THESIS ON INFORMATICS AND SYSTEM ENGINEERING C87

Hardware Modeling for Design Verification and Debug

ANTON TŠEPUROV

TUT PRESS

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

Dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer and Systems Engineering on May 13, 2013.

Supervisors: Prof. Jaan Raik,
Dr. Maksim Jenihhin
Department of Computer Engineering, TUT

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

Dr. Giovanni Squillero
Politecnico di Torino, Italy

Defence of the thesis: June 14, 2013

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

/Anton Tšepurov/

Copyright: Anton Tšepurov, 2013
ISSN 1406-4731
ISBN 978-9949-23-478-3 (publication)
ISBN 978-9949-23-479-0 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C87

Riistvara modelleerimine disaini verifitseerimise ja silumise jaoks

ANTON TŠEPUROV

TTÜ
KIRJASTUS

To my parents Alviina and Nikolai
who are responsible for the revival of this thesis

Abstract

Modern hardware design process relies entirely on CAD (Computer-Aided Design) tools. The main difficulties faced today by the EDA (Electronic Design Automation) industry and academia are the enormous scale and complexity of digital designs. This brings to the front the scalability property of design representation models and their ability to represent designs in the most efficient way for the given design task. Despite all the EDA efforts, the design productivity gap persists and manifests designers' inability to harness all that technology allows. Furthermore, verification productivity gap does not allow verification engineers to even ensure a 100% correctness of what yet can be designed.

This thesis contributes to closing these gaps using two powerful design representation models. Designers' productivity is improved through an automated debug method based on a scalable Instantiation Graph (IG) model. Researchers' capabilities are enriched through a scalable IG-based open source platform for hardware design and verification. It offers a solid ground for building other EDA tools and experimental environments. Verification accuracy is further improved through stringent code coverage measurement and fast assertion checking methods based on High-Level Decision Diagrams (HLDD) model and its modifications. If HLDD model suits best for academic research, then IG model is required for experimenting on large industrial designs and has also found industrial application.

First, we present several HLDD model modifications and extensions that target verification accuracy and speed. The thesis includes a methodology for automated generation of behavioral and structural HLDD models from VHDL descriptions. Different HLDD compactness levels are investigated with respect to code coverage measurement accuracy. Certain compactness levels are found to provide more stringent coverage metrics than classical metrics based on hardware description languages (HDLs). Different conditional statements representations by HLDDs are considered: some are found beneficial for condition coverage measurement, others — for automated bug location using statistical methods. The concept of mutation testing on HLDDs is discussed and shown to provide a better estimation of test quality than the code coverage approach. Temporally extended HLDD (THLDD) model is introduced and shown to provide faster verification assertion

checking than using an HDL-based state-of-the-art assertion checker. Corresponding methodology for THLDD generation from PSL assertions is also provided. Finally, a unified HLDD-based verification framework APRICOT is presented along with its implementation as ApricotCAD.

Second, to address the scalability problem unsolved in ApricotCAD, the thesis introduces IG model for scalable design representation and a methodology for comprehensive HDL descriptions elaboration into the IG model. The scalability property of the model is investigated on a system on chip consisting of 3584 LEON3 processor cores, which is far ahead of even tomorrow's scale. zamiaCAD scalable open source platform for design and verification powered by IG model is presented in detail. Tools for static code analysis and VHDL simulation based on zamiaCAD platform are described. Finally, a scalable automated method for precise bug localization in industrial-sized designs is also presented. It uses the both above mentioned tools and is experimentally proven to be particularly useful and accurate in debugging complex conditional statements in large industrial designs of today's scale.

The two proposed models and their derivatives offer wide possibilities for research and industrial application. The models can also be merged together to achieve promising synergic effect discussed shortly in the concluding chapter.

Kokkuvõte

Kaasaegne riistvaraarendus põhineb täielikult CAD (Computer-Aided Design) vahenditel. Tänapäevaste digitaalsete skeemide ülikõrge suurus ja keerukusaste on saanud põhilisteks raskusteks elektroonika arendus-automatiseerimistööstuse (EAA) ja vastavate teaduslike uuringute jaoks. See tõstab esiplaanile riistvara esitusmudelite skaleeritavusvõimet ja nende täpset sobivust ja efektiivsust riistvara esitusel antud arendusülesande täitmise kontekstis. Vaatamata EAA saavutustele, jätkuvalt säilib arendusvahendite kesisus, mis ei luba arendajatel realiseerida kõike, mida tehnoloogia parasjagu teha lubab. Ega ka verifitseerimisvahendite puudus luba tõestada selle 100% korrektsust, mis ikkagi realiseerida annab.

Antud väitekirja panuseks ongi nende kahe puuduse kõrvaldamine läbi kahe võimsa riistvara esitusmudeli kaasamise. Arendajate tootlikkus suurendatakse automaatse silumismeetodiga, mis põhineb hästi skaleeritaval eksemplar graaf (EG) mudelil. Uurijate võimalusi samuti rikastatakse läbi avatud lähtekoodiga riistvara arendamiseks ja verifitseerimiseks mõeldud laiendatava EG-põhise platvormi. Selle põhjal annab luua teisi EAA vahendeid ja skaleeritavaid eksperimentaalseid keskkondasid. Suurem täpsus ja kindlus verifitseerimisel saavutatakse range koodikatte mõõtmise ja kiire väidete kontrolli meetoditega, mis põhinevad kõrgtaseme otsustusdiagrammide (KTOD) mudelil ja selle laienditel. Kui KTOD mudel on eriti sobilik akadeemiliste uuringute läbiviimiseks, siis EG mudel on vajalik, et teha katseid suurte skeemidega, ja seega on leidnud rakendust tööstuses.

Esiteks me esitame mitmeid KTOD mudeli modifikatsioone ja laiendeid täpsuse ja kiiruse tõstmiseks verifitseerimisel. Väitekirjas tuuakse metodoloogia käitumuslike ja struktuursete KTOD mudelite automaatseks genereerimiseks VHDL kirjeldustest. Käsitletakse erinevaid KTOD kompaktsustasemeid. On leitud, et mõned neist annavad rangema koodikatte mõõtmise tulemuse võrreldes klassikaliste riistvara kirjelduskeeltel (RKK) põhinevate meetrikatega. Erinevad tingimuslausete esitusviisid on vaadeldud: mõned on osutunud kasulikeks tingimuskatte mõõtmiseks, teised — automaatseks vigade avastamiseks statistiliste meetoditega. On uuritud mutatsioontestimist KTOD mudeli peal ja see on osutunud koodikatte mõõtmisest veelgi paremaks testi kvaliteedi hindajaks. Ajalise laiendiga

KTOD (AKTOD) mudel on esitatud ja saavutavat kiirema väidete kontrolli kui lubab kaasaegne RKK-põhine väidete kontrollija. Vastav metodoloogia AKTOD genereerimiseks PSL väidetest on samuti lisatud. Lõpuks esitatakse ühtne KTOD-põhine verifitseerimiskeskkond APRICOT koos seda realiseeriva ApricotCAD rakendusega.

Teiseks, et pakkuda lahendust skaleerimise probleemile, mis jäi ApricotCADis lahendamata, väitekiri esitab EG mudeli skaleeritavaks disaini esituseks ja vastava metodoloogia selle mudeli saamiseks läbi põhjaliku RKK kirjelduste teisendamise. Mudeli skaleeritavusvõime hindamiseks kasutatakse 3584 LEON3 protsessor tuumast koosnevat kiipsüsteemi, mis vastab isegi kaugema kui homse päeva tööstuslikule ulatusele. Järgmisena esitatakse avatud lähtekoodiga skaleeritav zamiaCAD platvorm, mis põhineb EG mudelil ja on mõeldud riistvara arendamise ja verifitseerimise hõlbustamiseks. Kirjeldatud on kaks zamiaCAD põhjal loodud rakendust, millisteks on staatiline koodi analüsaator ja RKK VHDL kirjelduste simulaator. Lõpuks ka avaldatakse skaleeritav silumismeetod täpseks ja automaatseks disainivigade lokaliseerimiseks, milles kasutatakse mõlemat ülalmainitud rakendust. Järgnevas eksperimentaalses osas näidatakse meetodi erilist efektiivsust ja täpsust vigade avastamisel keerulistes tingimuslausetes ja selliste disainide suuruste puhul, mis vastavad tänapäeva tööstuslikule ulatusele.

Pakutud mudelid koos nende laienditega ja neist tuletatud teiste mudelitega on heaks aluseks laiale hulgale teaduslikele uuringutele ja ka tööstuses kasutamiseks. Ka mudelite kokkuviimine annab saavutada paljulubavat sünergilist efekti, mida arutatakse lühidalt väitekirja viimases peatükis.

Acknowledgements

I only have one chance to thank everyone who has contributed to this thesis, so here I take it.

First and foremost, I would like these leaves to constantly deliver words of gratitude to Prof. Raimund-Johannes Ubar, Prof. Jaan Raik and Dr. Margus Kruus. It is for the professional work of your team that I had been attracted to scientific research in the first place, supported diligently throughout my stay and had a chance to contribute and meet the top players in the field, both at the international conferences and here, in Tallinn. Your Teamwork is outstanding!

This work of mine would never exist without the generous advice and guidance of my supervisors Dr. Maksim Jenihhin and Prof. Jaan Raik. You have endowed me with a frame of reference for what the intense and fruitful teamwork should be like. You have taught me that hypotheses are nets: only he who casts will catch. And that to achieve a lot, one must do a lot. You were the Super-Wisers!

During the years, Günter Bartsch has shown me what it is to be a true engineer. The mastermind behind zamiaCAD, its founder and a brilliant versatile engineer, you have virtually shaped and inspired a half of this thesis. Herein I express my deepest gratitude to you and my anticipation of further collaboration. I would also like to thank Rainer Dorsch for all your valuable contributions and particularly for arranging our acquaintance with Günter Bartsch. You both played quite a Role!

I am also happy to have a pleasure of bringing up the rear of PhD defenses of the four close friends who have once started their BSc studies together: Dr. Anton Tšertov, Dr. Sergei Kostin and Dr. Igor Aleksejev. There was always more than just studies. While writing this thesis, I was constantly looking back at your own, fellows, work to make sure I remain on a par. We shall never forget that environment creates a man. And a man creates the Environment.

Dedicated gratitude goes to all the organizations that supported my PhD studies: Tallinn University of Technology, European Commission FP6 research project VERTIGO, European Commission FP7 research project DIAMOND, Centre of Research Excellence in Dependable Embedded Systems (CREDES), National Graduate School in Information and Communication Technologies (IKTDK) and

Estonian IT Foundation (EITSA). It is for your Support that new generations of researchers and citizens can concentrate on learning about the mistakes of the past and the experience of the past, so that the former would not be repeated, while the latter would be employed to the fullest extent possible.

Here I want it to be explicitly stated that it is not what I have learned that I value most, but the people I got acquainted with and the processes I observed during my stay at TUT.

Last but not least, I bring here my warmest thanks to my family who are at the core of my Universe, even if not always obvious. Alviina, Nikolai, Kirill, all I do is ultimately for you to be proud of. You are the endless source of Inspiration and Wisdom! And there were days when I did not Exist at all, to say nothing of the thesis.

It is also not a tradition to thank your beloved, but a great desire, a pleasant duty and an inevitable felicity one cannot help. I would love to thank my dear fiancée Olga for all your mildness and love, encouragement and help, which made my days both easier and brighter. You are my All!

Thank you!

Anton Tšepurov
Tallinn, May 2013

List of Publications

- V. Tihhomirov, A. Tšepurov, M. Jenihhin, J. Raik, R. Ubar, "Assessment of Diagnostic Test for Automated Bug Localization", *14th Latin American Test Workshop (LATW'13)*, April 3–5, 2013, Cordoba, Argentina, pp. 1–6.
- M. Jenihhin, A. Tšepurov, V. Tihhomirov, J. Raik, H. Hantson, R. Ubar, G. Bartsch, J. H. Meza Escobar, H. D. Wuttke, "Automated Bug Localization in Processor Designs", *IEEE Design & Test of Computers Magazine*, 2013, [Accepted].
- J. Raik, U. Repinski, A. Tšepurov, H. Hantson, R. Ubar, M. Jenihhin, "Automated design error debug using high-level decision diagrams and mutation operators", *Microprocessors and Microsystems Journal*, 2013, 37(4), pp. 1–10.
- A. Tšepurov, V. Tihhomirov, M. Jenihhin, J. Raik, G. Bartsch, J.-H. Meza Escobar, H.-D. Wuttke, "Localization of Bugs in Processor Designs Using zamiaCAD Framework", *Microprocessor Test and Verification (MTV'12)*, December 10–12, 2012, Austin, Texas, USA, pp. 1–6.
- A. Tšepurov, G. Bartsch, R. Dorsch, M. Jenihhin, J. Raik, V. Tihhomirov, "A Scalable Model Based RTL Framework zamiaCAD for Static Analysis". *IFIP/IEEE 20th International Conference on VLSI and System-on-Chip (VLSI-SoC)*, October 7-10, 2012, Santa Cruz, USA, pp. 171–176.
- J. Raik, T. Drenkhan, M. Jenihhin, T. Viilukas, A. Karputkin, A. Tšepurov, R. Ubar, "Generating Directed Tests for C Programs using RTL ATPG", *Proceedings of the IEEE 13th Workshop on RTL and High Level Testing (WRTLT'12)*, Niigata, Japan, November 22-23, 2012, pp. 1–6.
- J. Raik, U. Repinski, M. Jenihhin, A. Chepurov, "High-Level Decision Diagram Simulation for Diagnosis and Soft-Error Analysis". *Book chapter in "Design and Test Technology for Dependable Systems-on-Chip", Information Science Reference (IGI Global)*, *Ed. R. Ubar,* USA, 2011, pp. 294–309.

- M. Jenihhin, J. Raik, A. Chepurov, R. Ubar, “Application of High-Level Decision Diagrams for Simulation-Based Verification tasks”. *Estonian Journal of Engineering*, 2010, 16(1), pp. 56–77.
- J. Raik, U. Repinski, R. Ubar, M. Jenihhin, A. Chepurov, “High-level design error diagnosis using backtrace on decision diagrams”. *28th Norchip Conference (NORCHIP’10)*, November 15-16, 2010, Tampere, Finland, pp. 1–4.
- H. Hantson, J. Raik, G. Di Guglielmo, M. Jenihhin, A. Chepurov, F. Fummi,R. Ubar, “Mutation Analysis with High-Level Decision Diagrams”. *11th Latin-American Test Workshop (LATW’10)*, March 28-31, 2010, Punta del Este, Uruguay. IEEE Computer Society Press, pp. 1–6.
- R. Ubar, A. Jutman, J. Raik, S. Devadze, I. Aleksejev, A. Chepurov, A. Chertov, S. Kostin, E. Orasson, H.-D. Wuttke, “E-Learning Environment for WEB-Based Study of Testing”. *Proc. of the 8th European Workshop on Microelectronics Education (EWME’10)*. Darmstadt, Germany, pp. 47–52.
- M. Jenihhin, J. Raik, A. Chepurov, R. Ubar, “PSL Assertion-Checking Using Temporally Extended High-Level Decision Diagrams”. *Journal of Electronic Testing - Theory and Applications (JETTA)*, 25(6), 2009, pp. 289–300.
- M. Jenihhin, R. Raik, A. Chepurov, U. Reinsalu, R. Ubar, “High-Level Decision Diagrams based Coverage Metrics for Verification and Test”. *Proceedings of 10th IEEE Latin American Test Workshop (LATW’09)*, 2009, pp. 1–6.
- M. Jenihhin, J. Raik, A. Chepurov, U. Reinsalu, R. Ubar, “Code Coverage Analysis for Concurrent Programming Languages Using High-Level Decision Diagrams”. *Proceedings of the 12th European Workshop on Dependable Computing (EWDC’09)*, May 14-15, 2009, Toulouse, France, pp. 1–4.
- M. Jenihhin, J. Raik, A. Chepurov, R. Ubar, “Simulation-based Verification with APRICOT Framework using High-Level Decision Diagrams”. *East-West Design & Test Symposium (EWDTS’09)*, September 18-21, 2009, Moscow, pp. 13–16.
- J. Raik, M. Jenihhin, A. Chepurov, U. Reinsalu, R. Ubar, “APRICOT: a Framework for Teaching Digital Systems Verification”. *19th EAEEIE Annual Conference*, 2008, pp. 172–177.
- A. Chepurov, G. Di Guglielmo, F. Fummi, G. Pravadelli, J. Raik, R. Ubar, T. Viilukas, “Automatic Generation of EFSMs and HLDDs for Functional ATPG”. *11th Biennial Baltic Electronics Conference (BEC’08)*, 2008, pp. 143–146.

- M. Jenihhin, J. Raik, R. Ubar, A. Chepurov, "On reusability of verification assertions for testing". *11th Biennial Baltic Electronics Conference (BEC'08)*, 2008, pp. 151–154.
- K. Minakova, U. Reinsalu, A. Chepurov, J. Raik, M. Jenihhin, R. Ubar, P. Ellervee, "High-Level Decision Diagram Manipulations for Code Coverage Analysis". *11th Biennial Baltic Electronics Conference (BEC'08)*, 2008, Tallinn, Estonia, pp. 207–210.
- A. Jutman, A. Tsertov, A. Tsepurov, I. Aleksejev, R. Ubar, H.-D. Wuttke, "Teaching Digital Test with BIST Analyzer". *19th EAEEIE Annual Conference*, June 29 - July 2, 2008, Tallinn, Estonia, pp. 123–128.
- M. Jenihhin, J. Raik, A. Chepurov, R. Ubar, "Temporally Extended High-Level Decision Diagrams for PSL Assertions Simulation". *Proceedings of the 13th IEEE European Test Symposium (ETS'08)*, 2008, Los Alamitos, USA. IEEE Computer Society Press, pp. 61–68.
- J. Raik, R. Ubar, M. Jenihhin, A. Chepurov, "PSL Assertion Checking with Temporally Extended High-Level Decision Diagrams". *9th IEEE Latin American Test Workshop (LATW'08)*, February 17-20, 2008, Puebla, Mexico, pp. 49–54.
- M. Jenihhin, J. Raik, A. Chepurov, R. Ubar, "Assertion Checking with PSL and High-Level Decision Diagrams". *Proceedings of the IEEE 8th Workshop on RTL and High Level Testing (WRTLT'07)*, 2007, Beijing, China, IEEE Computer Society Press.
- A. Jutman, A. Tsertov, A. Tsepurov, I. Aleksejev, R. Ubar, H.-D. Wuttke, "BIST Analyzer: a Training Platform for SoC Testing". *Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports, 2007. 37th Annual Frontiers in Education Conference (FIE'07)*, October 10-13, 2007, Milwaukee, USA. IEEE Computer Society Press, pp. 1534–1539.
- V. Tihhomirov, A. Chepurov, S. S. Abrar, M. Jenihhin, J. Raik, "VHDL Design Debug Framework Based on zamiaCAD". *In University Booth section of Design, Automation and Test in Europe*, Grenoble, France, March 18-22, 2013. p. 1.
- A. Chepurov, V. Tihhomirov, S. A. Syed, M. Jenihhin, J. Raik, "Applications of the Open Source HW Design Framework zamiaCAD". *In University Booth section of Design, Automation and Test in Europe (DATE'12)*, March 12-16, 2012, Dresden, Germany, p. 1.

