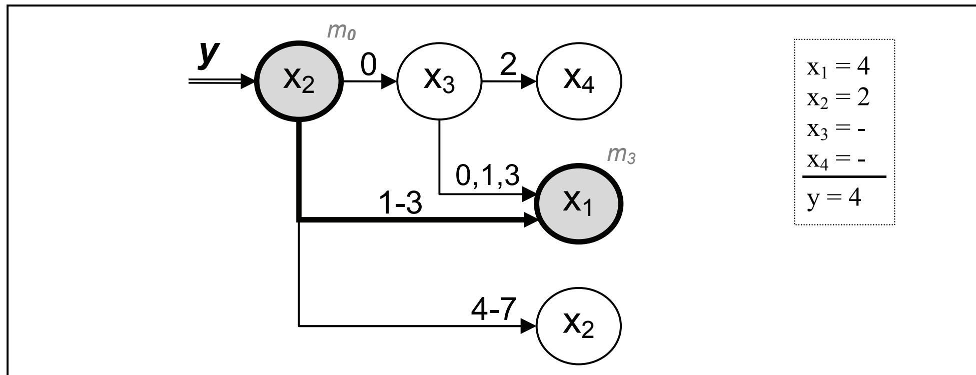


Figure 2.4. Design simulation on high-level decision diagrams

In Figure 2.4 simulation on the high-level decision diagram presented in Figure 2.3 is shown. Assuming that variable x_2 is equal to 2, a path (marked by bold arrows) is activated from node m_0 (the root node) to a terminal node m_3 labelled by x_1. Let the value of variable x_1 be 4, thus, $y=x_1=4$. Note, that this type of simulation is event-driven since we have to simulate only those nodes that are traversed by the main activated path (marked by grey colour in Figure 2.4).

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 labelling the nodes of a HLDD are calculated by other HLDDs of the system. The detailed algorithm for HLDD based systems simulation is provided in Subsection 3.4.1.

2.1.2.3 Pure RTL designs representation by HLDDs

Let us consider a design represented in a HDL at the *Register-Transfer Level* (RTL) of abstraction. We distinguish 2 styles of RTL description - *pure RTL* and *behavioural RTL*. While the first one more precisely targets the desired architecture, the second one describes the design in a more natural way. This and the following clauses introduce RTL design representations by HLDDs for these two description styles correspondingly.

A deign described in the pure RTL style is assumed to be partitioned into a *datapath* and a *control part*. Figure 2.5 shows this type of architecture. Here, the control part is a Finite State Machine (FSM) with a state register (represented by variable x_S in the corresponding HLDD model), next state logic and output logic. As input signals to the FSM are the primary inputs of the design (variables x_I), conditional signals originating from the datapath (variables x_N) and current value of the state variable x_S. Outputs of the FSM are the primary outputs of the design (variables x_O), control signals (variables x_C) and the next value of x_S. The signals' variables notations introduced for Figure 2.5 are used throughout this clause.

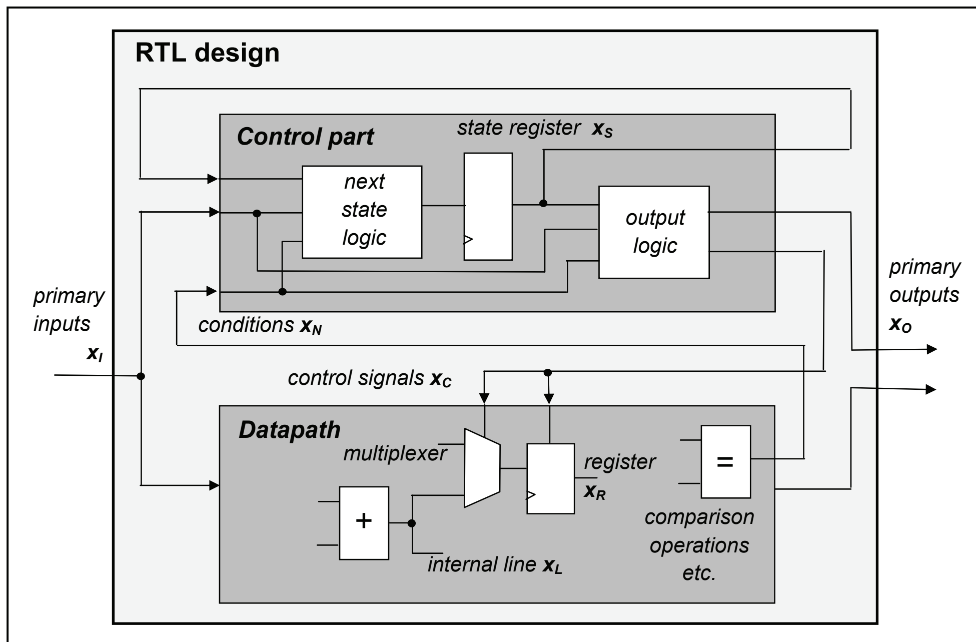


Figure 2.5. RLT view of a digital design

The *datapath* can be viewed as a network consisting of modules or blocks. These include registers, multiplexers and functional units (for implementing operations).

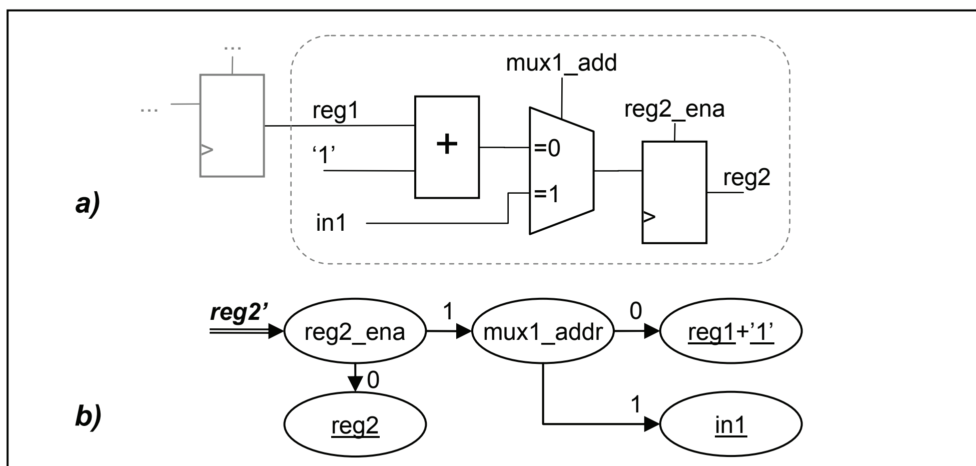


Figure 2.6. A datapath fragment (a) and its HLDD representation (b)

All the registers and some internal lines of the datapath can be represented by variables in the RTL HLDD model (variables x_R and x_L, respectively). Inputs for the datapath are the primary inputs x_I and control signals x_C (e.g. multiplexer

addresses and register enable signals). Outputs are the primary outputs x_O as well as conditional signals x_N (e.g. from comparison operators) leading to the control part FSM.

In HLDDs representing the datapath, the non-terminal nodes correspond to control signals (labelled by variables x_C). The terminal nodes represent operations (functional units). Register transfers and constant assignments are treated as special cases of operations. Figure 2.6 shows a simple example of a HLDD representation (Figure 2.6b) for the given datapath fragment (Figure 2.6a). In this example and further in this clause we use a notation where the prime symbol " ' " after diagram's variable denotes one clock cycle delay, i.e. next state of the variable (e.g *reg3'* vs. *reg3*).

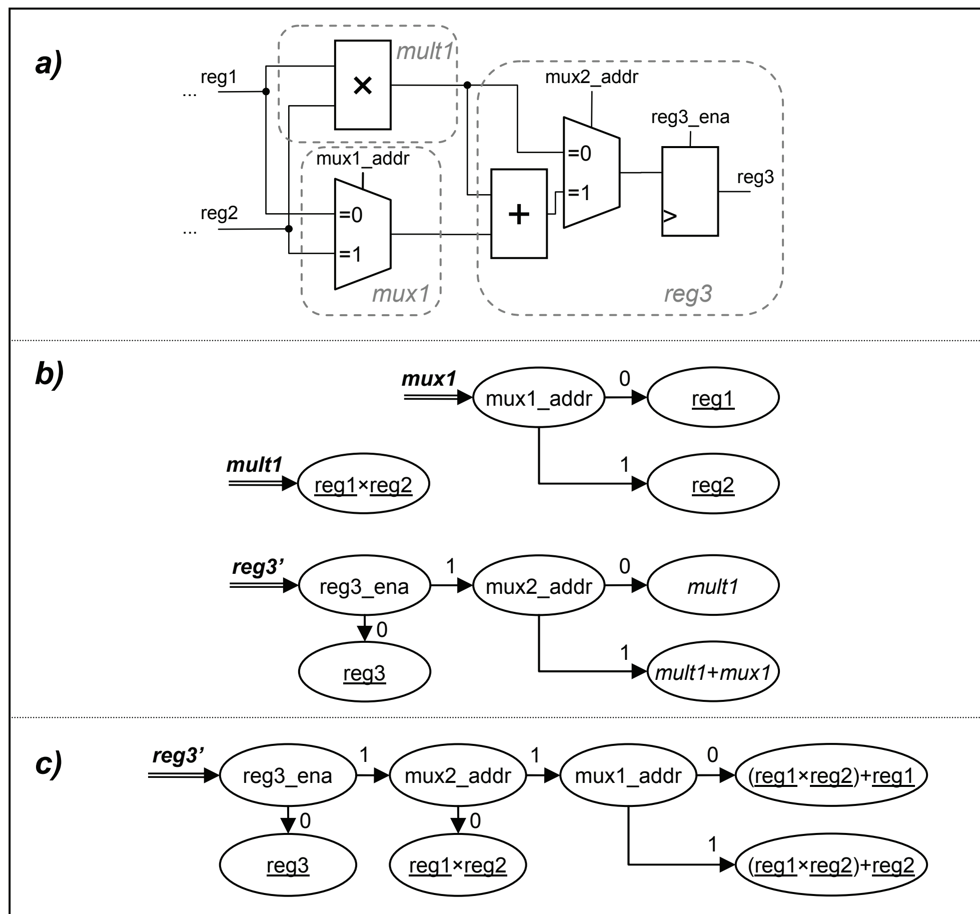
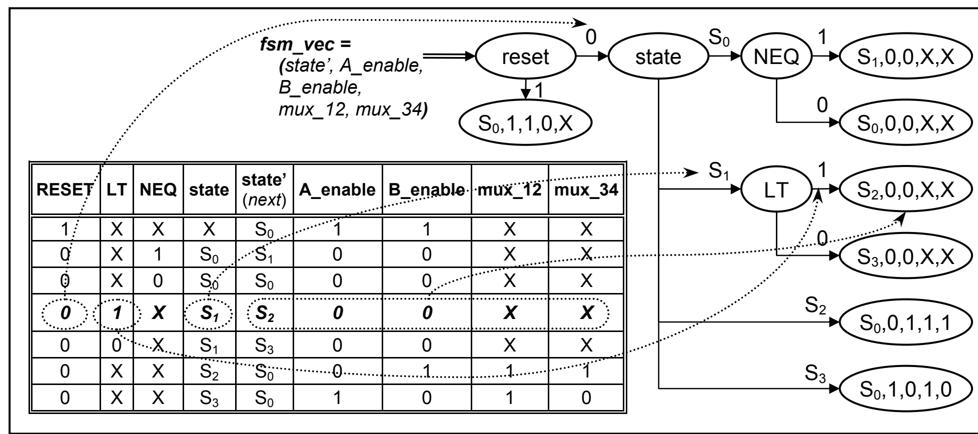


Figure 2.7. Datapath representing HLDD partitioning types

Usually, a datapath is represented by a system of HLDDs. Here, different partitioning strategies are possible. The most commonly used partitioning is the one, where for each primary output, fanout signal and register a HLDD

corresponds. In addition, multiplexers that are connected to inputs of a functional unit are represented by a separate HLDD. Figure 2.7b shows this type of HLDD system partitioning for the datapath given in Figure 2.7a. However, it is possible to use alternative partitioning. For example, Figure 2.7c shows an approach, where for each register of the datapath exactly one decision diagram corresponds. This type of partitioning is sometimes referred as *register-oriented HLDD*. Other types of HLDD partitioning can be used depending on the target model application.



RESET	LT	NEQ	state	state' (next)	A_enable	B_enable	mux_12	mux_34
1	X	X	X	S_0	1	1	X	X
0	X	1	S_0	S_1	0	0	X	X
0	X	0	S_0	S_0	0	0	X	X
0	***1***	***X***	***S_1***	***S_2***	***0***	***0***	***X***	***X***
0	0	X	S_1	S_3	0	0	X	X
0	X	X	S_2	S_0	0	1	1	1
0	X	X	S_3	S_0	1	0	1	0

Figure 2.8. Converting FSM state table into HLDD

A simple RTL design *control part* is usually represented by a single HLDD, however in case of complex or multiple FSMs different partitioning are possible here as well.

The control part HLDD calculates the values for a vector consisting of the state variable and control signals. In the HLDD, the non-terminal nodes correspond to current state (labelled by variable x_S) and conditional signals originating from the datapath (variables x_N). Terminal nodes hold vectors with the values of next state and control signals x_C.

Figure 2.8 shows an FSM state table and its corresponding HLDD representation. In the HLDD, *state'* denotes the next state and *state* denotes the current state value. Variables *A_enable*, *B_enable*, *mux_12* and *mux_34* are FSM outputs and belong to the control signals x_C. Variables *RESET*, *LT* and *NEQ* are FSM inputs and belong to x_N. The dashed circles and arrows in Figure 2.10 depict setting up the edges and the terminal node corresponding to the fourth row of the state table.

2.1.2.4 Behavioural RTL designs representation by HLDDs

Behavioural RTL HDL description style represents the design as an FSM structure nested with data assignments. It includes clocking information and is

therefore cycle-accurate. The control state is mapped to a *case* statement and conditions to *if* or *case* statements respectively. Each branch of the control state *case* statement corresponds to a certain control state and describes the datapath operations at the corresponding state and also the next state transitions.

This style is also synthesizable as well as pure RTL, but it is less target architecture specific. The behavioural RTL style is more commonly used in practical design HDL-based implementation than pure RTL style, where the design is strictly partitioned to datapath and control part. For example, ITC'99 benchmark circuits [76],[102] that are widely used in research community and partially represent real industrial designs, are described in behavioural RTL style.

A separate HLDD diagram is generated for each internal signal and output port of the behavioural RTL description. For each such signal *v* we generate a diagram by parsing the behavioural RTL code as follows:

1. From the nested *if/case* structure generate a diagram where nodes correspond to conditions in respective *if/case* statements and edges correspond to decisions and are marked by the activation values of the respective decisions.
2. The terminal nodes are labelled by the right-hand side of assignments to a signal *v*. If there is no assignment to the signal *v* in the corresponding decision branch then the respective terminal node will be labelled by *v*.

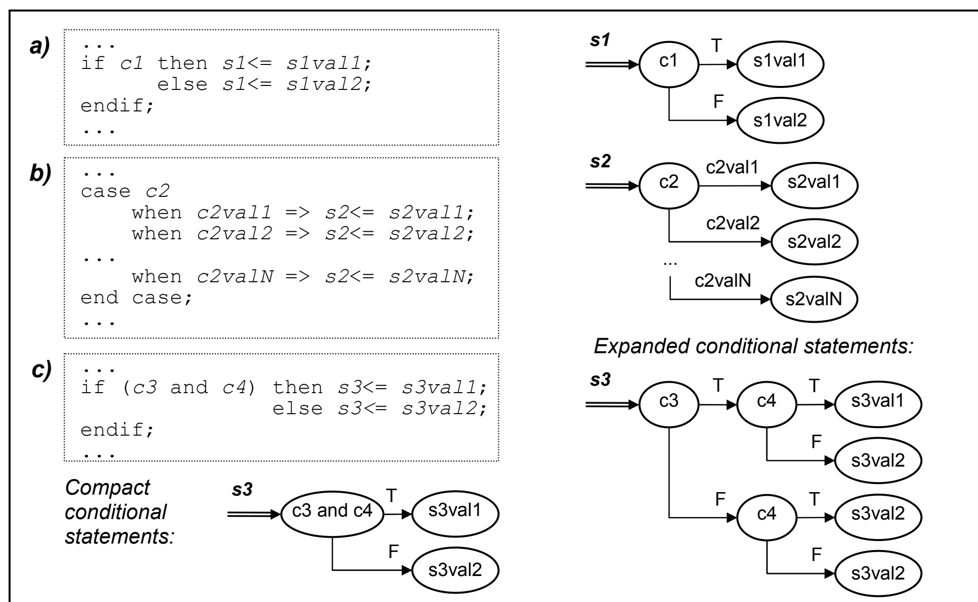


Figure 2.9. HLDD representation for the common constructs of the behavioural RTL VHDL

Figure 2.9 shows HLDD generation for the common behavioural RTL HDL (we consider VHDL) constructs. Figures 2.9a and 2.9b show simple *if* and *case*

conditional statements. Figure 2.9c shows a complex *if* construct, which consists of two conditions *c3* and *c4* joint by the logical *and* operator. For a complex conditional statement consisting of a set of conditions 2 HLDD variants are possible. Normally, the conditional statement evaluated as a whole and therefore can be represented by a HLDD with *compact conditional statements* (the bottom-left part of Figure 2.9c). However, in case if we are interested in several particular simulation coverage metrics (e.g. *condition coverage*) measurement, we may be interested in a HLDD representation with *expanded conditional statements* (the right-hand-side part of Figure 2.9c). This topic is discussed in detail in Chapter 4 (Section 4.3). Please note, that depending on the application we may be interested in a non-reduced HLDD. For example, *c4* is analyzed even after the *false-edge* of *c3* node in Figure 2.9c, however here the *reduction rule 1* from the Subsection 2.1.1 (Figure 2.2a) would be applicable.

Note, that since we do not support asynchronous latches in our approach the synthesizable RTL style must always include *else* and *default* branches of the *if* and *case* statements, respectively. Alternatively, default value assignments of signal v must be given.

Figure 2.10 shows an implementation GCD1 of a greatest common devisor design, which is actually a benchmark *gcd* from the HLSynth'92 benchmarks family [103]) in behavioural RTL VHDL and its corresponding HLDD. The comparison benchmark from the next clause (Figure 2.11) is an alternative pure RTL description implementation of the Greatest Common Devisor design (GCD2).

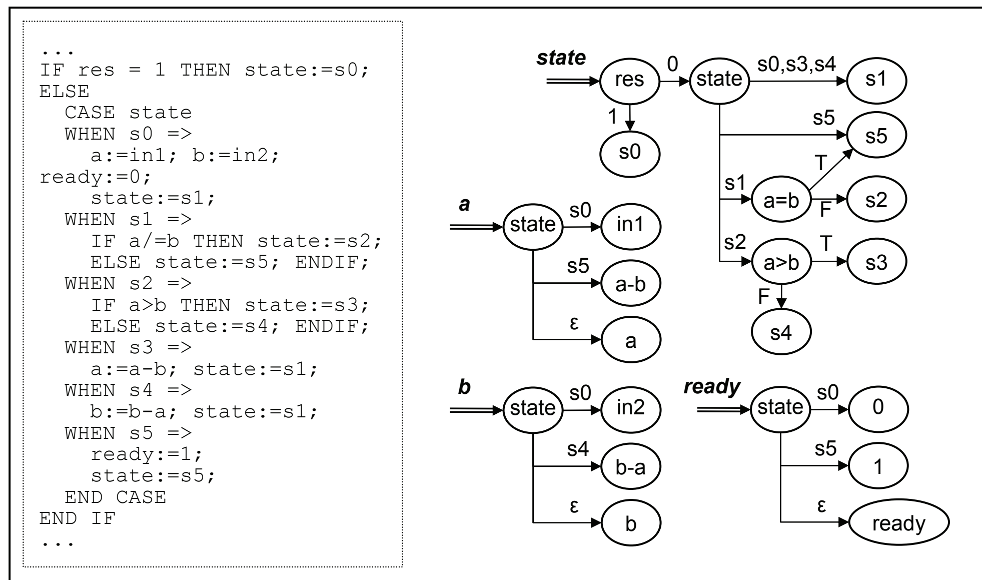
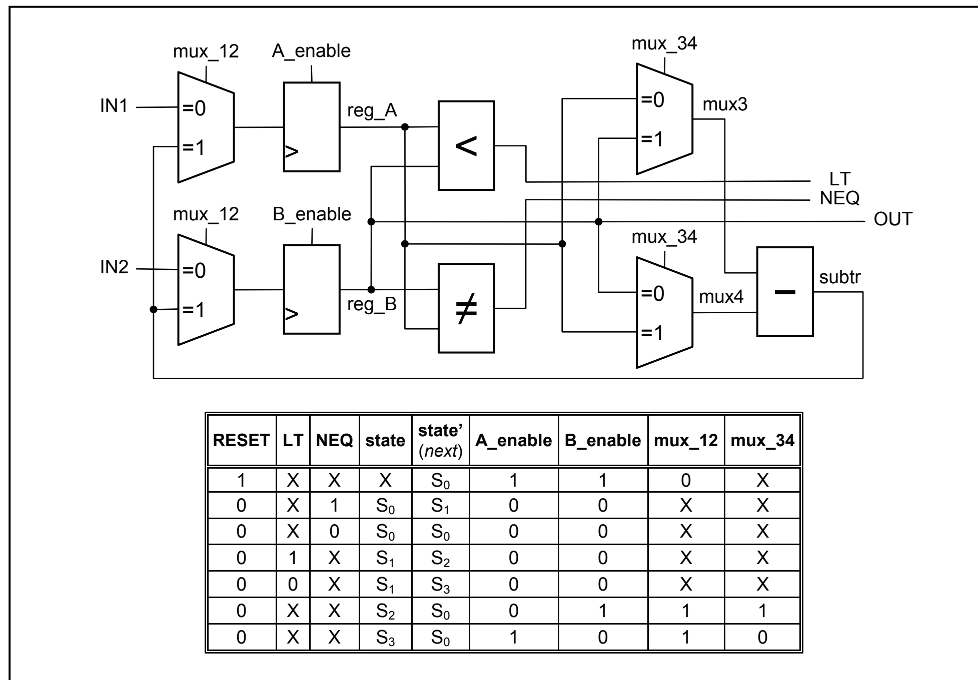


Figure 2.10. Behavioural RTL VHDL and HLDD representations for a design GCD1

2.1.2.5 HLDD vs. ADD representations comparison

This clause provides an example (proposed in [43]) of HLDD model comparison with a commonly used *Assignment Decision Diagram* (ADD) approach.

Figure 2.11 presents the schematic RTL description of a Greatest Common Divisor benchmark GCD2. Figures 2.12 and 2.13 show its corresponding HLDD and ADD representations respectively.



RESET	LT	NEQ	state	state' (*next*)	A_enable	B_enable	mux_12	mux_34
1	X	X	X	S_0	1	1	0	X
0	X	1	S_0	S_1	0	0	X	X
0	X	0	S_0	S_0	0	0	X	X
0	1	X	S_1	S_2	0	0	X	X
0	0	X	S_1	S_3	0	0	X	X
0	X	X	S_2	S_0	0	1	1	1
0	X	X	S_3	S_0	1	0	1	0

Figure 2.11. A RTL design GCD2

Apart from the fact that HLDD description contains less nodes, there are the following fundamental differences:

- ADDs structure closely matches the RTL design. Edges of ADD correspond to connecting nets in datapath. ADD for FSM is equivalent to its gate-level implementation. In contrast, HLDDs do not strictly follow the circuit structure. Here, a synthesis to extract data and control relationships from the circuit functionality has been carried out.
- ADD model includes four types of nodes (read, write, operator, assignment decision). In HLDD the nodes are treated uniformly and can be divided into nonterminal nodes (control) and terminal nodes (data).

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

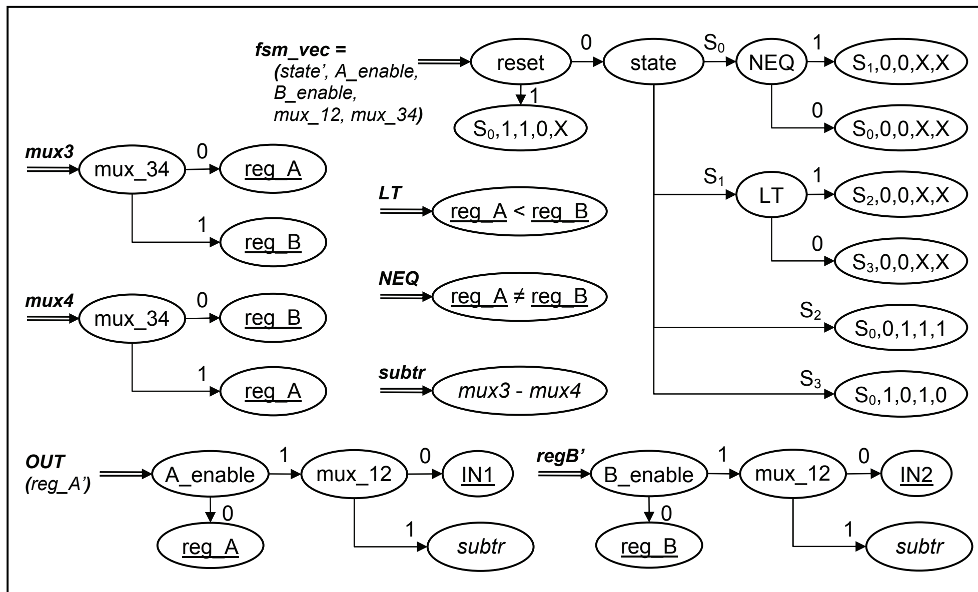


Figure 2.12. HLDD representation for the GCD2 design

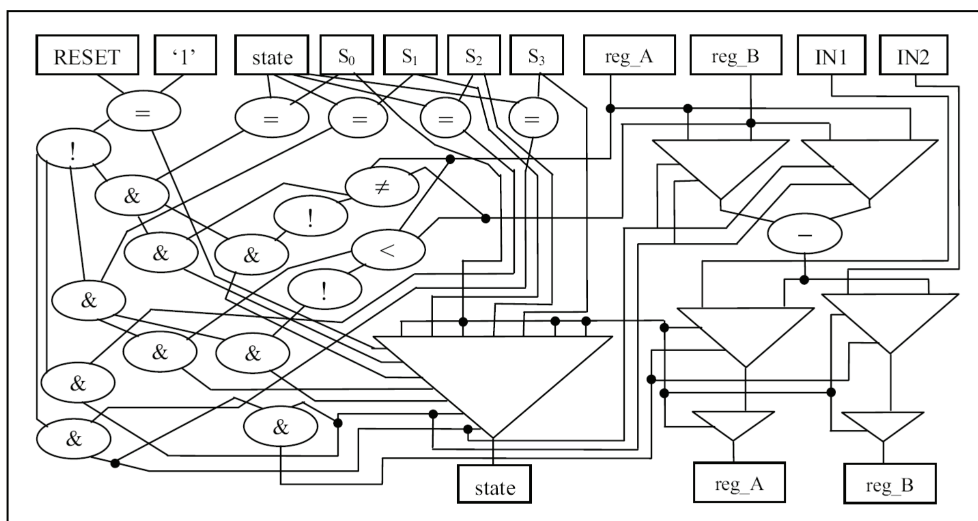


Figure 2.13. ADD representation for the GCD2 design

2.1.2.6 HLDD model advantages for debug in verification

High-level decision diagrams model has a set of advantages compared to HDL and other DD based design representation models. A comparison with ADD has been provided in the previous clause. This clause describes by example some advantages of HLDD model for debug process in verification.

As an example, consider a datapath of a design depicted in Figure 2.14a and its corresponding HLDD representation shown in Figure 2.14b. Here, R_1 and R_2 are registers (R_2 is also output), MUX_1, MUX_2 and MUX_3 are multiplexers, + and * denote adder and multiplier, *IN* is and input bus, y_1, y_2, y_3 and y_4 serve as input control variables, and *a, b, c, d* and *e* denote internal buses, respectively. In the HLDD, the control variables y_1, y_2, y_3 and y_4 are labelling internal decision nodes of the HLDD with their values shown at edges. The terminal nodes are labelled by a 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 grey nodes, a full activated path in the HLDD is shown from $Z(m_0)= y_4$ to $Z(m^T \in M^T)=R_1*R_2$, which corresponds to the pattern $y_4=2$, $y_3=3$, and $y_2=0$. The activated part of the network at this pattern is denoted by grey boxes.

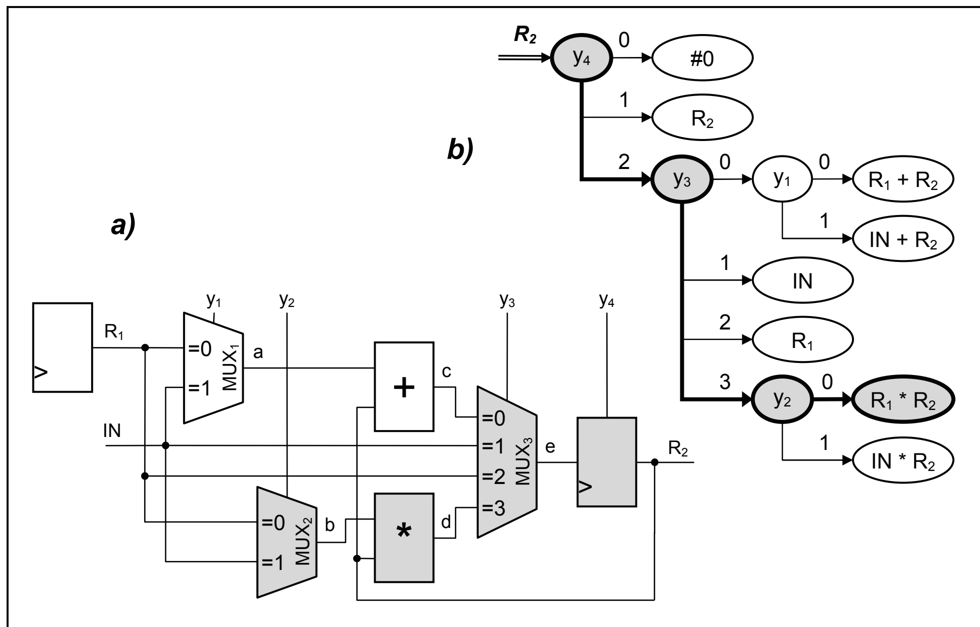


Figure 2.14. HLDD model advantages for debug

The main advantage and motivation of using HLDD model, compared to other design representation models relying on netlists of primitive functions, is the increased efficiency of simulation and diagnostic modelling. The efficiency is caused by direct and compact representation of cause-effect relationships. For

example, instead of simulating the control word $y_1, y_2, y_3, y_4 = 0032$ 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 y_4, y_3 and y_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 y_4, y_3 and y_2 without any diagnostic analysis inside the corresponding RTL netlist. As a result of such a quick reasoning the debugging of a system can be considerably simplified. A detailed analysis inside the RTL netlist is needed only if all the values of y_4, y_3 and y_2 are correct. In such a way, a very efficient hierarchical debugging procedure can be carried out with HLDDs: first, by a quick trace of faulty nodes in HLDDs, and then after locating the erroneous RTL region, by exactly locating the cause of error in this region.

The advantages for debug and proven [44],[45] faster design HLDD-based simulation are a strong motivation for HLDD application for simulation-based functional verification.

The first section of this background chapter has presented discussed the advantages of the design representation model called high level decision diagrams. This model is used for the approaches proposed in Chapters 3 and 4.

2.2 Property specification language

This section provides introductory information about a language for assertions expression *Property Specification Language* (PSL). Within this work PSL is not just a choice among available assertions expression languages like *System Verilog Assertions*, *Open Vera Assertions*, *e*, *Open Verification Library*, *SystemC Assertions*, etc. Based on the number of factors, discussed further, PSL serves in frames of this work as a reference for the supported set of properties and their classification.

Assertion-based verification popularity has encouraged a common property specification language development by the Functional Verification Technical Committee of Accellera. After a process in which donations from a number of companies were evaluated, the *Sugar* language [99] from IBM was chosen as the basis for PSL. The latest Language Reference Manual (LRM) for PSL version 1.1 was released in 2004 [91]. The language became an IEEE 1850 Standard in 2005 [92] and later IEC 62531 Standard [61] in 2007. The both above mentioned standards are based upon Accellera's LRM [91] with minor modifications (e.g. SystemC flavor introduction). This LRM together with the web resources listed on the PSL/Sugar Consortium webpage [94] can be considered for comprehensive PSL definition and explanation source. The information about PLS provided in this Chapter focuses on the parts of the language required for implementation and understanding of the approaches described in Chapter 3.