- A. Tsepurov, M. Jenihhin, J. Raik, "zamiaCAD: Open Source Platform for Advanced Hardware Design". *In University Booth section of Design, Automation and Test in Europe (DATE'11)*, March 14-18, 2011, Grenoble, France, pp. 1–2.
- A. Tsepurov, M. Jenihhin, J. Raik, "Simulator for ZamiaCAD Integrated Hardware Design Environment". *In University Booth section of Design, Automation and Test in Europe (DATE'10)*, March 8-12, 2010, Dresden, Germany, pp. 1–2.
- A. Chepurov, "Interface between VHDL and High-level Decision Diagram model descriptions", Master thesis, Tallinn University of Technology, Tallinn, June 2008, pp. 1–74.
- A. Chepurov, "Hybrid BIST Cost Visualization Tool", Bachelor thesis, Tallinn University of Technology, Tallinn, June 2006, pp. 1–34.

(30 publications in total)

List of Abbreviations

ABS	Anti-lock Braking System
ADD	Assignment Decision Diagram
ASIC	Application-Specific Integrated Circuit
ASM	Algorithmic State Machine
AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
CAD	Computer-Aided Design
CPU	Central Processing Unit
DBID	Database Identifier
DD	Decision Diagram
DUV	Design Under Verification
EDA	Electronic Design Automation
EFSM	Extended Finite State Machine
EHM	Extensible Hash Map
FSM	Finite State Machine
FSMD	Finite State Machine with a Datapath
GRLIB	Gaisler Research Library
GUI	Graphical User Interface
HDL	Hardware Description Language
HIF	HDL Intermediate Format
HLDD	High-Level Decision Diagram
IBM	International Business Machines

IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IG	Instantiation Graph
IP	Intellectual property
ISA	Instruction Set Architecture
ITC	International Test Conference
MIPS	Microprocessor without Interlocked Pipeline Stages
OOM	Out Of Memory
OS-VVM	Open Source VHDL Verification Methodology
PC	Program Counter
PPG	Primitive Property Graphs
PSL	Property Specification Language
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RO-BDD	Reduced Ordered Binary Decision Diagrams
RT	Register Transfer
RTL	Register-Transfer Level
SAT	Satisfiability Problem
SISD	Single Instruction Single Data
SMT	Satisfiability Modulo Theories
SoC	System-on-Chip
SSBDD	Structurally Synthesized Binary Decision Diagram
THLDD	Temporally Extended High-Level Decision Diagram
TLM	Transaction Level Modeling
TUT	Tallinn University of Technology
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuits
ZDB	zamiaCAD Database

Contents

CHAPTER 1 INTRODUCTION 21
1.1 Motivation 21
1.2 Objectives 23
1.3 Problem formulation 24
1.4 Contributions 24
1.5 Thesis structure 25
CHAPTER 2 BACKGROUND 27
2.1 Hardware design flow 27
2.1.1 Abstraction levels 29
2.2 Hardware modeling with HDLs 30
2.2.1 VHDL 31
2.2.2 HDLs for verification and debug 31
2.3 State-of-the-art models in hardware design and EDA 32
2.3.1 Exploitation of models in industry 33
2.3.2 Formal models in research 35
2.4 HLDD model preliminaries 36
2.4.1 HLDD model definition 37
2.4.2 Simulation on HLDDs at RTL 38
2.5 Summary 40
CHAPTER 3 HARDWARE MODELING WITH HIGH-LEVEL DECISION DIAGRAMS 41
3.1 HLDD generation 42
3.1.1 Handling of hierarchical designs 43
3.1.2 Handling of slices and sub-elements 44
3.1.3 Handling of memories 44
3.2 Application specific HLDD models 45
3.2.1 HLDD for test generation at RTL 45
3.2.2 HLDD for code coverage analysis 46
3.2.2.1 Compactness levels of HLDD model 47
3.2.2.2 Condition coverage measurement 52
3.2.3 Temporally extended HLDD model for assertion checking 55

3.2.3.1 Basic definitions..55
3.2.3.2 Recursive generation of THLDDs using Primitive Property Graphs 57
3.2.3.3 Assertion checking using THLDDs 59
3.2.4 HLDD for debug...61
3.2.5 HLDD for mutation testing 63
3.3 APRICOT verification framework 65
3.4 Summary 71
CHAPTER 4 HARDWARE MODELING WITH INSTANTIATION GRAPHS 73
4.1 Overview of zamiaCAD framework...74
4.1.1 Framework flow 75
4.1.2 Persistence and scalability..76
4.2 AST model and its limitations..79
4.2.1 Applications of AST model..81
4.3 IG model..81
4.3.1 Modules in IG...82
4.3.2 Objects, types and expressions in IG..84
4.4 Application of IG model to verification and debug............................85
4.4.1 Static analysis 85
4.4.2 Simulation 87
4.5 Scalability assessment 89
4.6 Debug 92
4.6.1 Bug localization with zamiaCAD...93
4.6.1.1 Static slicing...95
4.6.1.2 Suspiciousness ranking based on statement and branch coverage metrics 96
4.6.1.3 Hierarchical analysis based on condition coverage 98
4.6.2 Case study..98
4.6.2.1 ROBSY processor: functional test 98
4.6.2.2 Set of documented design errors...99
4.6.3 Details of automated bug localization 99
4.7 Summary 104
CHAPTER 5 CONCLUSIONS 105
5.1 Contributions...105
5.2 Future work 107
REFERENCES...109
CURRICULUM VITAE 119
ELULOOKIRJELDUS..121
APPENDIX 123

Chapter 1
INTRODUCTION

Modeling is indispensable in science and engineering. The purpose of modeling is to develop a concept and to analyze it through experimenting on it and measuring its properties. Verification of a concept is also an important goal of modeling as it allows correcting the inevitable mistakes on the conceptual level before approaching the real world and (in case of hardware) producing actual physical implementations of the concept.

In this thesis we will contemplate *hardware modeling* with a particular emphasis on those modeling techniques that improve the *quality* of the hardware produced and *speed up* the overall process of product development. In hardware design, the quality is guaranteed by comprehensive verification and testing, while the speed-up is achieved through automation and acceleration of design phases.

1.1 Motivation

The impact of embedded and consumer electronics on our life has been so profound we often take it for granted. We proclaim mobile access to the Internet one of our basic human rights. We get angry when our flight is delayed having no idea of how complex the hardware is that ensures our safety when on board. And we never expect the ABS system to fail in our car. Advances in feature scaling have also led to the situation when a dozen of appliances from the 90's are squeezed into a single smartphone of the 00's and it still fits into a pocket. When many were placing their orders for a new 4G phone, only few were listening to the complaints of Ericsson designers how it is next to impossible to debug a 4G cellular router. While many think building such complex hardware is as simple as putting together several Lego blocks, it is way far from that.

To build today's hardware systems we need the finest production technology and computer-aided design (CAD) tools to harness it. While many design steps are well automated already, we still face the *design productivity gap*, as well as *verification productivity gap* (Figure 1.1). The latter is there for two reasons. First, historically the main focus of CAD research was on automating the design step only. Secondly, the complexity of the verification task itself is much higher than design task complexity [1] with verification reported to occupy between 60% and 80% of design effort [2]. What adds to the importance of verification is the constantly increasing requirements towards the *quality* of verification [3], [4]. The quality requirements are stringent for the hardware systems have become ubiquitous today and even have made it into the many life-critical systems such as mentioned above. That is why, for every designer, the number of verification engineers may vary today from 2 to 4 depending on the design complexity.

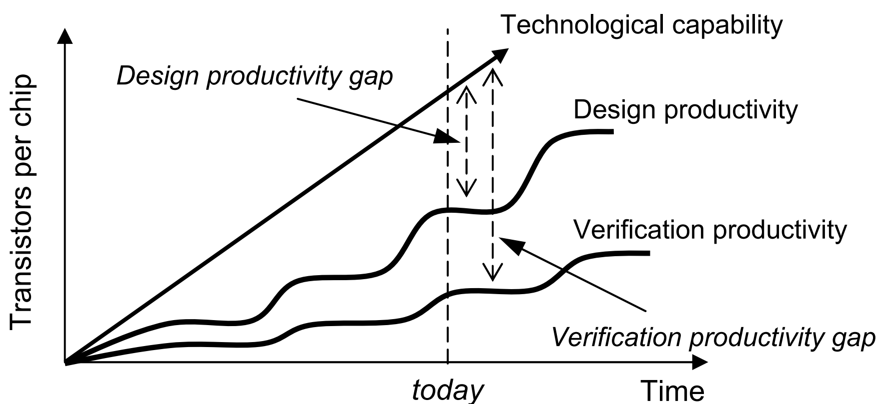


Figure 1.1 Productivity gaps

The *reuse* of pre-designed components is a widespread practice in engineering. While CAD tools allow composing enormous circuits out of smaller ones, the correctness of the resulting system still remains hard to guarantee. In 1996 a highly expensive Ariane 5 payload delivery rocket self-destructed just 37 seconds after the launch, because the control unit reused from Ariane 4 was not verified enough to meet the Ariane 5 specification and proved unable to handle Ariane 5's higher acceleration [5]. Pentium FDIV bug is another example of insufficient verification, which presumably cost Intel $475 million in total [6]. This motivates the creation of more stringent verification metrics and new means of revealing potentially erroneous places in the hardware — all attended closely in this thesis. Choosing the right model for representing hardware during verification in this case is as important as developing the proper algorithms that make use of the model.

Design productivity gap is caused by the weak handling of legacy designs, which are the bottleneck in simulation and verification of the entire system, and by the lack of certain automation. Despite the myth that design process is a single linear pass, it is in fact iterative and suffers most from its least automated phases.

Debug automation is one of such weakest phases. Also, because design is not just a top-down process, but requires both horizontal and vertical feature exploration by the team, re-design is very costly and particularly costly if done late in the process. And absolutely costly if a bug is discovered after the manufacturing, because, contrary to software bugs, hardware bugs cannot be patched. Given the ever more frequent application of embedded systems in the medical domain, bugs can even prove fatal [7].

It is therefore of the highest priority to further raise the quality of hardware through improved *verification accuracy* and to find ways of reducing the cost of development in terms of time and money by e.g. speeding up the *debug* step.

The main problem with today's scientific debug methods is that they rarely take the real life conditions and limitations into consideration. Not only do they scale bad, but they often ask for the availability of a *golden model* normally used in testing and obviously being a legacy thereof. This requirement does not have to be explicitly stated, but more often than not it is deducible from phrases like "It is required that the specification include the expected value of the condition expression" [8], "These dependences are compared with pre-specified dependences" [9] etc. Such phrases reveal that authors forget that during debug we are just in the process of obtaining such a fine-grained specification which, once achieved, will be called an implementation. By definition, the specification conveys information in different notions from those of the implementation and hence never mentions the internal signals and conditions. This can be translated into a *"No Golden Model"* rule often ignored for simplicity when adopting good old testing methods for the task of debug without thorough consideration.

Satisfying the scalability property is also not a trivial task. It leads to the situation where, on one hand, scientific methods are expected to scale well, but, on the other hand, development of a scalable and robust experimental platform is too hard of an intermediate task. Efficiency of new approaches is often only verified on small designs which have nothing to do with the huge industrial designs.

1.2 Objectives

In the previous section we have motivated the importance of raising the quality of verification and further automating the design process, e.g. its debug phase. To meet the demand, the related research should aim at the following objectives:

- Industry needs *efficient* and *accurate verification techniques* to catch all the design bugs before costly or catastrophic effects occur. Such techniques would help closing the verification productivity gap.
- Industry needs *scalable debug automation* to faster locate and correct bugs in designs of current scale. Automation of this frequently repeated design step would help closing the design productivity gap.

- Academia needs an *efficient scalable basis or environment* for carrying out research without ignoring the real life conditions and limitations.

1.3 Problem formulation

To reach the objectives formulated in the previous section, in this thesis we are solving the following specific problems:

- Fast and more accurate verification.
- Fast and scalable automated debug.
- A scalable design environment.

We want to improve the verification accuracy by modeling hardware in a way which is verification-friendly because it allows a fine-grained analysis of hardware descriptions expressed in popular hardware description languages (HDLs). We achieve this by extending the High-Level Decision Diagrams (HLDDs) hardware representation model towards stringent HDL code coverage measurement. Another extension introduces a temporal aspect to the HLDD model and allows us to verify design functionality using verification assertions in a fast and accurate way.

To automate the debug process and support the scale of today's designs we introduce the Instantiation Graph (IG) design representation model. We also provide a scalable debug method which employs the IG model and is particularly efficient in detecting bugs in complex conditional statements of HDL code.

Our final goal is also to publish the theory and the important details of HDL description elaboration into the scalable IG model and provide its working implementation as a scalable zamiaCAD platform. We also provide examples of how IG model can be efficiently used both in academic research and in industry.

1.4 Contributions

Below we list the main contributions of this thesis.

HLDD model modifications and extensions for improved verification and debug

- A *methodology for automated generation of behavioral HLDD* model from a wide subset of VHDL language and *structural HLDD* model synthesis from behavioral HLDD. *Implementation* of the two methodologies is included.
- *HLDD model extensions and modifications* for accurate code coverage measurement, fast PSL assertion checking (using THLDD model), mutation testing and automated debug. Modifications include different compactness levels of HLDD model, different representation of conditional statements

and a temporal extension of HLDD model (THLDD). A *methodology and implementation of THLDD generation* from PSL properties are included.

- A *unified HLDD-based verification framework APRICOT* and its *implementation as ApricotCAD tool* where all HLDD engines are integrated into a single flow.

IG model introduction for scalable debug and verification related research

- *Introduction of a methodology for HDL descriptions elaboration into a scalable IG model*.
- *Presentation of the IG-based design and verification platform zamiaCAD* well suitable for academic research and industrial use.
- A *scalable automated debug method* based on IG model and its *implementation* inside zamiaCAD platform.

1.5 Thesis structure

The rest of this thesis is organized as follows.

Chapter 2 contains background information about the contemporary hardware modeling techniques. The typical design flow is described first and the shortcomings of modeling hardware with HDLs are mentioned. Next, a number of application-specific models is discussed, both from industry and from academia. Finally, the background of HLDD model is described.

Chapter 3 is devoted to HLDD-based hardware modeling approaches. It starts by listing the most interesting aspects of the methodology for behavioral HLDD generation from HDL descriptions. The concept of structural HLDD synthesis is mentioned in the section devoted to test generation using HLDDs. Then a list of application-specific modifications and extensions of HLDD model follows. Different compactness levels of the HLDD model and various condition representations are discussed in frames of code coverage measurement. Temporally extended HLDDs are discussed in detail in the context of fast assertion checking. Finally, HLDD-based mutation testing and its application for high-level debug are considered. A dedicated section about APRICOT framework and ApricotCAD tool concludes this chapter.

Chapter 4 is completely devoted to the scalable IG hardware representation model, its elaboration process and application of this model to the task of debug. zamiaCAD platform which implements the IG model is described first. Then the concept of comprehensive HDL elaboration into IG model is discussed and experimental data provided so that the scalability level could be assessed. The details of convenient use of IG for simulation and static analysis are delivered along with a detailed explanation of the scalable and accurate debug approach.

Chapter 5 draws conclusions from the thesis and outlines the directions for future work.

Remark:

Throughout the thesis we use [*ref*][co-auth] and [*ref*][auth] superscript suffixes to distinguish co-authored and authored work from the rest of the referenced work.

Chapter 2
BACKGROUND

In this chapter, we describe the typical process of hardware development and the approaches used for modeling hardware at different stages of this process. We provide a short overview of hardware models employed in academia and stress the problems of modeling hardware of industrial scale. This background information helps for better understanding of the role of verification and debug phases of hardware design we address in this thesis and the models used during these phases.

Since a half of this thesis is devoted to extending high-level decision diagrams model of hardware representation to the area of verification and debug, we also provide in this chapter a detailed description of this model.

2.1 Hardware design flow

Let us consider a typical yet simplified hardware design flow depicted in Figure 2.1. The engineering part of hardware product development starts with composing the *specification* of product requirements in a natural language and setting up the constraints for timing, energy consumption, available chip area etc. Initial high-level *design model* is captured next as an executable behavioral description written in a hardware description language (HDL) or as a software program. This description gets then *verified* against the initial specification to make sure the latter is met. From this moment on, the design model gets decomposed recursively into a structure of lower-level models. This hierarchical *refinement* continues until the lowest level of abstraction is achieved, where the obtained models correspond to the components from the given library of available building blocks. If the product is a soft intellectual property (IP) core, then refinement is stopped when the

required technology independent level is achieved. During this refinement (either manual or using synthesis tools), the obtained models are constantly verified against the source model they were obtained from to make sure the initial functionality is preserved. Importantly, while descending through the levels of abstraction, new details become available and new properties become measurable (e.g. precise timing delays). This way different design properties and design compliance with the imposed constraints get verified.

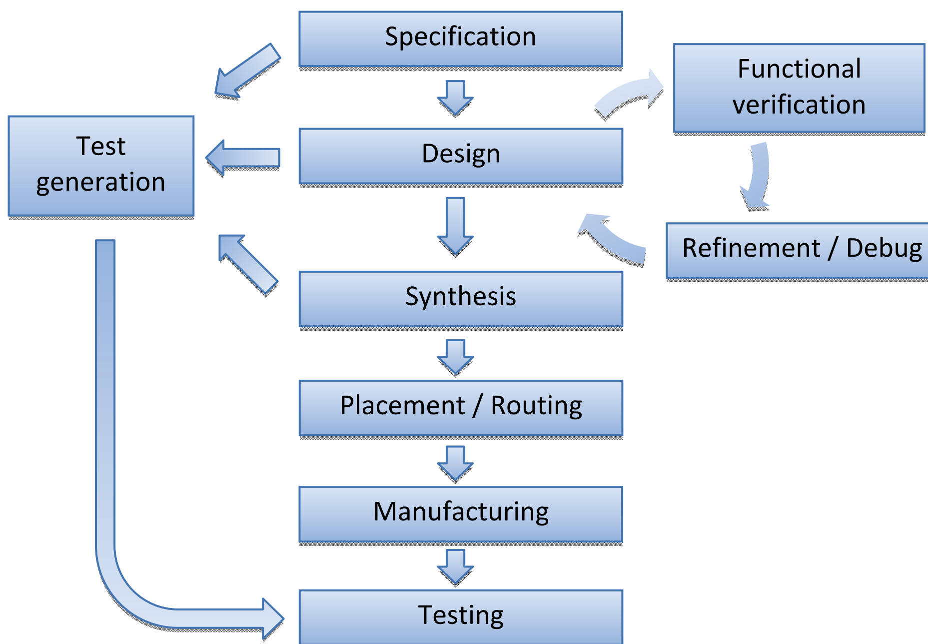


Figure 2.1 Simplified hardware design flow

To verify a model, two types of functional verification can be considered: *formal verification* (e.g. theorem proving, algebraic manipulation, symbolic computation) [10] and *simulation-based verification* [11]. Although formal verification allows proving the equality of two different models of the same design for all possible sets of input values, in practice it can only be applied to small designs because of the task complexity. The industry hence usually resorts to simulation when verifying today's large models due to its realistic runtime. The obvious drawback of simulation is that it can exercise the model with only a limited set of input stimuli. However, the same holds for the formal property checking which is limited to the set of actual properties. Therefore, with simulation just as with formal verification, it is of the utmost importance to develop a decent verification plan which would guide the verification process by accurately specifying what exactly, how and to what extent should be checked to ensure model correctness. It is also the main goal of this thesis: to improve the quality of

verification process and to help with the process of *debug* that follows right after a trouble has been identified during verification.