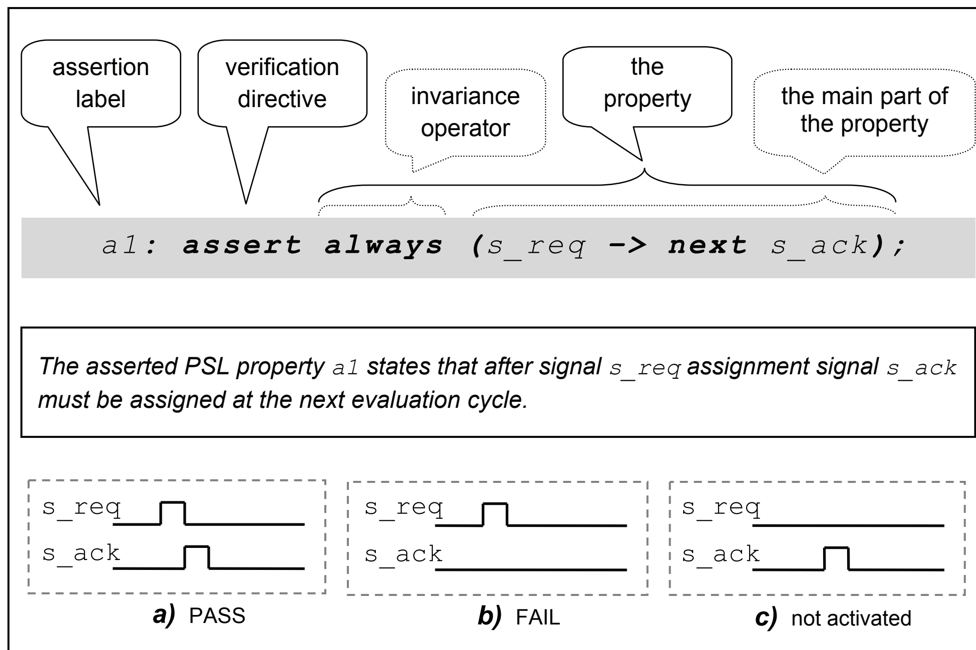


Figure 2.15. An example of PSL assertion with its structure explained and possible DUV's behavior timing diagrams

An example of a PSL assertion is shown in Figure 2.15. The assertion in the example consists of an optional label, the verification directive and the property to be checked. The last one is composed of the signals of interest and PSL operators. The timing diagrams in the bottom demonstrate 3 of many possible variants of the DUV's behaviour. In case of behaviour (a) the assertion is satisfied. However, the assertion will be violated in case of (b) and not activated (or vacuously passed) in case of behaviour (c).

2.2.1 PSL organization

As it was mentioned, PSL is primarily based on IBM's Sugar language. The latter one was, in turn, originally based on *Computation Tree Logic* (CTL) (first introduced in 1977 [62]), initially just providing "sugaring" to the CTL's complicated syntax for IBM tools users' convenience in early '90-ies (as it is stated in [4] and [2]). The main application of Sugar was Formal Verification. Later, before PSL standardization, Sugar has employed *Linear-Time (temporal) Logic* (LTL) (first introduced in 1981 [63]) capabilities. At present, PSL consists of 2 parts:

- *Foundation Language* (FL), that is based on LTL and applied for both simulation-based and formal verification
- *Optional Branching Extension* (OBE), that is based on CTL and finds its application in formal verification

The main emphasis of this thesis is put on simulation-based verification, therefore from now on we will consider mostly FL part of PSL.

2.2.1.1 Flavors

For convenience of the language users, PSL supports 5 flavors[1], each of them corresponding to one HDL. The main difference between the flavors is seen for Boolean expressions. At present the flavors are:

- SystemVerilog (IEEE Std 1800)
- Verilog (IEEE Std 1364)
- VHDL (IEEE Std 1076)
- SystemC (IEEE Std 1666)
- GDL (*General Description Language* [98]), which is known also as a placeholder for the future HDL from IBM

VHDL flavor:

```
-- psl property p1 is
--   always (sig1 ->
--      ((sig2(0 to 4)= "1010") or
--       (sig2(0 to 4)= "0101")))
--   @(clk'event and clk='1');
-- psl assert p1;
```

Verilog flavor:

```
// psl property p1 =
//   always (sig1 ->
//     ((sig2[0:4]== 4b'1010) ||
//      (sig2[0:4]== 4b'0101)))
//   @(posedge clk);
// psl assert p1;
```

Figure 2.16. The same PSL property expression in Verilog and VHDL flavors

An example of the same PSL property expression in Verilog and VHDL flavor is given in Figure 2.16. It is not allowed to mix flavors within one property (otherwise it could not be parsed).

2.2.1.2 Modes

In practice PSL properties related to a design under verification (DUV) may be expressed in one of the two modes [2].

- The s*tand-alone mode* means that all properties related to a DUV are grouped into a separate file or files, and also usually organised within the files into *verification units*. This approach is preferred by verification engineers, because of the convenience for complex verification plan organization.
- The second mode is the *embedded mode*. In this mode, usually preferred by designers, the DUV's properties are written directly into its HDL files.

[1] In the frames of PSL specification this term is normally used in the US transcription

Normally, special directives for HDL compilers with PSL support hidden into comments are used in this mode (e.g. like "-- *psl*" for VHDL flavor in Figure 2.16).

The embedded mode properties should be of the same PSL flavor as DUV's HDL. On the contrary, stand-alone properties can be of flavor *B* different from DUV's HDL *A*. However, it is necessary to check if the applied verification tool (e.g. simulator) with PSL support has the support for HDL *B* (e.g. SystemC).

The approaches proposed in the Chapter 3 of this work generally assume the VHDL flavor and the stand-alone mode for PSL properties expression.

2.2.1.3 Layers

PSL is a multi-layered language. The layers are:

- *Boolean layer* – the lowest one, it consists of Boolean expressions in HDL (e.g. `a and (b or c)`). These expressions are used as building blocks in the upper layer to compose complex properties. Boolean expression is an expression that is evaluated in a single (clock) cycle and has the value *true* or *false*. Boolean expressions may contain non-Boolean expressions.
- *Temporal layer* – it is the main part of the language. It builds temporal properties of Boolean expressions which describe DUV's (or its environment) behaviour over time (e.g. `a and (next[3]b or next_e[5 to 7]c)`). The main tools here are *temporal operators* that are introduced further. This layer also adds the support for *Sequential Extended Regular Expressions* (SERE) (e.g. `{a;{[*3];b} or {[*5 to 7]; c}}`).
- *Verification layer* - it provides directives that tell a verification tool what to do with specified properties. For example the directive *assert* specifies that the property is asserted (i.e. DUV's behaviour described by the property should hold) and makes an *assertion* out of it. The other directives are: *assume* (widely used to model DUV's environment), *cover*, *restrict* and others. It also includes declaration of verification units *vunit*-s used for properties organization, as it was mentioned in the previous subsection. Verification units can inherit from other ones.
- *Modelling layer* - additional helper code to model auxiliary combinational signals, state machines etc. that are not part of the DUV but are required to express the property. Usually (except SystemC and GDL flavor) the modelling layer instances are synthesizable.

The approaches provided in the Chapter 3 focus on the first two layers of PSL.

2.2.1.4 Styles

There are two styles in use for PSL properties expression.

- *LTL style* - is the approach when a PSL property is composed of Boolean expressions as operands for PSL *Boolean* and *temporal operators*, as well as pure HDL operators.
- *SERE*[2] *style* - makes use of sequences of Boolean variables or simple Boolean expressions. The sequences are enclosed curly braces and their *atoms* are usually separated by semi-colons (e.g. `{a;(b && c);d}`). As opposed to *Regular Expressions* (REs) from *pattern matching*, SEREs are allowed to have not only variables but also expressions as atoms for their sequences. A SERE may be an operand for some PSL temporal operators, a part of another SERE and can contain special *SERE repetition operators*. For example, the following sequence `{a;b[*5];{c;d}[=3]}` means that fist `a` is assigned, followed by 5 consecutive assignments of `b` and, finally, followed by sub-sequence `c` followed by `d` assignment for 3 non-consecutive times. Practically, in SERE style, a property is expressed as SEREs connected by *suffix implication operators* ("|−>" and "|=>").

The most of the properties can be expressed in both LTL and SERE styles, however with different levels of convenience. Usually it is more reasonable to express a property or a part of a complex property in one of the styles and therefore styles are normally mixed. According to [4], while every LTL style property can be translated to SERE style, there are some SERE style properties (with a particular form of counting, like *"p holds on every even cycle"*) that cannot be expressed by LTL (nor is it expressible by pure CTL).

The approaches from Chapter 3 include currently support for LTL style, however they do not have any principal constraint for SERE style usage. The support for SERE style PSL properties can be relatively easily added (as it will be explained further) and is scheduled for the future work. The most of the simple SERE style properties are known to have their formal equivalences in LTL style. Figure 2.19 from the next subsection presents a table with such equivalences [4].

2.2.2 PSL properties

It is important to distinguish between the two notions often used interchangeably: a property and an assertion. A *property* is some specified part of behaviour (of the DUV, its environment or the whole system). The property itself

[2] An interesting fact is that in Sugar before PSL standardization, SERE was acronym for *Sugar* Extended Regular Expressions (as opposed to the present *Sequential* Extended Regular Expression) [99]

does not state if it is expected to fail or hold, whether the DUV should be checked for it during simulation or it should be avoided during some actions. A property with the verification directive (see 2.2.1.3) *assert* (added directly or separately in the corresponding *vunit*) makes an *assertion* of the property. Assertions are the most often usage of the properties in verification, however a property can serve as an *assumption*, a *restriction* or other depending of it application. Figure 2.15 shows clear separation of the property part in the assertion *a1*.

The properties are composed of combinations of operators with operands that are variables, internal signals or primary inputs/outputs of interest. A property can state single signal assignment without an explicit operator (e.g. `p1: signal_a;` or `a1: assert signal_a;`).

2.2.2.1 Operators

This subsection discusses some commonly used PSL FL operators and their attributes. The complete list of PSL operators [92] sorted by their precedence is provided in Figure 2.17. The formal definitions for the operators are available in [92]. More details on PSL operators are also available in Appendix A of this work, which presents the *Primitive Properties Graphs* (PPG) library. *PPG library* is one of the contributions of this thesis and proposes decision diagrams based representation for a set of PSL operators.

Operator	Operator class	Associativity
and **or** **not** *etc.*	HDL operators	same as in HDL
union	Union operator	left
@	Clocking operator	left
[*] **[+]** **[=]** **[->]**	SERE repetition	left
within	SERE sequence within	left
& **&&**	SERE sequence AND	left
|	SERE sequence OR	left
:	SERE sequence fusion	left
;	SERE seq. concatenation	left
abort **async_abort** **sync_abort**	FL termination	left
next* **next_event*** **eventually!**	FL occurrence	right
until* **before***	FL bounding	right
|-> **|=>**	SERE seq. implication	right
-> **<->**	Boolean implication	right
always **never**	FL invariance operators	right

Figure 2.17. PSL FL operators (sorted by precedence from the highest on top)

In Figure 2.17 the asterisk at the end of a name (i.e. *name**) denotes the whole *family of operators*, which is a group of operators that are related. Operators of one family usually share a common family name (prefix) that is followed by suffix (can be empty). For example, *next*, *next_a*, *next_e*, *next!*, *next_a!*, *next_e!* are operators from *next** family.

Only particular operators can have their dedicated suffixes or particular a combination of them. Normally the suffixes have the following meaning:

- *"_a"* - the property should hold within *all the time* range of the operator (e.g. `p1: next_a[3 to 5](signal_1);`)
- *"_e"* - there should *exist a time moment* when the property holds within the time range of the operator (e.g. `p1: next_e[3 to 5](signal_1);`)
- *"_"* - it means the *overlapping* version of the operator (e.g. `p1: (x) until_ (y);`)
- *"!"* - it means the *strong* version of the operator (e.g. `p1: (x) until!_ (y);`)

Along with the set of operators, PSL has a number of built-it functions. Some of the functions are:

`prev()` - returns the previous value of the expression in the argument

`next()` - returns the value at the signal in the argument at the next smallest evaluation cycle

`rose()` - returns *'true'* if the signal in the argument has changed to *'1'* from *'0'* in the previous evaluation cycle, otherwise *'false'*

`fell()` - returns *'true'* if the signal in the argument has changed to *'0'* from *'1'* in the previous evaluation cycle, otherwise *'false'*

`stable()` - returns 'true' if the signal in the argument has not been changed since the previous evaluation cycle, otherwise 'false'

There are 7 more less frequently used PSL built-in functions. The formal definitions of the functions are available in [92].

It is important to notice that PSL clearly defines when each of the properties should be evaluated by adding @ clocking operator at the end of each property (e.g. `@(clk'event and clk='1');` like in Figure 2.16 for VHDL flavor). The other option if the same evaluation cycle is valid for all the properties to declare in *vunit* one default clock (e.g. `default clock is clk'event and clk='1';`). The examples given in this work and the approaches from Chapter 3 assume the latter case and the properties evaluation clock to be the same with DUV's one, if it is not stated otherwise.

2.2.2.2 Strong vs. weak operators

In the PSL the notion of strength is applicable to the properties as such and to PSL temporal operators. A property holds strongly if it holds on a finite path (simulation-based verification) and will hold on any extension of the path (e.g. `p1: eventually!(not a and b);`).

As it is mentioned above the strong version of an operator is distinguished by "!" at the end of it. Temporal operators may come in both strong and weak forms or only in one of them. For example, `always` has only weak version, `eventually!` is available only in the strong one, while `until` can have the both.

The *strong* operators require that the terminating condition eventually occurs, while the *weak* ones do not. Let's consider two properties `(busy until done;)` and `(busy until! done;)`. The first one will be satisfied even if `done` is never asserted and `busy` stayed asserted forever, while the second one requires `done` to finally occur.

2.2.2.3 Vacuous pass

The notion of vacuity is not PSL specific, however it is important for understanding the concept of assertion satisfaction and violation.

Vacuous pass occurs if a passing property contains Boolean expression that, in frames of the given simulation trace, has no effect on the property evaluation.

Let's consider a simple property `(p1: always (req -> next ack);)`. *p1* will pass on the simulation trace where signal `req` is never asserted without consideration when and if at all signal `ack` was asserted. It will be a vacuous pass because the property has passed not because of meeting all the specified behaviour but only because of non-fullfilment of logical implication activation conditions.

It is verification (simulation) tool dependant decision whether to treat vacuous passes as actual satisfactions of properties. The approaches presented in Chapter 3 separate vacuous passes from normal passes of a property.

2.2.2.4 PSL flexibility and common equivalences

Besides the convenience of 5 different flavors and different modes and styles, PSL gives the flexibility to express the same property in several possible ways. The most common equivalences for some simple properties expressed by means of different FL operators are provided in Figure 2.18. A set of commonly used equivalences between SERE and LTL style is provided in Figure 2.19. The other required equivalences can be written out as well or just used on-the-fly.

Property	An equivalent property
p **or** q	**not((not** p**) and (not** q**))**
p **and** q	**not((not** p**) or (not** q**))**
p **->** q	**(not** p**) or** q
always p	**not(eventually! (not** p**))**
always p	**never(not** p**)**
eventually! p	**not(always (not** p**))**
eventually! p	"true" **until!** p
next p	**not(next!(not** p**))**
next! p	**not(next(not** p**))**
p **until** q	**(**p **until!** q**) or (always** p**)**
p **until** q	**not((not** q**) until! ((not** p**) and (not** q**)))**
p **until!** q	**not((not** q**) until ((not** p**) and (not** q**)))**
p **until_** q	p **until (**p **and** q**)**
p **until!_** q	p **until! (**p **and** q**)**
p **before** q	**(not** q**) until (**p **and (not** q**))**
p **before!** q	**(not** q**) until! (**p **and (not** q**))**
p **before_** q	**(not** q**) until** p
p **before!_** q	**(not** q**) until!** p
next_event(b**)(**q**)**	**(not** b**) until (**b **and** q**)**
next_event!(b**)(**q**)**	**(not** b**) until! (**b **and** q**)**

Figure 2.18. Some common equivalences between PSL FL operators

This flexibility allows avoiding particular parts of the language (e.g. undesired operators or their special order, or even the whole style). The reason for this may be requirement to follow some specific rules dictated by software tools or method. In some cases, however, satisfaction of the rules forbids usage of some expressible by PSL properties in any form.

The most widely known set of such rules is the one defining the simple subset of PSL (Figure 2.21). It is a set of PSL supported by most of the available commercial hardware simulation tools with the ability to evaluate PSL assertions. The approaches described in Chapter 3 of this work have limited support for PSL as well. Some of the limitations were mentioned within this Chapter and will be discussed further.

The flexibility of the PSL gives the possibility to express a very wide set of possible properties within the frames of the method or software tool constraints.

LTL style property	An equivalent SERE style property
`eventually! b`	`{[*] ; b}!`
`eventually! s!`	`{[*] ; s}!`
`b until c`	`{b[*] ; c}`
`b until s!`	`{b[*] ; s}`
`b until! c`	`{b[*] ; c}!`
`b until! s!`	`{b[*] ; s}!`
`b and next c`	`{b ; c}`
`b and next s`	`{b ; s}`
`b and next! c`	`{b ; c}!`
`b and next! s!`	`{b ; s}!`
`next[i](b)`	`{[*i] ; b}`
`next[i](s)`	`{[*i] ; s}`
`next![i](b)`	`{[*i] ; b}!`
`next![i](s!)`	`{[*i] ; s}!`
`next_a[i to j](b)`	`{[*i] ; b[*j-i+1]}`
`next_a[i to j](s)`	`for k in {i to j}: & {[*k] ; s}`
`next_a![i to j](b)`	`{[*i] ; b[*j-i+1]}!`
`next_a![i to j](s!)`	`for k in {i to j}: & {[*k] ; s}!`
`next_e[i to j](b)`	`{[*i to j] ; b}`
`next_e[i to j](s)`	`{[*i to j] ; s}`
`next_e![i to j](b)`	`{[*i to j] ; b}!`
`next_e![i to j](s!)`	`{[*i to j] ; s}!`
`always{s} |-> p`	`{[*] ; s}! |-> p`
`always{s} |=> p`	`{[*] ; s}! |=> p`

Figure 2.19. Some common equivalences between SERE and LTL style

2.2.3 PSL simple subset

PSL provides very wide spectrum of applications, including complete design formal specification. As it was mentioned in section 2.2.1, only FL part of it is applicable for simulation-based hardware designs verification. However, not the whole FL part but usually only its subset known as *PSL simple subset* is supported by currently available commercial dynamic verification and simulation tools.

```
p1: always ((next sig_ack) -> sig_req);
p2: always (sig_ack -> next sig_ack);
```

Figure 2.20. *p1* does not belong to the PSL simple subset, while *p2* does

Loosely speaking the simple subset has a requirement for time to advance monotonically within the property expression. An example of two properties *p1*

PSL operator		Restriction
`not`		the operand is a Boolean
`never`		the operand is a Boolean or a sequence
`eventually!`		the operand is a Boolean or a sequence
`or`		at most one operand is a non-Boolean
`->`		the left-hand side operand is a Boolean
`<->`		both operands are Boolean
`until`	`until!`	the right-hand side operand is a Boolean
`until_`	`until!_`	both operands are Boolean
`before*`		both operands are Boolean
`next_e*`		the operand is Boolean
`next_event_e*`		the FL Property operand is Boolean

Figure 2.21. PSL simple subset rules

and *p2* is provided in Figure 2.20. Here, *p1* does not belong to the PSL simple subset, while *p2* does.

The simple subset is explicitly defined in [92] by the set of rules stating restrictions for several PSL operators operands types. The rules are provided in Figure 2.21. IBM, the authors of PSL's predecessor Sugar, have published in [60] formal proof for the PSL simple subset specification.

The aim of the simulation-based verification part of the approaches from Chapter 3 is the support for the complete PSL simple subset (not fully implemented yet and scheduled for the future work).

2.3 Chapter summary

This chapter has provided background information required for understanding of the main part of this thesis, where the work contributions are presented. The approaches proposed further rely on the design representation model based on high-level decision diagrams, proposed and developed in TUT. The first part of this chapter has discussed this model and emphasized its main advantages for application in simulation-based verification.

The second part of the Chapter has presented PSL, as the source and reference language for assertions used in assertion-based verification methods described further in this work. The language description presented was oriented towards the approaches from this thesis.

Chapter 3
ASSERTION-BASED VERIFICATION

"Functional verification is hard. Period. No disagreement here,"-
Harry Foster[3]

Hardware design verification phase is known to take even more computational and human resources than the design phase itself. This chapter proposes an approach of HLDD-based assertion checking aimed to assist with this problem.

The main scope of this chapter is simulation-based verification aided by assertions. The three main contributions of this chapter are the following. The first one is a temporal extension for the existing HLDD model, described in the previous chapter. The second one is a methodology for direct conversion of assertions expressed in PSL to temporally extended high-level decision diagrams (THLDD). The third contribution is HLDD-based simulator HLDD*sim* modification to support THLDDs and assertions checking.

A set of experimental results demonstrating the feasibility and effectiveness of the proposed concept is provided at the end of the chapter. Here the proposed approach is compared against a commercial simulator with PSL assertions support from a major CAD vendor.

Finally, the reusability of verification assertions for manufacturing testing development is briefly discussed.

---

[3] *Harry Foster* is the chair of IEEE-1850 WG, the chair of Accellera Formal Verification Technical Committee, principal engineer in Mentor Graphics, the author of several books about verification (i.a. [3] and [7]). (The citation is the first clause in the foreword to [4]).

3.1 Overview

3.1.1 Design flow

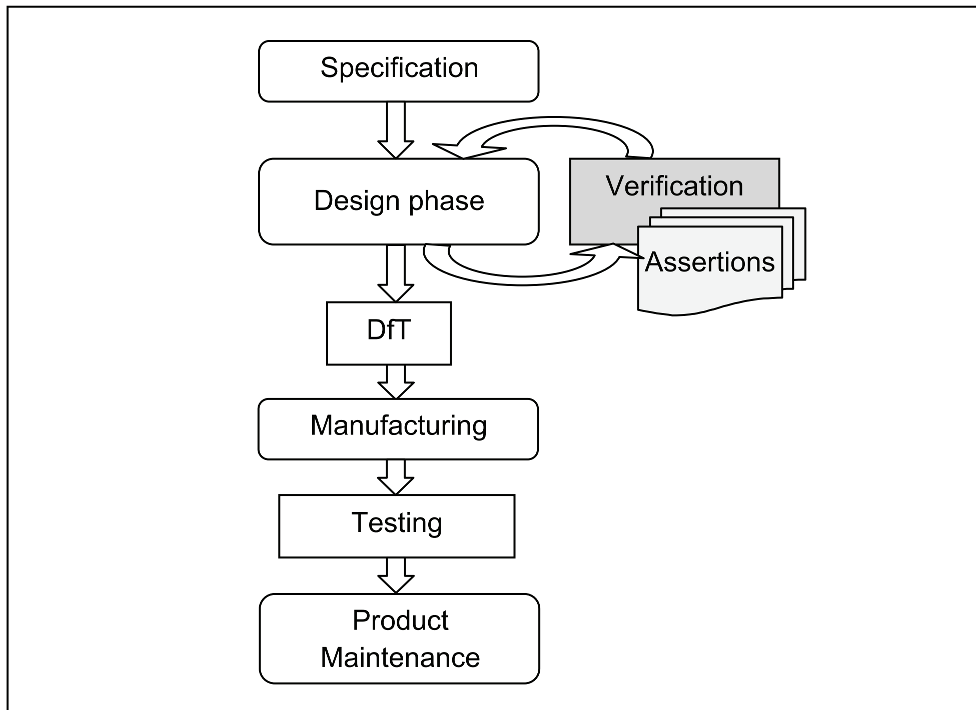


Figure 3.1. Design development flow (simplified)

The flowchart in the Figure 3.1 shows the typical flow of a design (e.g. ASIC, i.e. Application Specific Integrated Circuit) development process. The process starts from the *specification phase* which normally results in formal or partially-formal list of functionality and requirements for the future product. At the *design phase* the product implementation begins and usually goes through several levels of abstraction. The abstraction levels may include behavioural, TLM, RTL, gate and finally physical layout, required for the actual manufacturing and normally obtained by the synthesis tools. The design phase closely interacts with *verification phase* which checks the design's partial and complete implementations for their functional correctness. Further the verified design is manufactured (*manufacturing phase*) at the factory and passed to the customer. At this point product *maintenance phase* begins, which concludes the product development process. However, in order to ensure the physical manufacturing correctness of the product instances (chips), each one of them should pass the *testing phase* against physical defect. The

last one is a complex task itself and can be assisted by *Design for Testability* (DfT) *phase*.

At this stage it is a good time to distinguish between three important notions in the hardware design development process. The terms are sometimes used interchangeably in literature and some scientific publications. However, we would like to emphasize clear difference in their definitions (rephrased from [1] and [2]) used frames of this work.

- *Verification* is meant to ensure that design fulfils the specified functionality. Verification is performed in software using models (i.a. HDL-based) evaluated in PC environment.
- *Validation* aims at the same target as verification. However, here the object is real physical hardware prototype as opposed to model in verification.
- *Testing* aims at physical defects in each produced instance of the product.

While validation allows theoretically more accurate results, compared to verification, it is obviously more expensive to implement. In practice a *validation phase* can be added to the ASIC development flow after the design and verification phases in the simplified flowchart shown in Figure 3.1. Moreover, practically, validation often complements verification and used to check different more precise aspects of the implementation's functional correctness. While validation is not considered in the reminder of this thesis, manufacturing testing will be discussed in more detail in the Section 3.6.

3.1.2 Design verification

Functional design verification usually assumes comparison of one implementation against specification or another implementation (alternative or simplified, i.a. at a higher abstraction level). The other way is not to compare against a reference but to narrowly aim at specific design properties. The second approach is usually referred as Assertion-Based Verification (ABV).

The two main types of the hardware design verification approaches are *formal* (its alternative term is *static*) and *simulation-based* (alternative term is *dynamic*) ones. However, particular designs verification strategies (plans) may benefit from *semiformal* or other words mixed-type verification approaches.

Informal definitions of the verification types are the following:

- *Formal verification* assumes formal mathematical prove of the design correctness or its property validity. The formally proven aspect stays valid for any stimuli, if it is not stated otherwise.
- *Simulation-based verification* relies on design simulation by a set of predetermined or random stimuli. The simulation results (waveforms) are further analyzed for similarity with the reference simulation results (e.g. of an

alternative implementation) or checked for a particular behaviour specified by the DUV's property.

In the industry the simulation-based verification approaches find much wider application due to their lower requirements to computational resources. The pure formal verification methods are not practically applicable to real life large designs and normally used only for designs smaller parts of a particular functionality.

The decision diagrams based methods proposed in this chapter are dedicated to simulation-based verification. However, some of the contributions can be adapted to formal verification as well.

3.1.3 Assertion-based verification

A way to cope with the verification complexity is *Design-for-Verifiability* (DfV) techniques. They serve for the same purpose as widely used and well known DfT techniques for manufacturing testing. *Assertion-Based Verification* (ABV) can be classified as one of the DfV techniques. The main idea of DfV is application of complementary parts of HDL code introduced especially for design verification assistance. In case of ABV this complementary code is assertions.

ABV is meant to assist both formal methods and simulation-based verification and allow discovering Design under Verification (DUV) misbehaviour (causing an assertion violation) earlier and more effective. Another important advantage of ABV is its aid to debug process.

As it has been specified in Subsection 2.2.2, formally, assertions are asserted properties. Speaking loosely, assertions are formal representations of desired intents of the specification engineer or the designer. In the case of simulation-based verification they provide better *observability* on the design what allows detecting bugs earlier and closer to their origin. At the same time in the case of formal verification with model checking, the assertions increase the *controllability* of the design and direct verification to the area of interest. Each assertion violation discovered by model checking is reported as a counter-example.

ABV was initially separately aimed at the main drawbacks of simulation-based and formal verification approaches. For the first one it is known that the performance of simulation is low at the system level and the design coverage is inversely proportional to the complexity of the system. For the second it is the fact that formal verification can achieve very high coverage but it has very limited scalability that usually cannot go above the module level.

The question of the origin of assertions can be formulated as a separate topic for research itself. An important aspect here is the problem of *completeness*. Usually assertions do not describe all the possible properties of a design what would mean translation of the complete design specification to an appropriate formal specification language (e.g. PSL). Instead of this only design areas of concern,

sometimes referred as *verification hot spots*, are targeted. In practice they are often provided by design engineer and require deep knowledge of the DUV behaviour. Verification hot spot, as defined by [64], typically:

- ✓ contains a great number of sequential states
- ✓ deeply hidden in the design, making it difficult to control from the inputs
- ✓ has many interactions with other state machines and external agents
- ✓ has a combination of these properties

As it has been already mentioned in the Clause 2.2.1.2, in practice there are two modes [2] of assertions description indirectly implicitly related to the two origins of assertions. The first mode is named embedded and assumes that the assertions are written directly to the design's HDL files. It is preferred by design engineers who are the first group of assertions creators. The second mode is named stand-alone, which means that all assertions related to the design or its part (e.g. separated by a different HDL file) are grouped into a separate file. This mode allows more complex organization of assertions and normally preferred by verification engineers who form the second group of assertions creators.

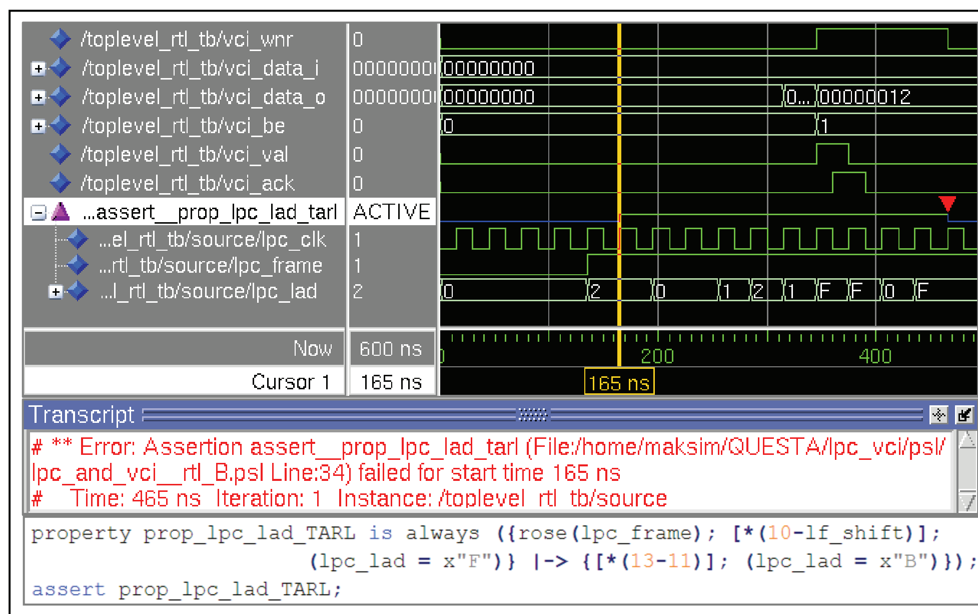


Figure 3.2. An example of assertion checking by a commercial tool

Figure 3.2 shows example of assertion checking by a commercial tool. The screenshot in the figure is waveform, transcript and assertion expression windows of QuestaSim environment from Mentor Graphics Corp. Here the asserted property *prop_lpc_lad_TRAL* is activated at 165 ns of simulation time and fails at 465 ns

(marked by reversed triangle), because signal *lpc_lad* of the DUV has taken value *x'F'* instead of the awaited *x'B'*. The simulator has reported an error to the transcript window. In the described simulation-based verification case, the application of this assertion has helped to detect DUV's functional misbehaviour. The preconditions for this detection are an appropriate assertion and stimuli capable to cover the assertion.

3.1.3.1 Diversity of assertion checking

The notion of *assertion checking* may refer to different processes depending on the type of verification approach and other factors.

In case of simulation-based verification, assertion checking means monitoring simulation results of the DUV for particular combinations notifying about satisfaction or violation of the property of interest. Such application of assertion checking has given assertions pseudonym *monitors* [6]. The properties with verification directives *assume* or *restrict* (for details, see Clause 2.2.1.3 and Subsection 2.2.2) instead of *assert* are sometimes named *generators* [6] and can serve for stimuli filtering or generation (this topic is discussed in more details in Subsection 3.6).

As it was mentioned above assertion checking in formal verification may result in creation of a counter-example in case if the assertion fails. The assertions to be checked in formal verification are often referred to as *checkers*.

At the same time there is a subset of assertions that can be expressed in the synthesizable form. The synthesizable form of assertions cannot be expressed in PSL and should be converted to constructs represented by another appropriate model (e.g. one of the HDLs, i.a. VHDL). These constructs are also referred to as *(hardware) checkers*. They are usually embedded to the version of DUV ready for synthesis and can be physically implemented as a single design on a chip, its prototype or in FPGA. These checkers can serve for many purposes including design validation and online test patterns generation for manufacturing testing (see Subsection 3.6). Usage of hardware checkers can lead to huge area overhead due to complex temporal relationships in the property implemented by the checker.

The main part of this chapter focuses on *assertion checking in simulation-based hardware verification*. Using alternative terminology it is possible to say that it is focused on *hardware assertions monitoring in software*.

3.1.4 State of the art

To the best of our knowledge this thesis, supported by the series of publications ([12],[11],[10],[13] and [14])[co-auth.], is the first attempt of PSL assertions conversion to and checking with decision diagrams based design representation model. Here

by decision diagrams we consider acyclic graphs, but not cyclic automata-based structures.