The *synthesis* step is no more than an automated model refinement which converts behavioral descriptions (of any abstraction level) into structural ones containing components from a given library [12]. One or several physical parameters can be optimized during synthesis, e.g. by minimizing the number of components, chip size, execution time, propagation delay, power consumption etc. While sometimes considered a part of the synthesis step, *placement and routing* are left stand-alone in our case so as to stress how synthesis is more concerned about the semantics of conversion. Placement and routing, on the contrary, are involved in the minimization of wire lengths, signal delays and the total chip area. This step produces a blueprint of the circuit used for circuit's *manufacturing* at the factory.

Finally, every single chip produced at the factory must be *tested* against manufacturing defects. To obtain good quality tests a *test generation* phase is required where information from specification and design models is employed. Thesis contributions to this phase will be discussed in sub-section 3.2.1.

2.1.1 Abstraction levels

When modeling hardware, several design representations or views, such as *behavioral*, *structural* and *physical*, are used to emphasize different information about a design. Different representations are used at different stages of the design process. First, *behavioral* view is used to conveniently describe required functionality of the design. *Structural* view is then used to describe the way to build or implement this functionality with the given components (transistors). The resulting layout of components and their physical properties are finally described using *physical* view. Although to manufacture a chip only structural and physical descriptions are needed, the required functionality is much easier and more convenient to be specified using behavioral descriptions, which are also easier to read and comprehend afterwards. Because of that, the process of converting behavioral descriptions into structural was automated and called *synthesis* [12].

These three different design views may be used at any of the four levels of abstraction or granularity: *algorithmic* level, *register-transfer* level (RTL), *gate* level and *transistor* level. Abstraction levels are defined by the type of objects they use as structural components. *Transistor* level uses transistors, resistors and capacitors to represent structure, and differential equations or other types of current-voltage relationships to represent functional behavior. *Gate* level uses logic gates and flip-flops as structural components described behaviorally with Boolean equations and finite state machines (FSMs). *RTL* operates with arithmetic and memory units such as adders, comparators, multipliers, counters, registers and multiplexers, and uses flowcharts, instruction sets, generalized FSMs and algorithms to describe the functionality of components when combined together. Finally, the *algorithmic* level uses processors, memories, controllers and

application-specific integrated circuits (ASICs) which, when combined together as a system, are described behaviorally in a natural language, in an HDL as an executable specification or as a software program.

The design flow depicted in Figure 2.1 goes through all the levels of abstraction from the algorithmic down to the transistor level when going from specification to placement and routing. The design process typically consists of defining/building a library for a certain abstraction level and using synthesis (manual or automated) to convert a behavioral description into a structure that can be implemented with the components from this library [12]. A list of high-level synthesis tools exist that accept algorithmic level of abstraction as an input specification and produce RTL descriptions [13], [14], [15] suitable for well-established RTL synthesis.

At highest levels of abstraction (e.g. algorithmic or system level) the interest lies in modeling interaction between entire processors and ASICs. Transaction-Level Modeling (TLM) [16] helps modeling communication among modules using the concepts of channels (for media) and transactions (for data). TLM allows abstracting away the implementation details of, both, the functional modules and the actual communication architecture and allows designers to concentrate on what to transmit instead of concentrating on how to transmit and how to process it.

All the different levels of abstraction are essentially different models of the design, since they provide some but not all of the information about the design. In this thesis we are only considering behavioral and structural descriptions at RTL, which still remains the most widely used level of abstraction for it is equally well understood by both human designers and logic synthesis tools. In this thesis, different hardware modeling techniques at RTL are considered which help raising the quality of verification and speed up the debug process.

2.2 Hardware modeling with HDLs

Originally, hardware description languages were introduced to document the behavior of integrated circuits, thus replacing massive and complex manuals with concise and, most importantly, standardized descriptions. It soon became obvious, however, that these concise descriptions are still subject to implementation-specific details and, hence, suffer from ambiguity issues. This understanding led to the idea of making hardware descriptions interpretable in a standardized way so that finally the first publicly available language-based hardware description method with fixed interpretable semantics appeared in 1987 as the IEEE Standard 1076 [17]. The first hardware model simulators implementing the standardized interpretable semantics followed shortly thereafter, along with the automatic circuit synthesizers.

Nowadays integrated circuits are described using *text models* written in HDLs such as VHDL, Verilog, SystemC. We have chosen VHDL [17] as a reference language. The text models are a very popular means of modeling because they

allow a great level of flexibility, resemble natural languages and can be efficiently processed by the machine. Object-oriented paradigm allows modeling both behavior and structure by text. Structured programming paradigm allows capturing hierarchy and supporting different levels of abstraction. To model the inherent concurrency of hardware, HDLs use concurrent statements. Concurrent execution is mimicked in simulators by representing every atomic time step as a sequence of sub-atomic intervals (delta-cycles) simulated sequentially.

2.2.1 VHDL

VHDL (VHSIC Hardware Description Language) [17] is one of the most widely used HDLs at RTL. It has been designed as a standard language from the ground up by incorporating expertise from many industry players of the time. It targets hardware description, synthesis and simulation on the gate, RT and (partially) algorithmic levels. The language is very general due to a large variety of description possibilities and independence from technologies, production processes and methodologies. This makes VHDL designs highly configurable and reusable.

The disadvantages of VHDL stem from its power and generality. Design descriptions are vague and idiomatic (agreed conventions are needed between designers and synthesis tools). While providing additional safety, strong typing also makes the language verbose which in turn makes designs prone to errors and hard to manage and analyze. Smart and scalable tools are required to both harness the potential of the language and avoid its pitfalls.

2.2.2 HDLs for verification and debug

The most important problem with HDLs is that, historically, they were designed to only describe the behavior of circuits. Nobody could envision at the time that aspects other than behavior will soon become of the utmost importance. Nowadays, because of the on-going technology and design scaling, *verification* of even a mid-sized contemporary design takes much *more effort* than the design process itself (up to 80% of total effort [2]). What adds to this effect is the constantly increasing requirements towards the *quality* of verification. The quality requirements are stringent because the integrated circuits have become ubiquitous today and even have made it into the many life-critical systems [7].

Although convenient for specifying behavior and structure, HDLs do not suit well for verification and debug of concurrent hardware. The awkwardness and even inability of HDLs to represent complex temporal assertions has caused introduction of dedicated hardware verification languages such as *e* [18], OpenVera, SystemVerilog, Property Specification Language (PSL) [19]. These languages in turn are not always supported by simulation tools or this support may be expensive. The attempts to unify design implementation and its properties' representation normally result in creation of large hardware checkers that assume significant

restrictions on the initial assertion functionality. Assertion checkers also impose considerable time and area overheads in case of simulation and emulation, respectively. Moreover, the concurrent nature of HDLs does not allow straightforward and comprehensive verification coverage measurement on HDL descriptions and requires either complicated analysis of HDL code or its transformation into other models [20].

Debugging HDL descriptions is also a tedious and difficult process, and is hard to automate. Because of the concurrent nature of HDLs, multiple execution contexts have to be handled by the designer at the same time, which is not easy for a human. Moreover, in practice synchronous halting of the concurrently operating modules during emulation is also not a trivial task, while the obtained system state dumps may occupy tens of gigabytes of hard to analyze data. Industrially scalable debuggers are only partially automated [21] and offer costly solutions for large businesses, but not for smaller ones. This situation with debugging contrasts with verification approaches where the understanding of the efficiency of verification-enablers has led to the emergence of the Open Source VHDL Verification Methodology (OS-VVM) [22]. OS-VVM helps a lot in creating high-quality testbenches by providing VHDL verification libraries (stimuli randomization and functional coverage evaluation packages) which also hide VHDL complexities. However, as of today these two packages are not enough for creating tests which would explicitly and unambiguously reveal the source of an error if one occurs.

In Chapter 4 of this thesis we will present a scalable method (and its implementation) for automatically localizing bugs in industrial-sized designs.

2.3 State-of-the-art models in hardware design and EDA

Electronic design automation (EDA) industry emerged to meet the demand for the tools that could harness the potential of HDLs and cope with complexity of ever-growing integrated circuits (ICs). Among the tasks addressed by the EDA industry are the efficient legacy HDL code exploration and code navigation, coping with verbosity of certain languages (e.g. VHDL) for the sake of design speedup and for fewer design errors, automation of verification, debug and automatic synthesis of HDL descriptions, and also simulation of large-scale ICs.

Nowadays the range of the offered EDA solutions is vast. Yet what remains common to all of them is having HDL descriptions as the primary input to these tools — something dictated by many practical concerns and previous experience. At the same time, having a common input format does not mean all these tools use the same underlying models and data structures. Vice versa, each of them relies on its own internal model that represents hardware designs in the most straightforward and convenient way for the given task. To illustrate this, let us consider the typical design flow in EDA tools presented in Figure 2.2.

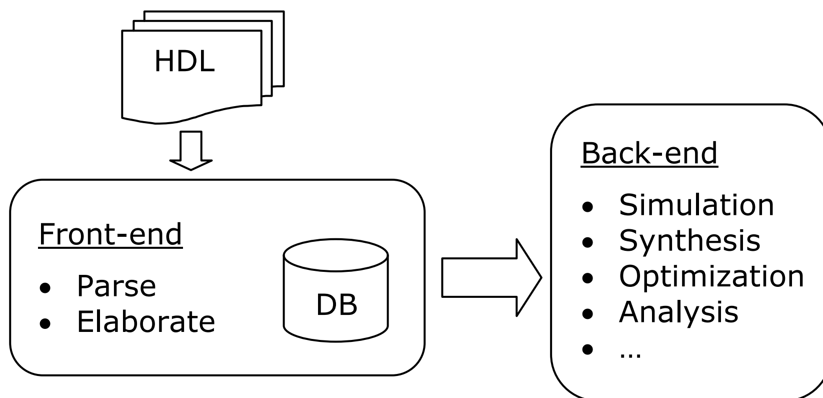


Figure 2.2 Typical RTL design flow in EDA tools

A typical RTL design flow involves a set of front-end and back-end tools. The *front-end* consists mainly of design parsing and design elaboration and storing the result in a database. Front-end steps are typically language dependent. The *back-end* applications read the elaborated design from the database and process it according to their needs. They might be targeted towards design analysis, design synthesis and optimization or simulation. Generally, to do the job in the most efficient fashion, each such EDA task requires a dedicated intermediate design representation adjusted for the given task and typically generated out of the elaborated design model taken from the database.

The rest of this section presents a brief overview of the different *intermediate design representation models* used by the state-of-the-art EDA tools, both, in industry and academia.

2.3.1 Exploitation of models in industry

The enormous size of today's RTL system on chip (SoC) designs together with the predicted technology scaling brings to the front the scalability property of design representation models. Since satisfying the scalability property is not a trivial task, such models are usually kept in-house and are as proprietary as the tools built upon them. This makes it difficult if not impossible to reason about the industrial models beyond what can be deduced from using them as black boxes. We list in this sub-section several common properties of industrial hardware models grouped by the corresponding design tasks the models are applied in.

HDL *code entry* assistance (HDL front-end) could be regarded as the easiest hardware design task. It only requires a parser and a parse tree as an underlying model. Tools like Veditor [23], Sigasi [24], Simplifide [25] propose good Eclipse-based solutions for this and provide code entry, syntax highlighting, code outline

and content assist for VHDL and Verilog. In addition, Simplifide offers hardware specific refactoring features and a coarse-grained design hierarchy view.

When it comes to design *navigation* and legacy *code exploration*, tools can no longer rely on a pure parse tree. It alone will allow neither precise navigation (e.g. to find the correct declaration of an overloaded function), nor a decent design comprehension (no elaborated chip hierarchy and other visual representations), nor global signal tracing etc. For such tasks, an elaborated model is required. In certain cases, though, one can still avoid building a proper elaborated model. For instance, it is possible to perform signal tracing, data flow and schematics view to some extent dynamically (e.g. [11]) relying on the testbench quality. However, proper design analysis for these and a set of other analysis tasks (e.g. hierarchy exploration, precise global signal tracing, type checks, sink-less or source-less signals analysis) can be performed only statically [26] and requires generation of a lossless elaborated design model. To grasp the design hierarchy at once and to speed-up the code entry, HDL Designer [27] from Mentor Graphics offers a sophisticated solution for both initial automatic code generation and further visualization and analysis thereof. However, the code gets generated out of non-standardized tables and graphics, which brings with it all the problems and overheads of invasive approaches, e.g. forces the whole team to use the same tool and only allows a minimal fine-tuning of the generated code afterwards.

The majority of EDA tasks require an elaborated model of the design. For instance Design Compiler [28] by Synopsys, or Verdi Automated Debug System [21] do create such models as an interim step for their further backend processing that are *synthesis* and *debug*, respectively. Verific Parser Platform [29] by itself is a naked *elaboration* engine for both VHDL and Verilog. Verific elaborates designs into a common language-independent HDL database, which is further used as a basis by several commercial hardware design tools.

Finally, when targeting the task of *simulation*, elaborated HDL models have another trait easily observed in such tools as ModelSim [11] and ghdl [30]. Even though the initial compilation/parse of the design is often fast, one has to start the simulator to launch the elaboration and produce the elaborated model for simulation. In practice, this simulation start not only takes a long time to complete, but it often fails by running out of either memory or time in case of large state-of-the-art designs. It confirms the importance of model scalability.

Even in case of industrial tools, the algorithms used in today's front-ends and back-ends are often based on methods developed years ago (Espresso-MV [31], Booledozer [32]) when scalability issues were less of a concern than they are now.

Last, but not least, most of these closed-source EDA tools are of no use to the open source community and of limited use to academia when it comes to building research tools on top of industrially robust models. Scalable hardware models used inside industrial EDA tools are typically kept in-house because of their high value. They are inaccessible to academia which still needs to experiment with large

designs. This leads to pretty severe consequences, namely, to the situation where, on one hand, scientific methods and approaches are required to scale well, and, on the other hand, development of a robust and scalable elaboration engine in the first place (to stay on a par with the industrial scale) is too hard of an intermediate task. No need to say it heavily impairs productivity of researchers and often hinders them from switching to new areas of research where they have no infrastructure yet, no adequate tools to support their work. Even in their area of expertise more often than not their tools can only support small designs which have nothing in common with those in the industry.

One of the contributions of this thesis (entire Chapter 4) lies exactly in filling up this gap. Namely, among other things, the thesis provides a description of a scalable hardware model and the open-source tool zamiaCAD which already implements this model in order to provide a scalable and robust basis for other scientists so that they can build their tools on top of it. The two hence represent what is called a front-end in Figure 2.2.

2.3.2 Formal models in research

In this sub-section we provide short descriptions of models widely used across academia to model hardware with the purpose of verification.

Decision Diagrams (DD) have been used in verification for about two decades. Reduced Ordered Binary Decision Diagrams (RO-BDD) [33] as canonical forms of Boolean functions have their application in equivalence checking and in symbolic model checking. Recently, a higher abstraction level DD representation, called Assignment Decision Diagrams (ADD) [34], have been successfully applied to, both, RTL verification and test [35], [36].

The main issue with the BDDs and ADDs is the fact that they only allow either logic or RTL modeling, respectively. In this thesis we consider a different decision diagram representation, High-Level Decision Diagrams (HLDD) that unlike ADDs can be viewed as a generalization of BDDs. HLDDs can be used for representing different abstraction levels from RTL to Transaction Level Modeling (TLM), and when combined with Structurally Synthesized BDDs (SSBDDs) [37], [38] can also support the gate level. Also, although highly useful for synthesis optimization because of their partial uniqueness property [39], ADDs do not suit for the tasks of dynamic verification and debug considered in this thesis. This is because, without labels on the edges, ADDs cannot represent activated paths through the diagram, and they also merge all individual conditions inside conditional statements [40] so that only the final output is known (i.e. atomic representation of conditions) and no debug inside conditional statements becomes possible.

When compared to the RO-BDD models which have worst-case exponential space requirements, HLDD size scales well with respect to the size of the RTL code. The main difference is that traditionally a decision diagram is generated for a

primary output of the system while nodes represent primary inputs. In HLDDs we generate a separate diagram for each variable (signal) v of the VHDL description and nodes represent variables (signals) assigned to v. Note, that the complexity of HLDDs is just $O(n)$ with respect to the number of processes n in the code. And if v is the average number of variables/signals inside a process and c is the average number of conditional branches in a process, then in the worst case the number of nodes in the HLDD model will be equal to $n \cdot v \cdot c$. Thus, very large realistic hardware systems can be represented in practice.

At RTL, hardware can also be modeled using algorithmic state machines (ASMs) [41]. ASMs are typically used during the design phase, because they allow specifying exact behavior and timing information for control part and operational part (datapath) in the form of charts, which can be later converted to HDL code. However, attempts exist to also use ASMs for synthesizing fast assertions checkers from PSL properties [42].

Extended finite state machines (EFSMs) [43] are often used for representing control dominated systems. EFSMs allow a more compact representation of design states than traditional FSMs without requiring the explicit enumeration of all the design states. Also, in EFSMs transitions may be associated with internal registers (i.e. not only with primary inputs/outputs, as is the case with conventional FSMs), while the registers themselves are not required to be explicitly represented using states. EFSMs are hence more resistant to the state explosion problem than FSMs. However, traversing an inconsistent EFSM may be more difficult than traversing an FSM [44] and requires solving the inconsistencies.

Petri-net [45] based models are widely used for modeling hardware at levels above RTL. In [46], Petri-net based Representation for Embedded Systems (PRES+) is used to formally verify components and their interaction at TLM. Petri-nets allow modeling concurrent interprocess communication while abstracting away the exact implementation of, both, the interconnection and the processes.

Hardware models can also be combined together to achieve a valuable synergic effect. In [47][co-auth], EFSM and HLDD models were exploited inside a functional test pattern generator. EFSM model was used for targeting control FSM transitions, while variable-oriented HLDD model targeted bit-coverage faults in the data variables. HLDD-based engine also provided information about untestable areas in the design to the EFSM-based engine, which improved the efficiency and speed of the overall test generation.

2.4 HLDD model preliminaries

A High-Level Decision Diagram (HLDD) [38] is a graph representation of a discrete function. It was proposed by Raimund Ubar in 1983 [48] for test generation and simulation due to its ability to efficiently and uniformly describe the

structure, function and faults of digital circuits. HLDDs can be considered as a generalization of Binary Decision Diagrams (BDDs), where as opposed to BDD the variables at the nodes can be of any scalar type, i.e. not just Booleans. HLDD model has proven to be an efficient model for simulation and fault modeling since it provides for a fast evaluation by graph traversal and for easy identification of cause-effect relationships [49], [50].

2.4.1 HLDD model definition

According to [51], 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 an HLDD.

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

The function $X(m_i)$ returns the variable x_i, which is labeling node m_i. Each node of an HLDD is labeled by a variable. In special cases, nodes can be labeled by constants or algebraic expressions. An edge $e \in E$ of an HLDD is an ordered pair $e=(m_1,m_2) \in M^2$, where M^2 is the set of all the possible ordered pairs in set M. Graphical interpretation of e is an edge leading from node m_1 to node m_2.

It is said that m_1 is a *predecessor node* of m_2, and m_2 is a *successor node* of the node m_1, respectively. D is a function on E representing the activating conditions of the edges for the simulating procedures. The value of $D(e)$ is a subset of the domain X_i of the variable x_i, where $e=(m_i,m_j)$ and $X(m_i)=x_i$. It is required that $Pm_i = \{ D(e) \mid e = (m_i,m_j) \in E\}$ is a partition of the set X_i.

Figure 2.3 presents an 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_3, m_4 and m_5 in Figure 2.3). In HLDD models representing digital systems the non-terminal nodes correspond to conditions or to control signals, and the terminal nodes represent operations (functional units), register transfers and constant assignments. 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 labeling the nodes of an HLDD are calculated by other HLDDs of the system.

$G_y=(M,E,X,D)$,

$M=\{m_1, m_2, m_3, m_4, m_5\}$;

$E=\{e_1, e_2, e_3, e_4, e_5\}$, $e_1=(m_1, m_2)$, $e_2=(m_1, m_4)$, $e_3=(m_1, m_5)$, $e_4=(m_2, m_3)$, $e_5=(m_2, m_4)$;

$X(m_1)=X(m_5)=(x_2,\{0,1,2,\ldots,7\})$, $X(m_2)=(x_3,\{0,1,2,3\})$, $X(m_3)=(x_4,\ldots)$, $X(m_4)=(x_1,\ldots)$;

$D(e_1)=\{0\}$, $D(e_2)=\{1,2,3\}$, $D(e_3)=\{4,5,6,7\}$, $D(e_4)=\{2\}$, $D(e_5)=\{0,1,3\}$.

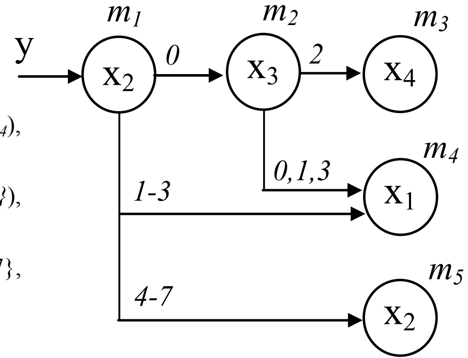


Figure 2.3 An HLDD for a function y=f(x1,x2,x3,x4)

2.4.2 Simulation on HLDDs at RTL

According to [50], 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_i of the variable $x_i=X(m_i)$ a certain output edge $e=(m_i,m_j)$, $v_i \in D(e)$ will be chosen which enters into its corresponding successor node m_j. Let us call such connections between nodes *activated edges* under the given values. Succeeding each other, activated edges form in turn *activated paths*. For each combination of values of all the node variables there exists always a corresponding activated path from the root node to some terminal node. This path is referred to as the *main activated path*. The simulated value of the variable represented by the HLDD will be the value of the variable labeling the terminal node of the main activated path.

In Figure 2.4 simulation on the HLDD 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_1 (the root node) to a terminal node m_4 labeled by x_1. The value of variable x_1 is 4, thus, $y=x_1=4$. Note that this type of simulation is event-driven since we have to simulate those nodes only (marked by bold circles in Figure 2.4) that are traversed by the activated path. This gives HLDDs advantages over netlists of primitive functions in terms of efficiency in simulation and diagnostic modeling because of the direct representation of cause-effect relationships. In fact, HLDD based simulation algorithms have also been shown to outperform commercial event-driven HDL simulators in 12 - 30 times and cycle-based simulators in 4 to 6 times [49]. This is achieved due to combining event-driven (path activation in the HLDD graphs) and cycle-based (HLDDs are synthesized into cycle-accurate models) paradigms. In other words, only a part of HLDDs should be traced during simulation (the activated path) and the time specific information inherent in HDL descriptions can be (and is) neglected during simulation [50]. Timing information is only considered once during HLDD generation.

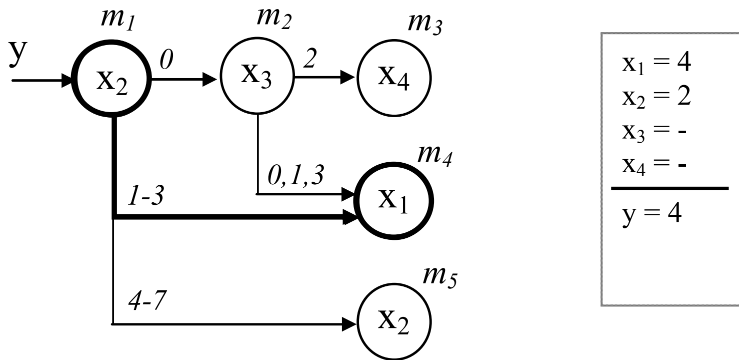


Figure 2.4 Design simulation on HLDDs

As it was previously mentioned, when representing systems and functions by decision diagram models, in general case, a network of HLDDs rather than a single HLDD is required. During the simulation in HLDD systems, the values of some variables labeling the nodes of an HLDD are calculated by other HLDDs of the system. The detailed algorithm for HLDD based system simulation was proposed in [49] and is presented in Figure 2.5. The algorithm supports both behavioral and structural design representations at RTL.

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

```
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 each edge e in m_Current
            If Value ∈ D(e_active), e_active =( m_Current, m_Next) then
               m_Current = m_Next
            End if
         End for
   End while
   Assign x_G = x_Current
 End for
```

Figure 2.5 Algorithm 1. Simulation on structural and behavioral HLDDs

2.5 Summary

In this chapter we have provided an overview of the typical hardware design flow and the hardware description models employed in the individual steps of this flow. We have shown how hardware is formally modeled in academic research and what problems appear when modeling hardware of industrial scale.

In particular, the background information about the high-level decision diagrams academic model was presented for better understanding of Chapter 3 where the HLDD model and its applications are extended beyond simulation and test generation covered so far by the model.

The rest of this thesis is devoted to solving the design problems mentioned in this chapter. Namely, in Chapter 3 and Chapter 4 we tackle the problems of scalability, accurate verification and fast debug with the help of academic high-level decision diagrams model and industrial scale instantiation graph model.

Chapter 3

HARDWARE MODELING WITH HIGH-LEVEL DECISION DIAGRAMS

This chapter discusses application of HLDD model to the tasks of hardware verification and debug.

Design representation with HLDDs was shown efficient for functional simulation and RTL test generation. One of the contributions of the author of this thesis lies in extending the use of HLDDs by:

1) developing methods for automated synthesis of behavioral and structural HLDDs from VHDL code, which allowed obtaining HLDD representations for larger designs than could be manually constructed before;
2) extending the application of HLDD model to other design tasks such as verification and debug by both participating in the development of the appropriate theories for HLDD exploitation in new areas and by adapting the model itself according to the given theory;
3) participating in the creation of a unified verification framework APRICOT and its implementation as ApricotCAD tool where all HLDD engines are integrated into a single solution.

All the three points are described in detail in sections 3.1, 3.2 and 3.3, respectively.

3.1 HLDD generation

The process of HLDD generation was described in detail in [52][auth].

To represent behavioral RTL descriptions, a separate HLDD diagram is generated for each VHDL signal, variable or output port. For those variables that change their value several times per clock a separate diagram is generated for each such value change. HLDD graph generation from VHDL code happens as follows:

1. Conditions from *if*/*case* statements or conditional assignments are represented by a non-terminal node with its edges marked by decisions of the condition.
2. Right-hand sides of assignments are represented by terminal nodes.
3. Those branches of conditional statements, where no assignment is made to the signal of interest, are filled with a default sub-graph. The default sub-graph is a graph constructed from preceding parallel assignment statements if such exist. Otherwise it is a value retaining terminal node. A warning is issued to inform about implicit latches when a value retaining terminal node is filled for a non-clocked variable.
4. The resulting HLDD graphs are topologically sorted to ensure causality during behavioral simulation (*Algorithm 1* in Figure 2.5).

Figure 3.1 presents an example of an HLDD generated for two variables, *state* and *RMAX* in the ITC99 benchmark b04.

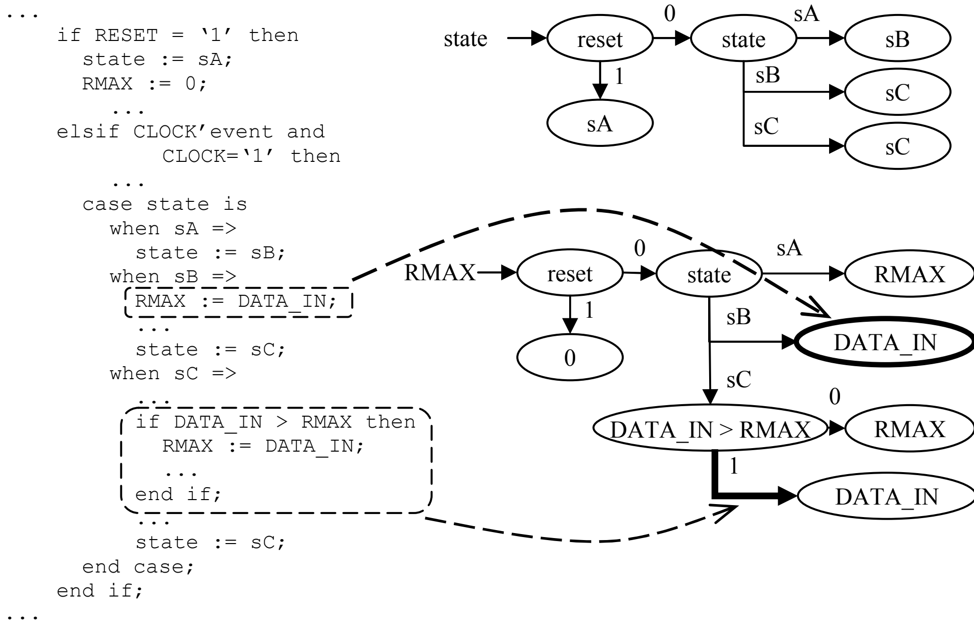


Figure 3.1 b04 example: behavioral HLDDs for variables *state* and *RMAX*

Several compactness levels of HLDD representations provide different tradeoffs between the HLDD-based analysis stringency and simulation speed with memory requirements. The three options of *model compactness* (full tree, reduced and minimized) will be discussed in sub-section 3.2.2.1 in frames of code coverage measurement. Different options for representing *conditional statements* are discussed in sub-section 3.2.2.2 in frames of condition coverage measurement. In HLDDs, conditional statements can be represented as built-in functions, as functions plus extra-graphs, as expansion graphs and as flattened graphs.

From modeling perspective, when generating HLDDs from VHDL descriptions, the appropriate selection among compactness levels and conditional statements representations depends on the target application.

In the following sub-sections we will add more details on HLDD generation not mentioned in [52][auth] because they had not been addressed at the time. The four new aspects are the support of design hierarchy, generation of temporal verification assertions and efficient handling of memories and assignments to signal slices/sub-elements. The synthesis of temporal assertions into special Temporally Extended HLDD graphs will be discussed in detail in section 3.2.3.

3.1.1 Handling of hierarchical designs

The newly added support for hierarchical designs (i.e. consisting of several HDL files / modules) was achieved in a straightforward way via variable/graph naming conventions. While traversing the design in a top-down manner, a hierarchical suffix is being constructed which gets finally appended to all the names of the functions, constants and HLDDs representing the final HDL signals/variables. Such suffixes are needed to facilitate manual debug of HLDD generation engines (suffixes are human-readable) and to avoid collisions when merging graphs, functions and constants from different hierarchy levels into a single system of HLDDs (which is ultimately a pure list of HLDDs, in some cases ordered, in others — not). An example of a hierarchical VHDL design supported by the automated VHDL-to-HLDD interface is UART16750 from OpenCores.org with 12 VHDL design units and 1662 lines of code [53].

From the modeling perspective, an important conclusion we can draw from how easy it was to add the hierarchy support is that the HLDD model inherently and naturally supports hierarchy, just in the same way it supports different levels of abstraction. If cleverly used, such a unified model of design representation would ideally allow performing different inter-level tasks to further improve the quality of designs. For example, it can allow analyzing of circuit aging on the gate level and propagating the results of the analysis up to the highest level of abstraction. This way a designer can see which places in his high-level description are expected to lead to premature device aging and can take the appropriate actions on that issue.

3.1.2 Handling of slices and sub-elements

The handling of slices/sub-elements of composite signals/variables was not previously supported and hence had not been mentioned in [52][auth]. While we can *access* slices/sub-elements from inside HLDDs by just equipping the nodes of HLDDs with the required slices/indices, the *assignments* to slices/sub-elements need to be handled separately. The general strategy of handling slices implies traversing HDL code structure and identifying all the atomic slices (i.e. non-intersecting regions) of signals/variables that are being written at least once, and creating a dedicated HLDD for every such non-intersecting region. If for a given signal the list of obtained regions contains intersecting slices, they will be split into the largest possible non-intersecting atomic regions. These regions are treated as independent signals with their own corresponding HLDDs. The HLDD of the initial signal is then constructed to contain exactly one node which basically concatenates all the slices into one. This HLDD is then accessed from other HLDDs in the system that refer to the original signal/variable.

3.1.3 Handling of memories

If we apply the approach from section 3.1.2 to memories accessed with dynamic indices, we will have to split the memory into individual elements. Such a non-scalable approach is depicted in Figure 3.2, where we use 1026 graphs to represent a memory of 1024 cells. 1024 graphs on the left are used for storing the actual value of each memory element and for writing to memory using dynamic index *w_adr*. Note how new value is only read from *D* when the value of dynamic index corresponds to the cell's index (the bold circles). 2 graphs on the right (*c(i)* and *c*) are used for reading from memory using dynamic index *i* or static index/slice.

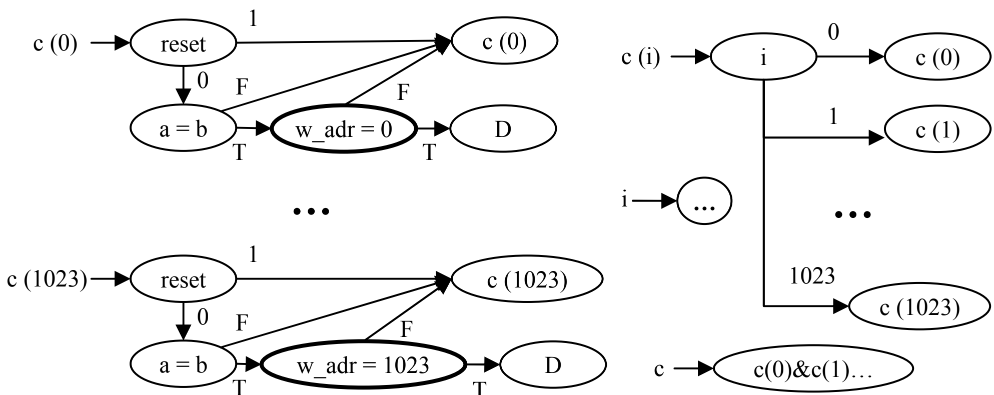


Figure 3.2 A "non-scalable" representation of memories by HLDDs

To make HLDDs support today's memories of millions of elements we delegate the job of representing memories to the simulator, effectively moving the memory structures out of HLDD model and into the simulator's internal arrays. Such a scalable approach is depicted in Figure 3.3, where a single graph is used for writing into memory (graph *c*) and a single graph *c(i)* or *c(i)(j)* is used for reading from it using dynamic indices *i* and *j* (i.e. possibly nested). Dashed arrows and nodes denote a special type of *memory variables*, *index nodes* and *array functions* that command the simulator to allocate internal arrays, index a memory cell for writing and for accessing, respectively. Chaining array functions allows nested indexing.

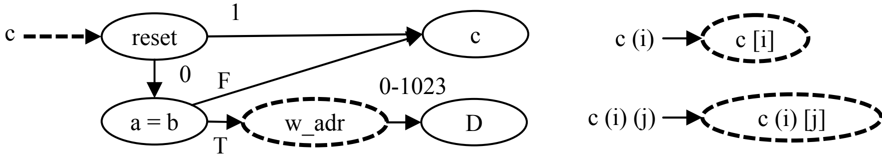


Figure 3.3 A scalable representation of memories by HLDDs

With the scalable approach we no longer have memories present in the model so they cannot be covered by the HLDD-based verification techniques described in section 3.2. In return, we retain the support of large designs and can apply simulation-based tools to the rest of the design's functional model. Fortunately, most of the simulation-based technique described in 3.2 can live with the "non-scalable" memory representation. Only the automatic RTL test pattern generator DECIDER [54] requires the version of HLDDs which cannot be generated in a scalable manner out of HLDDs with "non-scalable" memories inside. We describe such version of HLDDs in sub-section 3.2.1.

3.2 Application specific HLDD models

Previous works have shown that HLDDs are an efficient model for simulation [49], [50] and test pattern generation [55]. We have extended the use of HLDD model to other design tasks such as verification ([56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66])[co-auth] and debug ([67], [68])[co-auth]. To make test pattern generation applicable to larger designs, we have also developed an automated HLDD synthesizer [52][auth]. It produces structural HLDDs, suitable for test generation, out of behavioral HLDDs designed for simulation. In the following sub-sections we describe our contributions starting with the HLDD synthesizer.

3.2.1 HLDD for test generation at RTL

At RTL, designs are partitioned into a *control part* and a *datapath*. The datapath stores the data and performs operations on it. The control part guides this process

by remembering the state of the execution and by issuing control signals to the datapath. The behavior of the control part is usually specified by an FSM. To completely specify the behavior of the datapath it is enough to assign values to all its control signals at each state of the FSM. However, when there are too many control signals it is difficult to realize how the datapath will operate. To improve the comprehension of such specifications, the datapath is often described *behaviorally* using dataflow assignments to datapath variables inside the FSM states. A synthesis tool is then used to derive the actual control signals. Such *behavioral* model is called an FSM with a datapath, or *FSMD* [12], [41].

In [52][auth] an automated interface between FSMD VHDL and its HLDD counterpart referred to as *behavioral HLDD* was described. This interface made possible the simulation of such VHDL designs by HLDD engines and application of simulation-based HLDD techniques to the FSMD VHDL designs. However, the HLDD engine for test pattern generation (RTL ATPG DECIDER) requires a different type of HLDD, called *structural HLDD*, which corresponds rather to the *FSM+D structural* VHDL model [41]. In FSM+D model, the FSM and datapath are explicitly separated from each other, and the datapath is synthesized from behavioral description down to the structural use of standard RTL components such as functional units, multiplexors, registers etc. — hence the naming *structural HLDD*.