The research topic of PSL properties conversion to design representation models is gaining its popularity. There are several approaches published in recent time ([65] - [75]). The target application of the converted properties varies and includes among others software monitors and hardware checkers.

Yael Abarbanel et al. in [65] have proposed a tool called Formal Checkers (FoCs) [95] that has become very popular today and is widely used as a reference. It is capable of converting PSL properties of different flavors to synthesizable VHDL. This tool is discussed further in Clause 3.1.4.1.

Oddos, Morin-Allory et al. in [66],[73] have proposed a modular approach where sub-modules for each PSL property operator are built and interconnected according to the expression being implemented, so that they produce a RTL synthesizable design. Assertions produce a pair of signals that indicate the status of the assertion. Such generator can be connected to the design under test for verification by simulation or emulation.

Gheorghita et al. in [67] propose a competitor to FoCs, producing automata. It considers *e* and Verilog as output checkers HDLs. The time to produce the checkers is larger than in FoCs.

Bustan, Fisman et al. in [68] provide automata construction for the core logic of PSL defined by them and named as LTL-WR, which is an extension of LTL with regular expressions. In their work they show that for every LTL-WR formula there exists a non-deterministic Bäuchi automaton whose size is exponential in the size of the formula.

Pidan et al. in [69] proposed an algorithm with similar to [68] complexity for designing dynamic verifiers for PSL formulas. Firstly, they transformed a PSL formula to a non-deterministic finite automaton. Then they implemented the NFA with a discrete transition system, which, in turn, was translated into HDL codes.

Kotasek et al. in [70] propose a methodology for generating VHDL descriptions of hardware checkers for such components as communication protocols, counters, decoders, registers and comparators. The proposed application of the approach is the design of fault tolerant systems.

Boule, Zilic et al. in [71],[72] present a technique for automata-based checker generation of PSL properties oriented on dynamic verification and post-fabrication silicon debug. Their full automata-based approach (the tool's name is MBAC) allows an entire assertion to be represented by a single automaton, as opposed to modular approaches (e.g. [73]) where sub-circuits are created only for individual operators. For this purpose, automata algorithms are developed for the base cases, and a complete set of rewrite rules is developed and applied for all other operators. The checkers produce a single result signal for each assertion, as opposed to the paired signal result (e.g [73]).

Riazati, Navabi et al. in [74],[75] propose an approach for hardware checkers creation by synthesizing OVL assertions. The intended application area is online manufacturing test and fault tolerant systems. First, based on the ATPG results and fault simulation, they select a set of assertions with a good ratio of fault coverage over hardware area. Second, they merge similar assertions together and make a unified hardware checker in order to attain minimized resource usage for assertion circuits and reduce hardware overhead.

Direct PSL assertion checking in simulation-based verification is supported by a large number of commercial CAD tools and includes among others QuestaSim from Mentor Graphics Corp [77]. A list of more than 40 of such tools available by March 2006 was gathered by IBM in [100]. By current date the list should be significantly longer, considering that PSL has been approved as an IEEE standard in 2005.

The main difference of the approach proposed in this chapter from the listed above ones are:

- It allows *avoidance of the synthesizable HDL descriptions related constraints*. The approach aims verification by simulation in software, as opposed to validation by hardware emulation and post-silicon test/debug. This topic is discussed in more details in Clause 3.1.4.1.
- At the same time the PSL properties are converted in *straightforward* way to *acyclic decision diagrams* without any loose of information. No automata-like structures constructions are involved in the conversion process.
- The approach exploits a proven efficient for simulation *high-level decisions diagrams* design representation model.

3.1.4.1 An experience with FoCs

Our first attempt ([12],[24])[co-auth.] of PSL properties translation to HLDD was implying the generation of VHDL checkers by IBM's FoCs [65],[95] as an intermediate step. However this experience has revealed particular limitations and inefficiency for HLDD-based assertions creation. Moreover, checkers synthesis from PSL properties are efficient mainly for the case where checkers are to be used in hardware emulation. The application of the same checker constructs for simulation in software may lack efficiency due to target language concurrency and poor means for temporal expressions.

The details of the FoCs-based approach for PSL assertions conversion to HLDD model can be demonstrated by the example from [12][co-auth.]. The object for this example is a simple SERE style PSL assertion *fe_seq* (Figure 3.3).

```
fe_seq: assert always ({a; [*2]; b} |=> {c});
```

Figure 3.3. The *fe_seq* assertion for FoCs-based approach experience

The precondition of *fe_seq* assertion is the sequence of system behaviour when signal *a* is set to *'1'*, followed by a don't-care sequence 2 clock cycles long and then signal *b* set to *'1'*. This precondition activates the main part of the assertion and requires *c* to be set to *'1'* just after it (non-overlapping implication).

```
PROCESS (clk)
BEGIN
  IF ( ( clk = '1' ) ) THEN
    focs_ok <=
    ( focs_vout(4) OR NOT( c ) ) ;
  ELSE
    focs_ok <= '1' ;
  END IF;
END PROCESS;

PROCESS
...
VARIABLE focs_vout : std_logic_vector(4 DOWNTO 0);
BEGIN
  WAIT UNTIL (clk'EVENT AND clk = '1');
  ...
  focs_vout(4 DOWNTO 0) := reverse( ( ( ( ( ( ( focs_v(0) AND a ) )
    & ( ( focs_v(1) AND '1' ) ) ) & ( ( focs_v(2) AND '1' ) ) ) &
    ( ( focs_v(3) AND b ) ) ) & ( ( focs_v(4) AND NOT( c ) ) ) ) );
  ...
END PROCESS;
```

Figure 3.4. VHDL checker generated by FoCs for assertion *fe_seq*

Figure 3.4 shows a shortened form of the resulting VHDL code generated by FoCs from the *fe_seq* expressed in PSL. The VHDL checker can be converted to HLDD graphs by means of the standard VHDL to HLDD interface tool [59].

A possible system of HLDD graphs representing the checker is provided in Figure 3.5. In the figure we use a notation where the prime symbol " ' " after diagram variable denotes one clock cycle delay.

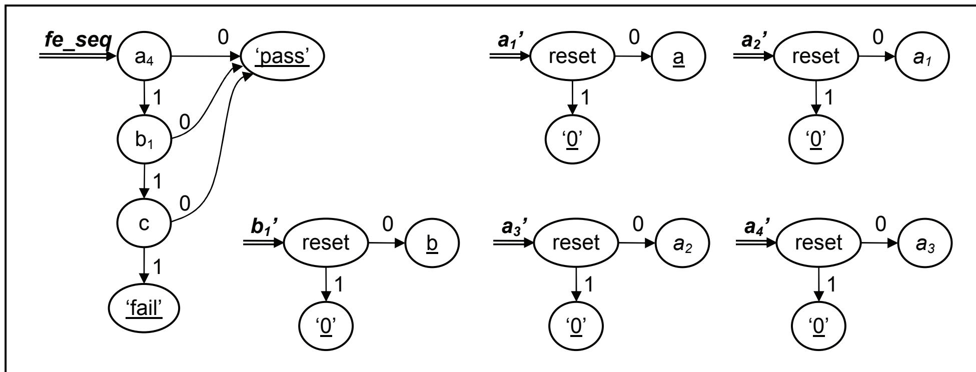


Figure 3.5. HLDD representation of the VHDL checker from Figure 3.4.

As it can be seen from the Figure 3.5, the FoCs-based approach result in very ineffective system of HLDD graphs, which can be unreasonably large in case of

simple but long-time temporal properties. Here a separate graph is required for every variable's evaluation cycle delay. Signals *a* and *c* are located in 4 cycles time distance and therefore caused 4 intermediate variables a_1-a_4. In case of physical hardware the intermediate variables may have to be substituted by memory elements (registers).

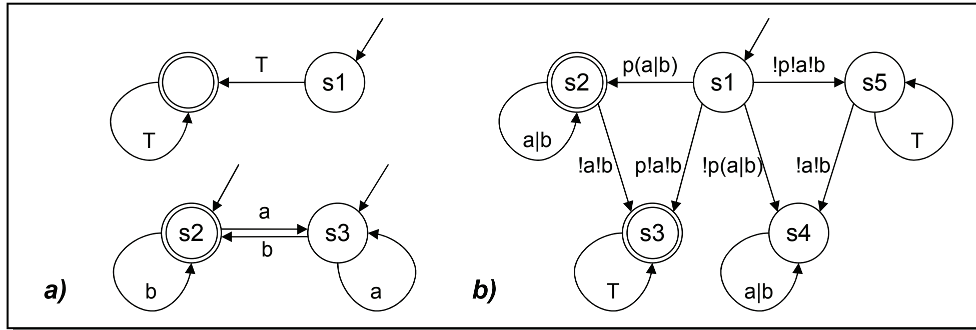


Figure 3.6. An example of NFA and DFA for a pure LTL property *!p→G((a|b)*)*

A solution to the inefficient for our task approach of checkers representation can lay in an explicit automata construction (FoCs may also in some cases implicitly construct automata-based structures). An example of such solution (among many others partially mentioned in Subsection 3.1.4) is provided in [6]. The approach requires a construction of NFA (Nondeterministic Finite Automation) followed by its defeminisation to DFA (Deterministic Finite-state Automaton). An example of NFA and DFA for a property expressed by original LTL with Regular Expressions is shown in Figure 3.6a and Figure 3.6b correspondingly (the semantics explanation is provided in [6]). However, this complicated solution does not suit for our target application which is verification by design simulation in software.

Therefore a different solution for PSL assertions conversion to the target HLDD model was required. It should use all the advantages of model simulation in software without involving unnecessary restrictions (e.g. the ones came with FoCs application as an intermediate step). Later, we have proposed the solution as idea in [11][co-auth.] and more detailed in [10][co-auth.] and [13][co-auth.]. The approach is one of the main contributions of this thesis and described in details further in this chapter.

3.1.5 APRICOT

APRICOT is an acronym for *Assertions checking (monitoring), formal PRoperty checkIng, verification COverage measurement and Test pattern generation*. This is the name [14][co-auth.] for a hardware verification framework developed by Tallinn University of Technology. As it follows from its name decryption, the framework supports a wide range of verification tasks. The novelty of APRICOT lies in the usage of the HLDD design representation model (see Subsection 2.1.2) advantages for the mentioned above verification tasks.

The direct APRICOT development has started during participation of TUT in Framework Program 6 European project VERTIGO [89]. The partners of the project besides TUT are ST Microelectronics (co-ordinator), Aerielogic, TransEDA and three other universities: LIU (Linköping, Sweden), SOTON (Southampton, UK) and UNIV (Verona, Italy).

The framework is aimed at both education and research. It has interfaces to commonly used design formats such as VHDL, SystemC, PSL and EDIF, as well as an intermediate format *HIF* (HDL Intermediate Format), developed by UNIV. The internal format for SSBDD, HLDD and THLDD models representation is *AGM* (Alternative Graph Model format). Figure 3.7 shows the APRICOT verification flow.

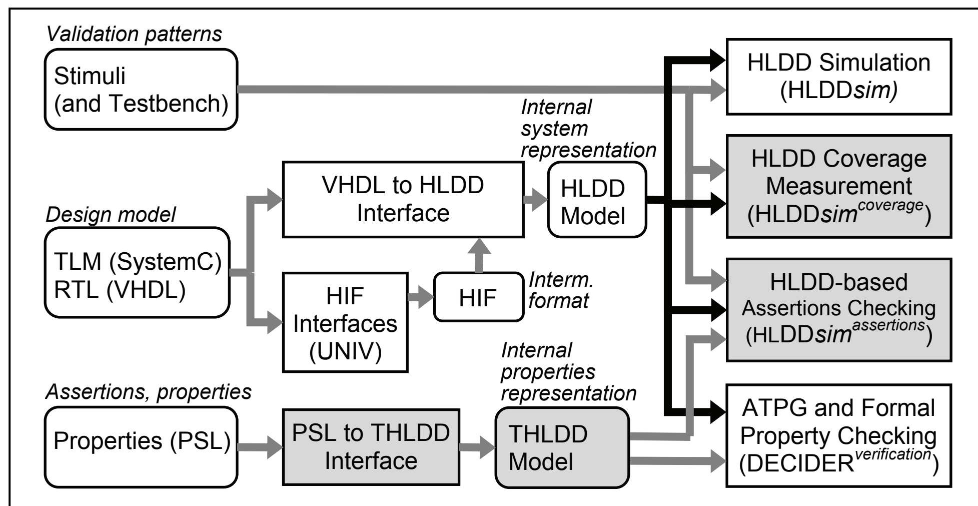


Figure 3.7. APRICOT verification flow

VHDL to HIF and SystemC to HIF interfaces are developed at UNIV and integrated to the flow as intermediate steps. HIF to HLDD interface is developed in cooperation [89] between TUT and UNIV, and the direct VHDL to HLDD interface is developed in TUT [59].

The formal methods implemented in the APRICOT framework include high-level Automated Test Pattern Generation (ATPG) and formal property checking. The former is based on DECIDER engine [19][co-auth.]. The property checking constituent is currently reduced to using the modified DECIDER ATPG with constraint solving support (DECIDER^{verification}) [46] as a model-checking engine.

The basis for verification coverage analysis and assertion monitoring is fast *HLDD-based simulator* (HLDD*sim*) with the corresponding modifications. The following parts (marked my grey) of the APRICOT are contributions of this thesis and will be discussed further in more details:

- HLDD-based assertions checking (HLDD$sim^{assertions}$)
- HLDD-based verification coverage analysis (HLDD$sim^{coverage}$)
- PSL to THLDD interface
- THLDD model for properties representation

The APRICOT verification framework is easy to use because of the variety of the available interfaces to the common design formats. It supports a wide range of verification tasks that alternatively would require a set of different commercial CAD tools. All the tools of APRICOT are based on the efficient high-level decision diagrams design representation model and allow homogeneous verification flow. The experimental results (partially presented in Sections 3.5 and 4.5) show the advantages of HLDD-based verification tools compared to the tools from the major CAD vendors.

The mentioned above contributions of this thesis to the APRICOT are described in details in the current and the following chapters.

3.2 Temporal extension for the HLDD model

The first attempts of assertions application in hardware verification have revealed inefficiency and sometimes inability of the traditional hardware description languages to express complex temporal relationships in the properties. This fact has encouraged development of PSL (Section 2.2), with its powerful temporal instruments, and several other assertions-oriented languages mentioned in the introduction to Section 2.2. In the last years these languages have found a wide application in ABV projects and proven their efficiency.

In the same way as PSL adds temporal instruments to assertion-based verification of the DUVs represented by HDL a temporal extension to HLDD model is required for a successful assertions application in HLDD-based verification flow.

One of the principles of HLDD-based verification is homogeneous verification flow. It means that all objects of the flow should be represented by the same or a compliant design representation model, i.e. HLDD or HLDD compliant. While HLDDs are proven [44],[45],[19][co-auth.] to be efficient for design representation, representation of temporal PSL properties by pure HLDDs has revealed inefficiency of this approach. This fact is discussed in details in Clause 3.1.4.1. The solution developed in ([11],[10],[13])[co-auth.] and provided in this chapter is a temporal extension to HLDD model. The model with this extension support is named as *Temporally extended High-Level Decision Diagrams* (*THLDD*).

Further in this section the definition of THLDDs is presented and the interface of the novel model is discussed.

3.2.1 THLDD model definition

Unlike the traditional HLDD described in the Subsection 2.1.2, the temporally extended high-level decision diagrams are aimed at representing temporal logic properties.

A temporal logic property P at the time-step $t_h \in T$ denoted by $P_{t_h} = f(x,T)$, where $x = (x_1, \ldots, x_m)$ is a vector defined on a finite domain $X = X_1 \times \ldots \times X_m$ and $T = \{t_1, \ldots, t_s\}$ is a finite set of time-steps. In order to represent the temporal logic assertion $P_{t_h} = f(x,T)$, a temporally extended high-level decision diagram G_P can be used.

Definition 2**:** A *Temporally extended High-Level Decision Diagram* (THLDD) is a non-cyclic directed labelled graph that can be defined as a sextuple $G_P=(M,E,T,Z,\Gamma,\Phi)$, where M is a finite set of nodes, E is a finite set of edges, T is a finite set of time-steps, Z is a function which defines the variables labelling the nodes, Γ is a function on E representing the activating conditions for the edges, and Φ is a function on M and T defining temporal relationships for the labelling variables.

The graph G_P has exactly three terminal nodes $M^{term} \in$ M labelled by constants, whose semantics is explained below:

- *FAIL* — the assertion P has been simulated and does not hold;
- *PASS* — the assertion P has been simulated and holds;
- *CHECKING* — the assertion P has been simulated and it does not fail, nor does it pass non-vacuously (See Clause 2.2.2.3 and Subsection 3.3.1 for discussions of vacuity).

The function $\Phi(m_i,t)$ represents the relationship indicating at which time-steps $t \in T$ the node labelling variable $x_l=Z(m_i)$ should be evaluated. More exactly, the function returns the range of time-steps relative to current time t_{curr} where the value of variable x_k must be read. We denote the relative time range by Δt and calculus of variable x_l using the time-range $\Phi(m_i,t)= \Delta t$ by $x_l^{\Delta t}$. We distinguish three cases:

- $\Delta t=\forall\{j,...,k\}$, meaning that $x_l^{\Delta tj} \wedge ... \wedge x_l^{\Delta tk}$ is true, i.e. variable x_l is true at every time-step between t_{curr+j} and t_{curr+k}.
- $\Delta t=\exists\{j,...,k\}$, meaning that $x_l^{\Delta tj} \vee ... \vee x_l^{\Delta tk}$ is true, i.e. variable x_l is true at least at one of the time-steps between t_{curr+j} and t_{curr+k}.
- $\Delta t=k$, where k is a constant. In other words, the variable x_l has to be true k time-steps from the current time-step t_{curr}. In fact, $\Delta t=k$ is equivalent to and may be represented by $\Delta t=\forall\{k,...,k\}$, or alternatively by $\Delta t=\exists\{k,...,k\}$.

Notation $event(x_c)$ is a special case of the upper bound of the time range denoted above by k and means the first time-step when x_c becomes true. This notation can be used in the three listed above THLDD temporal relationship functions $\Phi(m_i,t)$,

which creates the listed below variations of them. For $x_l^{\Delta t}$, where x_l and x_c are node labelling variables:

- $\Delta t=\forall\{0,...,event(x_c)\}$, which means that variable x_l is true at every time-step between t_{curr} and the first time-step when variable x_c becomes true, inclusive. This is equivalent to the PSL expression x_l *until_* x_c. The PSL expression x_l *until* x_c can be represented by $\Delta t=\forall\{0,...,event(x_c)-1\}$.
- $\Delta t=\exists\{0,...,event(x_c)\}$, which means that variable x_l is true at least at one of the time-steps between t_{curr} and the first time-step when variable x_c becomes true, inclusive. This is equivalent to the PSL expression x_l *before_* x_c. The PSL expression x_l *before* x_c can be represented by $\Delta t=\exists\{0,...,event(x_c)-1\}$.
- $\Delta t=(k\cdot event(x_c)+1)$, which means that variable x_l has to be true at the next time-step after the one where x_c is true for the kth time. This is equivalent to the PSL expression $next_event(x_c)[k]x_l$. k is a positive integer greater or equal to 1. If k=1, then $\Delta t=(event(x_c)+1)$ and its is equivalent to the PSL expression $next_event(x_c)\ x_l$.

For Boolean, i.e. non-temporal variables $\Delta t = 0$.

The notion of *event*(x_c) has been introduced in [13][co-auth.].

3.2.2 THLDD interface

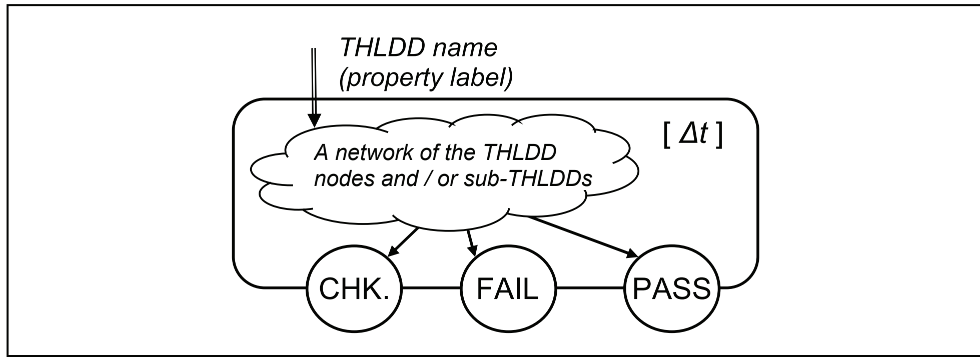


Figure 3.8. Standard THLDD interface

As it is mentioned above each THLDD graph has one root node and exactly 3 terminal nodes (*CHECKING*, *FAILED* and *PASSED*), as opposed to HLDD graphs that have an arbitrary number of terminal nodes. This constraint has been introduced [11][co-auth.] to allow strict hierarchy in complex THLDD graphs that imply THLDD sub-graphs. It has also an optional relative time range Δt, which shows when the assertion has to be checked. As it will be shown further, the standard interface is a precondition to the efficient and easy method for complex THLDD construction. The standard interface is shown in Figure 3.8.

The meanings of the terminal nodes of the interface are as specified in Definition 2. This interface is suitable for the subset of PSL properties with weak versions of operators only. The support of strong versions of PSL operators requires the fourth terminal node PENDING (and, as it will be discussed further, modification of some PPGs, see Section 3.3). The support of strong versions of PSL operators would not influence the principals of the approach described in this chapter and is scheduled for the future work.

3.2.3 THLDD temporal relationships

One of the motivations for introduction of PSL was the poor ability of standard HDL languages to express temporal relations between expressions in assertions. The main instruments for this purpose used in PSL are temporal operators of its own (LTL) and repetition operators of SERE. A powerful part of the temporal operators are their auxiliary suffixes (see Clause 2.2.2.1).

In this thesis we propose a temporal extension for the HLDD model that supports several temporal PSL constructs. The table in Figure 3.9 shows examples on how temporal relationships in THLDDs map to PSL expressions. The first two of the proposed in the table THLDD temporal relationship constructs are *basic*, while the following four are *derivative* from them.

Class	THLDD construct Φ	Formal semantics	Equivalent PSL expression
Basic	$x^{\Delta t=\forall\{j,\ldots,k\}}$	x holds at all time-steps between t_j and t_k	*next_a[j to k] x*
	$x^{\Delta t=\exists\{j,\ldots,k\}}$	x holds at least once between t_j and t_k	*next_e[j to k] x*
Derivative	$x^{\Delta t=k}$	x holds at k time-steps from t_{curr}	*next[k] x*
	$x^{\Delta t=\forall\{0,\ldots,event(x_c)\}}$	x holds at all time-steps between t_{curr} and the first time-step from t_{curr} where x_c holds	*x until_* x_c
	$x^{\Delta t=\exists\{0,\ldots,event(x_c)\}}$	x holds at least once between t_{curr} and the first time-step from t_{curr} where x_c holds	*x before_* x_c
	$x^{\Delta t=(k\cdot event(x_c)+1)}$	x holds at the next time-step after the one where x_c holds for the kth time from t_{curr}	*next_event*(x_c)*[k] x*

Figure 3.9. Table of temporal relationships in THLDDs

In addition, we introduce [11][co-auth.] the notion of t_{end} as a special value for the upper bound of the time range denoted by above by k. t_{end} is the final time-step that occurs at the end of simulation and is determined by one of the following cases:

- Number of test vectors

- The amount of time provided for simulation
- Simulation interruption

The special values for the time range bounds (i.e. *event*(x_c) and t_{end}) are supported by the HLDD-based assertion checking approach (please see Section 3.4 for the details). In the proposed approach design simulation, which calculates *simulation trace,* precedes assertion checking process. In practice, t_{end} is the final time-step of the pre-stored simulation trace.

Note, that the main purpose of THLDD as the proposed temporal extension for the existing HLDD model defined in Subsection 2.1.1 is transferring additional information (i.a. temporal) to the modified HLDD simulator HLDD*sim*^{assertions} that will be used for assertions checking. Further minor not principal extensions of the THLDD model would require, first, introduction of the new notation to the model (primarily its description in the AGM format) and, second, the corresponding "teaching" HLDD*sim* how to understand and process this extension.

3.3 PSL to THLDD conversion method

The idea of the proposed conversion method ([11],[10],[13])[co-auth.] relies on the principle of "*divide and conquer*". The method is based on partitioning of PSL properties into elementary entities containing only one operator. There are two main stages in the approach. The first one is preparatory and consists of *Primitive Property Graphs Library* creation for elementary operators. The second stage is PSL assertion expression *parsing* and recursive *hierarchical construction* of the THLDD for a complex property using the PPG library elements.

3.3.1 Primitive Property Graphs

Prior to the THLDD property construction procedure a *Primitive Property Graph* (PPG) should be created for every PSL operator supported by the proposed approach. All the created PPGs are combined into one *PPG library*. The library is extensible and should be created only once. It implicitly determines the supported PSL subset. The method currently supports only weak versions of PSL operators. However, by means of the supported operators a large set of properties expressed in PSL can be derived, as it was discussed in Clause 2.2.2.4.

Primitive Property Graph is always a THLDD graph. That means it has the standard interface with one root node and three terminal nodes (*CHECKING*, *FAIL* and *PASS*) as it was described in Subsection 3.2.2. Some potential modifications of the approach like strong PSL operators' support, and consequently modification of the interface, may lead to recreation of all PPGs.

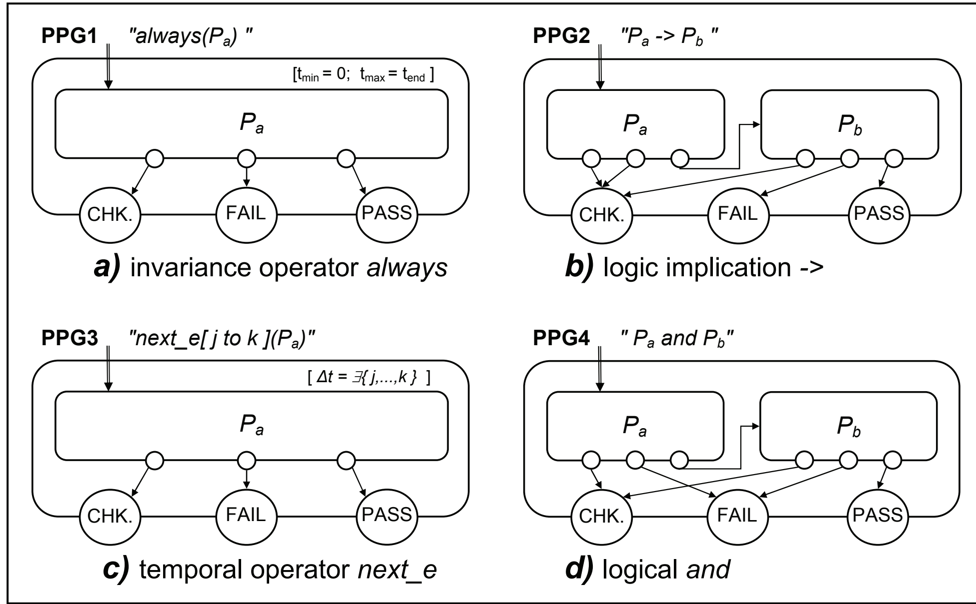


Figure 3.10. Example PPGs for a set of PSL operators

Example PPGs created for 4 PSL operators are shown in Figure 3.10. The complete set of PPGs from the current version of the PPG library with detailed complementary information is presented in Appendix A.

Note, that the logic implication operator '->' in Figure 3.10b exits to the terminal node *CHECKING* when the precondition P_a fails. This is due to the fact that in assertion checking the verification engineer is not interested in non-vacuous passes of the property (see Clause 2.2.2.3). The terminal node *CHECKING* is allowed to be eliminated from some graphs where it practically cannot be reached. This permission does not interfere with the proposed general THLDD structure. The PPGs, as well as complex THLDDs, without temporal relationships (e.g. logical *and* and logical implication) are evaluated to one of the terminal nodes at every time-step of the assertion checking. At the same time, the PPGs, as well as complex THLDDs, with temporal relationships (e.g. logical *always* and *next_e*) may evaluate to one of the terminal nodes at an arbitrary time-steps of the assertion checking, according to their particular temporal relationship function. In the following subsection an assertion checking algorithm is presented that is capable of handling such functions.

3.3.1.1 PPG Library

Appendix A at the end of this thesis presents the extensible *Library of Primitive Property Graphs* (PPG Library) in details. First, the format of *ppg.lib* file used for

THLDD properties constructor is given. Then PPGs for a set of supported PSL operators are provided. Every clause describing a PPG consists of:

- PSL notation in correspondence with [92]
- THLDD graph in AGM format
- THLDD graph graphical portrayal
- PPG operator related notes.

PPG Library section is separated to a separate Appendix because of its size and in order to keep the coherence of the PSL to THLDD properties conversion method explanation flow.

3.3.2 Parser

The second stage of the proposed method is implemented [10][co-auth.] as one tool but has a logical division into two separate tasks. The first task is PSL property expression parsing.

As it has been mentioned in Section 2.2 PSL is a very rich and powerful language. None of the commercial or academic tools available at present have the support for the whole language set, moreover such a support would be impractical. In practice each of the tools with PSL support has a set of rules for the PSL-expressed properties and assertions. At present, the proposed approach has the following restrictions and assumptions for the PSL expressions. The most of them are not principal and can be easily modified in future.

1. *Stand-alone mode*. As it has been specified in Clause 2.2.1.2 and Section 3.1.3, in this mode all the PSL properties related to the DUV are put into separate files, as opposed to their embedment into the DUV's HDL files.
2. *Single clock.* The present implementation of the approach assumes all the PSL assertions under conversion are clocked and their clock is the same as the one of the DUV. A consequent assumption is a single clock in the part of the DUV set up for co-simulation with the assertions. In terms of PSL there should present one default clock declaration for all the properties in the file (e.g. `default clock is clk'event and clk='1';`).
3. *Limited support for the verification and modelling layers of PSL*. Currently each property is supported and assumed to have only verification directive *assert* (i.e. the approach supports only assertions). It can be adjoined directly to the property body (Figure 3.11a) or stated separately with the support for combination of properties (Figures 3.11b and 3.11c). The most of the constructs of the verification and modelling layers are not supported directly; however their meaning can be expressed by means of *upper-layer THLDD properties* (discussed further in Subsection 3.3.4).

4. *Subset of supported PSL operators*.

   a. The current implementation supports only LTL style and does not support SERE style of PSL assertions expressions. This support can be easily added by extension of PPG Library and minor extension of the parser.

   b. Only weak versions of PSL operators are supported. The strong versions would require modification of the THLDD standard interface and several PPGs).

   c. PPGs for some PSL FL operators (e.g. *abort*) are not developed yet. However, this fact is not due to a principal constraint of the proposed approach.

   The subset of supported operators is implicitly defined by the PPG Library.

```
a)
   a1: assert always (a -> (b or c));
b)
   p2: always (a -> (b or c));
   a2: assert p2;
c)
   p3: (b or c);
   a3: assert always (a -> p3);
```

Figure 3.11. Possible combinations for the verification directive *assert* usage

In case if the above set of rule is satisfied the parser will partition the property under conversion into entities containing one operator only. Further the hierarchical set of the entities operators is passed to the constructor. The operands of the operators can be:

- primary inputs and outputs
- internal signals (variables)
- other operators

The precedence of the operators in the hierarchy is kept in accordance with [92] (see Figure 2.17) and their order of appearance. The order is passed to the parser from the PPG Library file *ppg.lib* where it is explicitly specified for the supported (described in this file) set of the operators. The details are described in Section A.1 of Appendix A.

3.3.3 Constructor

Complex properties are hierarchically constructed from elementary graphs in PPG Library in the top-down manner. The process of construction starts from the operators with the lowest precedence forming the top level. Then their operands that are sub-operators with higher precedence recursively form lower levels of the

complex property. For example, *always* and *never* operators have the lowest level of precedence and consequently their corresponding PPGs are put to the highest level in the hierarchy. The sub-properties (operands) are step-by-step substituted by lower level PPGs until the lowest level, where sub-properties are pure signals or HDL operations.

As it was mentioned earlier, the verification directive *assert* is assumed for all the properties in the current approach. Therefore, it does not have any reflection in the final property under conversion graphical portrayal or its code in the AGM format. Please note that this directive (as well as others e.g. *assume*) is not a PSL operator and therefore it cannot have a PPG. In the future work different verification directives can be represented as auxiliary suffixes to the properties names to pass this information to the THLDD processing tool (e.g. HLDD*sim*).