Thus, in order to generate RTL tests for the widely used FSMD VHDL descriptions, these designs are first converted to behavioral HLDDs, which are then synthesized into structural HLDDs. The synthesis process [52][auth] involves FSM recognition, optimal state encoding, memory elements inferring, mapping the behavior to the datapath RTL components and computation of the datapath control signals such as register enable, reset and multiplexor control signals. From modeling perspective, keeping the FSM separate from the datapath and explicitly driving the datapath using its control signals allows efficient test generation without overtesting.

3.2.2 HLDD for code coverage analysis

In sections 1.1 and 2.1 we have stressed the importance and high cost of verifying functional correctness of a hardware system being designed. Because of the rapid system scaling, design simulation by a set of test cases is often chosen as a means of (*dynamic*) verification, and HLDD simulation falls exactly into this category of verification methods. However, to control the verification effort the confidence level regarding the quality of the design must be quantified, so that we know when it is safe to stop verifying or simulating. Verification *coverage* is a measure of confidence and it is expressed as a percentage of items verified out of all possible items. Different definitions of items give rise to different coverage measures or coverage metrics, such as code coverage, parameter coverage and

functional coverage. In this thesis we will only use code coverage, which provides insight into how thoroughly the code of a design is exercised by a suite of stimuli.

The first attempt to use HLDD models in verification and code coverage analysis was presented in [69], where fast HLDD based simulation (see Section 2.4.2) was extended and HLDD model introduced for efficient code coverage analysis. It was shown how classical code coverage metrics can be mapped to HLDD constructs.

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

According to [58][co-auth], the statement coverage corresponds to the ratio of nodes $m_{Current}$ traversed during the HLDD simulation presented in *Algorithm 1* in Figure 2.5. As an example, Figure 3.1 depicts HLDD representations of state and data register variables of a VHDL design. Covering all nodes in the HLDD model corresponds to covering all statements in the respective HDL.

Similar to the statement coverage, branch coverage has also very clear representation in HLDD simulation. The ratio of edges e_{active} activated in the simulation process of *Algorithm 1* constitutes HLDD branch coverage. For example, the branch coverage item corresponding to DATA_IN > RMAX = true in the VHDL code of b04 design maps to the edge denoted by a bold arrow in the HLDD in Figure 3.1. The statement RMAX := DATA_IN is represented by the terminal node surrounded by bold circle in the corresponding HLDD.

3.2.2.1 Compactness levels of HLDD model

In [58][co-auth] we have researched on the *accuracy* of HLDD-based coverage assessment. We have proposed a set of HLDD manipulations to achieve more stringent code coverage metric than classical methods without sacrificing performance. The manipulation techniques include synthesis of HLDD trees from HDL descriptions and two types of HLDD collapsing methods, which are a generalization of the BDD reduction rules.

The most stringent coverage metric is path coverage, which can be achieved by synthesizing HLDD trees for each output signal and terminating the synthesis at the primary inputs. However, with the scale of today’s designs such HLDDs will be of exponential size. Therefore, we terminate HLDD tree synthesis at HDL variables. The achieved code coverage metric is closer to the path coverage metric (i.e. more stringent and accurate) than HDL based code coverage.

The method proposed for generating HLDDs suitable for code coverage analysis is similar to BDD reduction rules [33] and consists of the following steps:

1. Generate an HLDD tree for each system variable.
2. Reduce nodes with identical succeeding sub-graphs.
3. Unite identical terminal nodes.

The above steps are explained by an example presented in Figure 3.4, which depicts HLDD manipulations for the '*state*' variable of the b04 design presented in Figure 3.1. As the first step, an HLDD tree for variable *v* is generated by traversing the full control flow graph of the design and collecting the values assigned to *v* at each control step. If the value of *v* does not change at current control step then terminal node with the present value of variable will be created. Figure 3.4a shows the HLDD tree generated for the variable *state* in b04.

Then, reduction rules are applied to eliminate nodes for which all successor nodes (in general case, succeeding sub-graphs) are identical. As a result a reduced HLDD is obtained (Figure 3.4b). Finally, we create a minimized reduced HLDD by uniting identical terminal nodes (Figure 3.4c). HLDD generation experiments [58][co-auth] on a set of ITC'99 benchmarks [70] show that around 45-80% of nodes are removed by the reduction step from the initial HLDD tree. Further 40-60% of nodes are eliminated by the minimization step.

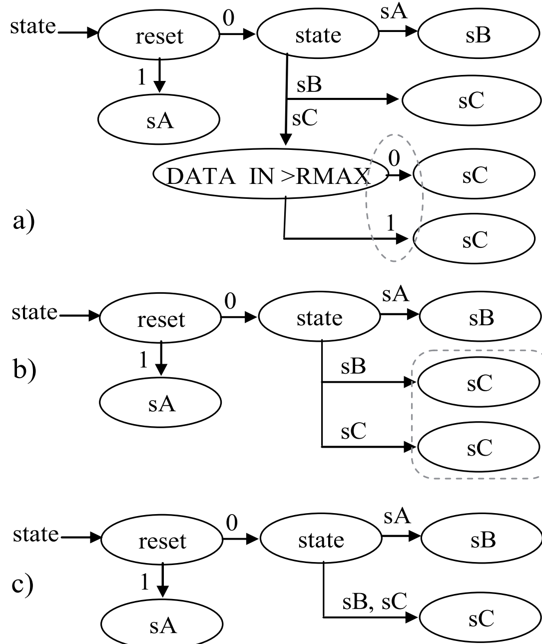


Figure 3.4 a) HLDD tree, b) reduced HLDD and c) minimized reduced HLDD

To summarize, the three types of HLDD design representation we distinguish in the proposed methodology are (in the increasing order of compactness):

1. *Full tree HLDD*: contains all control flow branches of the design. This type of representation includes a lot of redundancy. They introduce large space requirements and relatively slow simulation times.
2. *Reduced HLDD*: obtained from full tree HLDD by eliminating all the redundant nodes whose all edges point to the equivalent sub-graphs. This type of design representation is still a tree-graph. It combines the advantages of, both, full tree and minimized representations. The HLDDs of this type are reasonably compact and they allow more stringent coverage measurement than the minimized representation. Furthermore, the average HLDD path length, and therefore the simulation speed, is exactly equal to the more compact minimized type.
3. *Minimized HLDD*: obtained from reduced HLDD by making nodes share all the equivalent sub-graphs, thus eliminating all the redundancy within every single HLDD. The resulting HLDD is therefore no longer a tree structure, but a true graph with possible interconnecting crossovers. This representation is the most compact of the three. However, the minimization step may cause a loss in coverage accuracy.

In [58][co-auth] we propose *reduced* HLDDs as a suitable model for code coverage analysis because it provides for more stringent coverage metrics than minimized HLDDs and is more compact than the full tree and faster to simulate. Table 3.1 (ITC99 benchmarks) compares code coverage analysis comparing statement coverage and branch coverage assessment results on reduced HLDDs (red.), on minimized HLDDs (min) and on a well-known commercial tool using the same set of input stimuli for all three models. 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.

Table 3.1 Comparison of code coverage analysis results

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

The stringency of reduced HLDD model comes from HLDD's inherent support of *observation coverage*. Contrary to procedural languages, executing a statement in *concurrent languages* does not guarantee its effect to be propagated to the primary outputs of a design. Fallah et al. [71] propose *observation coverage* in their method called OCCOM, where simplified fault grading is carried out in order to asses, which code items have been covered and propagated to an observable output. They show that 100% code coverage corresponds to as low as 60-80% observable coverage in the worst case, which is similar to our results (Table 3.1). This indicates inability of conventional code coverage metrics designed for procedural languages to handle paths of e.g. non-blocking assignments.

On the contrary, HLDDs are generated for each HDL variable separately and are inherently dataflow-oriented. This adds the required observability context to the statement coverage, making the resulting metric closer to the actual path coverage.

To show how HLDD model helps unveiling the "lost" 40-20% of the observable coverage, let us consider VHDL as an example of a concurrent language with non-blocking assignments. VHDL uses a discrete event system to model time and deal with concurrency, and so is very flexible. The discrete event model is very general, but as a result, somewhat difficult to analyze [72]. VHDL uses, both, blocking and non-blocking assignments. All assignments to signals (with '<=') are non-blocking (i.e. they happen some (delta) time in the future), and all assignments to variables (with ':=')are blocking (i.e. they happen immediately).

Consider the following VHDL example *ex1* provided in Figure 3.5, which includes only non-blocking assignments to signals. The signals have the following naming notations: *V* - an output variable; *cS* - a conditional statement; *D* - a decision; *T* - a terminal node; *C* - a condition; *W* - a value. The keywords emphasized by bold determine if a line has a statement, a branch or conditions.

Stm	Dcn	VHDL code
1		**if** (cS1_C1 **and** cS1_C2)
	1	**then**
2		V1 **<=** V1_T1;
	2	**else**
3		V1 **<=** V1_T2;
		end if;
4		**case** cS2_C is
	3	**when** cS2_C_W1 =>
5		V2 **<=** V2_T1;
	4	**when** cS2_C_W2 =>
6		V2 **<=** V2_T2;
	5	**when** cS2_C_W3 =>
7		V1 **<=** V1_T2;
8		**if**(cS3_C1 **and** ((**not** cS3_C2) **or** cS3_C3))
	6	**then**
9		V2 **<=** V2_T2;
	7	**else**
10		V2 **<=** V2_T3;
		end if;
		end case;

Figure 3.5 A segment of the VHDL code of *ex1* design

Figure 3.6 and Figure 3.7 present reduced and minimized HLDD model representations for the *ex1* example.

In Figure 3.5, the numbers from the first column (*Stm*) correspond to the lines with 10 statements (both conditional and assignment ones). The 14 HLDD nodes of the two graphs in Figure 3.6 correspond to these statements. Covering all nodes in an HLDD model (i.e. full *HLDD node coverage*) corresponds to covering all statements in the respective HDL. However, the opposite is not true. To understand why HLDD node coverage is slightly more stringent than HDL statement coverage let us consider VHDL statements *1*, *2* and *3* and the respective nodes in Figure 3.6 *1a*, *1b*, *2a*, *2b*, and *3a*, *3b*. While VHDL statements *1*, *2* and *3* may well be covered by a test suite, their effect will be propagated down to the observable point only when not clobbered by statement *7*. In HLDD model this requirement is explicitly taken into account by inclusion of the corresponding sub-graphs *1a* and *1b* into HLDD of variable *V1* under the appropriate conditional node 4_1 (Figure 3.6). This way, additional nodes are included into computation of code coverage which allows achieving 100% observability coverage, otherwise unreachable if only statement coverage were used. It is indeed the case that HLDD model contains some functional duplication inside (sub-graphs *1a* and *1b* or identical nodes 4_1 and 4_2 denoted by subscript indices). Still, the model is guaranteed to resist exponential node explosion because HLDD construction is terminated at HDL variables and not at the primary inputs of a design. Also, 4_1 and 4_2 duplication is due to the fact that in HLDDs diagrams are normally generated for each data variable separately.

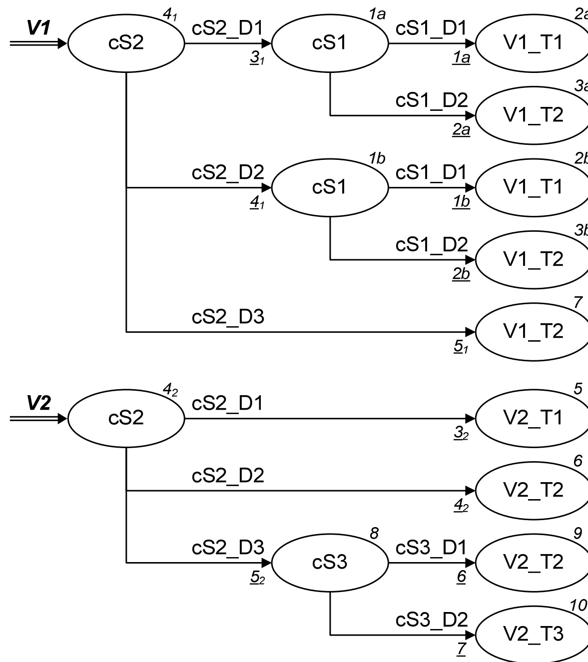


Figure 3.6 Reduced HLDD for *ex1*

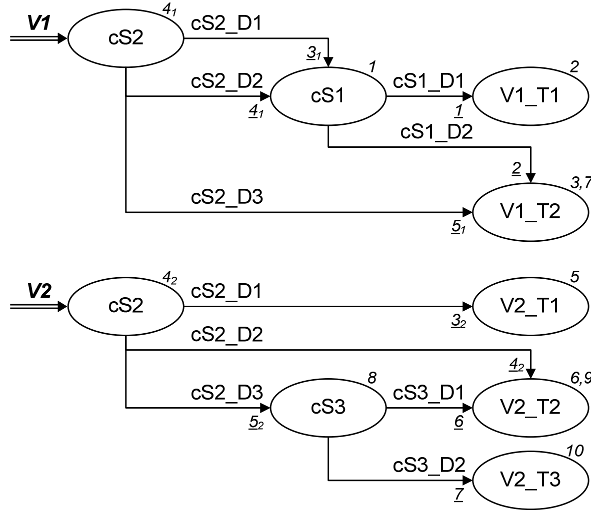


Figure 3.7 Minimized HLDD for *ex1*

Similar to the statement coverage, covering all edges in an HLDD model (i.e. full *HLDD edge coverage*) corresponds to covering all branches in the respective HDL. And, similar to the previously discussed statement coverage mapping, here the opposite is also not true and HLDD edge coverage is slightly more stringent that HDL branch coverage.

Finally, in [58][co-auth] the time overhead of coverage checking using HLDDs was reported to be much lower than using a popular commercial HDL simulator. For HLDDs, the penalty for coverage calculation is in a 1-4% range, while for the commercial simulator it is in a 28-78% range of extra time taken during simulation.

3.2.2.2 Condition coverage measurement

Branches typically consist of more than one atomic conditions combined with logical operators. To correctly estimate the quality of functional verification, we need a dedicated metric to distinguish among those atomic conditions.

Condition coverage metric ([73], [74]) reports all cases each sub-expression separated by logical operators in a conditional statement causes the complete conditional statement to evaluate to one of the decisions (e.g. 'true' or 'false') under the given set of stimuli. Branch coverage, on the contrary, only takes into account the final decision determining the branch. The importance of condition coverage metric is stressed by e.g. DO-254 [3] and DO-178B [4] standards for hardware and software system quality used in airborne systems. These standards state that condition coverage along with statement and branch coverages has to be applied in the cases where system failures would cause catastrophic results.

However, while condition coverage metric allows discovering many corner cases of the design under verification (DUV), its calculation based on HDLs is a sophisticated multi-step process. On the contrary, the flexible representation of *conditional statements* by HLDDs makes HLDD model particularly suitable for several important design tasks, including straightforward condition coverage evaluation. When conditional statements are represented:

- *as functions*, the whole VHDL condition is represented as a single node in the corresponding HLDD. This node either contains a single logical expression or refers to the built-in conditional functions represented in HLDDs as special function variables which may refer to other functions recursively. This is the most compact representation of conditions in the HLDD model, as the function variables can be reused by any HLDD in the system. Since built-in functions are fast, this is also the fastest representation for simulation.
- *as functions + extra-graphs*, a set of graphs (extra-graphs) is added to the resulting HLDD system making each atomic logical sub-expression from the source VHDL file be represented by an HLDD on its own. This allows automatic bug localization inside conditional statements by using statistical methods. An approach similar to HLDD-based statistical bug localization, but employing the IG model, will be described in sub-section 4.6.1.
- *as expansion graphs*, the whole VHDL condition is represented by a full-blown HLDD on its own, containing all the possible evaluation paths through the condition. This "conditional" HLDD is then referred to from all the HLDDs that used the original VHDL condition — in the same way function variables are shared in the first option described above. This representation of conditions allows measuring condition coverage which is discussed below.
- *as flattened graphs*, the "conditional" HLDDs from the previous bullet are injected into each main HLDD graph which refers to them. This representation of conditions should provide even more stringent coverage metric.

In [63][co-auth] we have used HLDDs with *expansion graphs* for conditional nodes to measure condition coverage. Conditional statements with complex logical expressions (normally represented by single nodes in HLDD graphs) are expanded into separate HLDD graphs. Otherwise this is the same HLDD model we previously exploited for increasing coverage accuracy and for supporting non-blocking concurrent statements. This allows us to achieve a homogeneous verification flow (i.e. one model and one tool).

Let us consider the example design *ex1* provided in Figure 3.5 and Figure 3.6. The HLDD expansion graphs for the 3 conditional statements from *ex1* are provided in Figure 3.8. For better readability, the terminal nodes are marked by background colors according to different decisions. These 3 expansion graphs can be considered as sub-graphs representing "virtual" variables (because they are not real variables of the *ex1* VHDL representation) *cS1*, *cS2*, *cS3*. Thus, together with the two HLDD graphs for variables *V1* and *V2* from Figure 3.6 these sub-graphs compose design's hierarchical HLDD representation.

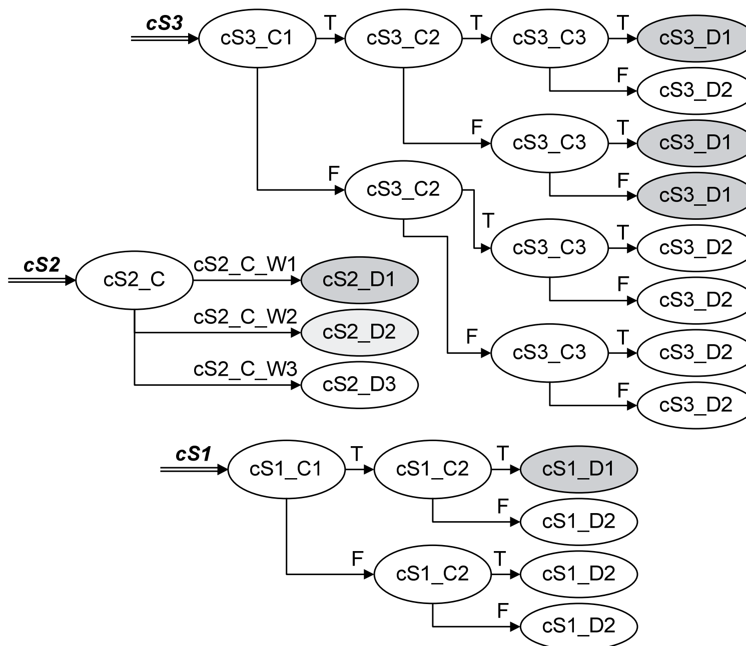


Figure 3.8 Expansion graphs for conditional statements of *ex1*

The full condition coverage metric maps to the full coverage of *terminal nodes* of the conditional statements expansion graphs during simulation. The main advantage of this approach is low computational overhead. Once the HLDD is constructed, the quality of every given stimuli set is evaluated in a straightforward manner during HLDD simulation (*Algorithm 1*, Subsection 2.4.2).

In [63][co-auth] we have carried out an example study for the proposed HLDD-based coverage analysis methodology based on the *ex1* design. With its HDL description, a set of 4 stimuli was enough to achieve 100% statement coverage and 100% branch coverage. However, the HLDD-based node coverage was only 86%, edge coverage only 78% and, most importantly, condition coverage only 47%. The HLDD items remained uncovered were edges *2a*, *1b* and nodes *3a*, *2b* (Figure 3.6). In the minimized HLDD (Figure 3.7), where instead of sub-graphs *1a* and *1b* only one sub-graph *1* exists, this benefit in stringency is lost with minimization.

This indicates that, even with full statement and branch coverages achieved, the full condition coverage still requires additional test vectors. This adds orthogonal dimension of confidence in terms of stringency and verification/test accuracy.

From hardware modeling perspective, it is important to emphasize that all coverage metrics (i.e. statement, branch, condition or a combination of them) are analyzed by a single HLDD simulation tool which relies on HLDD design representation model. Different levels of coverages are distinguished by simply generating a different form of HLDD (i.e. minimized, reduced, or hierarchical with expanded conditional nodes).

3.2.3 Temporally extended HLDD model for assertion checking

In simulation-based verification assertions play the role of monitors for particular system behavior during simulation. Because of the poor ability of HDLs to express *temporal relations* in assertions, dedicated languages were developed, such as the IEEE-1850 standard Property Specification Language (PSL) [19]. To specify temporal logic properties in PSL, repetition operators are used with their auxiliary suffixes (e.g. for next* family they are next_a, next_e!, next_event etc) [75]. In order to introduce the equivalent semantics into the HLDD model and make HLDD simulation engine support such PSL constructs we have proposed [60][co-auth] a *temporal extension* for HLDD (Table 3.2).

The extension to the traditional HLDD model defined in Section 2.4.1 is referred to as *Temporally extended High-Level Decision Diagrams (THLDDs)*. In [60][co-auth] we present the definition of THLDDs and propose an algorithm for their hierarchical generation based on a concept of templates for PSL constructs referred to as Primitive Property Graphs (PPGs). THLDD extends HLDD by using temporal relationships functions to transfer additional information and directives to the HLDD simulator that are used for assertions checking. In the proposed approach design simulation, which calculates *simulation trace,* precedes assertion checking.

3.2.3.1 Basic definitions

In order to represent a temporal logic assertion *P*, a temporally extended high-level decision diagram G_P can be used, which is defined as follows.

Definition 2: A *Temporally extended High-Level Decision Diagram* (THLDD) is a non-cyclic directed labeled graph that can be defined as a sixtuple $G_P=(M,E,T,Z,\Gamma,\Phi)$, 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 labeling the nodes and their domains, *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 labeling variables.

The graph G_P has exactly three terminal nodes $M^{term} \in$ M labeled by constants, whose semantics is explained below (see graphical representation in Figure 3.11):

- *FAIL* — the assertion *P* has been simulated and does not hold;
- *PASS* — the assertion *P* has been simulated and holds;
- *CHK*. (from CHECKING) — the assertion *P* has been activated and simulated, but it does not fail, nor does it pass non-vacuously. (See section 3.2.3.2 for explanation of vacuity).

The first two terminal nodes correspond semantically to the two of the possible states of a PSL property being checked. As an example, consider PSL property *reqack* shown in Figure 3.9. One its possible timing diagram is illustrated by Figure 3.10a (corresponds to the PASS node in THLDD). It states that *ack* must

become high next after *req* being high. A system behavior that activates *reqack* property but violates it is demonstrated in Figure 3.10b (FAIL node in THLDD). Figure 3.10c shows the case when the property was not activated at all.

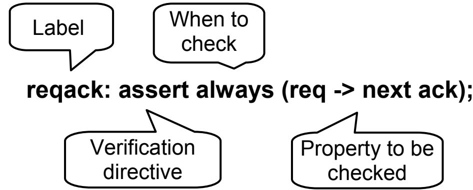


Figure 3.9 PSL property *reqack*

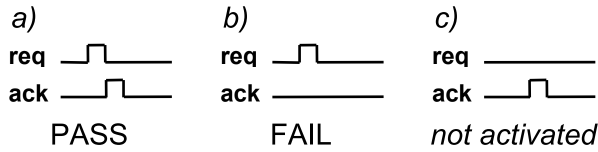


Figure 3.10 Timing diagrams for PSL property *reqack*

The function $\Phi(m_i,t)$ returns the range Δt of time-steps relative to current time t_{curr} where the value of node variable $x_l=Z(m_i)$ must be evaluated/read. The calculus of variable x_l using the relative time range $\Phi(m_i,t)= \Delta t$ is denoted by $x_l^{\Delta t}$. We distinguish three general cases of relative time ranges Δt (upper three in Table 3.2). Table 3.2 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 their *derivatives*. For non-temporal variables $\Delta t = 0$.

Table 3.2 Temporal relationships in THLDDs

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
	$x^{\Delta t=event(x_c)}$	x *holds at the first time-step from* t_{curr} *where* x_c *holds*	next_event(x_c) x

3.2.3.2 Recursive generation of THLDDs using Primitive Property Graphs

The method is based on partitioning PSL properties into elementary entities containing one operator only. 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 recursive *hierarchical construction* of a THLDD for a complex property using the PPG Library elements.

A *Primitive Property Graph* (PPG) is created for every supported PSL operator. All PPGs are combined into one *PPG Library*, which is extensible and should be created only once. It implicitly defines the supported PSL subset. The method currently supports only weak versions of PSL Linear-Time Logic (LTL) operators [75]. Yet, they are enough to derive a large set of properties expressed in PSL.

PPG is a THLDD graph. That means it uses an HLDD model with a temporal extension and has a *standard THLDD interface*. The standard interface was introduced to support the hierarchy in the recursive construction of a complex property. A PPG has one root node and exactly 3 terminal nodes (CHK., FAIL and PASS), as opposed to an arbitrary number of terminal nodes in a usual HLDD graph. It has also an optional relative time range Δt, which shows when the assertion has to be checked. The standard THLDD interface is shown in Figure 3.11.

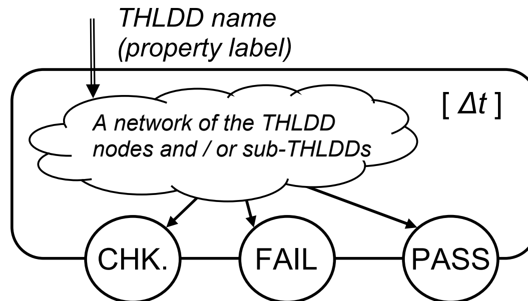


Figure 3.11 Standard THLDD interface

Sample PPGs created for 4 PSL operators are shown in Figure 3.12. The value for a PPG is obtained by evaluating the THLDD sub-graphs (e.g. P_a, P_b) and the intraconnections. The THLDD sub-graphs may be PPGs as well as Boolean expressions or variables. Note, that the logic implication operator '->' in Figure 3.12b exits to the terminal node 'CHK.' when the precondition P_a fails (as opposed to the logical *and* PPG in Figure 3.12d). This is because in assertion checking a verification engineer is not usually interested in vacuous passes of the property. *Vacuous passes* occur not because the property has met all the specified behaviour, but because the activation conditions of logical implication were not fulfilled.

THLDDs (and PPGs) without temporal relationships are evaluated to one of the terminal nodes at every time-step. THLDDs with temporal relationships may evaluate at arbitrary time-steps according to their temporal relationship function.

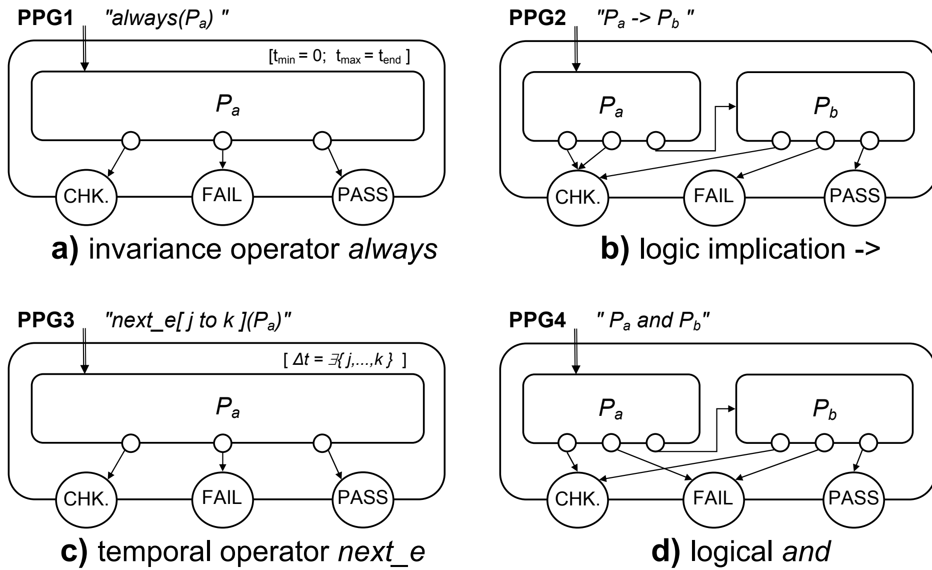


Figure 3.12 PPGs for a set of PSL operators

Complex THLDDs are constructed from elementary PPGs in a top-down way. Source PSL property is first partitioned into entities containing one operator only, following the standard PSL operators precedence [19]. THLDD construction starts from the lowest precedence operators which form the top level. Their operands (PSL operators with higher precedence) recursively form lower levels of the complex property. E.g. since *always* and *never* are operators with lowest precedence, 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.

Figure 3.13 depicts the resulting THLDD for PSL property *gcd_ready*:

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

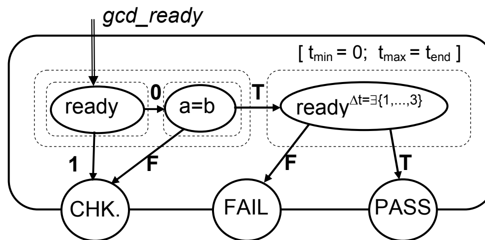


Figure 3.13 A THLDD representation of PSL property *gcd_ready*

The construction of the property *gcd_ready* implies usage of the four PPGs shown in Figure 3.12 and a PPG for the logical *not* operator. The nodes in the final THLDD contain pure variables and an HDL expression *(a=b)*.

3.2.3.3 Assertion checking using THLDDs

THLDD-based assertion checking utilizes the existing HLDD simulator and its extension for assertion checking presented by *Algorithm 2* in Figure 3.14. The execution of *Algorithm 2* is preceded by simulation of the design to calculate the *simulation trace*. Assertion checking analyses the trace by taking into account temporal relationships information at the THLDD nodes that represent an assertion.

Figure 3.15 shows an example of time windows for a THLDD converted from a two-operator PSL assertion *two_win*, which states that *x* should hold between the jth and k^{th} time-step starting from every time-step in the simulation trace:

two_win: assert always(next_a(j to k)(x))

Here the light-grey time window limited by t_{min} and t_{max} belongs to *always*. The dark-grey time window belongs to *next_a*. It is dynamic (moving along the time axis), denoted by $\Delta t=\forall\{j,...,k\}$, with size t_k-t_j and relative to t_{curr} (current position in time). Normally, depending on its complexity, a THLDD has one static (caused by invariance operators) and several dynamic time windows that can overlap.

```
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 until m_Current ∉ M^term
            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
        End repeat
        Assign x_G = x_Current at time-step t_now
    End for /* t= t_min...t_max */
End for
```

Figure 3.14 Algorithm 2. Assertion checking based on THLDDs

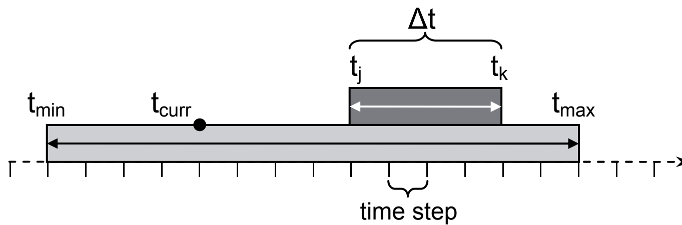


Figure 3.15 THLDD time windows in assertion checking

A general flow of the THLDD-based assertion checking process is given in Figure 3.16. The input data for the first step (simulation) are an HLDD model representation of the DUV and stimuli. This step results in a simulation trace stored

in a text file. The second step (checking) uses this data and the set of THLDD assertions as input. The output of the second step is the assertions checking results that include information about both the assertions coverage and their validity.

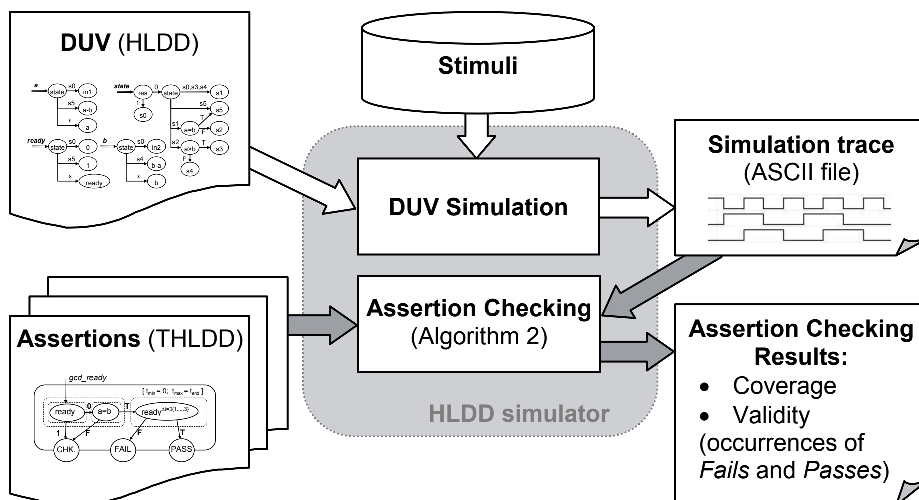


Figure 3.16 THLDD-based assertion checking process flow

To compare THLDD-based assertion checking *execution times* with those of a commercial state-of-the-art simulator supporting assertion checking, we have used a GCD implementation and 3 designs from the ITC’99 [70] benchmarks family.

For each benchmark, a set of 5 realistic assertions was created to contain:

- different types of operators (Boolean, temporal, implication, ‘until’);
- different resulting outcomes (fail, pass, both);
- various frequency of failures/passes (frequent, infrequent).

The assertions selected for GCD 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));

Assertions for b00, b04, b09 had the same temporal complexity as those for GCD. Each assertion checked 2-5 signals; the invariance operator (*always* or *never*) contained 1-3 LTL temporal operators from Table 3.2. Both simulators were supplied with the same realistic stimuli providing a good coverage for assertions.

Table 3.3 compares assertion checking execution times (in seconds per 10^6 stimuli) of THLDD simulator and the commercial tool. THLDD time is comprised of the simulation and assertion checking (*Algorithm 2*) execution times. The fourth (highlighted) and the fifth columns are the total execution times taken by the THLDD and the commercial simulator, respectively. Both tools have shown identical responses about assertion satisfactions and violations occurrences.

Table 3.3 Execution time comparison

Design	Checking time, s/10^6 stimuli			
	THLDD-based assertion checking			Commercial tool
	Simulation	Checking	Total	Total
gcd	2.07	4.87	**6.94**	13.52
b00	3.43	2.95	**6.38**	13.84
b04	5.47	3.61	**9.08**	19.23
b09	2.21	4.55	**6.76**	12.4

The experimental results show a significant speed-up (2 times) in assertion checking by the THLDD tool compared to the state-of-the-art commercial tool.

The speed-up is achieved by introducing the temporal aspect to HLDD model and by handling it inside the simulator instead of making HLDDs cyclic. Temporal windows are added recursively during THLDD construction from PSL properties.

3.2.4 HLDD for debug

The HLDD applications mentioned so far already allow us to quickly simulate a design, generate tests for it, measure the quality of existing tests in terms of code coverage, and check the verification assertions expressed in PSL in an efficient way during simulation — all with a single model. Once we have verified the design implementation with a good test set and discovered a bug in it, we would like to have a way to automatically locate the bug and correct it if possible. There is a good rationale behind such a desire given that bug location and correction together constitute around 1/3 of the total time spent on product design [76], [77].

In [67][co-auth] and [68][co-auth] we have shown how HLDDs can be used to accurately and automatically locate design errors. The proposed method was based on *backtracing* of the mismatched and matched outputs on HLDDs. In fact, all the errors injected in the experiments were identified as top suspects by the proposed diagnosis algorithm.

The motivation behind applying HLDDs for bug diagnosis was threefold. First, serious scalability problems appear [78], [79] once you try to apply to sequential circuits the diagnosis methods developed for combinational circuits (both, fault-model-based [80], [81] and fault-model-free approaches [78]). Second, most of the approaches do not scale well because of relying on formal SAT/SMT engines [82],

[83], [84], which despite being constantly improved are still solving an NP-complete problem. Our approach is different in that we rely on HLDD simulation that executes in polynomial time. This means much larger designs could be potentially handled by the proposed method. Third, SAT-based approaches reduce the diagnosis problem to logic-level formal engines. On the contrary, in [67][co-auth] and [68][co-auth] we operate directly on the RTL. This results in a readable diagnostic feedback and is therefore better understandable to the engineer than logic-level debug information provided by previous methods.

The method proposed in [67][co-auth] consists of generating diagnostic trees for primary outputs using backtrace during HLDD simulation and statistically analyzing the set of trees in two steps (basically, discovering unique nodes in failing trees that are missing in the passing trees and ranking them). The result of diagnosis is a statistically ranked list of HLDD nodes suspected of containing an error inside. Each diagnostic "tree" is effectively a set of pairs (x_i, t_j) that show at which time step t_j the variable x_i was backtraced. The pairs are arranged into a directed graph, where the vertices represent a subset of the time-expansion model of the design and directed edges show relations between the variables in the simulation process.

Examples of such diagnostic trees can be found in [68][co-auth]. The complete algorithm for diagnostic tree generation and the two diagnostic analysis steps can be found in [67][co-auth]. We are not providing them in this thesis, because a similar yet much more powerful approach based on IG model will be described in detail in section 4.6 of the thesis. What we want to mention though is that HLDD hardware description model inherently provides for a straightforward critical path tracing on a design. This makes possible a bug locating approach which in our experiments in [67][co-auth] in its step 1 always gave the injected error the top score in the suspected faults list. Therefore the method is accurate even though designs with sequential loops were targeted. It is also not limited to the single fault assumption. Last but not least, being based on polynomial-time HLDD simulation, the method is applicable to larger designs than can be handled by existing formal approaches.

From modeling perspective, it is important to notice that HLDD-based diagnosis is related to known debugging techniques such as program slicing [85] and critical path tracing [86]. Modeling discrete systems by a system of HLDDs may be regarded as a form of program slicing, because a separate diagram is generated for each variable x in the program, reflecting the control flow branches where assignments are made to x and including the data assigned to x. Activating paths in HLDD diagrams using *Algorithm 1* is equivalent to critical path tracing. The technique of critical path tracing consists of simulating the fault-free system (true-value simulation) and using the computed signal values for backtracking all sensitized paths from primary outputs towards primary inputs in order to determine the faults that would affect the primary output. In HLDDs the same task is solved in a single run as a byproduct of simulation.