Let us consider an example PSL assertion *gcd_ready* for the *GCD1* design (see Figure 2.10) provided in Figure 3.12.

```
gcd_ready: assert always((not ready) and (a=b) -> next_e[1 to 3](ready));
```

Figure 3.12. An example PSL assertion *P* for DUV *GCD*

The step-by-step construction of this complex property is presented in Figures 3.13 and 3.14 (in the intermediate steps the *gcd_ready* property is denoted by *P*). The process consists of 6 steps starting from the lowest level precedence operator *always* of the *gcd_ready* (denoted by *P*) property (Figure 3.13a). In this step the remaining part of the PSL property under construction, put into the brackets of this operator, is considered as its operand P_1. The PPG corresponding to *always* is taken from the PPG Library and put as the starting THLDD of the property. The figure contains the portrayal of this THLDD for illustration, while the tool itself "thinks" in terms of AGM format for THLDD representation. In the second step (Figure 3.13b) the operand of *always* P_1 in the THLDD is substituted by the PPG for logical implication -> with two operands P_2 and P_3. The process continues until the Step 6 (Figure 3.14c) where all the sub-properties P_5, P_6, P_7 from Step 5 (Figure 3.14b) that are operands for the upper level operators are whether pure signals (e.g. *ready*) or HDL expressions (e.g. *a=b*).

Please note that the Step 6 includes also elimination of a number of redundant edges outgoing from terminal nodes CHECKING of P_5, P_6, P_7 from Step 5. These sub-properties have been considered as temporal (aka *TOP* in PPG Library terminology, see Appendix A) in Step 5 and therefore had the third terminal node. After their substitution by Boolean and bit type operands (aka *BOP* in PPG Library terminology, Appendix A) their terminal nodes CHECKING are eliminated.

Figure 3.14c shows the final THLDD representation for the example complex PSL property *P* given in Figure 3.12.

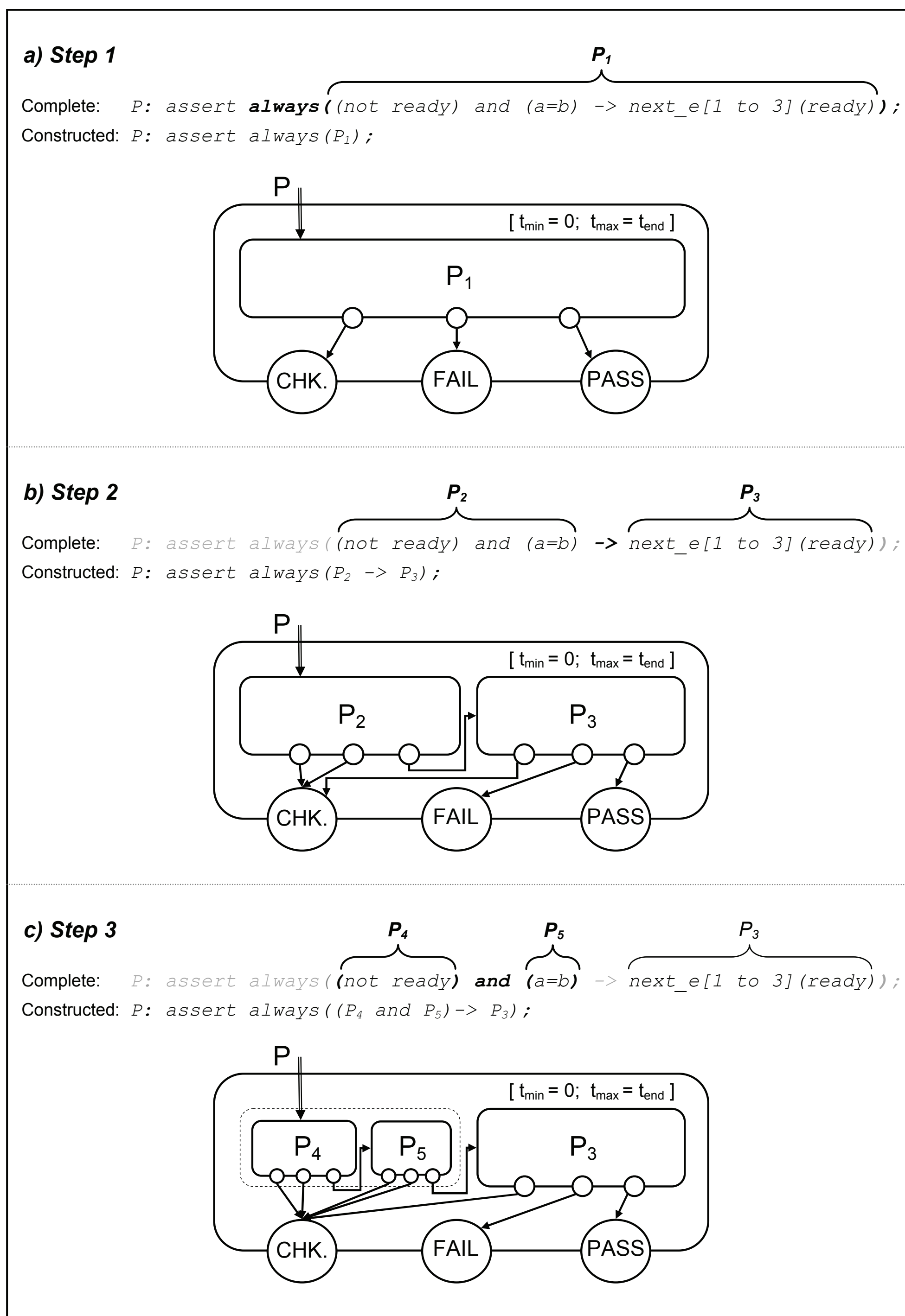


Figure 3.13. A THLDD property construction process
(continued in Figure 3.14)

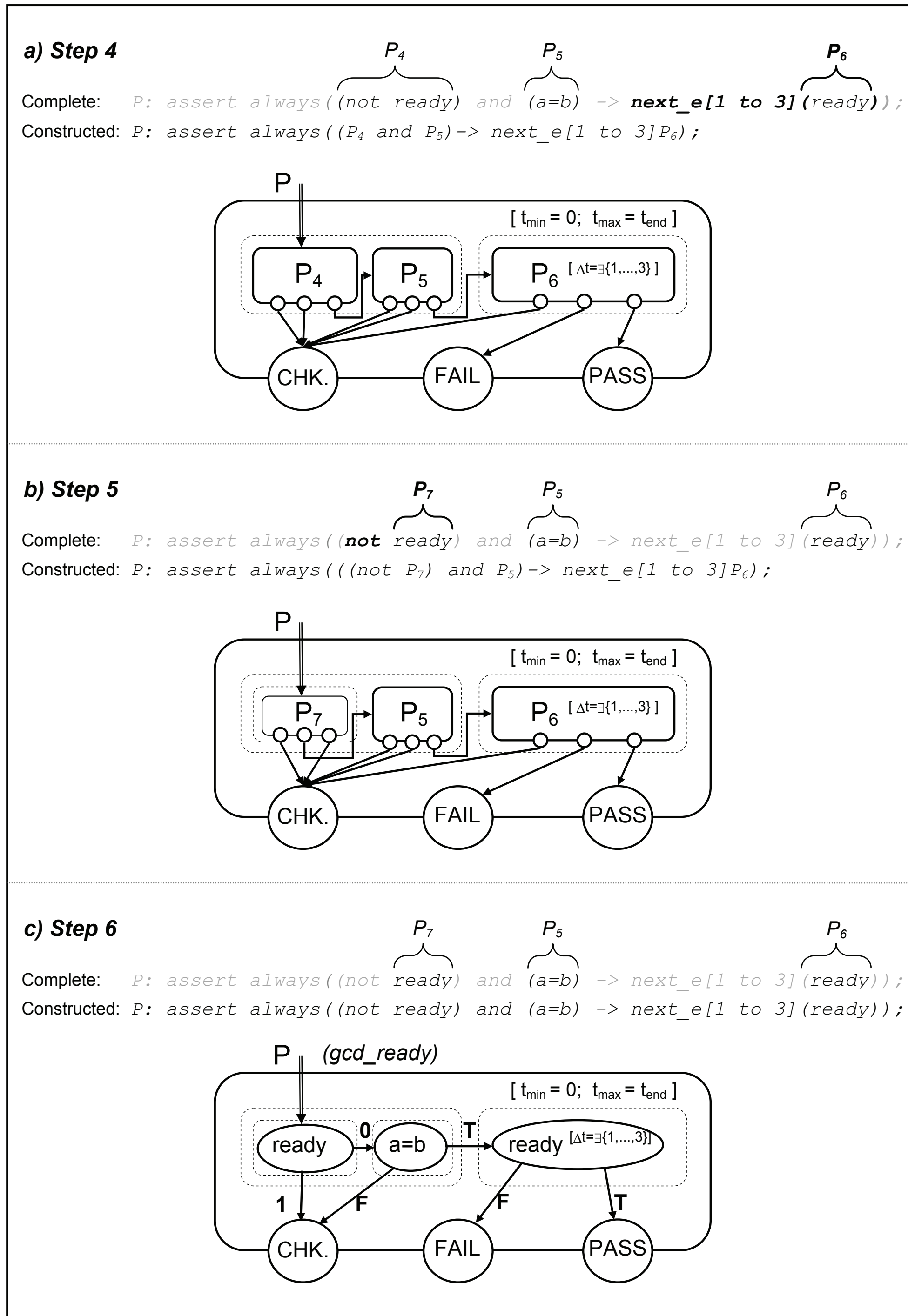


Figure 3.14. A THLDD property construction process (continued from Figure 3.13)

The presented PSL to THLDD conversion method supports PSL files with a set of PSL assertions and pure properties. The properties are allowed to have hierarchical dependencies and multi-layered properties for verification layer expression assistance. This topic is discussed in more details in the next subsection. The resulting THLDD properties are stored in AGM format file. This file serves as an input for HLDD based assertions checking tool presented in the next section.

3.3.4 Representation types of THLDD properties

The THLDD properties may have the following three types of representation:

- Flattened
- Partially flattened
- System-based

The first type assumes the whole property to be represented by a single graph. An example THLDD property given in Figure 3.14c is of this type. *Flattened* representation is the most optimized for checking by HLDD*sim*^{assertions}, because it reduces the number of nested calls and therefore reduces the checking time. However, not all the generally supported PSL properties can have this type of THLDD representation. In case if a basic Boolean sub-property within the given property of interest has a complex activity time window (i.a. not a single range), then the property can have only partially flattened or system-based representations.

The second type of representation was introduced to overcome the constraints of the first type. *Partially flattened* representation of a property is a system of graphs, where several temporal sub-properties are detached to separate graphs in order to keep all the sub-properties activity windows simple (describable by a single continuous range). *Optimal* partially flattened representation of a property has the minimal number of graphs capable to express the complex system of time windows in THLDD. The number of separate graphs in the optimal partially flattened representation is less than or equal to the number of temporal operators in the property. An example of a PSL property that requires partially flattened (or system-based) representation is given in Figure 3.15. The detailed discussion of the complex and simple time windows is given in the next subsection.

```
complex_tw_1: assert always (A -> next_a[3 to 7](next_e[1 to 4](B)) );
```

Figure 3.15. An example of a property with a complex activity time window

System-based representation of a property is the system of PPGs corresponding to the property's operators. The number of graphs in such representation is always equal to the number of all operators (must be listed in PPG Library and not treated as a HDL Boolean expression or otherwise) of the property. This type of representation is maximal and may cause HLDD*sim*^{assertions} non-optimal

performance in terms of checking time due to higher number of nested calls. However, the main benefit of this type of representation is its flexibility and reduction of principal constraints for an arbitrary property representation by THLDD. System-based represented property is also more flexible in terms of simple modifications for its reusability within verification plan. It means, a similar property differing by a simple part (please consider the example in Figure 3.16) can be derived from the original one by touching only one THLDD sub-graph in the system leaving the rest of its sub-graphs and links as they were. An implicit argument for system-based THLDD representation type usage is the design representation model HLDD system-based nature.

```
P1: assert always (A -> (next_a[3 to 7](B)) or (next(C)));

P2: assert always (A -> (next_a[3 to 7](D)) or (next(C)));
P3: assert always (A -> (next_a[3 to 8](B)) or (next(C)));
```

P2 and P3 are similar to P1, and can be derived by modifying only one graph in the system.

Figure 3.16. Similarity in properties

The system-based structure is also used for *multi-layered* THLDD properties. They are applied for the Verification and Modelling layers of PSL and can be utilized for modelling of ABV plan or its parts. The upper-layer THLDD properties are very useful for properties internal reuse management – an important part of a real life verification process configuration.

```
low_prop1: assert always (A -> (next_a[3 to 7](B)) or (next(C)));
low_prop2: assert always (D -> ((E) until (F));
low_prop3: assert always (C -> (next[2](not (C))));
conf1: low_prop1 and low_prop2;
conf2: low_prop2 and low_prop3;
```

Figure 3.17. An example of upper-layer properties application

An example of upper-layer property usage is given in Figure 3.17. Here assertions *low_prop1*, *low_prop2*, *low_prop3* are used in two different configurations of an ABV plan. The major shortcoming for the upper-layer properties flexibility is the constraints of PSL simple subset (see Subsection 2.2.3). For example, some of them are the restriction for the negation operator *not* operand to be Boolean and the restriction for the logical *or* operator at least one of the operands to be Boolean. These restrictions prohibit convenient combinations of temporal lower-layer properties in upper-layer ones.

Figure 3.18 shows system-based representation of the property *gcd_ready* (Figure 3.12) for GCD1 design (Figure 2.10).

```
gcd_ready: assert always((not ready) and (a=b) -> next_e[1 to 3](ready));
```

The THLDD flattened representation for this property was shown in Figure 3.14c.

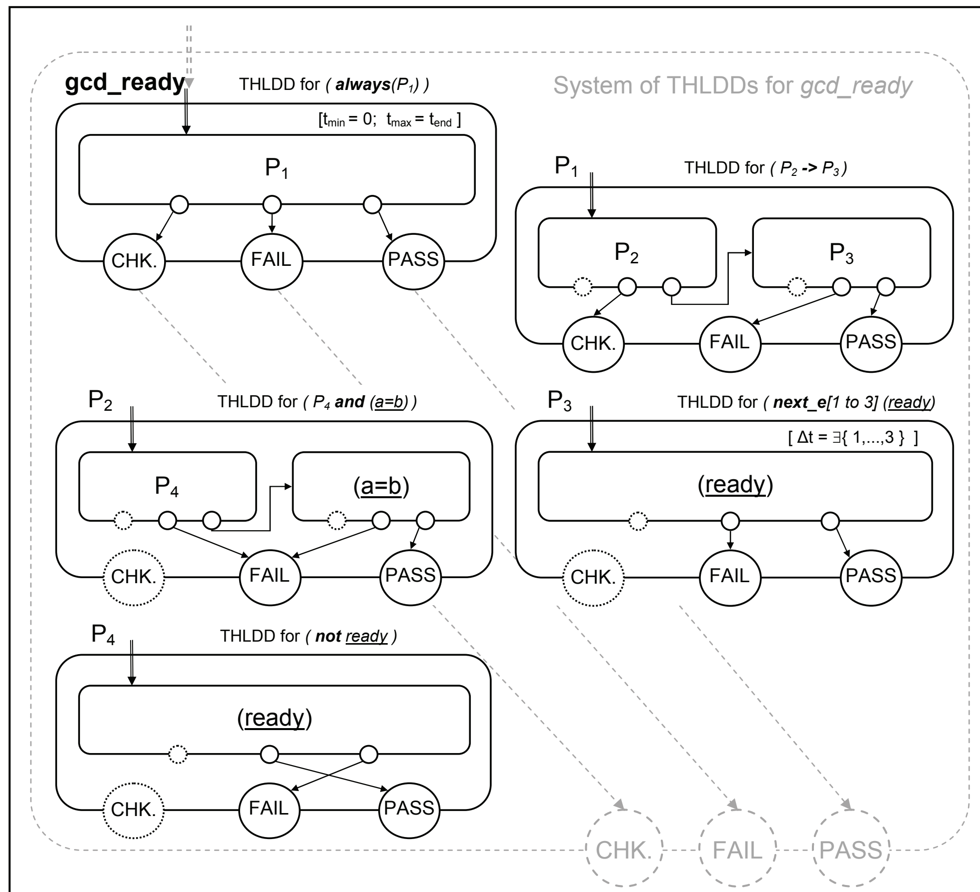


Figure 3.18. A system-based type representation of the property *gcd_ready*

The process of creation of a flattened type THLDD property was described in details in this section. The process of creation of partially-flattened and system-based types properties is very similar. The difference is (partial) replacement of, the direct substitution of operands by lower-level sub-properties' PPGs, by creation of links (i.e. calls) between them. In frames of this thesis by default we consider optimal partially flattened representation type for THLDD, if it is not stated otherwise.

3.4 The method for assertions checking with HLDD*sim*

This section presents a method for HLDD-based assertions checking. First the existing HLDD design simulator HLDD*sim* is discussed. Further its complementary modification for assertions checking support is presented. The section also presents the general flow of HLDD-based assertion checking process and discusses in details the THLDDs' checking timing issues.

3.4.1 HLDD*sim* algorithm

The basis for assertion checking proposed in this thesis is the HLDD model simulator (HLDD*sim*) engine. An algorithm for it (Algorithm 1) is presented in Figure 3.19. It supports both behavioural and RTL (pure RTL and behavioural RTL, see clauses 2.1.2.3 and 2.1.2.4) design abstraction levels and has been proposed in [45]. This algorithm is briefly explained below and it is used for DUV simulation. The following description uses HLDD model data structure notations provided in Clause 2.1.2.1.

In the RTL style, the algorithm takes the previous time step value of variable x_j labelling 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 behavioural HDL coding style HLDDs are generated and ranked in a specific order to ensure causality. For variables x_j labelling 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.

```
SimulateHLDD()
  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  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
        For {Γ | Value ∈ Γ(e_active), e_active =( m_Current, m_Next)}
           m_Current = m_Next
        End if
     End while
     Assign x_G = x_Current
  End for
End SimulateHLDD
```

Figure 3.19. Algorithm 1. RTL/behavioural simulation on HLDDs

3.4.2 HLDD*sim* modification for assertions checking

The support of assertion checking in HLDD*sim* implies an extra step added on top of the existing functionality.

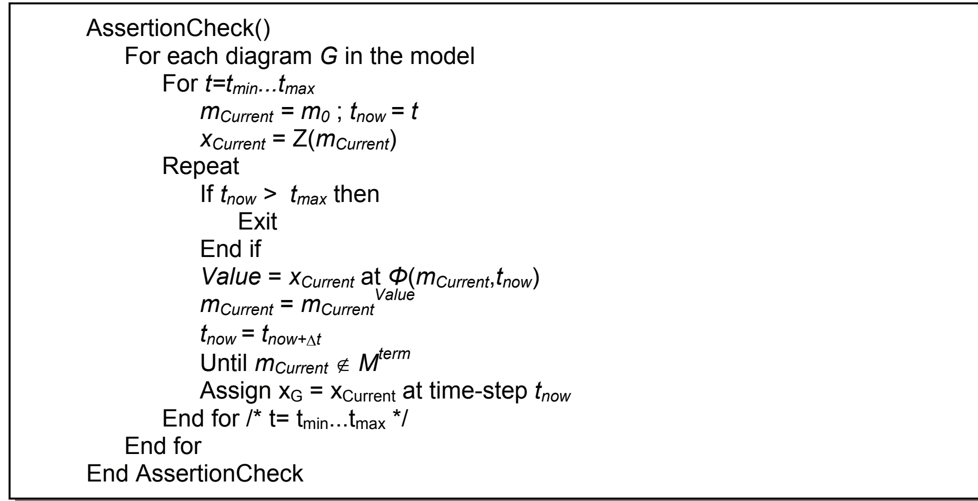

```
AssertionCheck()
    For each diagram G in the model
        For t=t_min...t_max
            m_Current = m_0 ; t_now = t
            x_Current = Z(m_Current)
        Repeat
            If t_now >  t_max then
                Exit
            End if
            Value = x_Current at Φ(m_Current,t_now)
            m_Current = m_Current^Value
            t_now = t_now+Δt
            Until m_Current ∉ M^term
            Assign x_G = x_Current at time-step t_now
        End for /* t= t_min...t_max */
    End for
End AssertionCheck
```

Figure 3.20. Algorithm 2. Assertion checking based on THLDDs

This step is preceded by executing Algorithm 1 (Figure 3.19) from the previous subsection which calculates the *simulation trace* (i.e. values of variables at the HLDD nodes during the simulation time). This trace is a starting point for assertion checking. This step is formally explained by Algorithm 2 in Figure 3.20. It takes into account temporal information at the nodes and has an exit condition in order to avoid eternal loops that are due to the cyclic nature of the general case of THLDDs.

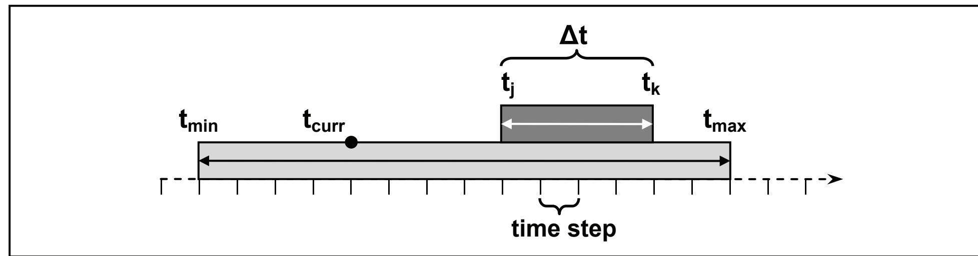


Figure 3.21. THLDD time windows in assertion checking

Figure 3.21 shows an example of time windows for a THLDD graph converted from a two-operator PSL assertion `two_win: assert always(next_a(j to k)(x))`. Here the light-gray time window limited by t_{min} and t_{max} belongs to *always*. The dark-gray time window belongs to *next_a*. It is dynamic (moving along the time axe), denoted by $\Delta t = \forall\{j,...,k\}$, with size $t_k - t_j$ and relative to t_{curr}, which is the current position in time. Normally, depending on its complexity, a THLDD has one

static and several dynamic time windows that can overlap. The next subsection (Subsection 3.4.3) provides a discussion on time window dependencies and possible transformations.

A general flow of the HLDD-based assertion checking process is given in Figure 3.22. The input data for the first step (simulation) are HLDD model representation of the design under verification and stimuli. This step results in simulation trace stored in a text file. The second step (checking) uses this data as well as the set of THLDD assertions as input. The output of the second step is the assertions checking results that include both information about the assertions coverage and validity.

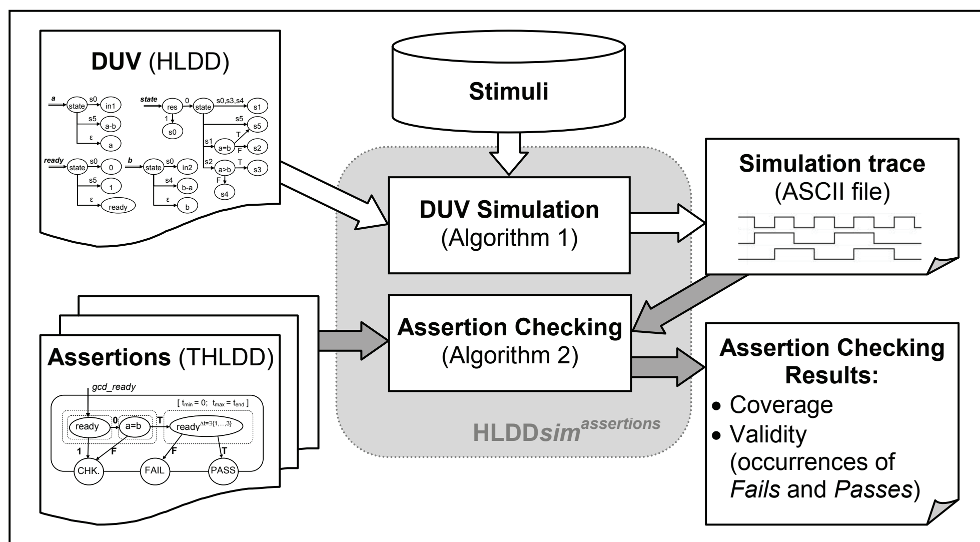


Figure 3.22. HLDD-based assertion checking process flow

The validity state (i.e. *CHECKING* or *FAIL* or *PASS*) of the monitored assertions is stored for every time step. That allows further analyzing which combinations of stimuli and DUV states have caused assertions violations and passes. This data also implicitly contains information about the monitored assertions coverage (i.e. assertion activity: *active* or *inactive*) by the given stimuli (testbench). A lower than expected assertions coverage may warn about insufficient stimuli.

The process of DUV assertions-based verification with the HLDD-based approaches presented in this chapter is performed according to the common ABV flow.

3.4.3 THLDD assertions checking timing issues

As it was stated in Subsection 3.2.3 the temporal extension for HLDD model proposed in this thesis supports 2 basic and a number of derivative temporal PSL constructs (please refer to Subsections 3.2.1 and 3.2.3). A wide set of complex temporal relationships is obtainable by means of this main constructs and special values for the upper bounds of the time range (i.e. $event(x_c)$ and t_{end}). In this subsection we will discuss in more detail temporal relationships in THLDD graphs and especially their treatment during assertions checking by HLDD*sim*.

PSL operator *next_e* has a constraint application determined by the rules of PSL simple subset (see Subsection 2.2.3). Namely it is not allowed to have a temporal operand. Operator *next_a* does not have such a constraint and can embrace both *next_a* and *next_e* temporal operators as its operands.

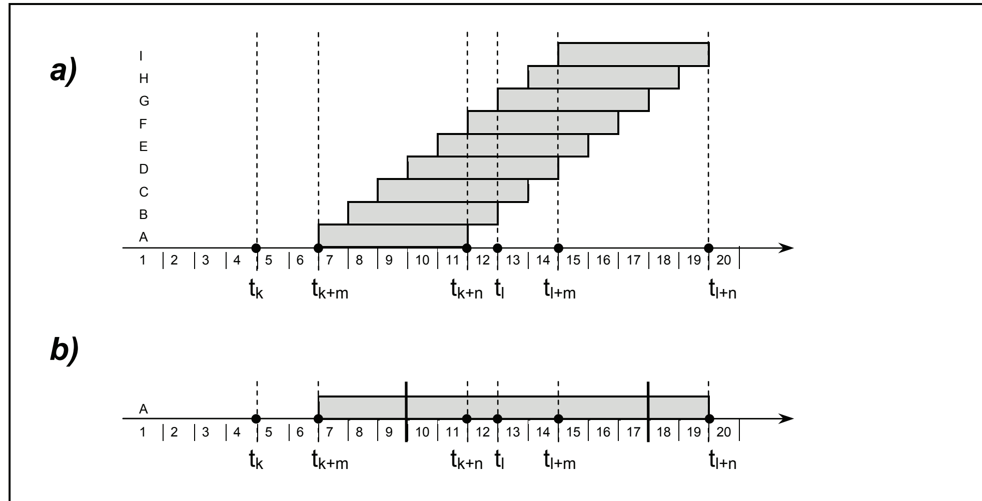


Figure 3.23. Overlapping of time windows for *next_a* + *next_a* combination

Let us consider two possible cases of *next_a* and n*ext_e* combinations of embracement depth equal to 2.

The first case when operator *next_a* embraces another instance of *next_a* is shown in Figure 3.23a. Here a PSL property `win_P1: next_a[k to l](next_a[m to n](x))` (where $(k{:=}4) < (l{:=}12)$, $(m{:=}2) < (n{:=}7)$ are integers and x is a Boolean property) is considered. The first dynamic time window is relative to the time step *0* (it does not float further because there is no PSL invariance operator *always*), it starts at time step k and lasts till time step l. The second dynamic time window is dynamically relative to the time steps in the range from k to l, starts at the mth and lasts till the nth time step from the reference point. The gray rectangles (*A-H*) in Figure 3.23a denote the set of these time windows. The property *win_P1*

will be satisfied only in case if the Boolean property x will hold at all time steps in the range $\{t_{k+m}, \ldots, t_{l+n}\}$. Some of the time windows *A-I* overlap, for example at time step t_l the validity of sub-property x is checked 5 times, i.e. within time windows *B*, *C*, *D*, *E*, *F*. The property *P1* can be substituted by functionally equivalent `win_P2: next_a((k+m) to (l+n))(x).` Figure 3.23b shows the single time window of property *win_P2*. A set of dynamic time windows of property *win_P1* can be represented by a single time window and therefore this property can have a flattened representation. However, the proposed substitution may lead to a certain loose of potential debug information, i.e. the information about in which of the overlapping time windows in particular the violation has happened.

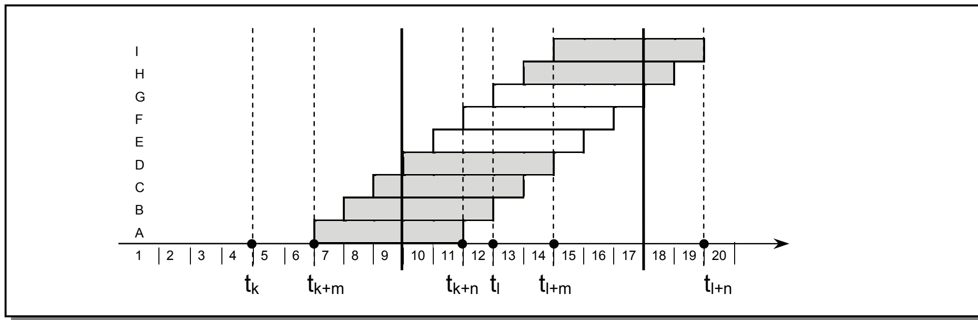


Figure 3.24. Overlapping of time windows for *next_a* + *next_e* combination

The second case is when operator *next_a* embraces operator *next_e* is shown in Figure 3.24. Here a PSL property `win_P3: next_a[k to l](next_e[m to n](x))` (where $(k{:=}4) < (l{:=}12)$, $(m{:=}2) < (n{:=}7)$ are integers and x is a Boolean property) is considered. This property also has 9 final time windows *A-I*, however, they cannot be substituted by a single time window or at least a less number of time windows. In case if the sub-property x holds at two time steps 10th and 18th marked by bold lines in figure, then *win_P3* will hold only for the time windows A-D, H-I and violate for 3 time windows E, F, G. *win_P3* cannot have a flattened representation, because it has more than one dynamic time window. Therefore its minimal representation is a system of 2 THLDD graphs (optimal partially flattened). If the Boolean sub-property x (which is an operand in the considered above properties) is complex, then it can cause additional THLDD graphs for system-based representation.

PSL temporal operator *always* can be considered as a special case of the first basic temporal construct from Figure 3.9 temporal operator *next_a*. Namely, if we use the notion of t_{end} from Subsection 3.2.3, then for a Boolean property x:

```
always(x) ≡ next_a(0 to 'end')(x)
```

Therefore, a static time window of a TLHLDD property [t_{min} ... t_{max}] defined by *always* can also be considered as a special case of a dynamic window $\Delta t = \forall\{0,...,t_{end}\}$ relative to the starting point of the simulation trace t_0.

In practice, the majority of PSL assertions start with the invariance operator *always*. The external window was introduced for the practical simplification purposes. In some sense, PSL can be considered as a "sugaring" for the formal logics LTL and CTL. An optimal THLDD-based temporal property representation would require initially their optimal formal representation in LTL. However, the proposed in this thesis approach is PSL oriented.

The examples from this subsection demonstrate the necessity of THLDD properties with complex temporal relationships to be represented by more than one graph.

3.5 Experimental results

This section provides experimental results [10][co-auth.] of assertion checking execution times comparison between the proposed HLDD$sim^{assertions}$ simulator and a state-of-the-art commercial tool from a major CAD vendor.

The experiments were performed with 4 experimental benchmarks. The first one is *gcd* design from the HLSynth92 benchmarks family [103]. Its VHDL and HLDD representations were provided in Figure 2.10 (Clause 2.1.2.4). The remaining designs are two from ITC'99 benchmarks family [76],[102],[105] and one created in University of Verona.

Design	Characteristic, number				
	VHDL lines	inputs	outputs	signals	HLDD nodes
gcd	75	4	1	8	25
b00	76	4	2	7	37
b04	84	6	1	14	58
b09	102	4	1	9	44

Figure 3.25. Benchmark characteristics table

The characteristics of the benchmarks are provided in Figure 3.25 and their functionality is described below:

- *gcd* is a design implementing functionality of a greatest common divisor.
- *b04* is a design implementing functionality to compute minimum and maximum.
- b09 is a design implementing functionality of serial to serial converter.

- *b00* is a benchmark created in University of Verona. It is specially designed to contain hard-to-test branches and addresses their testability / verifiability analysis problems. The design contains conditional statements where one branch has probability ($1 - 1/2^{32}$)of being satisfied, while the other has probability ($1/2^{32}$).

A set of 5 realistic assertions has been created for each benchmark. The assertions selected for GCD1 are the following:

```
p1: assert always(((not ready) and (a = b)) -> next_e[1 to 3](ready));
p2: assert always (reset -> next next((not ready) until (a = b)));
p3: assert never ((a /= b) and ready);
p4: assert never ((a /= b) and (not ready));
p5: assert always( reset -> next_a[2 to 5](not ready));
```

The assertion *p1* has been discussed in the previous sections as *gcd_ready*. Its THLDD representations of different types were provided in Figures 3.14c and 3.22. THLDD properties of the optimal partially flattened representation type were considered in the current experimental setup.