There are still several problems with the proposed approach. Despite the high accuracy of locating injected bugs and the short run time of the proposed method, all the injected bugs were of artificial nature. It is therefore unclear how probable they are, i.e. how well they generally map to HDL source code. Also, while the diagnostic method is potentially applicable to large designs, it has not been proven so experimentally because of the limited power of VHDL-to-HLDD translator. Finally, in our method we rely on the availability of a golden design, which is a rare case in real life and may impede application of the approach to the real life situations. These three problems of scalability, direct HDL mapping and dependence on golden models were, however, successfully solved by the IG model and the debug method based on it. Both are described in Chapter 4 of this thesis.

3.2.5 HLDD for mutation testing

So far we have been using HLDDs as they are without changing them after they have been generated out of HDL descriptions. However, HLDD model also provides for straightforward model mutation. This allows further test quality improvement in terms of stringency of its results [66][co-auth].

Mutation testing implies creation of several mutated versions of the source code by introducing syntactically correct functional changes. Perturbation of the behavior of the code allows us to measure the effectiveness of the test suite in detecting the difference between the behaviors of the original and mutated codes.

In [66][co-auth] we have performed mutation testing on HLDDs and observed a clear advantage of mutation testing over the HDL code coverage approach. Such verification methods as HDL code coverage and functional coverage suffer from the observability problem. This means, while activating a bug (exercising the corresponding code), they guarantee neither propagation of the bug to an observable point, nor detecting the bug (i.e. observing a value mismatch at the given point). Mutation testing on HLDDs, on the contrary, guarantees observation and is thus more powerful in bug detection capabilities.

In HLDD models, a perturbation means a simple replacement of an operator, variable or constant labeling the HLDD node by another operator, variable or constant. Although not studied yet, edge mutation is also possible, but would require a special care inasmuch as arbitrary edge redirections may result in unrealistic bugs when translated back to HDL. Figure 3.17 illustrates HLDD graph perturbations for implementing the five key mutation operators [87] on a sample diagram G_{y_out}. Table 3.4 shows the list of replacements for each mutation operator implemented in [66][co-auth]. In the experiments, the original operator was substituted in every test by another operator from the group until all operators were covered.

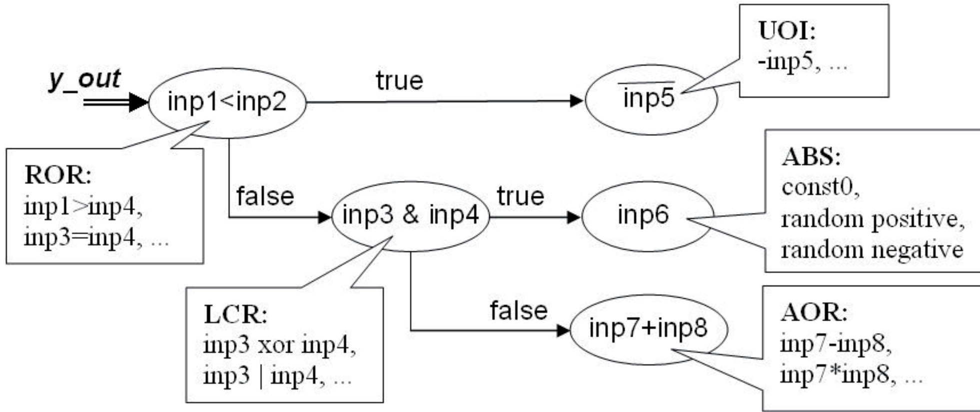


Figure 3.17 Key mutation operators as HLDD perturbations

Table 3.4 Five key mutation operators

Mutation operator	List of replacements
LCR (logical connector replacement)	AND, NAND, OR, NOR, XOR
AOR (arithmetic operator replacement)	ADDER, SUBTR, MULT, DIV, MOD
UOI (unary operation insertion)	NEG, INV
SOR (shift operator replacement)	SHIFT_LEFT, SHIFT_RIGHT, U_SHIFT_RIGHT
ROR (relational operation replacement)	EQ, NEQ, GT, LT, GE, LE, U_GT, U_LT, U_GE, U_LE

Experiments on several ITC99 benchmarks and an industrial example were performed. It was shown [66][co-auth] that, given the shortest possible test with 100% code coverage, mutation testing only achieves as low as 8% observation coverage in the worst case and 8-47% in the general case. With increased test lengths (1.5 to 100 times), the numbers for mutation test coverage grow up to 12-53%, still remaining very low (21% for the industrial example). This clearly states the need for better test sets, and gives an idea of how small observation coverage is guaranteed by 100% code coverage tests. This clear advantage of mutation testing over the coverage approach is due to considering fault observation.

Not only can mutations of the HLDD model be used for mutation testing directly, but these mutations can also be viewed as high-level faults and thus be used for high-level fault injection, as was the case with HLDD-based diagnosis described in the previous subsection. For a model, being able to provide for different tasks at once with minor or almost no adjustments is a highly valuable property a model can have.

HLDD-based mutations also have some disadvantages. First, there is only a limited number of possible mutations (either edges or nodes mutations) one can

have in the prepared set, and even out of those — not all are meaningful. These mutations are also bad in a sense for they are not mapped to HDL directly and hence may represent unrealistic bugs in the source code. Also, there is no guarantee that a simple atomic bug in HDL code would manifest itself through an equally simple atomic mutation in HLDD (i.e. it may be expressed as a relatively large change in HLDDs, e.g. imagine how a mutation of target in an assignment statement would be represented in HLDDs). Moreover, HLDD-based mutations are not intuitive to designers who are used to think in terms of HDL notions. But they are also faster than other alternative methods of error diagnosis, easy to implement, with the obvious cause-effect relationship. In the next chapter in section 4.6 we will propose an alternative way of error diagnosis which is based on HDL code directly and hence is free of the shortcomings mentioned above.

3.3 APRICOT verification framework

The author of this thesis has been actively participating in the development of HLDD-based debug and verification techniques since 2006, when the first attempts to automatically synthesize HLDDs from VHDL descriptions were made [52][auth]. Since then an entire hardware *functional verification framework* has been developed based on HLDDs which was named APRICOT (acronym for *Assertions checking, formal PRoperty checkIng, verification COverage measurement and Test pattern generation*) [57][co-auth], [64][co-auth], [66][co-auth]. The list of verification tasks supported by the framework includes assertion checking, code coverage analysis, simulation, test generation and automatic debug comprised of bug location and correction. Apart from evolving the HLDD-based verification theories as such, the author was also responsible for implementing several parts of the framework as separate tools and integrating the different parts into a single tool.

Hence, one of the author's main contributions is the development of *ApricotCAD verification environment* (a screenshot in Figure 3.21). ApricotCAD is based on the APRICOT framework and integrates different verification and debug HLDD tools into a single flow. The motivation behind the ApricotCAD tool is to ease the hardware debug and verification process through the unified environment and by adding different visualizations to the main verification engines, to increase the usability of the different tools and thus facilitate further research in the field, to polish and to debug the tools themselves in the first place, and to popularize HLDD-based tools and make them easier to use.

The APRICOT framework itself was developed during participation of TUT in Framework Program 6 European project VERTIGO [88]. Apart from TUT, the partners in the project were ST Microelectronics (coordinator), Aerielogic, TransEDA and three other universities: UNIV (Verona, Italy), LIU (Linköping, Sweden) and SOTON (Southampton, UK).

Within the APRICOT framework and its implementation as ApricotCAD tool we have targeted the aspects of *speed*, *accuracy*, *complexity* and *diagnosability* of hardware functional verification — all discussed previously in this chapter. The novelty of the whole framework lay in taking advantages of hardware design uniform representation by HLDD model.

Figure 3.18 shows verification flow within the APRICOT framework. The format of the DUV natively supported by APRICOT is VHDL. The framework has an automated interface from this design's description to HLDD representation model. There are also available additional interfaces to the APRICOT partners' internal and intermediate formats and therefore indirect support for several standard formats such as SystemC, EDIF and the intermediate format *HIF* (HDL Intermediate Format), produced by HIFSuite [89] and developed by UNIV.

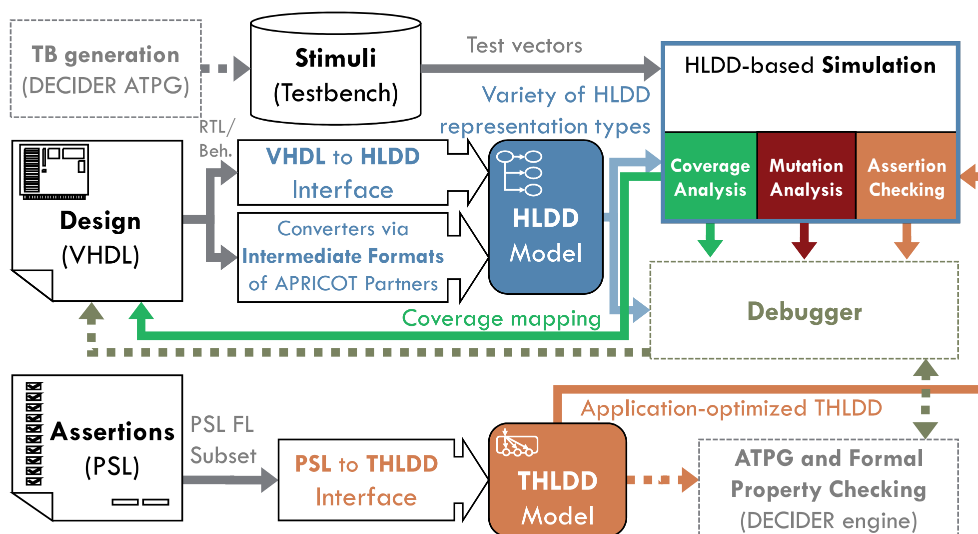


Figure 3.18 Hardware verification framework APRICOT

Once the DUV is converted to HLDD it can be simulated with HLDD-based simulator under the precomputed set of stimuli. The simulator also supports code coverage analysis, mutation analysis and assertion checking. The latter requires assertions to be represented by THLDD graphs. The framework has automatic interface for THLDD graphs creation from PSL assertions. The framework also considers hierarchical test pattern generation and formal property checking, as well as an HLDD-based debugger that exploits the model's easy cause-effect relationship diagnosability.

Figure 3.19 depicts ApricotCAD converters and assertion checker.

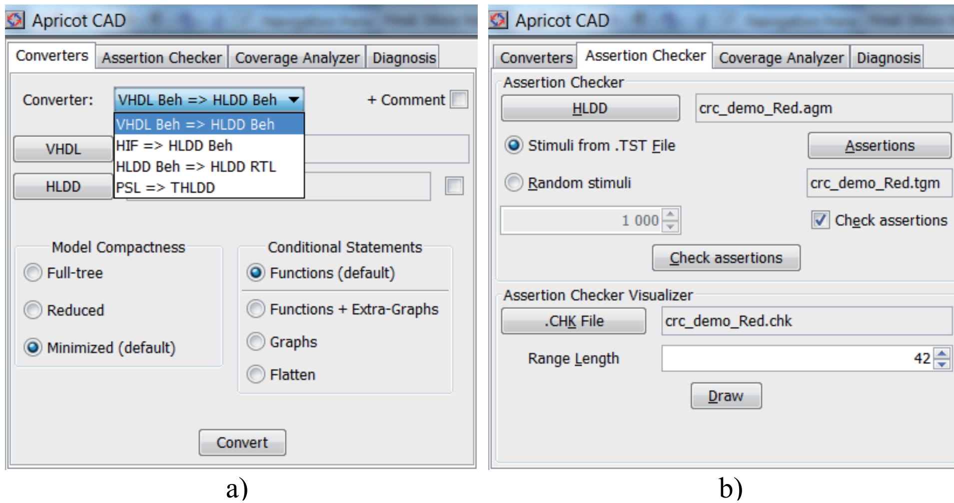


Figure 3.19 ApricotCAD converters (a) and assertion checker (b)

ApricotCAD features the following *converters* (Figure 3.19a):

- *VHDL Beh => HLDD Beh*: given the top-level VHDL file for a design, synthesizes its HLDD representation suitable for simulation and simulation-based verification engines (i.e. coverage analysis, assertion checking, mutation analysis, diagnosis and debug).
- *HIF => HLDD Beh*: same as the first one, but uses ApricotCAD's partner's intermediate format as the input format.
- *HLDD Beh => HLDD RTL*: synthesizes behavioral HLDDs into structural HLDDs suitable for test generation with RTL ATPG DECIDER [54] (see section 3.2.1 for more details).
- *PSL => THLDD*: synthesizes THLDD property representations from PSL descriptions and PPG library. The corresponding theory behind the synthesis process has been described in detail in section 3.2.3.

The main new extensions of the converters, as compared to [52][auth], comprise the support for hierarchical designs (multiple VHDL files), a PSL-to-THLDD synthesizer with a PPG library included, efficient handling of hardware memories, and extended support for widely used VHDL constructs. That means, in addition to introducing target application specific changes to the HLDD model being generated, a particular attention was devoted to supporting a possibly larger VHDL subset. This was important for, first, making the HLDD-based theories applicable to the real life designs and, second, to understand the limitations of the model itself so as to come up with solutions to overcome those.

When synthesizing HLDDs from VHDL files, ApricotCAD can distinguish between several compactness levels of HLDD representations that provide different tradeoffs between the analysis stringency and simulation speed with

memory requirements. The three options of *model compactness* (full tree, reduced and minimized) were discussed in section 3.2.2.1 in detail. Different options for representing *conditional statements* (see Figure 3.19a) were discussed in section 3.2.2.2. The appropriate options should be selected during HLDD generation to explicitly target specific HLDD-based applications. Similar HLDD compactness options (e.g. for collapsing/expanding conditional statements) are available in ApricotCAD when generating HLDDs from *HIF* intermediate format [89].

The *assertion checker* is shown in Figure 3.19b and Figure 3.20 below. In ApricotCAD, once a THLDD representation for a PSL assertion is created, it is co-simulated with an HLDD representation of the DUV by the same single simulation tool and the activation, violation or satisfaction times of the assertion are obtained. This information is both stored into a text file for later analysis and visualized in the ApricotCAD tool. Figure 3.20 shows simulation waveforms with assertion activations (a green line), violations (a red triangle) and satisfactions (a green triangle). If no assertions file is provided, the assertion checker works as a pure simulator.

Coverage analyzer is depicted in Figure 3.21. The user can choose to measure node, edge, toggle and condition coverages during simulation. Again, if no metrics is selected, the engine performs as a pure simulator. And again, the result of the measurement is both stored in a file and visualized in the framework. Visualization is performed in the following two ways. First, tabular data in the top-right panel conveys information about HLDD-based coverage and VHDL statement coverage (note the difference between them, how HLDDs are more stringent, i.e. 80% vs. 87%). Second, once you open any VHDL file from the given design in any of the two bottom panels, those lines (statements) of the file which were simulated by the given stimuli will be highlighted with orange color.

Debugger is also depicted in Figure 3.21. If there is at least one failing test in the test set, then, once the debugger is run from the Diagnosis tab, two lists of bug candidates are generated and the corresponding places in the design files containing the candidates are marked with either red or yellow colors (see Figure 3.21, bottom-left panel). The two lists of candidates correspond to the two diagnosis steps mentioned in section 3.2.4. Each list is obtained by using a different formula for ranking the candidates. With the red list, the debugger tries to reduce the number of candidates to a much shorter list which would lead to a greater speed-up in the overall debug process. In Figure 3.21, both red and yellow candidates happen to contain the bug on line 105. In our experiments we were injecting mutations ourselves using the mutation engine, so in that case ApricotCAD could frame the place with the actual mutation in a green box. In real life, though, we would not know where the bug actually is and only the candidates would then be shown (i.e. red and yellow lines, without the green box). Those lines will have to be analyzed manually by the designer, if the debugger fails to automatically correct the bug using simple automated mutations described in 3.2.5.

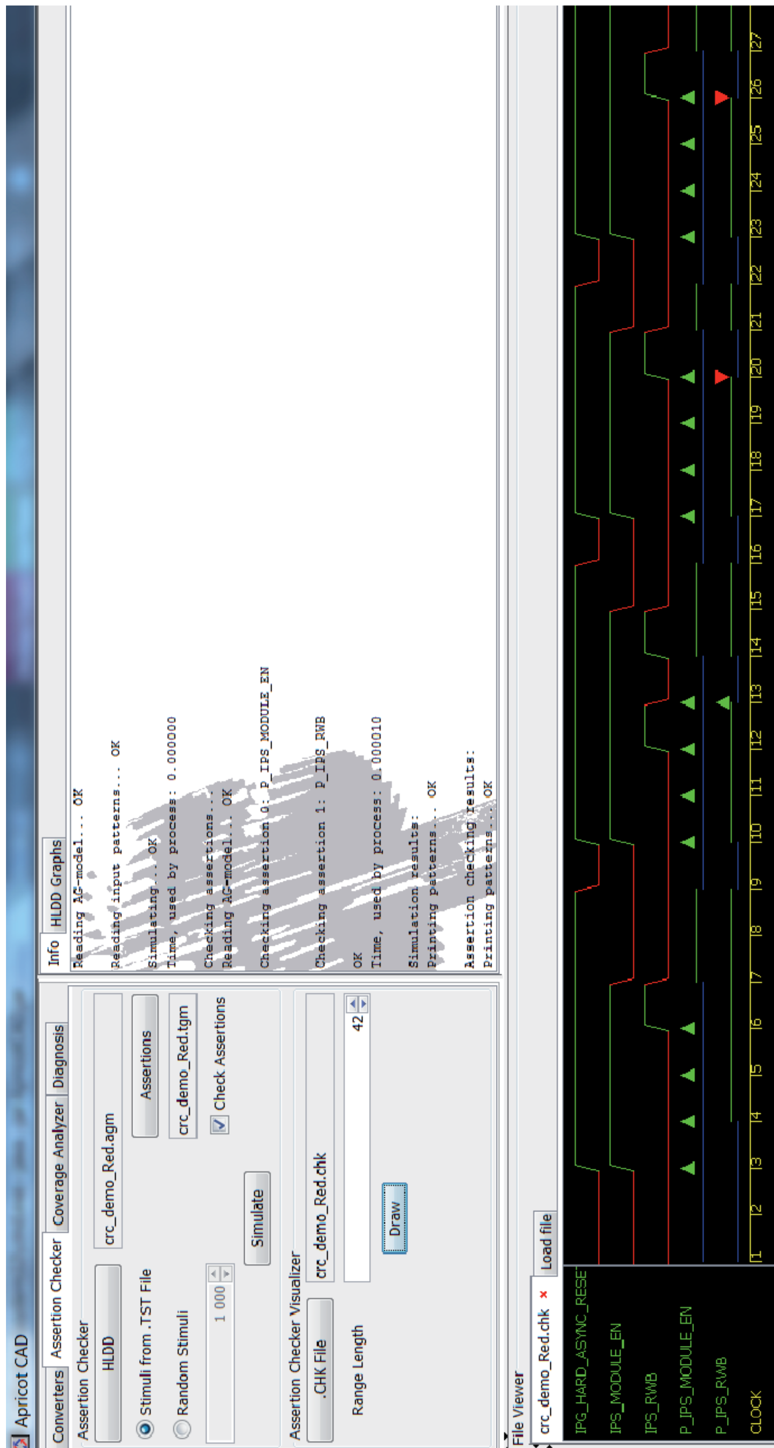


Figure 3.20 Assertion checker in ApricotCAD

Figure 3.21 Coverage analyzer, debugger and waveform viewer in ApricotCAD

3.4 Summary

This chapter discussed exploitation of HLDD model and its extensions in performing various tasks of hardware verification and debug.