Design	Stimuli Length (clocks)	The proposed approach (HLDD*sim*)			Commercial tool
		Simulation Time (seconds)	Checking Time (seconds)	Total Time (seconds)	Total Time (seconds)
gcd2	10,000	0.02	0.04	**0.06**	0.67
	100,000	0.20	0.40	**0.60**	1.71
	1,000,000	2.07	4.87	**6.94**	13.52
b00	10,000	0.03	0.03	**0.06**	0.79
	100,000	0.30	0.30	**0.60**	1.83
	1,000,000	3.43	2.95	**6.38**	13.84
b04	10,000	0.05	0.03	**0.08**	0.84
	100,000	0.54	0.28	**0.82**	2.21
	1,000,000	5.47	3.61	**9.08**	19.23
b09	10,000	0.02	0.04	**0.06**	0.72
	100,000	0.22	0.39	**0.61**	1.74
	1,000,000	2.21	4.55	**6.76**	12.4

Figure 3.26. Table of assertion checking execution time comparison

Figure 3.26 shows the results of the experiments. Both simulators were supplied with the same sequences of realistic stimuli providing a good coverage for the assertions. (The stimuli were pre-generated by the appropriate testbenches of the DUVs). The test lengths are shown in the second column of the table. The third and fourth columns show the simulation (Algorithm 1, see Section 3.4.1) and assertion checking (Algorithm 2, see Section 3.4.2) execution times required for the HLDD$sim^{assertions}$. The fifth (highlighted) and the sixth columns are the total execution time taken by the proposed approach and the commercial tool,

respectively. The values in the sixth column include approximately 0.5 sec of simulation initialization time for the commercial tool that was impossible to exclude from the measurement.

The both tools have shown the identical responses about the assertion satisfactions and violations. Though minor differences in the PSL assertions interpretations are possible for different tools, the compared tools interpret PSL in identical way. (An example of such difference was provided in [78]).

The experimental results show the feasibility of the proposed approach and a significant speed-up (2 times) in the execution time required for design simulation with assertion checking by the proposed approach compared to state-of-the-art commercial tool.

3.6 Verification assertions reuse for manufacturing testing

In this section we would like to depart from the main topic of the thesis, which is HLDD-based approaches for simulation-based verification. Here we propose ([17],[23])[co-auth.] directions for the discussed in this chapter verification assertions reuse for *manufacturing test* ([19]-[22],[25] and [26]-[33])[co-auth.] development to enhance its quality.

The verification assertions contain valuable knowledge and sometimes "insider" information about the design's functionality and implementation. Normally they are cleaned out after the verification phase in the design's synthesizable description and the information is lost. The proposed directions for assertions reuse are *Test Pattern Generation* (TPG), embedded *Built-In Self-Test* (BIST) observability improvement and *Design for Testability* (DfT) enhancement. Figure 3.28 repeats the typical design development flow presented in Figure 3.1 and shows the main directions for assertions reuse.

Several approaches for a variation of BIST for combinational circuits called *Hybrid BIST* (the stimuli consists of both pseudorandom and deterministic test patterns) have been proposed in ([28] - [35])[co-auth.]. Some approaches for manufacturing test TPG proposed in ([21],[22])[co-auth.] consider fault models which represent *physical defects* behaviour more accurately than the traditional stuck-at fault model.

The approaches presented in ([26],[27])[co-auth.] consider test pattern generation for sequential circuits based on design's properties, however the properties themselves were not automatically obtained and required comprehensive study of the design's under test functionality. In this section we propose to extract this information partially from verification assertions and assumptions.

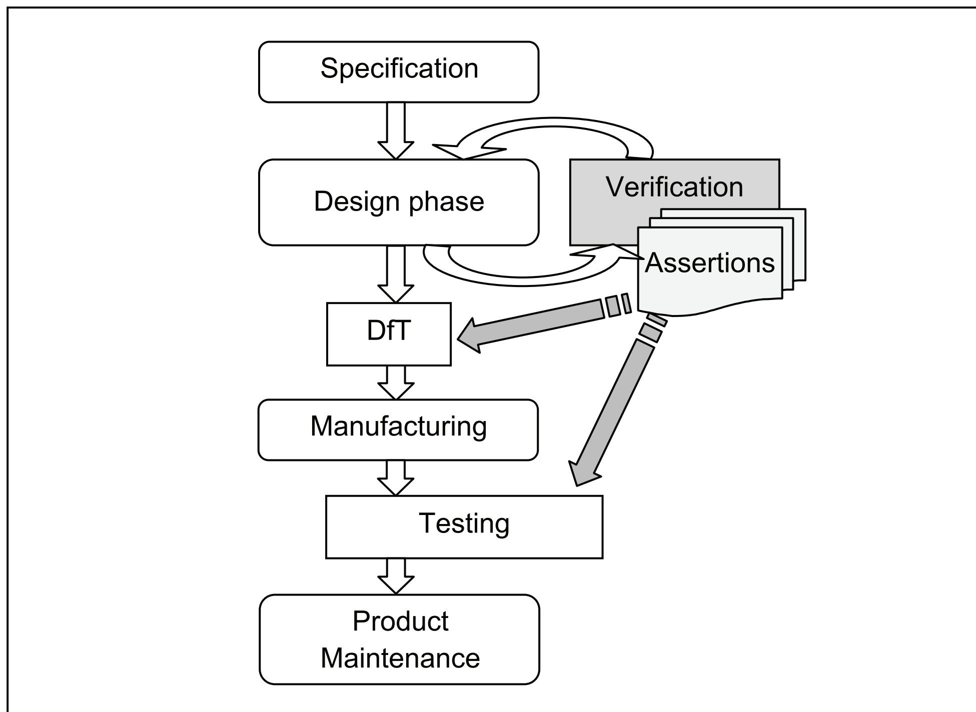


Figure 3.28. Design development flow (simplified) with assertions reuse

As it has been mentioned in Subsection 3.1.4, there are a number of approaches proposed for aiming hardware checkers creation from assertions meant for prototype verification/validation by emulation (a development flow phase standing before mass-production). A few research groups propose to use verification assertions for manufacturing testing. However they have several limitations and focused only on test pattern generation by the synthesized checkers. The approaches do not consider other aspects of test plan development and usually lead to large area overhead. In this section we propose a wider range of manufacturing test plan development areas where assertions can be reused.

3.6.1 Assumption-based test generation

A part of today manufactured ASICs contains scan-chains embedded during DfT that increase observability and controllability of the design by providing an access to the internal registers. This method allows reaching high fault coverage, however it results in over-testing of the core (see e.g. [79]). In other words, it covers faults that could never influence the functional behaviour of the circuit thus reducing the yield. On the other hand, it has been shown that non-scan testing based on pseudorandom test sequences can be highly feasible for several types of designs (such as crypto cores as it was proven in [80]). Finally, not all the circuits

may have the embedded scan. The ideas proposed in this section consider non-scan sequential circuits. However they can be partially applied to scan circuits as well.

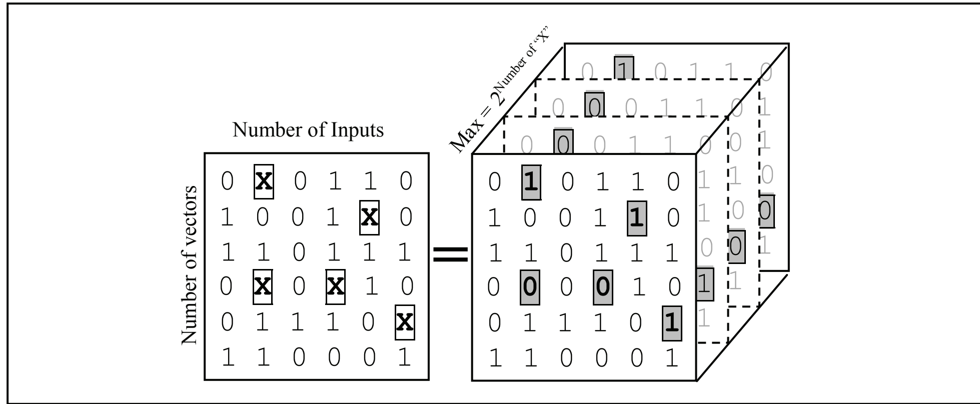


Figure 3.29. Test cube

The notion of test cubes is applied for test set representation when several bits in the test vectors contain unassigned values *X* (see Figure 3.29). These bits can be assigned to the normal binary values *'1'* or *'0'* whether based on some rules or just randomly. The *X*-s add the 3rd dimension to the fully determined 2-dimensional test set.

In case if the test engineer has no prior information about the design's input data dependencies (assumptions) he has to start with a test set filled with all *X*-s (an empty set). The assumptions provide information how to assign a part of the bits in the test cube which increases the final test quality and eases the test generation process, because the assigned bits reduce the remaining search space exponentially. For example if an assumption provides us information that some particular input (pin) of the design is its *RESET* signal, then the test engineer may decide to set its value to *'1'* only once in some sequence of cycles. The assumptions may be much more complex and origin from a design specification and its implementation (design) phase.

As it is discussed in Subsection 2.2.2, both assumptions (sometimes referred also as environmental constraints) and assertions are design properties. The first ones contain information about the design's environment and therefore its expected inputs dependencies while the second ones describe the dependencies of the design's internal signals and outputs [4]. The assumptions can be described explicitly by the designer or implicitly follow from some of the given assertions of the connected designs or design cores. The latter case is shown in Figure 3.30 and known as assumption-assertion dualism [6]. This method allows obtaining a wider set of assumptions.

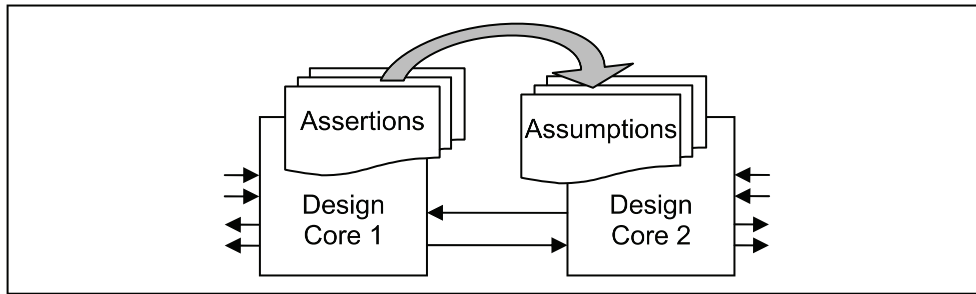


Figure 3.30. Assumption-assertion dualism

The final set of the assumptions can be divided in two groups:

- The first group is the assumptions *explicitly assigning a set of bits* in the test cube. The rest of the undetermined bits should be assigned by Pseudorandom or Genetic TPG. Deterministic TPG may be not reasonably efficient in this case.
- The second group representing a set of *more complex rules* not suitable for straightforward bits assignment is meant for an appropriate Deterministic TPG constraints representation. The both groups of the assumptions should be utilized for the maximum efficiency.

The main difference of the proposed approach from the one described in ([26],[27])[co-auth.] is the formal source and format of the assumptions.

3.6.2 Assertion-based BIST

Another possible application of the information provided by verification assertions is *Built-In Self-Test* (BIST) ([26]-[33])[co-auth.] response analyzer observability enhancement. The BIST has been proven to be an efficient approach for manufacturing testing. However in case of non-scan circuits its main bottleneck is observability. The mentioned in the state-of-the-art approaches for hardware checkers obtained from assertions may find their application in online testing. However, they usually consider TPG to be performed offline. We propose to use these checkers for BIST observability enhancement. Here also two approaches are possible:

- Assertion-based checkers act as *separate observers* in BIST response analyzer architecture.
- Assertions *aid building complex controller-based BIST observer*. For example, one of the task of which would be to inform the analyzer when to check and when to stay in “Silence Mode”.

The main issue here is the coverage of observability by the existing assertions. Therefore, the efficiency of this technique should be very much dependable on the particular DUV and the designer's assertions choice style.

3.6.3 Assertion-based DfT by test points insertion

The two main components of design's testability are observability and controllability. The both of them can be significantly improved during DfT phase. These improvements usually include minor rearrangements inside the design or addition of an extra logic (for example scan-chains as it was mentioned in Subsection 3.6.1).

One of the DfT techniques is *test points insertion*, when an additional auxiliary input or output pin is routed to an internal net of the design. The main drawback of this technique is hardware area- and input/output pins overhead. The second ones are very costly and increase the significance of this drawback. Therefore, only the nets providing good controllability coverage should be chosen. The identification of such nets is a very complex task, usually solved by heuristic approaches.

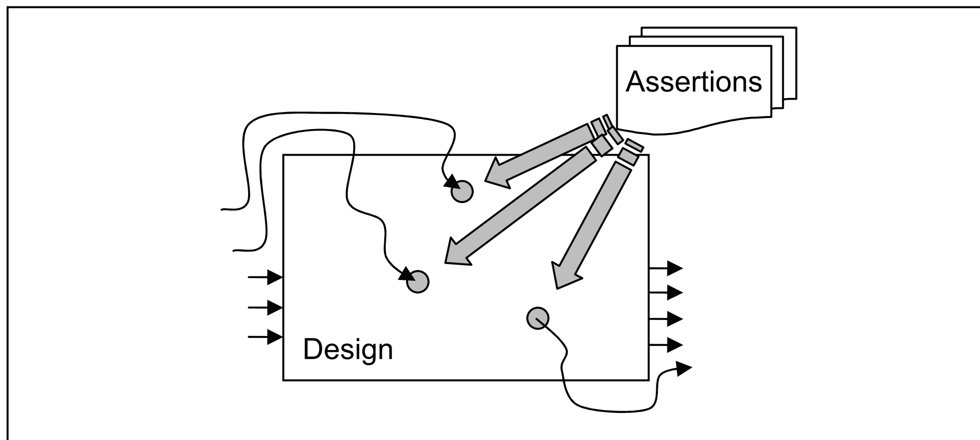


Figure 3.31. Assertions for test points insertion

We propose to use for test points locations the nets corresponding to the signals (operands) from assertions (Figure 3.31). The good candidates for test points' locations are the nets with hard-to-test faults, due to very low controllability. The other intent for test point insertion would be a fan-out that provides at once a good controllability to a set of middle-level controllability places. Such a test point would shorten the test length. The both criteria can be usually satisfied by normal assertions, because the assertions reflect the designer's conception of the most significant cornerstones of the design. At the same time it is necessary to note, that test points insertion to the sequential circuits may lead at some extend to the same

drawback of over-testing as in case of scan-chains insertion mentioned in Subsection 3.6.1.

In this section we have shown that verification assertions, which are normally cleaned out after the verification phase in the synthesizable description, can be reused for manufacturing testing in several ways. The potential of the first direction was proved by experimental results in [27][co-auth.], where it was shown that the proposed approach of the properties-based random BIST has dramatically improved generated test set fault coverage for non-scan design. However, further development of the approaches is required and the feasibility of the proposed directions must by proven by extensive experimental results. These tasks are scheduled for the future work.

3.7 Chapter summary

This chapter has discussed simulation-based hardware verification and proposed a *new approach for HLDD-based assertions checking*. The three main contributions of this chapter are the following.

The first one is a *temporal extension for the existing HLDD model*. The new extended model is aimed at temporal properties expression and named Temporally extended High-Level Decision Diagrams (THLDD). The extension supports a set of commonly used temporal constructs that can be used to express a wide set of possible complex temporal relationships.

The second contribution is a *methodology for direct conversion of assertions* expressed in Property Specification Language (PSL) to THLDD. The proposed hierarchical approach introduces an extendable library of Primitive Property Graphs (PPG Library). The components of this library serve as building blocks for a complex THLDD property construction.

The third contribution is HLDD-based *simulator* HLDD*sim modification* to support THLDDs and assertions checking. This part is supported by discussion of properties' activity time windows and variety of THLDD types.

The feasibility of the proposed approaches is proven by the presented experimental results.

The chapter has also briefly discussed verification assertions reuse for manufacturing testing.

Chapter 4
VERIFICATION COVERAGE ANALYSIS

Hardware verification coverage analysis is aimed to estimate quality and completeness of the performed verification. It plays a key role in simulation-based verification and aids to find an answer to the important yet sophisticated question of *when the design is verified enough*.

This chapter discusses a basic classification of verification coverage metrics and the main aspects related to their measurement. The main focus is on the structural coverage for simulation-based verification as the most widely used today in practice.

The main contribution of this chapter is approaches for HLDD model based verification coverage analysis. First, approaches for mapping commonly used verification coverage metrics to HLDD-based coverage are proposed. Further, an approach employing a hierarchical decision diagrams' model for the condition coverage measurement is presented. Finally, HLDD model manipulations for the verification coverage analysis are discussed.

The HLDD-based verification coverage analysis has a set of advantages compared to the commonly used HDL-based methods. In this chapter these advantages are discussed in detail and illustrated on a common example design. The feasibility and efficiency of the proposed approaches are supported by the presented experimental results.

4.1 Verification coverage overview

As it has been noticed in Section 3.1 there are two main approaches in design verification: *formal* and *simulation-based*. Although the notion of verification coverage is also applicable for the first one, it is a fundamental part of simulation-

based verification process. From now on we will consider the notion of verification coverage only in frames of the simulation-based verification approach.

The main purpose of *verification coverage* is to estimate how well we have verified the DUV, in other words the progress of the verification process. The three main aspects of simulation-based verification are:

- Stimuli generation
- Coverage measurement
- Response analysis

In practice, the verification process's actions aimed at these three aspects can be cycled as it is shown in Figure 4.1 (inspired by [5]).

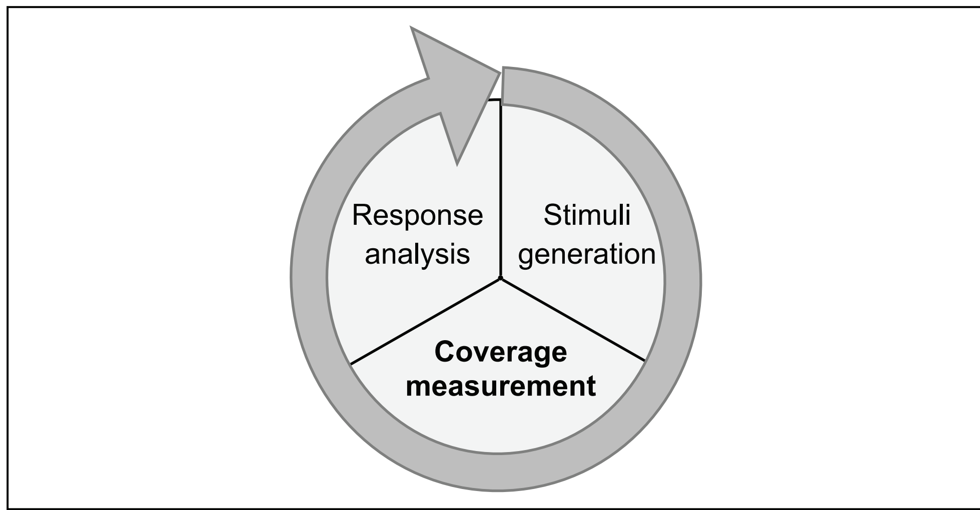


Figure 4.1. Simulation-based verification in cycle

Once the stimuli are generated the process of DUV simulation takes place. It includes verification coverage measurement (assertion checking, if it is assumed by the verification plan, takes place at this stage as well). The stage of simulation is followed by the response analysis, when the simulation results (e.g. waveform) can be compared with the responses of a reference (e.g. the design's simplified implementation or its implementation at a different abstraction level) or somehow studied for expected/unexpected behaviour.

There are three main reasons for new cycle iteration:

- Independently from the response analysis results, verification coverage value determines how thoroughly the DUV was examined. In case, if the value of verification coverage is *not sufficient* (this criterion is discussed further in Subsection 4.1.2), then the stimuli should be improved and new cycle iteration may be initiated.

- The second reason is the case, when the response analysis may have discovered an inconsistency in the responses. Further, it has been debugged and led to detection of an *implementation error* (or multiple ones) in the DUV. After the errors are corrected, new cycle iteration may be initiated.
- The third reason is an *error in the stimuli* (especially if it is a complex testbench) detected by the debug process. Then the stimuli are improved and new cycle iteration may be initiated.

A common practice in simulation-based verification is application of (pseudo[4]-) random stimuli or variation of a constraint-random (i.e. (pseudo-) randomly generated with regards to DUV's particular properties) generated on-line during the simulation. In this case "stimuli improvement" basically means simulation time increase and/or the constraints review.

4.1.1 Verification coverage classification

There are two main types of verification coverage: *functional* and *structural* (books [1],[2],[5] and a state-of-the-art commercial tool reference manual [77]). Although [1] singles out a third type *parameter coverage*, which is a metric for the variety of unit's parameters (e.g. range and depth for a FIFO) examined, we do not consider this type of verification coverage separately in this thesis.

Functional coverage is a metric for DUV's functionality exercised during its simulation. The main advantage of this type of verification coverage is that it relies on design's specification and does not depend on the particular implementation under verification. Its measurement is a sophisticated process and it is less used in practice for simulation-based verification. *Assertion coverage*, discussed in the following subsection, can be considered as a subset of functional coverage.

Structural coverage is also commonly referred as *code coverage*. Its main drawback is its quality dependence on the current implementation of the DUV. It means that, even an implementation does not include at all a part of the specified functionality, its structural coverage can still be 100%. The second drawback [81] is that while it is perfectly suitable for software verification (testing) and smaller hardware designs it may lack efficiency for covering corner cases of complex designs due to their extended concurrent functionality. However, structural coverage measurement, as opposed to functional one, is relatively easy to implement and therefore it is widely used in practice. The alternative term, i.e. code coverage, is widely used because the HDL code is a common representative of the design's structure. Although in the HLDD model-based design

[4] It is almost impossible to implement a purely random data generator. However, even available solutions with relatively good data randomness are often substituted with more deterministic pseudorandom ones. The main advantage of the latter ones is the reproducibility of the generated data.

representation the structure of a design is described by graphs, in this chapter we will still refer to structural coverage as code coverage.

Code coverage in its turn can be separated to several more narrowly defined practically used coverage metrics, such as *statement coverage*, *branch coverage*, *toggle coverage*, *(FSM) state coverage*, *data flow*, *condition coverage* and others. Section 4.2 describes in details the listed above code coverage metrics and proposes an approach for their analysis based on the HLDD model.

4.1.2 Sufficiency of verification coverage

The criterion of *sufficiency* (i.e. minimum acceptable) for hardware verification coverage metrics is very much dependant on a particular verification plan.

The methodology of code coverage measurement is very much developed in *software testing* discipline. A lot of research has been performed in this area in software testing before this topic became vital in hardware verification (due to DUVs' complexity increase). There are many similarities in the problems of code coverage measurement and some of the existing solutions can be reused from the former discipline. In the area of software testing there are known [106] several standards for software quality such as DO-178B [82] for software used in airborne systems and [83], [84]. The second one [83] recommends full statement coverage or full branch coverage, depending on the criticality of the object. Please consider Figure 4.2, where DO-178B states a sequence of coverage combinations depending on the software product target application criticality in increasing order (the terms are adapted for the hardware verification terminology).

Level	Effect of System Failure	100% statement coverage	100% branch coverage	100% condition coverage
E	*No effect*	-	-	-
D	*Minor*	-	-	-
C	*Major*	✓	-	-
B	*Hazardous*	✓	✓	-
A	*Catastrophic*	✓	✓	✓

Figure 4.2. DO-178B: minimal acceptable code coverage for software testing

There exists a standard also developed by RTCA and named DO-254 [85], which is a counterpart to the DO-178B, but aiming hardware used in avionics. None of the above-mentioned standards for both software testing and hardware verification determine a general minimal numerical threshold for a sufficient coverage. Normally, the decision which coverage value is satisfactory depends on a

particular verification plan. The coverage value is considered more informative in comparison with the values of other iterations of measurement.

The numerical value of coverage metric is a ratio of an amount of its executed units to the amount of the total units of this metric. The *hits* of the units' executions are usually counted and a particular metric can contain a requirement for a *threshold* number of hits for the units to be counted. The approaches proposed in this chapter always consider hits counting and the threshold value equal to *1*, if it is not stated otherwise.

Verification coverage analysis allows directing the effort of stimuli generation to the crucial parts of the DUV and warns if particular stimuli improvement activities do not give any increase in terms of verification coverage, i.e. do not actually provide for any benefit. The strategy for design verification process evaluation based on the verification coverage measurement results is known as *Coverage-Driven Verification* (CDV) [5].

4.1.3 Assertion coverage

As it is emphasized in [5], the notion of *assertion coverage* has several meanings (similar to the notion of *assertion checking* diversity, discussed in Clause 3.1.3.1). The first option is its usage to refer to the ratio of number of assertions to the number of lines in the HDL code (or any other metric of design implementation size, e.g. number of nodes in corresponding HLDDs). An alternative term for this ration is *assertion density*. The second option is association of this notion with the amount of functionality implemented by assertions. However, in frames of this thesis we use the notion of assertion coverage for the information about the assertions evaluation, i.e. activity/inactivity and pass/fail times.

The assertions can be classified by their purpose of application: *checking* or *coverage*. This classification only partially correlates with division of assertions to *safety* (i.e. something should never happen) and *liveness* (i.e. something should eventually happen) ones. *Checking assertions* usually inserted to detect assertions violation, in other words to they are aimed at capturing an undesired event. At the same time, *coverage assertions* report expected behaviour. The second type of assertions can be emphasized by PSL verification directive *cover* instead of *assert*. Loosely, speaking the directives in this case only influence the way how the simulator will process the assertion evaluation results (e.g. just store to file or take an immediate action such as warning or maybe even simulation interruption).

The approach discussed in Chapter 3 and HLDD*sim* do not currently support other than *assert* PSL verification directives, and process all the assertions in the same way, i.e. store their evaluation results to a file for further analysis and debug process. However, the classification of the assertions into checking and coverage ones is important for coverage-driven verification strategy. The improvement of stimuli can be motivated not only by insufficient code coverage, but also by

insufficient assertion coverage. For the second case, the coverage of *coverage assertions* is more important than coverage of *checking assertions*. The situation when some of the latter ones were not activated may only signal about the correctness of the DUV.

In assertions-based verification the assertion coverage measurement may be used as addition to or instead of the structural verification coverage measurement stage. However, it is necessary to keep in mind that if an error is discovered and the DUV was corrected before another iteration of the cycle (Figure 4.1), the assertions should be re-examined beforehand. Unlike the verification coverage constituents that are parts of the implementation and that have been directly changed with the DUV's modification, the assertions may require a separate effort for their explicit modification. For example, the modified DUV could have eliminated a signal monitored in some of its assertions.

The assertion coverage can be considered as a subset of functional coverage. Its measurement implementation referred as HLDD-based assertion checking was discussed in details in the previous chapter. The rest of this chapter will consider only code coverage part of the verification coverage.

4.2 HLDD-based analysis of code coverage

In this section six traditional code coverage metrics mentioned in Subsection 4.1.1, i.e. *statement coverage*, *branch coverage*, *toggle coverage*, *state coverage*, *data flow coverage* and *condition coverage,* are proposed to be analyzed based on the HLDD model. The proposed analysis ([15],[16])[co-auth.] is more efficient compared to the standard HDL approaches due to the nature of the HLDD model. Once a correct mapping of coverage metric to HLDDs is created, the coverage measurement overhead during design simulation is significantly reduced.

Let us consider a common example design *CovEx*. Its behavioural RTL VHDL representation (only the functional segment) is provided in Figure 4.3. There are three columns with numerical values to the left from the VHDL code. The third column *Ln.* is basically the line number, while the other two are explained further.

The variables' names in *CovEx* follow the following unification rules: {*V*- an output variable; *cS* - a conditional statement; *D*- a decision; *T*- a terminal node; *C*- a condition}.Correspondingly, *cS'x'_D'y'* is the yth decision for the xth conditional statement, *V'x'_T'y'* is a yth possible value (terminal node) for the xth output (variable), etc. Please note, that the condition *cS3_C* is equal to condition *cS6_C* (they correspond to the same signal in the design), this is discussed further in Section 4.3.

The HLDD representation of the example design *CovEx* is provided in Figure 4.4. It consists of 2 graphs for the variables *V1* and *V2* correspondingly. All the nodes and edges in the HLDD representation are numbered, where the edges'

numbers are underlined. The nodes of the HLDD graphs correspond to both conditional and assignment statements, while edges correspond to decisions. The statements' keywords are emphasized by bold in the VHDL representation of *CovEx* (Figure 4.3).

4.2.1 Statement coverage mapping

The s*tatement coverage* is a ratio of statements executed during simulation to the total number of statements under the given set of stimuli ([1],[2],[5],[106],[77]).

This metric has several variations. For example *line coverage* [5], which counts lines as opposed to statements. It is more HDL coding style dependant and less accurate than the *statement coverage*, which counts several statements separately even if they are on the same line.

An observation that once a control statement is executed then a group of the following statements is usually executed as well has lead to introduction of *block coverage*. Here dividing lines for blocks are branching statements (see next subsection) and several other statements such as *wait* and *loop* [1]. Although block coverage can be consider an advanced version of statement coverage, in practice usually exactly statement coverage metric is used (e.g. [77]). In our approach, application of block coverage is more complicated then statement coverage due to the system-based representation of a DUV by HLDDs. The statements form one HDL block can be contained in different HLDD graphs and therefore may require an effort for their (virtual) grouping back.

Please consider the VHDL description of the *CovEx* design presented in Figure 4.3. Here the third column of the numbers (*Ln.*) shows the line numbers, i.e. constituents of the line coverage metric. The numbers from first column (*Stm.*) correspond to the lines with statements (both conditional and assignment). While, depending to the coding style the number of lines may vary (e.g. lines 2 and 3 may be placed to the end of the first line), the number of statements remains constant. The 3 lines from 4 to 6 can represent one "block", in case of block coverage measurement.

The statement coverage metric has a straightforward mapping to HLDD-based coverage ([15],[16])[co-auth.]. It maps directly to the ratio of nodes $m_{Current}$ traversed during the HLDD simulation presented in Algorithm 1 (Subsection 3.4.1) to the total number of the HLDD nodes in the DUV's representation. As an example, the 20 HLDD nodes of the two graphs in Figure 4.4 correspond to the 18 statements of the VHDL segment. Covering all nodes in a HLDD model (i.e. full *HLDD node coverage*) corresponds to covering all statements in the respective HDL. Please note that some of the HDL statements have duplicated representation by the HLDD nodes. This is due to the fact that in HLDD-based design representation the diagrams are normally generated for each data variable separately. For example,

Stm.	Dcn.	Ln.	A functional segment ***CovEx*** of an VHDL file.
*1***		1	**if** (cS1_C1 **or** cS1_C2)
	*1***	2	**then**
*2***		3	**case** cS2_C is
	*2***	4	**when** cS2_C_W1 =>
3		5	V2 **<=** V2_T1;
4		6	**if** (cS3_C) -- *where, cS3_C = cS6_C*
	3	7	**then**
5		8	V1 **<=** V1_T2;
	4	9	**else**
6		10	V1 **<=** V1_T1;
		11	end if;
	*5***	12	**when** cS2_C_W2 =>
7		13	V2 **<=** V2_T1;
8		14	**if** (cS4_C1 **and** ((**not** cS4_C2) **or** cS4_C3))
	6	15	**then**
9		16	V1 **<=** V1_T3;
	7	17	**else**
10		18	V1 **<=** V1_T1;
		19	end if;
	*8***	20	**when** cS2_C_W3 =>
11		21	V1 **<=** V1;
12		22	**if** (cS5_C1 **and** cS5_C1)
	9	23	**then**
13		24	V2 **<=** V2_T2;
	10	25	**else**
14		26	V2 **<=** V2_T1;
		27	end if;
		28	end case;
	*11***	29	**else**
15		30	V2 **<=** V2_T2;
16		31	**if** (cS6_C) -- *where, cS3_C = cS6_C*
	12	32	**then**
17		33	V1 **<=** V1_T2;
	13	34	**else**
18		37	V1 **<=** V1_T1;
		36	end if;
		37	end if;

Figure 4.3. The VHDL file functional segment of an example design *CovEx*

consider statements 1 and 2 emphasized by subscript asterisk character '*' in Figure 4.3 and by additional subscript indexes in Figure 4.4. They are represented twice by the nodes of both variables *V1* and *V2* graphs, and therefore there are 20 HLDD nodes in total.

4.2.2 Branch coverage mapping

The *branch coverage* metric reports the ratio of branches in the control flow graph of the code that are traversed under the given set of stimuli. The conditional statements of HDL code are *if*- and *case*- statements. The branch coverage indicates separately evaluations of the statements *if* and *elsif* to *'true'* and *'false'* as well as evaluations of *case* statement to all of its solutions. This metric is also

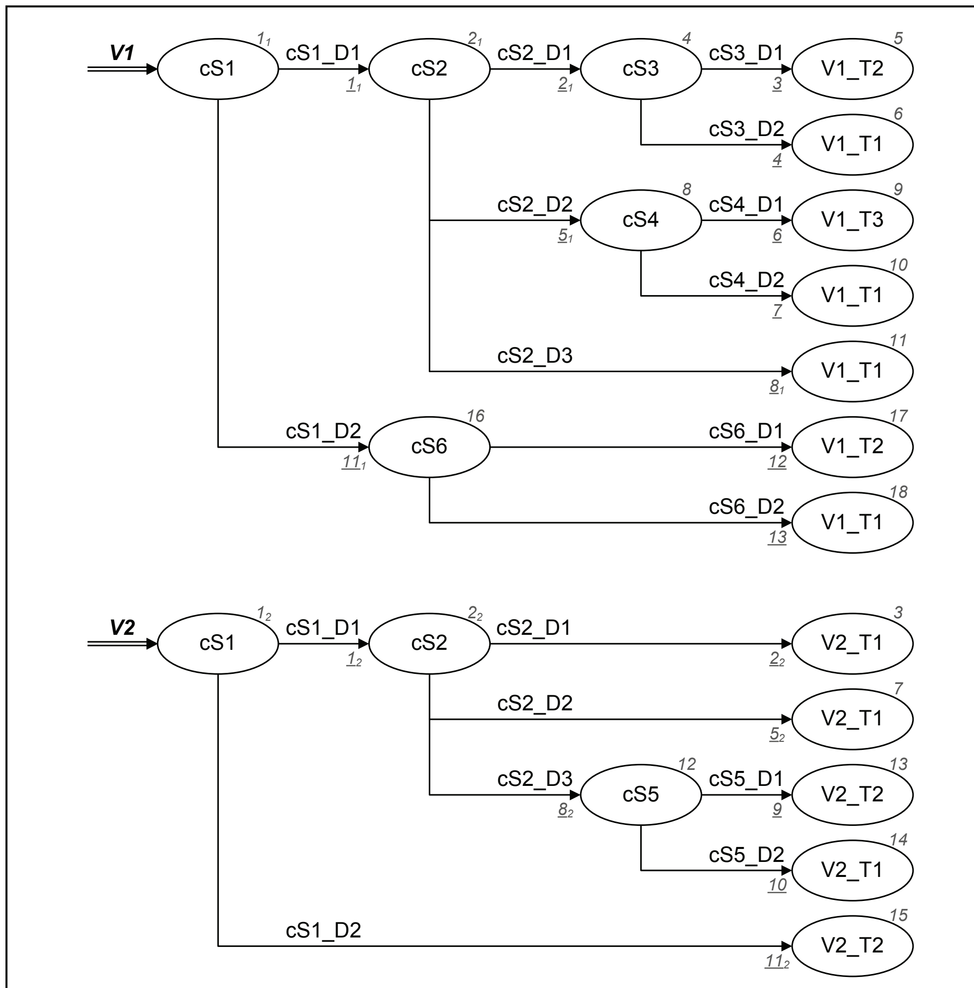


Figure 4.4. HLDD representation of the example design *CovEx*

known as *decision coverage*, especially in software testing [106], *all-edge coverage* and *arc coverage*. In a typical application of branch coverage measurement, the number of every decision's hits is counted. Note, that the full branch coverage comprises full statement coverage.

Conditional statements create different *paths* of execution over time. *Path coverage* reports the ratio of paths in the control flow graph executed during simulation under the given stimuli to the total amount of possible paths. This metric is more stringent compared to branch coverage. However, its main disadvantage is that the number of paths grows exponentially with the number of conditional statements. Therefore, it is rarely used in practice for reasonably large real designs.

Similar to the statement coverage, branch coverage also has very clear representation ([15],[16])co-auth. in HLDD model. It is the ratio of every edge e_{active} activated in the simulation process presented by Algorithm 1 (Subsection 3.4.1) to the total number of edges in the corresponding HLDD representation of the DUV.

Please consider the VHDL segment in Figure 4.3. Here, the second column (*Dcn.*) numbers all 13 branches (aka decisions) of the code. The edges in the HLDD graphs provided in Figure 4.4 represent these branches and are marked by the corresponding numbers (underlined). Covering all edges in a HLDD model (i.e. full *HLDD edge coverage*) corresponds to covering all branches in the respective HDL. Please note that some of the HDL branches also have duplicated representation by the HLDD edges. This is due to the same reason as with HLDD nodes. While several conditional statements appear in the graphs of both variables (*V1* and *V2*), some of their decisions do too. These decisions in the *CovEx* example design are *1*, *2*, *5*, *8*, *11* (Figures 4.3 and 4.4), i.e. decisions of *cS1* and *cS2*. They are marked by '*' in Figure 4.3 and by additional subscript indexes in Figure 4.4.

We will refer to statement and branch coverage as *base coverage metrics*. These metrics cover the constituents of decision diagrams, i.e. nodes and edges. Also, the majority of the other coverage metrics employ them in some way.

4.2.3 Toggle coverage mapping

The *toggle coverage* metric, depending on its implementation, reports how many times each design signal, variable (or sometimes each bit of a register, or a bus) toggles, i.e. changes its state from *'1'* to *'0'* and vice versa under the given set of stimuli. Some code coverage measurement aided simulators measure in addition the toggles to and from *'X'* (undefined) and *'Z'* (tristate) values.

In the current implementation ([15],[16])co-auth., the toggle coverage analysis based on HLDD model does not have any particular advantages caused by HLDD model application. It implies similar counters as HDL simulators do.

4.2.4 State coverage mapping

State coverage, also known as FSM coverage [5], implies several variations of the DUV's finite state machine behaviour analysis for the given set of stimuli. In this thesis we consider the following two of them:

- The first approach is to analyze which states were visited during the simulation and count the number of the visits (hits).
- The second approach is to analyze the amount of fired transitions from one state to another.

A support for the state coverage measurement requires from a appropriate HDL tool ability to identify and extract the DUV's FSM from its RTL HDL description.

As it has been shown in Clauses 2.1.2.3 and 2.1.2.4 the HLDD-based representation of a RTL design clearly distinguishes graphs for separate variables including the *state* variable. Please consider Figures 2.8 and 2.10 for example.

The first approach (from the listed above) for the state coverage measurement maps to the HLDD-based measurement of the coverage for the *terminal nodes* (assignments) in the *state* variable's graph. The second approach maps to the HLDD edges based measurement of sub-paths between the *current state* node and terminal nodes coverage (please consider Figures 2.8 and 2.10). HLDD-based analysis of the state coverage implies the advantages of base coverages metrics HLDD-based analysis.

4.2.5 Data flow coverage mapping

Data flow coverage metric ([86],[106]) reports covered and uncovered sub-paths from data variables' assignments to their subsequent references (for the given set of stimuli). It can be considered as a simplified for calculation yet powerful variation of the path coverage metric (please see the Subsection 4.2.2). The main advantage of the data flow metric is its direct relevance to the actual design data flow. For example, in case if we have the full branch coverage for a particular DUV/stimuli pair, it does not guarantee the sub-path between a DUV's variable assignment and its reference (use) to be covered. Therefore, data flow coverage provides for an extra potential of corner cases misbehaviour (implementation errors) discovery. However, the analysis of this metric based on HDL design representations has high complexity and, therefore, it is rarely used.

On the other hand, HLDD-based design representation contains separate graphs for each variable and signal of the design. Full data flow coverage of a design maps to the coverage of all single paths from terminal nodes to the root nodes separately in all variables' HLDD sub-graphs. Data flow coverage HLDD-based analysis strictly requires HLDD model of the reduced typed and partitioned by variable. These requirements are discussed in Section 4.4.

4.2.6 Condition coverage mapping

Condition coverage metric ([5],[106]) reports the number of times each Boolean sub-expression, separated by logical operators *or* and *and*, in a conditional statement causes the complete conditional statement to evaluate to one of the decisions (e.g. *'true'* or *'false'* values) under the given set of stimuli. It differs from the branch coverage, by the fact that in the branch coverage only the final decision determining the branch is taken into account. In case, if we have *n* conditions joined by logical *and* operators in a logical expression of a conditional statement, it means that the probability of evaluating the statement to the decision *'true'* is $1/2^n$ (considering pure random stimuli for the condition values). Calculation of the

condition coverage based on HDL representation is a sophisticated multi-step process. However, the condition coverage metric allows discovering information about many corner cases of the DUV.

The approach for HLDD-based analysis of condition coverage is proposed and described in detail in Section 4.3.

4.3 A hierarchical approach for HLDD-based condition coverage analysis

In this section we propose an approach for HLDD-based condition coverage analysis. The approach is based on a hierarchical DUV representation where the conditional statements with complex logical expressions (normally represented by single nodes in HLDD graphs) are representation by BDDs. A brief introduction to this hierarchical representation was shown in Figure 2.9 (Clause 2.1.2.4). Please note that the BDD graphs for expanded conditional statements use the alternative description style presented in Subsection 2.1.1 (Figure 2.1c).

Let us consider the example design *CovEx* provided in Figures 4.3 and 4.4. It contains 6 conditional statements *cS1*-*cS6*, repeated in Figure 4.5.

```
cS1: if (cS1_C1 or cS1_C2) then cS1_D1 else cS1_D2;

cS2: case (cS2_C)   when cS2_C_W1: cS2_D1;
                    when cS2_C_W2: cS2_D2;
                    when cS2_C_W3: cS2_D3;

cS3: if (cS3_C) then cS3_D1 else cS3_D2;             -- where, cS3_C = cS6_C

cS4: if (cS4_C1 and ((not cS5_C2) or cS5_C3))then cS4_D1 else cS4_D2;

cS5: if (cS5_C1 and cS5_C2) then cS5_D1 else cS5_D2;

cS6: if (cS6_C) then cS6_D1 else cS6_D2;             -- where, cS3_C = cS6_C
```

Figure 4.5. The conditional statements of *CovEx*

The design contains 3 conditional statements with single conditions. They are equivalent statements *cS3* and *cS6* and *cS2* with a non-Boolean single condition. Moreover, the *case*- conditional statements always contain a single, usually, non-Boolean condition (in case of a Boolean condition it has two decisions and can be substituted by an *if*- conditional statement). A BDD expansion graph for a conditional statement with a single condition has the same number of edges as the number of terminal nodes, and therefore does not contain any additional information compared to the pure HLDD-based representation of this statement. In other words the condition coverage for a HLDD with conditional statements nodes containing only single conditions is equivalent to its branch coverage. The conditional statements *cS1*, *cS4* and *cS5* contain complex logic expressions with multiple conditions.

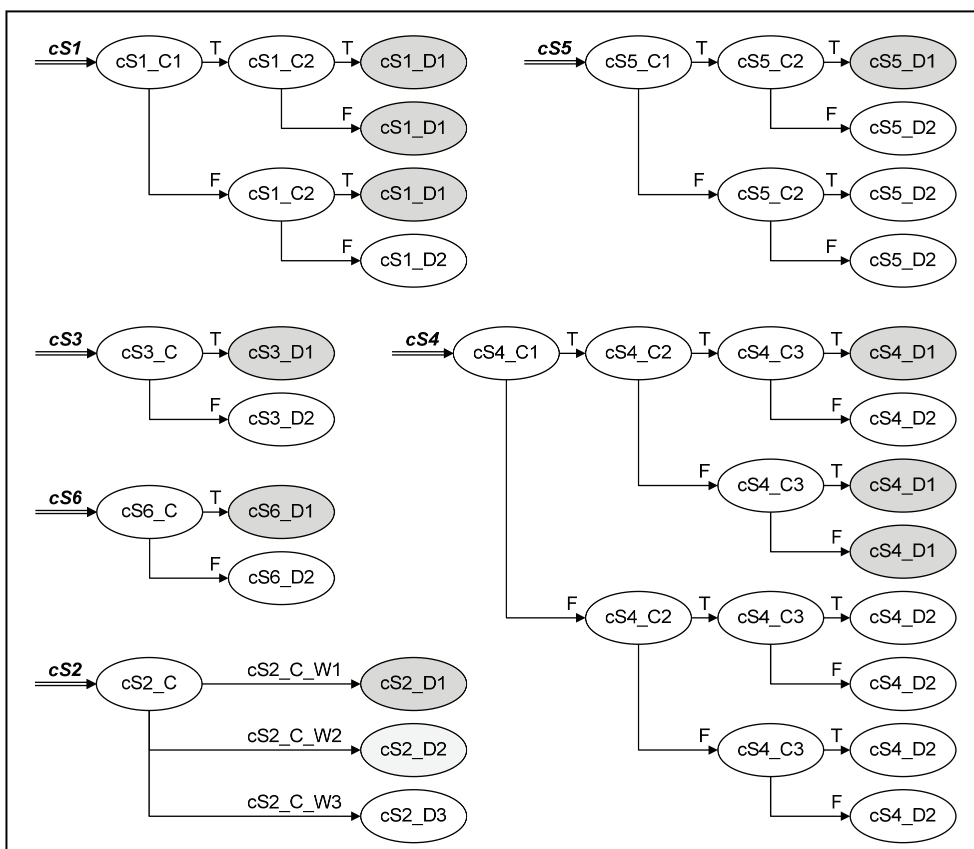


Figure 4.6. BDD-expanded multi-condition conditional statements of *CovEx*

The BDD-based expanded representations for all conditional statements of *CovEx* design are provided in Figure 4.6. Here the terminal nodes are marked by background colours according to different decisions for better readability. These 6 graphs can be considered as sub-graphs representing "virtual" variables (because they are not real variables of the *CovEx* VHDL representation) *cS1*-*cS6*. Thus, together with the two HLDD graphs for variables *V1* and *V2* from Figure 4.4 these sub-graphs compose design's *hierarchical DD* representation, which is a BDD-aided HLDD.

The complete (i.e. not system-based) hierarchical DD graph for the variable *V2* of *CovEx* design is provided in Figure 4.7.

The full condition coverage metric maps to full coverage of terminal nodes of the BDD graphs from the system-based hierarchical DD representation during the complete system simulation with the given stimuli. The size of the items list for

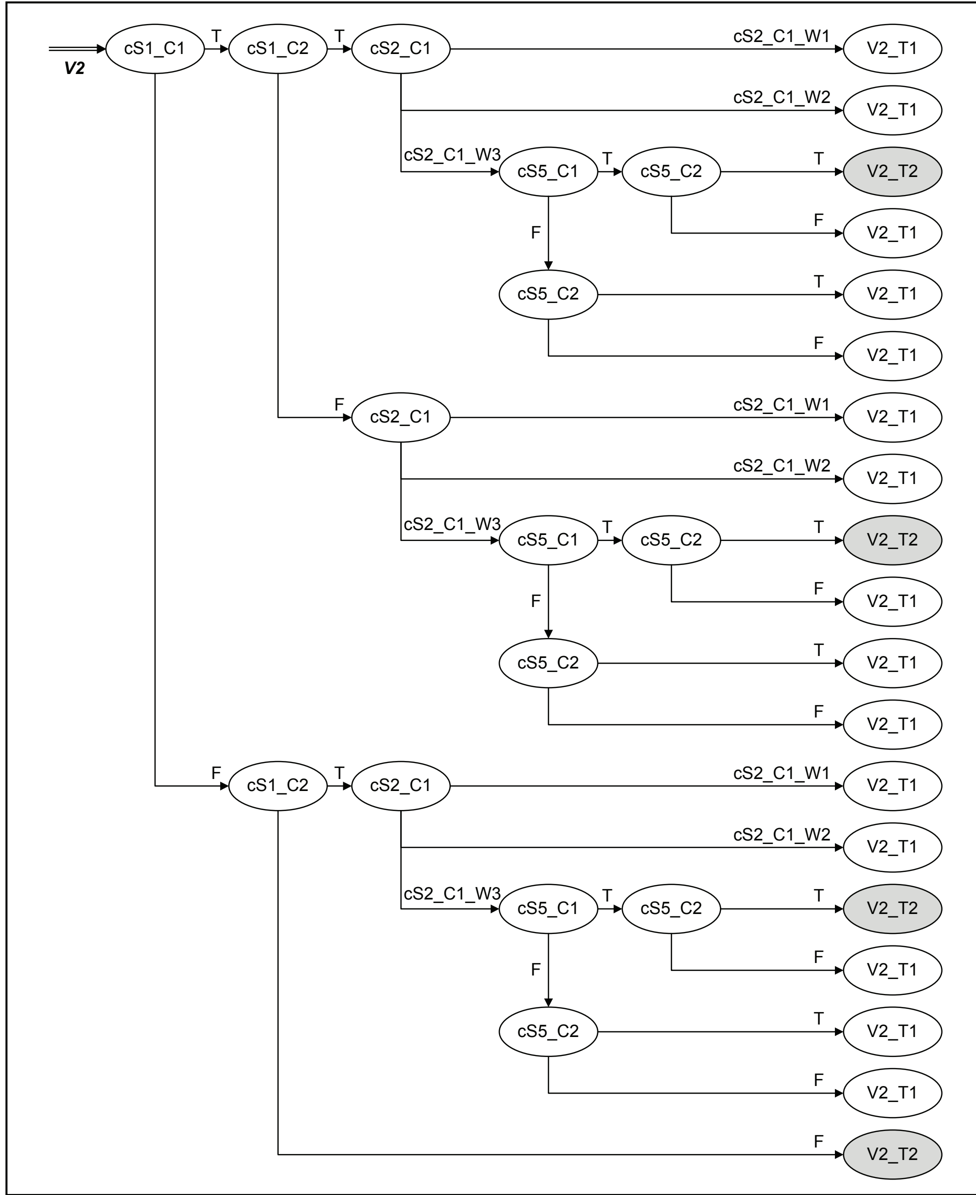


Figure 4.7. Complete HLDD graph with expanded conditional statements for *V2*

this coverage metric is $\sum_{i=1}^{n_{cS}} 2^{n_{c_i}}$ where n_{cS} is the number of conditional statements and n_{c_i} is the number of conditions in the ith conditional statement.

In case if we measure the amount of covered terminal nodes for all complete hierarchical DD graphs (an example of such graph for *V2* of *CovEx* is provided in

Figure 4.7) of the design, we will get a combination of the data flow and condition coverage metrics (*data flow/condition coverage*). This metric is adequately stringent for hardware simulation-based verification and has relatively good ratio of stringency and calculation overhead. The obvious advantage of this combination is that the enhancement of data flow coverage by the condition coverage metric adds the competence of the DUV's structure coverage in an orthogonal axis. An analog for this coverage, however less stringent one, is the popular [106] in software testing *Modified Condition Decision Coverage* (MC/DC) metric.

The main advantage of the proposed approach is low computational overheads for condition coverage and data flow/condition coverage analysis. Once the system-based- (for condition coverage) or complete- (for dataflow/condition coverage) hierarchical DD is constructed, the analysis for an every given stimuli set is evaluated in a straightforward manner by the same tool (HLDD$sim^{coverage}$).

The size of the DDs with the expanded conditional statements may grow exponential to the number of conditions and therefore there is significant increase of the memory consumption. However, the length of the average sub-path from the root to terminal nodes grows linear to the number of the conditions. Therefore, since the simulation time of a HLDD has a linear dependency to the average sub-path from the root to terminal nodes, it will grow only linearly with respect to the number of conditions.

4.4 HLDD model reduction manipulations for code coverage analysis

In this section we propose [16][co-auth.] to distinguish three types of HLDD representation according to their compactness, and with consideration of the *HLDD reduction rules*. These rules are similar to the reduction rules for BDDs [40] presented in Subsection 2.1.1 and can be generalized as follows (the differences are underlined):

HLDD reduction rule1: Eliminate all the redundant nodes whose <u>all</u> edges point to an <u>equivalent</u> sub-graph.

HLDD reduction rule2: Share all the equivalent sub-graphs.

The three representation types in the increasing order of compactness are:

- *Full tree HLDD* contains all control flow branches of the design.
- *Reduced HLDD* is obtained by application of the *HLDD reduction rule 1* to the *full tree* representation. This HLDD representation is still a tree-graph.

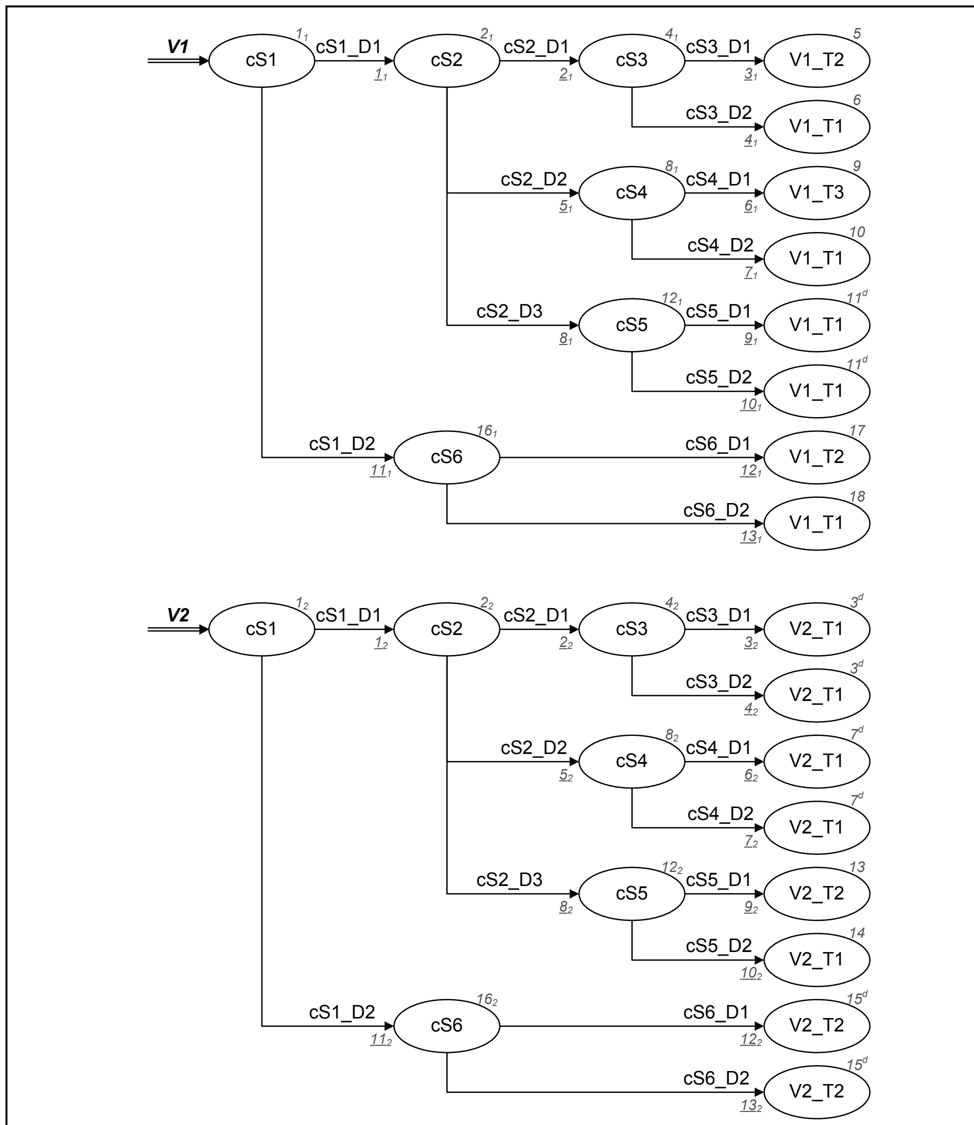


Figure 4.8. The *full tree* HLDD representation for the *CovEx* design

- *Minimized HLDD* is obtained by application of both *HLDD reduction rules 1* and *2* to the *full tree* representation. This representation is no longer a tree.

The presented in Figure 4.4 HLDD representation for the example design *CovEx* is of the *reduced* type. Figures 4.8 and 4.9 present the *full tree* and *minimized* HLDD representations for the same design, correspondingly.

A less compact HLDD representation contains more items, i.e. nodes and edges. It means it requires more memory for the data structure storage and possibly longer

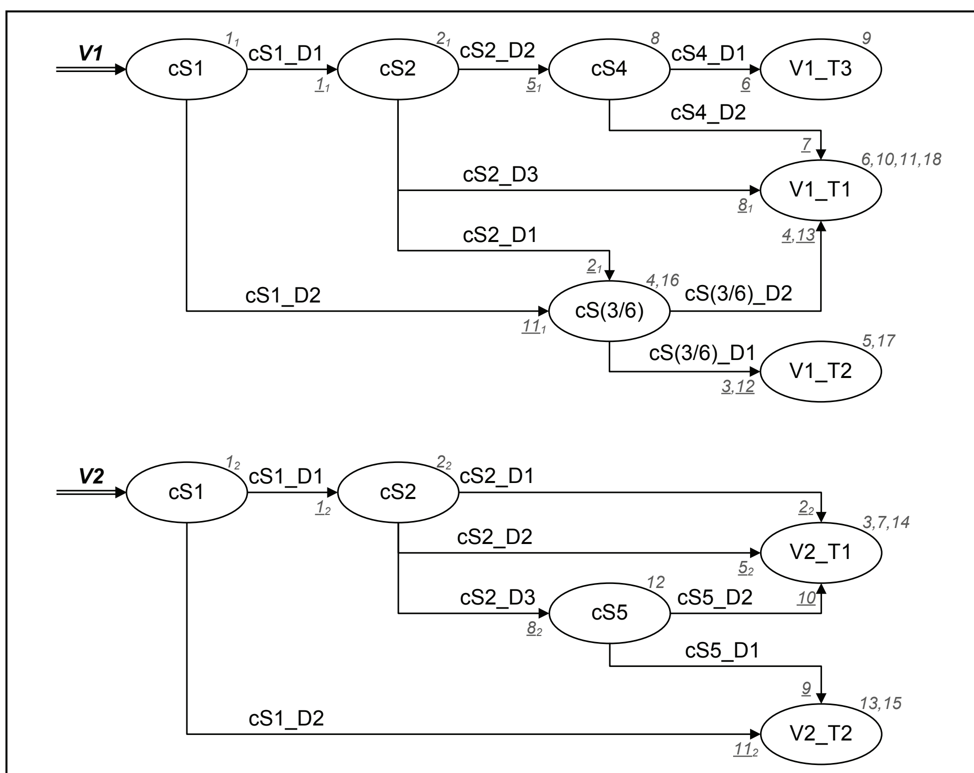


Figure 4.9. The *minimized* HLDD representation for the *CovEx* design

simulation time, if the average sub-path from the root to terminal nodes becomes longer. However, it is potentially capable to represent the design's structure more accurately and therefore the coverage measurement may be more accurate as well. In fact, this dependency is not "linear" and particular HLDD representation types may be more convenient or not suitable at all for particular application.

It has been shown in [16][co-auth.] (please see also Section 4.5) that analysis results for the base code coverage metrics, i.e. statement and branch coverages, performed on reduced HLDDs is more stringent that the ones with the minimized HLDDs. Moreover, compared to HDL-based analysis the reduced HLDD-based results are always more stringent, while minimized HLDD-based ones are often less. However, it is necessary to note that the direct comparison to HDL is not very accurate due to its dependency to the coding style.

At the same time the performance of the base coverage metrics analysis based on reduced and minimized HLDD model is equivalent due to the fact that both models have the same average length of sub-paths from the root to terminal nodes. Compared to the *full tree* HLDD representation, the *reduced* HLDD model usually has significant performance improvement while the accuracy of the design's structure representation remains the same for the base code coverage metrics.

The conclusions for application of the three types of the HLDD representations compactness for the structural coverage base metrics' analysis are the following:

- The *minimized* HLDDs provide the most compact design representation, and, therefore, it has the lowest memory requirements. However, a possibility of coverage stringency loss should be considered for the structural coverage analysis based on this type of DUV's representation.
- The *reduced* HLDDs require more memory, but they do not lose in coverage metrics' stringency. The simulation speed compared to the minimized HLDDs remains the same.
- The *full tree* HLDDs-based coverage analysis is slower and does not provide any gain in the stringency of the considered metrics, compared to the reduced HLDDs-based one. However, this representation type may be required for an accuracy lossless analysis of several other coverage metrics. A comprehensive analysis of these metrics is scheduled as a future work.

As it was discussed on the previous section the measurement of the *condition coverage* is performed on the hierarchical DD model employing both HLDD model and BDDs for expanded conditional statements representation. For this analysis we assume *reduced* HLDD model and *full tree* type of the BDD representations. *Toggle* and *data flow* and *state* coverage metrics are also proposed to be analyzed based on reduced HLDDs.

Please note, that the correct analysis of the *condition* and *data flow* and *state* coverage metrics may involve also other HLDD manipulations discussed in Clause 2.1.2.3 (Figure 2.7). It is the correct *partitioning* of the HLDDs.

- The *data flow coverage* analysis requires strict partitioning by variables, i.e. exactly one graph for every variable or signal. In case of the data flow/condition coverage the expanded conditional statements must be contained in the compete hierarchical DD (Figure 4.7).
- The pure *condition coverage* analysis requires separation of the BDD graphs representing conditional statements. The resulting design representation must be system-based .
- The *state coverage* analysis accepts partitioning by variables and strictly requires a separate graph for the DUV's *state* variable.

The proposed in this chapter approaches and statements are supported by experimental results presented in the next section.

4.5 Experimental results

This section provides experimental results ([15],[16])[co-auth.] for the proposed HLDD-based verification coverage analysis approaches. First, the experimental results present information for the comparison of the proposed approaches implemented as HLDD*sim*^{coverage} with a state-of-the-art commercial tool from a major CAD vendor. Second, the comparisons are performed for verification coverage measurements based on different HLDD model representation types.

The experiments were carried with the benchmarks presented in Section 3.6 *gcd*, *b00*, *b04*, *b09* and also 3 other benchmarks from the ITC'99 benchmarks family [76],[102],[105] *b01*, *b02* and *b06*.

- *b01* is a design implementing functionality a finite state machine that compares serial flows
- *b02* is a design implementing functionality of a finite state machine that recognizes binary-coded decimal numbers
- *b06* is a design implementing functionality of an interrupt handler

Design	Coverage measurement time overhead, (%)	
	Commercial HDL simulator	HLDD*sim*
b00	28.0	1.0
b04	32.2	0.9
b09	78.9	4.3
gcd	31.7	3.2

Figure 4.10. Coverage analysis penalty: traditional vs HLDD

The comparative experiments between the HLDD-based code coverage analysis tool HLDD*sim*^{coverage} and a state-of-the-art commercial HDL simulation tool from a major CAD vendor have been presented in [15][co-auth.]. Their results have shown that the time overhead of verification coverage measurement in the popular commercial tool environment is much higher than in the case of HLDD-based approach. When HLDDs have coverage measurement time overhead in a range of 1% to 4%, the commercial simulator uses from 28% up to 78% extra time for coverage measurement (see Figure 4.10).

Design	Number of nodes			Number of edges	
	min	*red.*	*f.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

Figure 4.11. Characteristics of different HLDD manipulations

Figure 4.11 presents the characteristics [16][co-auth.] of the different HLDD representations introduced in Section 4.4. The columns *min*, *red*. and *f.tree* show the number of nodes and edges in *minimized*, *reduced* and *full tree* HLDD model representations, respectively. As it can be seen from the figure, around 45-80 % of nodes were removed by the reduction step from the initial HLDD full tree. Further 40-60 % of nodes were eliminated by the minimization step.

Design	Stimuli, (vectors)	Statement coverage, (%)			Branch 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

Figure 4.12. Comparison of code coverage analysis results

Figure 4.12 shows the comparison results [16][co-auth.] of the base code coverage metrics analysis based on *reduced* HLDDs, *minimized* HLDDs and a state-of-the-art commercial HDL simulation tool from a major CAD vendor 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. However, as it has been noticed earlier, the comparison to HDL can be slightly inaccurate due to coding style variations.

4.6 Chapter summary

This chapter has discussed the notion of verification coverage together with its classification for simulation-based hardware verification. The main focus of the chapter was structural coverage that is also known as code coverage. Several practically used metrics of this coverage have been described and approaches for their HLDD-based analysis have been presented. One of them is an approach for condition coverage measurement that employs hierarchical decision diagrams consisting of HLDDs and BDD-based representations of the conditional statements. The chapter also discusses how the accuracy and performance of HLDD-based coverage analysis depend of the HLDD model's reduction manipulations.

The HLDD-based structural verification coverage analysis has the following main advantages:

- HLDD can be generated or manipulated further in accordance with its target application for particular coverage metric analysis.
- All coverage metrics' measurements and analysis are performed by the same tool HLDD*Sim*^{coverage}.
- HLDD-based analysis has a better performance than HDL-based one due to, first, faster HLDD-based simulation and, second, lower percentage ratio for the measurement overhead.
- The proposed HLDD-based coverage metrics are more stringent than HDL-based ones and therefore allow discovering more corner cases and assessing stimuli more precisely.

The HLDD-based verification coverage analysis approaches also consider *observability coverage* [15][co-auth.] that is not discussed in this thesis.

Chapter 5
CONCLUSIONS AND FUTURE WORK

This thesis has presented several approaches addressing simulation-based hardware verification issues. The approaches target assertion checking and structural coverage measurement and exploit advantages of high-level decision diagrams design representation model.

This chapter summarizes the thesis and points out open problems and interesting directions for future work.

5.1 Conclusions

5.1.1 Contributions

The contribution of this thesis is twofold:

A new approach for HLDD-based assertions checking

- A *temporal extension for the existing HLDD mode*. The new extended model is aimed at temporal properties expression and named *Temporally extended High-Level Decision Diagrams* (THLDD). The extension supports a set of commonly used temporal constructs that can be used to express a wide set of possible complex temporal relationships.
- A *methodology for direct conversion of assertions* expressed in Property Specification Language (PSL) to THLDD. The proposed hierarchical approach introduces an extendable library of Primitive Property Graphs (*PPG Library*). The components of this library serve as building blocks for a complex THLDD property construction.

- An *approach for HLDD-based assertion checking*. A *modification* of the existing HLDD-based *simulator* (HLDD*sim*) was proposed to support THLDDs and assertion checking. This part was supported by explanations of temporal issues and different varieties of THLDD properties.

The feasibility of the proposed approaches was proven by the presented experimental results. A minor contribution includes discussions of verification assertions reuse for manufacturing testing.

A new approach for HLDD-based coverage analysis

- An approach for *mapping* traditional verification structural coverage metrics to HLDD-based coverage. In addition to the base code coverage metrics such as statement and branch coverage, the approach considers also more sophisticated ones, including FSM and data flow coverage metrics.
- An approach for *condition coverage* analysis. The approach employs a *hierarchical decision diagrams* model consisting of HLDDs and BDD-based representations of the conditional statements.
- An approach for *HLDD model manipulations* targeted to different aspects of verification coverage analysis.

The feasibility of the proposed approaches was proven by the presented experimental results.

5.1.2 Advantages

The main advantages of the proposed HLDD-based approaches for simulation-based verification are outlined in the following:

- ✓ The proposed approaches rely on a *homogeneous hardware verification flow* based on High-Level Decision Diagrams (HLDD) design representation model. Once an appropriate input objects' representation is created the analysis is performed by the same tool HLDD*Sim*.
- ✓ HLDD-based analysis has a *better performance* than HDL-based one due to, first, faster HLDD-based simulation and, second, *lower percentage ratio overheads* for both assertion checking the coverage measurement processes.
- ✓ THLDD model is capable to represent complex temporal properties and supports a wide set of PSL language.
- ✓ HLDD can be generated or manipulated further in accordance with its target application for particular coverage metric analysis.

5.2 Future work

In this section we outline a few issues which can be considered further in order to improve and advance the approaches proposed in this thesis:

HLDD-based assertions checking:

- The presented approach, including its all three constituents (i.e. the model, PSL to THLDD properties conversion methodology and assertion simulation-based checking process), supports a *wide set of PSL language*. The supported part is close to the PSL simple subset and it is a powerful instrument to express the majority of practical temporal properties. However, in our future work we would like to target the remaining PSL FL LTL operators and add a support for SERE. The extension should necessarily include the support for the strong version of the PSL operators. As soon as the HLDD-based design representation finds its application in formal verification, the supported language subset should support CTL and full LTL as opposed to currently targeted PSL FL LTL simple subset.
- The proposed approach for HLDD-based assertion checking implies 2-step process. First the DUV simulation trace for the given stimuli is calculated. Second the assertions are evaluated based on the simulation trace. This approach is convenient in the most of the cases, but in some situations (e.g. very long simulations) the *dynamic assertion checking* may be preferable. Its support may be addressed by HLDD*sim* modification.
- In this thesis we have drawn a number of ideas for *verification assertions reuse for manufacturing testing*. The proposed ideas are scheduled for further development and integration with the previously performed research in the area of manufacturing testing.

HLDD-based coverage analysis:

- In this thesis we have briefly proposed a new approach for *data flow coverage metric analysis* based on HLDD model. This metric seems very attractive in terms of DUV's structure representation accuracy. At the same time its mapping to HLDD coverage has obvious convenience. A comprehensive analysis of this metric application and detailed development of an appropriate approach for its measurement are other attractive directions for future work.
- The proposed HLDD-based verification coverage analysis assumes structural coverage (aka code coverage). This coverage type is widely applied in practice, however it has several drawbacks. The latter ones are partially caused by the extended concurrency of the state-of-the-art complex designs functionality. Therefore, a comprehensive verification plan for such designs is preferred to include functional coverage analysis. At present it is partially

supported in the HLDD-based verification flow by assertion coverage analysis. However, as a longer term future work we see application of more comprehensive coverage models for *functional coverage HLDD-based analysis*.

In general:

- More comprehensive experimental results with additional benchmarks would be beneficial. For this purpose we plan to try large complex real-life industrial designs.
- Further development of the presented tools from HLDD-based verification flow is relevant. Here the target is stand-alone reliable tool set that is convenient to use.
- In this thesis we have considered simulation-base hardware verification. It is widely used in practice and capable to handle large state-of-the-art designs. However, a number of verification issues are more reasonable to address by formal verification approach. We also consider to our research efforts towards its support by HLDD-based verification flow.

Finally, the presented research has paved the way for future development of HLDD model application in hardware functional verification. This thesis has revealed not only advantages of this approach but also its potential for the future.

References

Books:

[1] William K. Lam, "Hardware Design Verification: simulation and Formal Method-Based Approaches", *Prentice Hall, Pearson*, 2005

[2] Bruce Wile, John C. Goss, Wolfgang Roesner, "Comprehensive Functional Verification: The Complete Industry Cycle", *Elsevier*, 2005

[3] Harry D. Foster, Adam C. Krolnik, "Creating Assertion-Based IP", *Springer*, 2008

[4] Cindy Eisner, Dana Fisman, "A Practical Introduction to PSL", *Springer,* 2006

[5] Andrew Piziali, "Functional Verification Coverage Measurement and Analysis", *Springer*, 2008

[6] Jun Yuan, Carl Pixley, Adnan Aziz, "Constraint-Based Verification", *Springer*, 2006

[7] Douglas L. Perry, Harry D. Foster, "Applied Formal Verification", *McGraw-Hill*, 2005

[8] Katarzyna Radecka, Zeljko Zilic, "Verification by Error Modeling: Using Testing Techniques in Hardware Verification", *Kluwer*, 2003

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

Co-authored papers:

[10] Maksim Jenihhin, Jaan Raik, Anton Chepurov, Raimund Ubar "Temporally Extended High-Level Decision Diagrams for PSL Assertions Simulation", *Proc. of 13th IEEE European Test Symposium (ETS'08),* Verbania, Italy, May 25-29, 2008, pp. 61-68

[11] Maksim Jenihhin, Jaan Raik, Anton Chepurov, Raimund Ubar, "PSL Assertion Checking with Temporally Extended High-Level Decision Diagrams", *Proc. of 9th IEEE Latin American Test Workshop (LATW'08)*, Puebla, Mexico, February 17-20, 2008, pp. 49-54

[12] Maksim Jenihhin, Jaan Raik, Anton Chepurov, Raimund Ubar, "Assertion Checking with PSL and High-Level Decision Diagrams", *Digest of the IEEE 8th Workshop on RTL and High Level Testing (WRTLT'07)*, Beijing, China October 12-13, 2007, pp. 105-110

[13] Maksim Jenihhin, Jaan Raik, Anton Chepurov, Raimund Ubar, "PSL Assertion Checking Using Temporally Extended High-Level Decision Diagrams", *Journal of Electronic Testing: Theory and Applications (JETTA) [submitted on September 30, 2008]*

[14] Jaan Raik, Maksim Jenihhin, Anton Chepurov, Uljana Reinsalu, Raimund Ubar, "APRICOT: a Framework for Teaching Digital Systems Verification", *Proc. of 19th EAEEIE Annual Conference, IEEE,* Tallinn, Estonia, June 29 - July 2, 2008, pp. 1-6

[15] Jaan Raik, Uljana Reinsalu, Raimund Ubar, Maksim Jenihhin, Peeter Ellervee, "Code Coverage Analysis using High-Level Decision Diagrams", *Proc. of the 11th IEEE Workshop on Design and Diagnostics of Electronic Systems (DDECS'08)*, April, 2008, pp. 201-206

[16] Karina Minakova, Uljana Reinsalu, Anton Chepurov, Jaan Raik, Maksim Jenihhin, Raimund Ubar, Peeter Ellervee, "High-Level Decision Diagram Manipulations for Code Coverage Analysis", *Proc of the 11th IEEE Biennial Baltic Electronics Conference (BEC'08),* Tallinn, Estonia, October 2008, pp. 207 - 210

[17] Maksim Jenihhin, Jaan Raik, Raimund Ubar, Anton Chepurov, "On reusability of verification assertions for testing", *Proc. of 11th IEEE Biennial Baltic Electronics Conference (BEC'08),* Tallinn, Estonia, October 2008, pp. 151 - 154

[18] Maksim Jenihhin, "Assertion-based verification and testing with Decision Diagrams", *PhD Forum (abstract + poster), Design, Automation and Test in Europe (DATE'08)*, Munich, Germany, March 10-14, 2008

[19] Jaan Raik, Raimund Ubar, Taavi Viilukas, Maksim Jenihhin, "Mixed Hierarchical-Functional Fault Models for Targeting Sequential Cores", *Journal of Systems Architecture, 54(3-4), Elsevier,* 2008, pp. 465 - 477

[20] Raimund Ubar, Sergei Devadze, Maksim Jenihhin, Jaan Raik, Gert Jervan, Peeter Ellervee, "Hierarchical Calculation of Malicious Faults for Evaluating the Fault-Tolerance", *Proc. of IEEE International Symposium on Electronic Design, Test and Applications (DELTA'08)*, Hong Kong, January 23 - 25, 2008, pp. 222-227

[21] Maksim Jenihhin, Jaan Raik, Raimund Ubar, Witold Pleskacz, Michal Rakowski, "Layout to Logic Defect Analysis for Hierarchical Test Generation", *Proc. of 10th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems(DDECS'07)*, Kraków, Poland, April 11-13, 2007, pp. 35-40

[22] Witold Pleskacz, Maksim Jenihhin, Jaan Raik, Michal Rakowski, Raimund Ubar, Wieslaw Kuzmicz, "Hierarchical Analysis of Short Defects between Metal Lines in CMOS IC", *Proc. of the 11th Euromicro Conference on Digital System Design (DSD) Architectures, Methods and Tools*, Parma, Italy, September 2008, pp 729 - 734

[23] Jenihhin, Maksim, "On reusability of verification assertions for testing", *Proc. of the 3rd IKTDK Conference*, Voore, Estonia, April 25-26, 2008, pp. 43-46

[24] Jenihhin, Maksim, "PSL Assertions based Verification with HLDD Tools", *Proc. of the 2nd IKTDK Conference*, Viinistu, Estonia, May 11-12, 2007, pp. 17 - 20

[25] Jenihhin, Maksim, "Case Study: Defect-Oriented Testing of a Combinational Circuit", *Proc. of the 1st IKTDK Conference*, Janeda, Estonia, May 12-13, 2006, pp. 78 - 81

[26] Knut Hermann, Jaan Raik, Maksim Jenihhin "TTBist: a DfT Tool for Enhancing Functional Test for SoC", *Proc. of the Baltic Electronics Conference*, Laulasmaa, Estonia, 2006, pp. 191-194

[27] Jaan Raik, Maksim Jenihhin, Rain Adelbert, "Sequential Circuits BIST Synthesis from Signal Specifications", *Proc. of IEEE Norchip Conference*, Oulu, Finland, November 21-22, 2005, pp.196 - 199.

[28] Gert Jervan, Petru Eles, Zebo Peng, Raimund Ubar, Maksim Jenihhin "Test Time Minimization for Hybrid BIST of Core-Based Systems", *Journal of Computer Science and Technology, 21(6)*, 2006, pp. 907 - 912

[29] Raimund Ubar, Maksim Jenihhin, Gert Jervan, Zebo Peng "Hybrid BIST Optimization for Core-based Systems with Test Pattern Broadcasting", *Proc. of the IEEE International Workshop on Electronic Design, Test and Applications (DELTA 2004)*, Perth, Australia, January 28-30, 2004, pp. 3 - 8.

[30] Raimund Ubar, Maksim Jenihhin, Gert Jervan, Zebo Peng "An Iterative Approach to Test Time Minimization for Parallel Hybrid BIST Architecture" *Proc. of 5th IEEE Latin-American Test Workshop (LATW'04)*, Cartagena, Colombia, March 8-10, 2004, pp. 98 - 103.

[31] Gert Jervan, Petru Eles, Zebo Peng, Raimund Ubar, Maksim Jenihhin "Hybrid BIST time minimization for core-based systems with STUMPS architecture", *Proc. of the 18th IEEE International Symposium on Defect and*

[32] Gert Jervan, Petru Eles, Zebo Peng, Raimund Ubar, Maksim Jenihhin "Test Time Minimization for Hybrid BIST of Core-Based Systems", *Proc. of the 12th IEEE Asian Test Symposium (ATS03)*, Xian, China, November 17-19, 2003, pp. 318 - 323

[33] Raimund Ubar, Maksim Jenihhin, Gert Jervan, Zebo Peng "Test Time Minimization for Hybrid BIST with Test Pattern Broadcasting", *Proc. of the 21st NORCHIP Conference*, Riga, Latvia, November 10-11, 2003, pp. 112 - 116

[34] Maksim Jenihhin, "Test Time Minimization for Parallel Hybrid BIST Architectures", *Master thesis, Tallinn University of Technology,* Tallinn*,* June 2004

[35] Maksim Jenihhin, "Test Time Minimization for Hybrid BIST of Systems-on-Chip", *Bachelor thesis, Tallinn University of Technology*, Linköping*,* June 2003

Papers:

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

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

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

[39] R. Ubar, "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

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

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

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

[43] J.Raik, R. Ubar, T. Viilukas, "High-Level Decision Diagram based Fault Models for Targeting FSMs", *Proc. of the 9th IEEE Euromicro Conference on*

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

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

[46] J. Raik, and R. Ubar. "Sequential Circuit Test Generation Using Decision Diagram Models," *Proc. of the DATE Conference*, Munich, Germany, March 9-12, 1999, pp. 736–740

[47] J. Raik, R. Ubar, "DECIDER: A System for Hierarchical Test Pattern Generation", *East-West Design & Test Conference - EWDTC'03, Scientific-Technical Journal Radioelectronics and Informatics, No. 3 (24),* July-September, 2003, pp. 40-45

[48] H.-T. Liaw, C.-S. Lin, "On the OBDD-representation of general Boolean functions", *IEEE Trans. on Computers*, Vol. C-41, No. 6, June 1992, pp. 61-664

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

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

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

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

[53] S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems", *Proc. of the 30th ACM/IEEE DAC*, June 1993, pp. 272-277

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

[55] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, M. Perkowski, "Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams", *Proc. of the 31st ACM/IEEE DAC*, June 1994, pp. 415-419

[56] R. Drechsler, B. Becker, S. Ruppertz, „K*BMDs: a new data structure for verification“, *Proc. of European Design & Test Conf.*, 1996, pp. 2-8

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

[58] M. Aarna, E. Ivask, A. Jutman, E. Orasson, J. Raik, R. Ubar, V. Vislogubov, H.-D. Wuttke. "Turbo Tester - Diagnostic Package for Research and Training", East-West Design & Test Conference - EWDTC'03, Scientific-Technical Journal Radioelectronics and Informatics, No. 3 (24), July - September 2003, pp. 69-73

[59] Anton Chepurov, "Interface between VHDL and High-level Decision Diagram model descriptions", *Master thesis, TUT, Tallinn,* 2008

[60] S. Ben-David, D. Fisman, S. Ruah, "The safety simple subset", *Proc. of IBM Verification Conference*, 2005 *(available at [96])*

[61] IEC 62531 (Edition 1.0, 2007-11), "Standard for Property Specification Language (PSL)", *International Electrotechnical Commission*, 2007 *(available at [97])*

[62] A. Pnueli, "The temporal logics of programs", Proc. of the Annual IEEE Symposium on Foundations of Computer Science (FOCS'77), 1977, pp. 46-57

[63] E.M. Clarke, E.A. Emerson, "Design and synthesis synchronization skeletons using branching time temporal logic", *Proc. of Logic of Programs Workshop, volume 131 of LNCS, Springer*, 1981, pp. 52-71

[64] P. Yeung, K. Larsen, "Practical Assertion-based Formal Verification for SoC Designs", *Proc. of International Symposium on System-on-Chip 2005*, 15-17 Nov. 2005 pp. 58-61

[65] Y. Abarbanel et al., "FoCs: Automatic generation of simulation checkers from formal specifications," *In Computer Aided Verification*, Chicago, USA, 2000, pp. 538-542

[66] Y. Oddos, K. Morin-Allory, D. Borrione, "Prototyping Generators for on-line test vector generation based on PSL properties", *Proc. of IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS '07),* 11-13 April 2007, pp. 1 - 6

[67] S. Gheorghita and R. Grigore, "Constructing Checkers from PSL Properties", *Proc. of 15th International Conference on Control Systems and Computer Science (CSCS15), vol. 2*, 2005, pp. 757–762

[68] D. Bustan, D. Fisman, J. Havlicek, "Automata construction for PSL", *The Weizmann Institute of Science, Technical Report MCS05-04,* May 2005

[69] D. Pidan, S. Keidar-Barner, M. Moulin, D. Fisman, "Optimized algorithms for dynamic verification", *Technical Report Delivery 3.1/1, PROSYD* [101], March 2005

[70] M. Straka, Z. Kotasek, J. Winter, "Digital Systems Architectures Based on On-line Checkers", *Proc. of the 11th Euromicro Conference on Digital System Design (DSD) Architectures, Methods and Tools*, Parma, Italy, September 2008, pp 81-87

[71] M. Boulé and Z. Zilic, "Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties", *Proc. of the 2006 IEEE International High Level Design Validation and Test Workshop (HLDVT'06)*, 2006, pp. 69-76

[72] M. Boule, J.-S. Chenard, Z. Zilic, "Assertion Checkers in Verification, Silicon Debug and In-Field Diagnosis", *Proc. of 8th International Symposium on Quality Electronic Design (ISQED '07),* 26-28 March, 2007, pp. 613-620

[73] K. Morin-Allory, D. Borrione, "Proven correct monitors from PSL specifications", *Proc. of Design, Automation and Test in Europe (DATE'06)* , 2006, pp 1-6

[74] M. Riazati, S. Mohammadi, A. Afzali-Kusha, Z. Navabi, "Improved Assertion Lifetime via Assertion-Based Testing Methodology", *Proc. of International Conference on Microelectronics (ICM '06),* 16-19 December 2006, pp. 48 - 51

[75] M. Kakoee, M. Riazati, S. Mohammadi, "Enhancing the Testability of RTL Designs Using Efficiently Synthesized Assertions", *Proc. of 9th International Symposium on Quality Electronic Design (ISQED 2008)*, 17-19 March 2008, pp. 230 - 235

[76] F. Corno, M.S. Reorda, G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results", *Journal, Design & Test of Computers, IEEE, Volume 17, Issue 3*, July - September 2000, pp. 44 - 53

[77] Mentor Graphics Corporation, "QuestaSim User's Manual", *version 6.1d*, Published on January 16, 2006

[78] Rainer Findenig, "Behavioral Synthesis of PSL Assertions", *Diploma Thesis, Upper Austrian University of Applied Sciences, Austria*, June 2007

[79] I. Pomeranz, S.M. Reddy, "On Generating Tests that Avoid the Detection of Redundant Faults in Synchronous Sequential Circuits with Full Scan", *IEEE Transactions on Computers, Volume 55, Issue 4,* April 2006, pp 491 – 495

[80] A. Schubert, W. Anheier, "On random pattern testability of cryptographic VLSI cores", *Proc. of IEEE European Test Workshop*, May 1999, pp. 15–20

[81] M. Lange, T. Boer, "Effective Functional Verification Methodologies for DO-254 Level A/B and Other Safety-Critical Devices", *White paper, rev. 1.1, Mentor Graphics Corp.,* 2007

[82] RTCA DO-178B / EUROCAE ED-l2B, "Software Considerations in Airborne Systems and Equipment Certification", *RTCA Inc., Washington,* December 1992

[83] ANSI/IEEE Std 1008-1987, "IEEE Standard for Software Unit Testing", *IEEE*, Reaffirmed, December 2, 1993

[84] U.S. Food and Drug Administration, "General Principles of Software Validation", Final Guidance for Industry and FDA Staff, renewed at January 11, 2002

[85] RTCA DO-254 / EUROCAE ED-80 "Design Assurance Guidance for Airborne Electronic Hardware", *RTCA Inc., Washington,* April, 2000

[86] Q. Zhang, I. Harris, "A data flow fault coverage metric for validation of behavioral HDL descriptions", *Proc. of IEEE/ACM International Conference on Computer Aided Design (ICCAD'00)*, 5-9 November, 2000 pp. 369 - 372

[87] J. L. Lions, the chairman of the board "ARIANE 5 Flight 501 Failure", *Report by the Inquiry Board*, Paris, July 19, 1996

[88] Intel Corp., "FDIV Replacement Program", *White paper*, November 30, 1994

Web resources:
(All listed URLs are valid as for the state of October 2008.)

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

[90] Turbo Tester web page,
[*http://www.pld.ttu.ee/tt/*]

[91] Accellera, "Property Specification Language Reference Manual", v1.1, June 9, 2004,
[*http://www.eda.org/vfv/docs/PSL-v1.1.pdf*]

[92] IEEE Std 1850-2005, "IEEE standard for Property Specification Language (PSL)," 2005, (the .pdf file is available at [93])
[*http://www.eda.org/ieee-1850/*]

[93] IEEE Std 1850-2005 at "IEEE Xplore" ,
[*http://ieeexplore.ieee.org/iel5/10222/32588/01524461.pdf*]

[94] PSL/Sugar Consortium webpage
[http://www.pslsugar.org/]

[95] IBM AlphaWorks, "FoCs Property Checkers Generator ver. 2.04",
[*http://www.alphaworks.ibm.com/tech/FoCs/*]

[96] IBM, "The safety simple subset", *web resource*,
[*http://www.haifa.ibm.com/Workshops/verification2005/papers/verification/sss.pdf*]

[97] "IEEE Xplore", database web page,
[*http://ieeexplore.ieee.org*]

[98] IBM, "General Description Language", GDL flavor of PSL, a complementary document to the IEEE Std 1850-2005,
[*http://www.haifa.il.ibm.com/projects/verification/sugar/gdl.ps*]

[99] Project "Sugar", IBM Research,
[*http://www.haifa.ibm.com/projects/verification/sugar/*]

[100] PSL-based Verification Tools, IBM Research,
[*http://www.haifa.ibm.com/projects/verification/sugar/tools.html/*]

[101] EU's 6th Framework Programme research project PROCYD web page,
[*http://www.prosyd.org/*]

[102] ITRS, "International Technology Roadmap for Semiconductors report", 2007 Edition, Design section,
[*http://www.itrs.net/*]

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

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

[105] ITC'99 Benchmarks webpage, Scott Davidson, Sun Microsystems, Inc.,
[*http://www.cerc.utexas.edu/itc99-benchmarks/bench.html*]

[106] Web-articles: "Code Coverage Analysis" and "Minimum Acceptable Code Coverage", Bullseye Testing Technology,
[*http://www.bullseye.com/*]

Appendix A
PPG LIBRARY

This appendix describes the extendable Library of Primitive Property Graphs (PPGs) and is a part one of the main contributions of this thesis. The section consists of the following parts. First, the format of *ppg.lib* file used for THLDD properties Constructor (see Section 3.3.3) is given. Then a set of supported operators is provided, where each operator is presented by its

a) PSL notation based on PSL Standard IEEE 1850 [92]

b) THLDD graph in AGM format (refer to Appendix B)

c) THLDD graph graphic portrayal

d) the operator related notes

The supported set of PSL operators was discussed in Sections 2.2 and 3.3. Loosely speaking, the library mostly includes PSL FL operators of LTL style (vs. SERE style). It conforms with the PSL simple subset rules and supports weak versions (vs. strong) of the operators. Several FL operators such as *abort* and *next_event_e/next_event_a* are not included in this version. PSL operators' precedence together with their classification was presented in Figure 2.17 (Clause 2.2.2.1). The PPG Library is constantly developing which leads to the extension of the supported operators set. Many PSL operators have an equivalent expression by means of other operators.

A.1 Format of the ppg.lib file

There are 2 types for *operand_type* available in *ppg.lib* file format.

- *BOP* – Boolean OPerand, may consist of
    a) Boolean type signal (primary I/O or internal)
    b) Boolean expression, processed by HLDD constructor as VHDL Boolean expression (e.g. comparison operators "<", ">=", etc.)
    c) Boolean operator, processed THLDD Constructor (e.g. logical *and*)

- *TOP* – Temporal OPerand is a complex operand that contains a temporal operator (also includes BOP)

Figure A.1.1 shows the template for the *ppg.lib* file.

```
; PPG Library file
; Version: yy.mm.dd
; Notes: plain text with notes about the current version

operators
{
        List of all described in this file operators together with operands types
        The operators appear in the precedence order.
}

operator_name(operand_type){
        operator graph in the AGM format
}
```

Figure A.1.1 ppg.lib file template

The THLDD graphs representing PSL operators have the same precedence as the original operators. The precedence is specified by the IEEE-1850 [92]. The THLDD Constructor obtains this information related to the supported operators from the *ppg.lib* file's *operators* section.

Figure A.1.2 shows an example of a truncated *ppg.lib* file with two PPGs.

```
; PPG Library file
; Version: 08.07.03
; Notes: This is a truncated version with 2 PPGs

operators
{
    BOP -> TOP;
    next[n] TOP;
}

BOP -> TOP {
    STAT# 5 Nods, 6 Vars, 1 Grps, 2 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "BOP"    <1:0>
    VAR#    1:    (i____)    "TOP"    <1:0>

   ;terminal node constants
    VAR#    2:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    3:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    4:    (c____)    "CHECKING"    <1:0>    VAL = 2

   ;property PPG
    VAR#    5:    (o____)    "PROPERTY"    <1:0>
    GRP#    0:    BEG = 0, LEN = 5 -----
        0    0:     (n___) (0=>4 1=>1 2=>4)    V = 0 "BOP"    <1:0>
        1    1:     (n___) (0=>2 1=>3 2=>4)    V = 1 "TOP"    <1:0>
        2    2:     (____) (    0    0)        V = 2 "FAIL"    <1:0>
        3    3:     (____) (    0    0)        V = 3 "PASS"    <1:0>
        4    4:     (____) (    0    0)        V = 4 "CHECKING"    <1:0>
}

next[k] TOP {
    STAT# 4 Nods, 5 Vars, 1 Grps, 1 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "TOP"    <1:0>

   ;terminal node constants
    VAR#    1:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    2:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    3:    (c____)    "CHECKING"    <1:0>    VAL = 2

   ;property PPG
    VAR#    4:    (o____)    "PROPERTY"    <1:0>
    GRP#    0:    BEG = 0, LEN = 4 -----
        0    0:     (n___) (0=>1 1=>2 2=>3)    V = 0 "TOP"    <1:0> @k
        1    1:     (____) (    0    0)        V = 1 "FAIL"    <1:0>
        2    2:     (____) (    0    0)        V = 2 "PASS"    <1:0>
        3    3:     (____) (    0    0)        V = 3 "CHECKING"    <1:0>
}
```

Figure A.1.2. An example of *ppg.lib* file

A.2 Set of the supported operators

A.2.1 always

a) PSL notation: `FL_Property ::= always FL_Property`

Example: `P: always(TOP);`

Explanation:

An "always" property holds in the current cycle of a given path iff the FL Property that is the operand holds at the current cycle and all subsequent cycles.

b) THLDD graph in AGM format

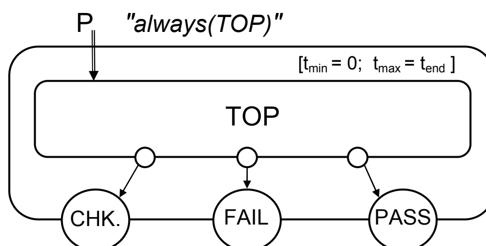
```
always TOP {
    STAT# 4 Nods, 5 Vars, 1 Grps, 1 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "TOP"    <1:0>

   ;terminal node constants
    VAR#    1:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    2:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    3:    (c____)    "CHECKING"    <1:0>    VAL = 2

   ;property PPG
    VAR#    4:    (o____)    "PROPERTY"    <1:0>
    GRP#    0:    BEG = 0, LEN = 4 -----
        0    0:    (n___)  (0=>1 1=>2 2=>3) V = 0 "TOP" <1:0> @[0 to END]_a
        1    1:    (____)  (    0    0)    V = 1 "FAIL"    <1:0>
        2    2:    (____)  (    0    0)    V = 2 "PASS"    <1:0>
        3    3:    (____)  (    0    0)    V = 3 "CHECKING"    <1:0>
}
```

c) THLDD graph portrayal:



d) Notes:

None.

A.2.2 never

a) PSL notation: `FL_Property ::= never FL_Property`

Example: `P: never(BOP);`

Explanation:

A "never" property holds in the current cycle of a given path iff the FL Property that is the operand does not hold at the current cycle and does not hold at any future cycle.

b) THLDD graph in AGM format

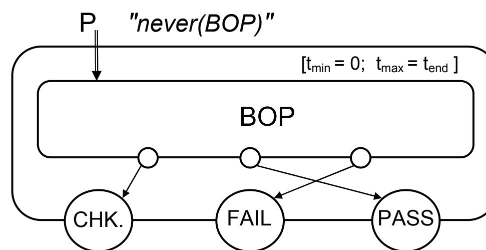
```
never BOP {
    STAT# 4 Nods, 5 Vars, 1 Grps, 1 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "BOP"    <1:0>

   ;terminal node constants
    VAR#    1:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    2:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    3:    (c____)    "CHECKING"    <1:0>    VAL = 2

   ;property PPG
    VAR#    4:    (o____)    "PROPERTY"    <1:0>
    GRP#    0:    BEG = 0, LEN = 4 -----
        0    0:    (n___)  (0=>2 1=>1 2=>3)  V = 0 "BOP" <1:0> @[0 to END]_a
        1    1:    (____) (    0    0)    V = 1 "FAIL"    <1:0>
        2    2:    (____) (    0    0)    V = 2 "PASS"    <1:0>
        3    3:    (____) (    0    0)    V = 3 "CHECKING"    <1:0>
}
```

c) THLDD graph portrayal:



d) Notes:

The operand of a "never" operator is BOP because of the PSL simple subset requirements. It is allowed to be a Sequence, but Sequences are not currently supported. According to good practice of PSL application (refer to [4]), logical implication "->" within an operand of "never" is rarely used. Normally it should be substituted by "and", or a combination of "always" and negation "not" operators should be used instead of "never".

A.2.3 Logical implication

a) PSL notation: `FL_Property ::= FL_Property -> FL_Property`

Example: `P: (BOP -> TOP);`

Explanation:

A logical implication property holds in a given cycle of a given path iff:
- The FL Property that is the left operand does not hold at the given cycle, or
- The FL Property that is the right operand does hold at the given cycle.

b) THLDD graph in AGM format

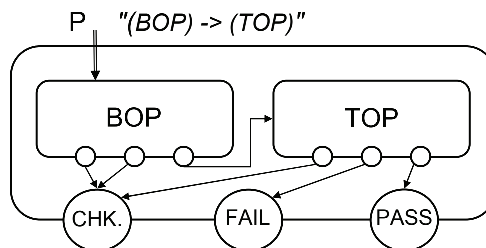
```
BOP -> TOP {
    STAT# 5 Nods, 6 Vars, 1 Grps, 2 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "BOP"    <1:0>
    VAR#    1:    (i____)    "TOP"    <1:0>

   ;terminal node constants
    VAR#    2:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    3:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    4:    (c____)    "CHECKING"    <1:0>    VAL = 2

   ;property PPG
    VAR#    5:    (o____)    "PROPERTY"    <1:0>
    GRP#    0:    BEG = 0, LEN = 5 -----
        0    0:    (n___) (0=>4 1=>1 2=>4)    V = 0 "BOP"    <1:0>
        1    1:    (n___) (0=>2 1=>3 2=>4)    V = 1 "TOP"    <1:0>
        2    2:    (____) (    0    0)    V = 2 "FAIL"    <1:0>
        3    3:    (____) (    0    0)    V = 3 "PASS"    <1:0>
        4    4:    (____) (    0    0)    V = 4 "CHECKING"    <1:0>
}
```

c) THLDD graph portrayal:



d) Notes:

The left-hand side operand of a logical implication is BOP because of the PSL simple subset requirements.

A.2.4 Logical iff

a) PSL notation: `FL_Property ::= FL_Property <-> FL_Property`

Example: `P: (BOP1 <-> BOP2);`

Explanation:

A logical "iff" property holds in a given cycle of a given path iff:
- Both FL properties that are operands hold at the given cycle, or
- Neither of the FL properties that are operands holds at the given cycle.

b) THLDD graph in AGM format

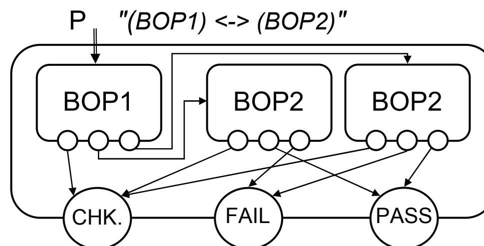
```
BOP1 <-> BOP2 {
    STAT# 6 Nods, 6 Vars, 1 Grps, 2 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "BOP1"    <1:0>
    VAR#    1:    (i____)    "BOP2"    <1:0>

   ;terminal node constants
    VAR#    2:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    3:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    4:    (c____)    "CHECKING"    <1:0>    VAL = 2

   ;property PPG
    VAR#    5:    (o____)    "PROPERTY"    <1:0>
    GRP#    0:    BEG = 0, LEN = 6 -----
        0    0:    (n___) (0=>1 1=>2 2=>5)    V = 0 "BOP1"    <1:0>
        1    1:    (n___) (0=>4 1=>3 2=>5)    V = 1 "BOP2"    <1:0>
        2    2:    (n___) (0=>3 1=>4 2=>5)    V = 1 "BOP2"    <1:0>
        3    3:    (____) (    0    0)    V = 2 "FAIL"    <1:0>
        4    4:    (____) (    0    0)    V = 3 "PASS"    <1:0>
        5    5:    (____) (    0    0)    V = 4 "CHECKING"    <1:0>
}
```

c) THLDD graph portrayal:



d) Notes:

Both operands of a logical "iff" are BOP because of the PSL simple subset requirements.

A.2.5 Logical not

a) PSL notation: `FL_Property ::= NOT_OP FL_Property`

Example: `P: not(BOP);`

Explanation:

A logical "not" property holds in a given cycle of a given path iff the FL property that is the operand does not hold at the given cycle.

b) THLDD graph in AGM format

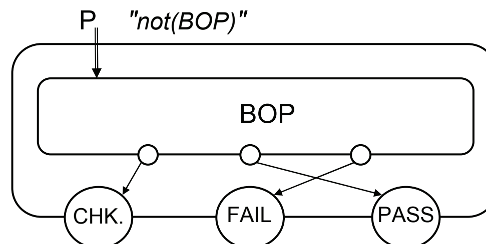
```
not BOP {
    STAT# 4 Nods, 5 Vars, 1 Grps, 1 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "BOP"    <1:0>

   ;terminal node constants
    VAR#    1:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    2:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    3:    (c____)    "CHECKING"    <1:0>    VAL = 2

   ;property PPG
    VAR#    4:    (o____)    "PROPERTY"    <1:0>
    GRP#    0:    BEG = 0, LEN = 4 -----
        0    0:    (n___) (0=>2 1=>1 2=>3)    V = 0 "BOP"    <1:0>
        1    1:    (____) (    0    0)    V = 1 "FAIL"    <1:0>
        2    2:    (____) (    0    0)    V = 2 "PASS"    <1:0>
        3    3:    (____) (    0    0)    V = 3 "CHECKING"    <1:0>
}
```

c) THLDD graph portrayal:



d) Notes:

The operand of a logical "not" operator is BOP because of the PSL simple subset requirements.

A.2.6 Logical and

a) PSL notation: `FL_Property ::= FL_Property AND_OP FL_Property`

Example: `P: (TOP1 and TOP2);`

Explanation:

A logical "and" property holds in a given cycle of a given path iff the FL properties that are operands both hold at the given cycle.

b) THLDD graph in AGM format

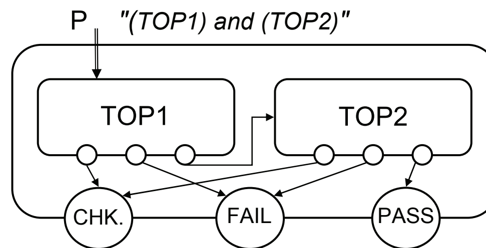
```
TOP1 and TOP2 {
    STAT# 5 Nods, 6 Vars, 1 Grps, 2 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "TOP1"    <1:0>
    VAR#    1:    (i____)    "TOP2"    <1:0>

   ;terminal node constants
    VAR#    2:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    3:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    4:    (c____)    "CHECKING"    <1:0>    VAL = 2

   ;property PPG
    VAR#    5:    (o____)    "PROPERTY"    <1:0>
    GRP#    0:    BEG = 0, LEN = 5 -----
        0    0:    (n___)  (0=>2 1=>1 2=>4)    V = 0 "TOP1"    <1:0>
        1    1:    (n___)  (0=>2 1=>3 2=>4)    V = 1 "TOP2"    <1:0>
        2    2:    (____)  (    0    0)    V = 2 "FAIL"    <1:0>
        3    3:    (____)  (    0    0)    V = 3 "PASS"    <1:0>
        4    4:    (____)  (    0    0)    V = 4 "CHECKING"    <1:0>
}
```

c) THLDD graph portrayal:



d) Notes:

None.

A.2.7 Logical or

a) PSL notation: `FL_Property ::= FL_Property OR_OP FL_Property`
   Example: `P: (BOP or TOP);`
   Explanation:
   A logical "or" property holds in a given cycle of a given path iff at least one of the FL properties holds at the given cycle.

b) THLDD graph in AGM format

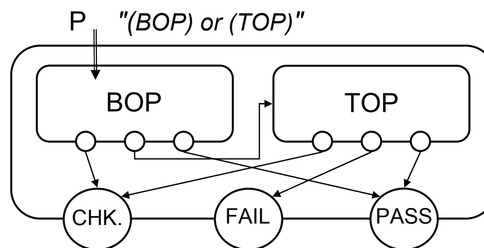
```
BOP or TOP {
    STAT# 5 Nods, 6 Vars, 1 Grps, 2 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "BOP"    <1:0>
    VAR#    1:    (i____)    "TOP"    <1:0>

   ;terminal node constants
    VAR#    2:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    3:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    4:    (c____)    "CHECKING"    <1:0>    VAL = 2

   ;property PPG
    VAR#    5:    (o____)    "PROPERTY" <1:0>
    GRP#    0:    BEG = 0, LEN = 5 -----
        0    0:    (n___) (0=>1 1=>3 2=>4)    V = 0 "BOP"    <1:0>
        1    1:    (n___) (0=>2 1=>3 2=>4)    V = 1 "TOP"    <1:0>
        2    2:    (____) (    0    0)    V = 2 "FAIL"    <1:0>
        3    3:    (____) (    0    0)    V = 3 "PASS"    <1:0>
        4    4:    (____) (    0    0)    V = 4 "CHECKING"    <1:0>
}}
```

c) THLDD graph portrayal:



d) Notes:
   One of the operands of a logical "or" is BOP because of the PSL simple subset requirements.

A.2.8 next

a) PSL notation: `FL_Property ::= next[Number](FL_Property)`

Example: `P: next[k](TOP);`

Explanation:

A next[n] property holds in a given cycle of a given path iff:
- There is not an kth next cycle or
- The FL property that is the operand holds at the kth next cycle

b) THLDD graph in AGM format

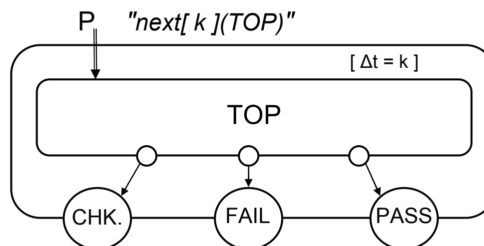
```
next[k] TOP {
    STAT# 4 Nods, 5 Vars, 1 Grps, 1 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "TOP"    <1:0>

   ;terminal node constants
    VAR#    1:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    2:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    3:    (c____)    "CHECKING"    <1:0>    VAL = 2

   ;property PPG
    VAR#    4:    (o____)    "PROPERTY" <1:0>
    GRP#    0:    BEG = 0, LEN = 4 -----
        0    0:    (n___)  (0=>1 1=>2 2=>3)    V = 0 "TOP"    <1:0> @k
        1    1:    (____)  (    0    0)    V = 1 "FAIL"    <1:0>
        2    2:    (____)  (    0    0)    V = 2 "PASS"    <1:0>
        3    3:    (____)  (    0    0)    V = 3 "CHECKING"    <1:0>
}
```

c) THLDD graph portrayal:



d) Notes:

- The number "k" should be positive integer and statically computable. However, "k" is allowed to take value END (see Subsection 3.2.3) in frames of this approach. Value '0' for "k" is generally allowed, "next[0](TOP)" means TOP holds at the current cycle.

- Property "next(TOP)" is equivalent to the property "next[1](TOP)".

- Property "next[k](TOP)" can also be expressed as "next_a[k to k](TOP)"; for BOP "next[k](BOP)" is also expressible by "next_e[k to k](BOP)".

A.2.9 next_a

a) PSL notation: `FL_Property ::= next_a[finite_Range](FL_Property)`

Example: `P: next_a[j to k](TOP);`

Explanation:

A next_a[j to k] property holds in the current cycle of a given path iff the FL property that is the operand holds at all cycles between the jth and k^{th} next cycle, inclusive. (If not all those cycles exist, then the FL Property that is the operand holds on as many as do exist.)

b) THLDD graph in AGM format

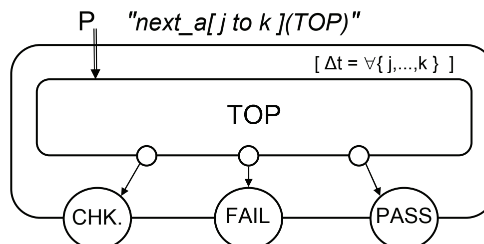
```
next_a[j to k] TOP {
    STAT# 4 Nods, 5 Vars, 1 Grps, 1 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "TOP"    <1:0>

   ;terminal node constants
    VAR#    1:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    2:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    3:    (c____)    "CHECKING"    <1:0>    VAL = 2

   ;property PPG
    VAR#    4:    (o____)    "PROPERTY"    <1:0>
    GRP#    0:    BEG = 0, LEN = 4 -----
        0    0:    (n___)  (0=>1 1=>2 2=>3)  V = 0 "TOP"  <1:0>  @[j to k]_a
        1    1:    (____)  (    0    0)    V = 1 "FAIL"  <1:0>
        2    2:    (____)  (    0    0)    V = 2 "PASS"  <1:0>
        3    3:    (____)  (    0    0)    V = 3 "CHECKING"    <1:0>
}
```

c) THLDD graph portrayal:



d) Notes:

- Both bounds of the range "j" and "k" must be positive integers and statically computable, where "j"<= "k".

- The value of number "j" is allowed to be '0'. In this case the property TOP that is operand of "next_a[0 to k](TOP)" holds starting from the current cycle and for the next "k" cycles.

- The right bound "k" is allowed to take value END (see Subsection 3.2.3) in frames of this approach.

A.2.10 next_e

a) PSL notation: `FL_Property ::= next_e[finite_Range](FL_Property)`

Example: `P: next_e[j to k](BOP);`

Explanation:

A next_e[j to k] property holds in the current cycle of a given path iff:
- There are less than "j" next cycles following the current cycle, or
- There is some cycle between the ith and j^{th} next cycle, inclusive, where the FL property that is the operand holds.

b) THLDD graph in AGM format

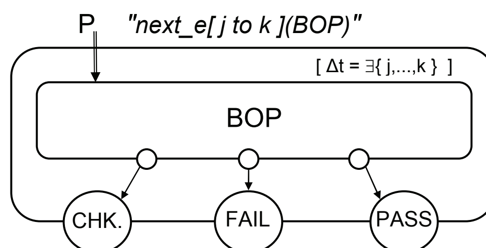
```
next_e[j to k] BOP {
    STAT# 4 Nods, 5 Vars, 1 Grps, 1 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "BOP"    <1:0>

   ;terminal node constants
    VAR#    1:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    2:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    3:    (c____)    "CHECKING"    <1:0>    VAL = 2

   ;property PPG
    VAR#    4:    (o____)    "PROPERTY" <1:0>
    GRP#    0:    BEG = 0, LEN = 4 -----
        0    0:    (n___) (0=>1 1=>2 2=>3) V = 0 "BOP"  <1:0> @[j to k]_e
        1    1:    (____) (    0    0)    V = 1 "FAIL"    <1:0>
        2    2:    (____) (    0    0)    V = 2 "PASS"    <1:0>
        3    3:    (____) (    0    0)    V = 3 "CHECKING"    <1:0>
}
```

c) THLDD graph portrayal:



d) Notes:

- Both bounds of the range "j" and "k" must be positive integers and statically computable, where "j"<= "k".

- The value of number "j" is allowed to be '0'. In this case the property BOP that is operand of "next_e[0 to k](BOP)" holds either in the current cycle or in one of the next "k" cycles. The right bound "k" is allowed to take value END (see Subsection 3.2.3) in frames of this approach.

- The operand of a "next_e" operator is BOP because of the PSL simple subset requirements.

A.2.11 next_event

a) PSL notation: `FL_Property ::=`
`next_event(Boolean)[positive_Number](FL_Property)`

Example: `P: next_event(BOP)[ k ](TOP);`

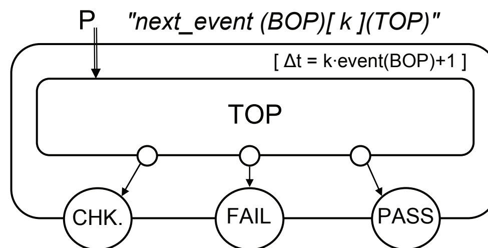
Explanation:

A "next_event[k]" property holds in the current cycle of a given path iff:
- The Boolean expression that is the operand does not hold at least "k" times, starting at the current cycle, or
- The Boolean expression that is the operand holds at least "k" times, starting at the current cycle, and the FL Property that is the operand holds at the "k^{th}" occurrence of the Boolean expression.

b) THLDD graph in AGM format

```
next_event BOP[k] TOP {
    STAT# 4 Nods, 6 Vars, 1 Grps, 2 Inps, 1 Outs, 3 Cons
    VAR#    0:    (i____)    "TOP"    <1:0>
    VAR#    1:    (i____)    "BOP"    <1:0>
   ;terminal node constants
    VAR#    2:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    3:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    4:    (c____)    "CHECKING"    <1:0>    VAL = 2
   ;property PPG
    VAR#    5:    (o____)    "PROPERTY" <1:0>
    GRP#    0:    BEG = 0, LEN = 4 -----
        0    0:    (n___)  (0=>1 1=>2 2=>3) V = 0 "TOP" <1:0> @ k*event BOP
        1    1:    (____) (    0    0)    V = 2 "FAIL"    <1:0>
        2    2:    (____) (    0    0)    V = 3 "PASS"    <1:0>
        3    3:    (____) (    0    0)    V = 4 "CHECKING"    <1:0>
}
```

c) THLDD graph portrayal:



d) Notes:

- The number "k" must be a statically computable positive integer, where $k \geq 1$.

- Property "next_event(BOP)(TOP)" is equivalent to "next_event(BOP)[1](TOP)".

- The formula "next_event('true')(TOP)" is equivalent to the formula "next[0](TOP)". Similarly, if "BOP" holds in the current cycle, then "next_event(BOP)(TOP)" is equivalent to "next_event('true')(TOP)" and therefore to "next[0](TOP)". However, none of these is equivalent to "next(TOP)".

A.2.12 until

a) PSL notation: `FL_Property ::= (FL_Property) until (FL_Property)`

Example: `P: (TOP)until(BOP);`

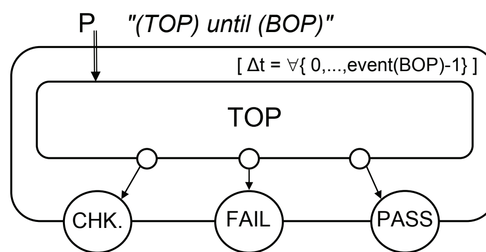
Explanation:

An "until" property holds in the current cycle of a given path iff:
- The FL property that is the left operand holds forever, or
- The FL property that is the right operand holds at the current cycle or at some future cycle, and the FL property that is the left operand holds at all cycles up to, but not necessarily including, the earliest cycle at which the FL Property that is the right operand holds.

b) THLDD graph in AGM format

```
TOP until BOP {
    STAT# 4 Nods, 6 Vars, 1 Grps, 2 Inps, 1 Outs, 3 Cons
    VAR#    0:    (i____)    "TOP"    <1:0>
    VAR#    1:    (i____)    "BOP"    <1:0>
   ;terminal node constants
    VAR#    2:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    3:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    4:    (c____)    "CHECKING"    <1:0>    VAL = 2
   ;property PPG
    VAR#    5:    (o____)    "PROPERTY" <1:0>
    GRP#    0:    BEG = 0, LEN = 4 -----
      0    0:    (n___)  (0=>1 1=>2 2=>3) V = 0 "TOP" <1:0> @[0 to
                                                  event BOP -1]_a
      1    1:    (____) (    0    0)    V = 2 "FAIL"    <1:0>
      2    2:    (____) (    0    0)    V = 3 "PASS"    <1:0>
      3    3:    (____) (    0    0)    V = 4 "CHECKING"    <1:0>
}
```

c) THLDD graph portrayal:



d) Notes:

- The same PPG but with the time window "Δt= ∀{ 0,...,event(BOP)}" corresponds to the PSL operator "(TOP)until_(BOP)"

- The right hand side operand of a "until" operator is BOP because of the PSL simple subset requirements.

A.2.13 before

a) PSL notation: `FL_Property ::= (FL_Property) before (FL_Property)`

Example: `P: (BOP1)before(BOP2);`

Explanation:

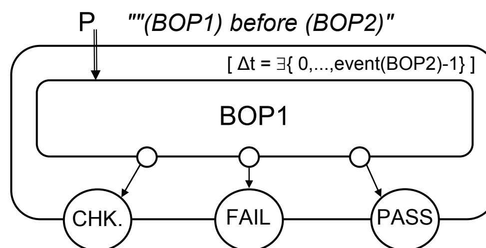
A "before" property holds in the current cycle of a given path iff:
- Neither the FL Property that is the left operand nor the FL Property that is the right operand ever hold in any future cycle, or
- The FL Property that is the left operand holds strictly before the FL Property that is the right operand holds.

b) THLDD graph in AGM format

```
BOP1 before BOP2 {
    STAT# 4 Nods, 6 Vars, 1 Grps, 2 Inps, 1 Outs, 3 Cons

    VAR#    0:    (i____)    "BOP1"    <1:0>
    VAR#    1:    (i____)    "BOP2"    <1:0>
   ;terminal node constants
    VAR#    2:    (c____)    "FAIL"    <1:0>    VAL = 0
    VAR#    3:    (c____)    "PASS"    <1:0>    VAL = 1
    VAR#    4:    (c____)    "CHECKING"    <1:0>    VAL = 2
   ;property PPG
    VAR#    5:    (o____)    "PROPERTY" <1:0>
    GRP#    0:    BEG = 0, LEN = 4 -----
        0    0:     (n___) (0=>1 1=>2 2=>3) V = 0 "BOP1"   <1:0> @[0 to
                                                       event BOP2 - 1]_e
        1    1:     (____) (    0    0)    V = 2 "FAIL"    <1:0>
        2    2:     (____) (    0    0)    V = 3 "PASS"    <1:0>
        3    3:     (____) (    0    0)    V = 4 "CHECKING"    <1:0>
}
```

c) THLDD graph portrayal:



d) Notes:

- The same PPG but with the time window "Δt=∃{ 0,...,event(BOP)}" corresponds to the PSL operator "(BOP1)before_(BOP2)"

- Both operands of a "before" operator are BOP because of the PSL simple subset requirements.

Appendix B
AGM FORMAT

This appendix describes syntax of AGM file format. This format is used to represent the following design representation models proposed and used in TUT:

- SSBDD
- HLDD *(RTL, TLM and behavioural abstraction levels)*
- THLDD

This format is not a contribution of this thesis but rather presented here for explanatory purposes. Only several minor modifications have been introduced to the format to support new THLDD model.

AGM stands for *Alternative Graph model Format*. This abbreviation has historical roots in the first publications of Prof. Raimund Ubar on topic of decision diagrams, where they were referred to as *alternative graphs* (e.g. [38]).

AG model format is case sensitive. It is a line-based format where maximum line length can be 256 characters. In the following the BNF syntax of AG model format is presented. The meta-syntax used obeys the following rules:

1) Syntactic categories (nonterminals) are printed in *italics*; literal words, characters and constants are enclosed to ‘quotes’.
2) If a construct is enclosed to [square brackets], it is optional.
3) If a construct is enclosed to {curly brackets}, it may be repeated zero or more times.
4) A choice is indicated with a vertical bar |.
5) If a construct is enclosed in <chevrons>, it can occur at most once.

B.1 AGM syntax

ag_model *:=*
statistics
mode
[control_signals]
ag_description

statistics *:=*
'STAT#' *natural* 'Nods,' *natural* 'Vars,' *natural* 'Grps,' *natural* 'Inps,' *natural* 'Outs,' *natural* 'Cons' *[*',' *natural* 'Funs'*]* *[*',' *natural* 'Mems'*]* *[*',' *natural* 'C_outs'*]*

The *natural* values reflect the number of nodes, variables, graphs, inputs, outputs, constants, functions, memory arrays and control part outputs, respectively. The number of functions and memory arrays are meaningful in the high-level descriptions. The number of control part outputs is used with the RTL descriptions divided into a control part and a datapath only.

mode *:=*
'MODE#' 'STRUCTURAL' | 'RTL' | 'BEHAVIORAL' | 'TEMPORAL'

Indicates whether a structural gate-level model, a RTL model, a behavioral model, or a temporally extended model is being described. RTL and behavioral models have the following difference: RTL descriptions contain clocking information while behavioral descriptions do not. Other intermediate abstraction levels' models, such as TLM, are logically included in one of these two groups depending on the clocking information presence. Temporally extended model is used for THLDD representation. It differs from the behavioral mode by presence of additional temporal relationships information for properties.

control_signals *:=* 'COUT#*' natural {',' natural}*

Shows the variable indexes of control signal variables. Used in RTL descriptions partitioned to datapath and control parts.

ag_description *:=*

[{input_definition}]
[{memory_definition}]
[{constant_definition}]
[{function_definition}]
[{control_definition}]
{graph_variable_definition}

The definitions are ranged according to the order shown above. There are no memory definitions or function definitions in structural gate-level AG models.

control_definitions are used only in the RTL descriptions partitioned into control and datapath parts.

input_definition *:=*
'VAR#' *var_index* ':' '(' *variable_flags* ')' *var_name var_range*

Defines a primary input of the model.

memory_definition *:=*
'VAR#' *var_index* ':' '('*variable_flags*')' *var_name var_range [row_range]*
column_range
memory_row
{memory_row}

Defines a memory array. The optional *row_range* is used with two-dimensional arrays, and it determines the range of row addresses used in memory. In one-dimensional arrays, *row_range* is omitted. In similar way, *column_range* determines the range of column addresses used in the memory variable.

memory_row *:=* '{' *integer {*',' *integer}* '}'

Defines the contents of a memory variable. The number of integers in *memory_row* is determined by *column_range.*

row_range *:= mem_range*

Row_range is used with two-dimensional arrays, and it determines the range of row addresses used in memory. In one-dimensional arrays, *row_range* is omitted.

column_range *:= mem_range*

Determines the range of column addresses used in the memory variable.

mem_range *:=* '[' *integer* '–' *integer* ']'

In *mem_range* the first integer must be less than the second one.

constant_definition *:=*
'VAR#' *var_index* ':' '('*variable_flags*')' *var_name var_range* 'VAL' '=' *integer*

Defines a constant. The integer value shows the value of the constant.

function_definition *:=*
'VAR#' *var_index* ':' '('*variable_flags*')' *var_name var_range*
'FUN#' *function_type arguments_definition*

Defines an operation or function.

function_type *:= identifier*

Shows the type of the operation.

arguments_definition *:=* '(' *[argument] {*',' *argument}* ')'

Defines the arguments (if any) of an operation.

argument *:=* ‘A’*argument_index* ‘<=’ *argument_variable range*

The *range* shows the bit-slice of the variable *argument_variable* that is used as a function argument.

argument_index *:= natural*

Shows the index of the function argument.

argument_variable *:= natural*

Shows the index of the variable used as the function argument.

control_definition *:=*
‘VAR#’ *var_index* ‘:’ ‘(‘ *variable_flags* ’)’ *var_name var_range*

+

Defines a control signal. Used to define control part output signals of the RTL designs partitioned into datapath and control parts.

graph_variable_definition *:=*
‘VAR#’ *var_index* ‘:’ ‘(‘*variable_flags*’)’ *var_name var_range*
graph_definition

Defines a variable for which a graph corresponds.

graph_definition *:=*
‘GRP#’ *graph_index* ‘:’ ‘BEG’ ‘=’ *natural* ‘,’ ‘LEN’ ‘=’ *natural* ‘-----’
node_definition | parallel_node_definition
{node_definition | parallel_node_definition}

Defines a graph in the AG model. The ‘BEG=’ construct shows the absolute index of the first node in the graph. The ‘LEN=’ construct in turn shows the number of nodes in the graph.

node_definition *:=*
nod_abs_index nod_index ‘: (‘*nod_flags*’) (’ *successors* ‘) V =’ *nod_var nod_name nod_range*

Defines an AG node. *nod_abs_index* and *nod_index* represent the absolute (inside the model) and relative (inside the graph) indexes of the node. Construct *successors* shows the successor nodes of current node which are chosen with different node values. Index of the variable labeling the node is determined with *nod_var.*

parallel_node_definition *:=*
nod_abs_index nod_index ‘: (v___)’ ‘(‘ ‘0’ ‘0’ ‘)’ ‘VEC =’ *nod_var_vector*

Defines a terminal node of the FSM graph of RTL description. *nod_abs_index* and *nod_index* represent the absolute (inside the model) and relative (inside the graph) indexes of the node, respectively. Indexes of the variables labeling the node are determined with *nod_var_vector.*

nod_var_vector *:= ‘ ”’ state_value {signal_value} ’ ”’*

state_value shows the value of the next state. The *signal_value* constructs show the values of the control signals defined in the *control_signals* construct.

state_value *:= natural*

Shows the value of the next state.

signal_value *:=* ‘0*’* | ‘1*’* | ‘X*’*

The *signal_value* constructs show the values of the control signals defined in the *control_signals* construct.

nod_var *:= natural[[* ‘[‘ ‘V’ ‘=’ *row_index* ’]’ *]* ‘[‘ ‘V’ ‘=’ *column_index* ’]’ *]*

Shows the index of the variable labeling the node. Optional constructs *row_index* and *column_index* are used with memory variables labeling the node. These constructs determine the indexes of the variables used for addressing rows and columns, respectively.

nod_name *:= string*

Shows the name of the node.

nod_range *:= range*

nod_range determines the bit-slice of the variable that labels the node. AG model format allows slices of variables to be used for labeling a node.

row_index *:= natural*

Determines the indexes of the variables used for addressing rows of the memory variable.

column_index *:= natural*

Determines the index of the variable used for addressing columns of the memory variable.

nod_abs_index *:= natural*

Shows the absolute (inside the model) index of the node.

nod_index *:= natural*

Shows the relative (inside the graph) index of the node inside the graph.

graph_index *:= natural*

Shows the index of the graph.

variable_flags *:=* < ‘i’ | ‘m’ | ‘c’ | ‘f’ | ‘o’ | ‘n’ | ’_’ | ‘F’ > *{*<‘d’> | ‘_’*}*

The variable flags have the following interpretation:
‘i’ - input variable

‘m’ - memory variable (RTL, behavioral)
‘c’ - constant variable
‘f’ - function variable (RTL, behavioral)
‘o’ - output variable
‘d’ - clock cycle delay, e.g. in registers, flipflops. (Gate-level, RTL)

The following flags are used with RTL descriptions only:
‘n’ - control part output signal
‘F’ - FSM graph variable
‘r’ - reset variable
‘s’ - state variable

nod_flags *:=* < ‘i’ | ’_’ > *{* ‘n’ | ‘v’ | ‘_’*}*

The node flags have the following interpretation:
‘i’ - inverted node (in gate-level descriptions only)
‘n’ - non-terminal node (RTL, behavioral)
‘v’ - control part terminal node (RTL)

successors *:=*
nonterminal_successors | *terminal_successor* | *boolean_successors*

Construct *successors* shows the successor nodes of current node which are chosen with different node values.

nonterminal_successors *:=*
node_values ‘=>’ *successor_index* *{node_values* ‘=>’ *successor_index }*

This construct shows the indexes of successor nodes which will be selected with corresponding node values. (Used with RTL and behavioral models only).

terminal_successors *:= ‘0’ ‘0’*

Terminal nodes are nodes which have no successor nodes.

boolean_successors*:= natural natural*

This type of construct can be used with Boolean AGs only. The first natural number indicates the relative index of the successor node when the value of current node is ‘0’, and the second number shows the relative index of the successor node when current node is ‘1’, respectively. If the index of the successor node is ‘0’, it shows that there is no successor nodes to current node with corresponding value.

node_values *:= natural {* ‘,’ | ‘–’ *natural}*

Determines the set of node values that activate the corresponding branch. The comma ‘,’ character is used for separating the indexes; the minus sign ‘-‘ is used for index ranges, e.g. ‘3-5’, which can be alternatively written as ‘3,4,5’.

successor_index *:= natural* | ‘X’

If *successor_index* is a natural number, it shows the index of the successor node. Otherwise, if *successor_index* is ‘X’, it means that the successor is undetermined.

var_index *:= natural*

Shows the index of the variable.

var_name *:= string*

Shows the name of the variable.

var_range *:= range*

Shows the bitwidth of the variable.

range *:= [* ‘<’ *natural* ‘:’ *natural* ‘>’ *]*

Range is a construct for describing bit-vectors. The first natural shows the index of the most significant bit and the latter is for the least significant bit, respectively. If range is omitted, it will default to ‘<0:0>’.

string *:=* ‘ ” ’ *{character}* ‘ “ ’

Character can be any character, except newline and double quote ‘ ” ’.

integer *:=*
[‘–’]natural
Any integer number.
natural

Natural can be any non-negative number.

identifier *:=*
alphabetic_character{alphabetic_character | *digit* | ‘_’*}*
alphabetic_character := ‘A’| ...| ’Z’ | ‘a’ | ...| ‘z’
digit := ‘0’ | ‘1’ | ...| ‘9’

Curriculum Vitae in English

Personal data

Name	Maksim Jenihhin
Date and place of birth	09.06.1981, ESTONIA
Citizenship	Estonian

Contact data

Address	Raja 15, Tallinn 12618, ESTONIA
Phone	+372 620 2262
E-mail	maksim@computer.org

Education

2004 - ...	Ph.D. student in Computer Engineering, Tallinn University of Technology
2003 - 2004	M.Sc. in Computer Engineering, TUT
1999 - 2003	B.Sc. in Computer Engineering, TUT
1988 - 2003	1999, Secondary Education from Russian High School of Jõhvi

Career

2007 - ...	Tallinn University of Technology, Faculty of Infotechnology, Dept. of Computer Engineering Chair of Computer Engineering and Diagnostics, Researcher
2004 - 2007	ELIKO Ltd. Competence Centre in Electronics-, Info- and Communication Technologies, Development Engineer
2003 - 2004	Borthwick-Pignon Solutions Ltd., Test Engineer

Honours & Awards

Ustus Agur grant, Estonian Association of Information Technology and Telecommunications (ITL), May 16, 2007

"Tiger University" grant for ICT PhD students, Estonian Information Technology Foundation (EITSA), 2007

AS Eesti Energia grant, Development fund of TUT, November 22, 2007

AS Tallinna Sadam grant, Development fund of TUT, November 22, 2006

Jaan Poska grant, Tallinn Council, May 15, 2006

"Tiger University" grant for ICT PhD students, Estonian Information Technology Foundation (EITSA), 2006

Jaan Poska grant, Tallinn Council, May 15, 2005

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

Yearly Award, Natural Sciences and Engineering - Maksim Jenihhin; Estonian National Contest for Young Scientists II Prize, 2004

Curriculum Vitae
Eesti keeles

Isikuandmed

Nimi	Maksim Jenihhin
Sünniaeg ja -koht	09.06.1981, EESTI
Kodakondsus	Eesti

Kontaktandmed

Aadress	Raja 15, Tallinn 12618, EESTI
Telefon	+372 620 2262
E-post	maksim@computer.org

Hariduskäik

2004 - ...	doktorant, Arvutitehnika Instituut, Tallinna Tehnikaülikool
2003 - 2004	tehnikateaduste magister, Arvuti- ja Süsteemitehnika eriala, Tallinna Tehnikaülikool
1999 - 2003	tehnikateaduste bakalaureus, Arvuti- ja Süsteemitehnika eriala, Tallinna Tehnikaülikool
1988 - 2003	keskharidus, Jõhvi Vene Gümnaasium

Teenistuskäik

2007 - ...	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Arvutitehnika instituut, Arvutitehnika- ja diagnostika õppetool, teadur
2004 - 2007	ELIKO OÜ Tehnoloogia Arenduskeskus, arendusinsener
2003 - 2004	Borthwick-Pignon OÜ, testinsener

Teaduspreemiad ja -tunnustused

ITL-i Ustus Aguri nimiline stipendium, 16. mai 2007.a

"Tiigriülikooli" stipendium IKT doktorantidele (EITSA), 2007.a

AS Eesti Energia stipendium, TTÜ arengufond, 22. november 2007.a

AS Tallinna Sadam stipendium, TTÜ arengufond, 22. november 2006.a

Jaan Poska stipendium, Tallinna Linnavalitsus, 15. mai 2006.a.

"Tiigriülikooli" stipendium IKT doktorantidele (EITSA), 2006.a

Jaan Poska stipendium, Tallinna Linnavalitsus, 15. mai 2005.a.

"Tiigriülikooli" stipendium IKT doktorantidele (EITSA), 2005.a

Aasta preemia, Loodusteadused ja tehnika - Maksim Jenihhin; Üliõpilaste teadustööde riikliku konkursi II preemia, 2004.a