First, *methodologies for generation of behavioral and structural HLDDs* from a wide subset of VHDL constructs were presented. This allowed obtaining HLDD hardware descriptions for larger VHDL designs than could be previously achieved.

Second, different *HLDD modifications* were proposed to address a set of verification tasks. *Expansion graphs and reduced compactness level of HLDDs* were proposed to achieve a more accurate code coverage measurement than commercial tools allow, while at the same time avoiding the large time overhead of coverage checking in commercial tools. *Temporal extension of HLDDs (THLDD)* was developed for fast assertion checking using a modified HLDD simulator. *HLDD model mutation* showed more stringent results in estimating the quality of a test set than code coverage measurement. HLDD mutations also proved a convenient means of high-level bug injection employed in estimating the efficiency of *automated bug localization method*. Backtracing was performed directly on the HLDD model as a byproduct of simulation. All these applications provide higher verification quality.

Third, the chapter introduced a *hardware verification framework APRICOT* and its implementation as *ApricotCAD* tool. The framework combines all HLDD-based tools into a single flow, which helps to ensure higher quality of produced hardware and to speed up the debug process during hardware design.

Chapter 4
HARDWARE MODELING WITH INSTANTIATION GRAPHS

Contrary to academic research, many industrial tasks in hardware design and verification require a scalable and fully elaborated model for representing large industrial designs. The full support of HDL constructs is also a typical requirement.

In this chapter we introduce a scalable Instantiation Graphs (IG) hardware representation model and a methodology for elaborating large HDL designs into IG model. This model is implemented inside an open source zamiaCAD platform which can be used as a scalable basis for building academic tools and complex industrial tools. As an example, a scalable automated debug method and its implementation inside zamiaCAD are introduced.

The contributions of this chapter are summarized as follows. First, it introduces the concepts of a new scalable elaborated design model IG. Persistence and scalability of the model are guaranteed by a custom designed object database referred to as ZDB (zamiaCAD Database) that is highly optimized for performance. The details of the comprehensive RTL design elaboration front-end are also presented. Second, it evaluates the model for different back-end applications. In particular, as a reference for the experimental part on IG scalability we have taken widely used commercial state-of-the-art tools and have considered for benchmarks very large RTL designs including a system-on-chip (SoC) with multiple thousands of LEON3 CPU cores. The considered back-end applications of IG are the static through-signal-assignment reference search, design simulation and automated debug. As the third contribution, the new RTL debug methodology is introduced and evaluated on a large industrial design.

4.1 Overview of zamiaCAD framework

zamiaCAD ([90], [91], [92])[co-auth] is an open source platform based on scalable models that includes, both, a comprehensive elaboration front-end for RTL designs and design processing back-end flows. The ultimate goal of zamiaCAD project is to allow its users to *elaborate* large RTL designs on relatively modest hardware, such as a typical hardware engineer's laptop workstation. And also to provide a *solid ground* for building other EDA tools on top of the scalable and fully elaborated model, which represents RTL designs as is (i.e. precisely as described by the designer). The framework puts particular emphasis on the following properties of hardware design models:

- lossless representation of hardware designs,
- scalability,
- accessibility to research and engineering communities,
- non-intrusiveness (standing HDL-centric),
- standing HDL-agnostic.

Non-intrusiveness is achieved by having an HDL as the only data format and requiring almost no configuration for pre-existing design projects. Functionally, the framework addresses advanced hardware RTL design, analysis, verification and debug. Currently zamiaCAD supports IEEE 1076-2002 VHDL [17] plus many extensions of the later standard VHDL 2008 in the front-end. Furthermore, the front-end may be extended to other languages due to the use of generic internal models described further on in this chapter. On the back-end side zamiaCAD supports design entry and comprehensive analysis. It includes an Eclipse IDE plugin based Graphical User Interface (GUI) and contains a built-in VHDL simulator, RTL debugger and work-in-progress stubs for synthesis. It handles very large open source (e.g. LEON3 [93] based) and industrial designs. Altogether zamiaCAD contains currently more than 100,000 lines of Java code (excluding generated parts).

zamiaCAD supports *bidirectional interaction* with external tools using Python scripts. This allows it to serve as an *integration platform* for academic tools and industrial zamiaCAD based EDA tools. From the outside, zamiaCAD’s internals are fully accessible from within Python scripts, so that existing zamiaCAD applications can be included into automated design flows. From the inside, zamiaCAD’s interfacing with external tools (either via scripts or directly) also allows gaining sizeable synergic effect, e.g. by importing VCD files and visualizing them in zamiaCAD, or taking timing information from external timing tools and annotating this information inside zamiaCAD’s HDL editor. This bidirectional communication along with the scalable and comprehensive HDL elaboration is expected to be of interest to, both, academia and industry.

For academia, zamiaCAD’s internal models serve as a *scalable basis* for further processing, so that researchers would not have to elaborate HDL from scratch

every time for every new task. For example, zamiaCAD allows one to create an RTL *simulator* by only requiring a thin executable layer to be implemented on top of the elaborated model. Moreover, zamiaCAD allows *combining the tools* together in order to gain a valuable synergic effect. One such example is described in section 4.6, where a decent real-life RTL *debugger* is produced by combining a simulator with static slice computation — a tool otherwise impossible because of the closed-source nature of EDA tools/models. Another example is the ongoing research on using zamiaCAD's internal model as a starting point for *translation* of RTL VHDL into TLM (thus raising the level of abstraction). This is useful for legacy code exploitation, because it allows avoiding execution bottlenecks during high-level simulation of multi-IP SoCs with legacy RTL IP cores in them.

zamiaCAD project was founded by Günter Bartsch. In [91][co-auth] we have first presented the details of the framework's main scalable internal model. In this thesis we augment zamiaCAD description by covering other hardware models that power EDA applications built on top of zamiaCAD main scalable model. We also contribute by extending zamiaCAD from being a code editor/navigation tool to becoming a debug environment with a built-in simulator.

4.1.1 Framework flow

Figure 4.1 shows a simplified overview of the zamiaCAD framework flow. zamiaCAD follows the previously described partitioning into a front-end and a back-end (Figure 2.2). First, in the front-end part, the HDL code is parsed which results in a (language specific) *Abstract Syntax Tree* (AST). AST is a pure tree data structure where the leaves of the tree are simple strings without semantic meaning. This tree is then elaborated into the *Instantiation Graph* (IG) model (designed to be HDL-agnostic). During the elaboration, the following three actions happen. First, all semantic rules as defined by the source HDL are applied to the AST, so that the tree of strings obtains semantic meaning. Now that we have this meaning, we can perform different checks: due to having the concept of types instead of pure strings we can check type matches, syntax rules, scoping rules, static array boundaries etc. But most importantly during the elaboration process identifiers (names) in the tree are resolved and replaced by references to the actual objects they denote.

Elaboration produces IG which is a very fundamental data structure in zamiaCAD as it is used as a basis for all the back-end applications offered by the framework such as analysis, RTL synthesis, simulation and automated debug. As of this writing, the front-end fully supports VHDL (2002 standard [17] in particular and many parts of 2008 standard excluding PSL), while Verilog only has a parser.

Both AST and IG models are stored in the zamiaCAD Database (ZDB) which is there for two reasons. ZDB guarantees the *persistence* of the models and ensures zamiaCAD's *scalability*. It is a custom-designed object database, highly optimized for EDA purposes. Next sections discuss each model and the database in detail.

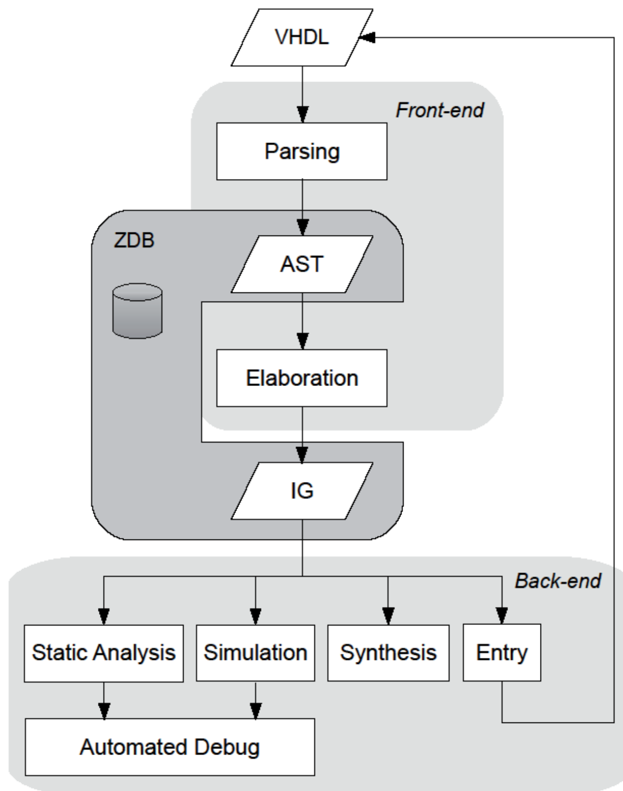


Figure 4.1 Simplified zamiaCAD flow

Back-end tools either utilize the IG capabilities directly or translate the IG model into a dedicated task-oriented design representation more suitable for the given task than IG itself. However, since IG model is at the core of the whole framework, there is a steady tendency to express as much of EDA specific details in IG as reasonably possible without slowing down IG computation. Thus, if there is any information which can be statically computed once and for all and later on reused by the back-end applications, then that information will be statically incorporated into IG during elaboration. Another benefit of having IG as a shared basis for all the back-end applications is the synergic effect we achieve when combining together several of these applications. We will demonstrate this effect in section 4.6 by an example of an automated RTL debugger.

4.1.2 Persistence and scalability

Since scalability is the primary issue when elaborating large designs, let us first see how ZDB solves this issue, and only thereafter look closer into how AST and IG models make use of it.

Especially for today's industrial designs, the data structures tend to grow very large, often larger than the available RAM on a typical hardware engineer's personal workstation. While many will argue that the affordable amount of RAM per workstation will increase exponentially with time, unfortunately the same holds true for the increase in size of the designs being processed so in effect we will always face the scalability issue.

Quite naturally, the solution is to hold only small parts of these data structures in memory at any given point in time while keeping the whole data structure on disk [94] — hence the requirement for a database. Of course, we have to make sure this is done in a quick and transparent way such that users would not even notice any effects apart from being able to handle large designs on modest hardware. Using disk space to store these data structures also solves the problem of persisting them since disk memory is persistent.

Also, persistence is very desirable as parsing and elaboration can take a lot of time, especially for large projects. It would be reasonable to keep the results from one zamiaCAD session to the next. Furthermore, zamiaCAD exploits principles of incremental elaboration. If only a part of the design has been modified, then only these updated design units will have to be re-elaborated to IG.

It is important to mention that experiments with Oracle database and several XML databases revealed that these are all way too slow in the EDA context, in fact, several orders of magnitude slower than the current working solution. The obvious decision was thus to implement zamiaCAD's own ad-hoc object database [95] and tune it for performance. For this, the benefits of Java programming language were exploited. Contrary to other mainstream languages, Java provides off-the-shelf a built-in object serialization feature which allows storing plain java objects on disk with almost no effort and ultra fast.

At its core, ZDB uses Java's built-in object serialization [96] to read and write objects from/to disk. Each object stored in ZDB gets assigned a numeric identifier called *DBID* which can later be used to retrieve the object. DBIDs are essentially pure integers and thus are represented by just 32 or 64 bits each (depending on the host CPU architecture). ZDB hence allows saving on the large objects which in the EDA field can sometimes occupy several hundreds of megabytes of RAM. Similarly, the lists of DBIDs which may also grow beyond the available memory are handled in an equally scalable manner described below in detail. ZDB also offers various indexing features that allow strings to be used to denote objects making the underlying *DBIDs* transparent for any code that uses ZDB.

Figure 4.2 gives a slightly simplified overview of ZDB's internal structure. As already mentioned, ZDB is highly tuned for performance. Generic performance features include the use of aggressive *caching* of objects in memory, including a least-recently used cache eviction strategy, as well as the use of standard *extensible hash maps* (EHM) [97] for maintaining potentially large indices on disk. The rationale behind EHMs is the weakness of the standard Java hash maps which do

not suit for representing very large indices typically met in EDA environment. Standard maps are very memory inefficient and once a hash map grows beyond the available memory, it crashes the Java Virtual Machine. Instead, EHMs can be partially stored on disk and partially kept in memory. Other optimizations are specific to the EDA use case: typically, we want to build large data structures very quickly and want to be able to traverse and index them as efficiently as possible. Deleting and modifying data, on the other hand, happen much less frequently. Therefore, ZDB does not really implement these last two operations, but only mimics them: object deletion will only result in it being removed from indices so it becomes inaccessible, but it will continue to be stored on disk (*deleted obj* in Figure 4.2). Object modification is not supported at all, but will require the application using ZDB to delete the old object from ZDB and store it again — a technique widely used in enterprise databases (e.g. *vacuumdb* utility in Postgres [98]).

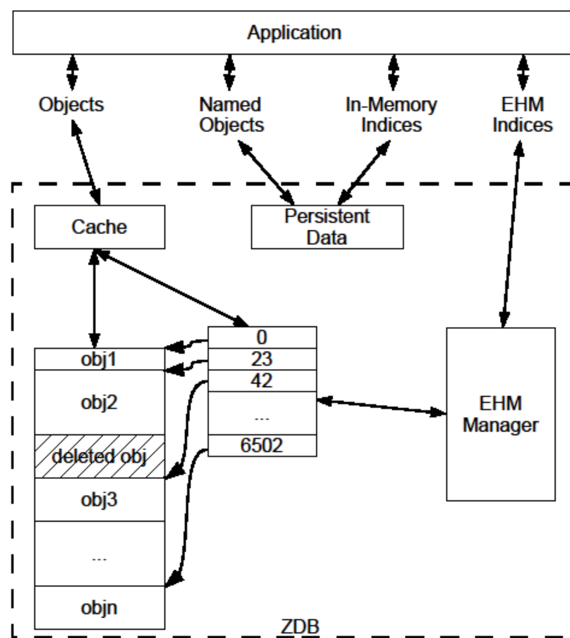


Figure 4.2 zamiaCAD database ZDB

Based on the same principles (creating/traversal should be fast, deletion and modification are infrequent) throughout ZDB and in the rest of zamiaCAD we use two custom data structures *HashSetArray* and *HashMapArray* which are essentially crossovers between regular arrays and hash sets or hash maps. This combination allows preserving the order of added elements (the ordered set/map semantics) while providing a constant time access to elements by, both, their keys and indices.

To wrap up, from the user's perspective, ZDB is used by feeding a java object to ZDB and getting in return a unique key (DBID) for this object — just like any other database, but much faster. And then one can store in memory only this key instead of the whole object. In addition to DBIDs one can also use strings for referencing objects: an indirection layer translates strings to DBIDs transparently.

Now that we have this possibility of replacing large objects with small DBIDs (pure integers), let us see how it can be used in the construction of AST and IG.

4.2 AST model and its limitations

The AST model in the zamiaCAD flow serves mainly as an intermediate step in the IG elaboration process. Its only particularity with respect to traditional AST approaches is its scalable handling by ZDB. This allows the model to accommodate large net-lists, weakly supported by even proprietary tools that rely on AST-like design models. This is particularly important, because today it is a common practice in industry to use net-lists for certain IP cores in a design.

The AST generated by zamiaCAD's parsers is a language-specific, traditional tree structure closely modeled after the grammar productions as specified by the respective HDL language reference manual. It is closely correlated to the source code processed by the parser as there have been very few semantic rules applied to it yet. The AST is a true tree data structure which is due to the context-free nature of the grammars used.

We will illustrate this by looking at the AST generated from a very simple VHDL example given in Figure 4.3. Figure 4.4 shows a slightly filtered and simplified AST for the architecture part.

```
entity foo is
  port (A, B : in   bit;
        Z    : out  bit);
end;

architecture rtl of foo is
  signal  t : bit;
begin
  T <= A xor B;
  Z <= T;
end;
```

Figure 4.3 VHDL Source Code Example

It is important to notice that there is no semantic information present in this data structure. Instead, any VHDL object is still denoted by names (which are, in this example, simple identifiers, but could be more complex). If we look at the way the signal declaration is represented here, this is very obvious: the signal *T* is declared as a subtype of *BIT*, but we have no information what the *BIT* type really is (while the experienced VHDL designers, of course, know this is one of the types declared in the *STANDARD* package). It is exactly what semantic information means.

The same problem can be observed when looking at the concurrent signal assignments. The relevant signals are denoted by names which have not been resolved yet, since that would require semantic knowledge which a parser does not have. To be more specific, the two signal assignments in the figure refer to the same object: the first assignment has its target denoted by identifier *T*, while the second one has *T* as its driver. These are two different identifiers in the tree, denoting the same object, but this object only gets created during elaboration.

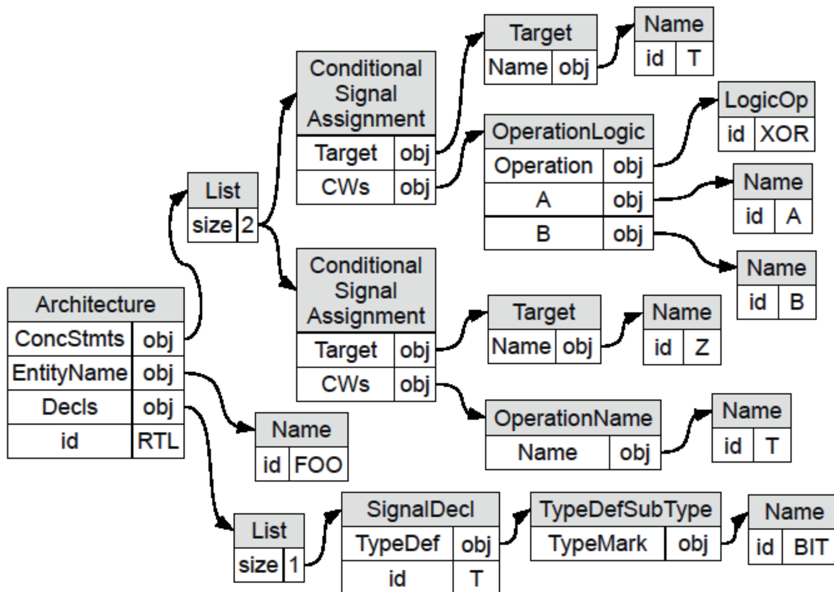


Figure 4.4 Abstract Syntax Tree

To address the scalability issue, all AST nodes which mostly consist of strings are persisted using ZDB. Due to the pure tree nature of the AST (that is having no cycles in it), persistence can be done simply by storing individual library/design units as ZDB objects while keeping track of them using indices, so they can be found quickly during elaboration. This kind of simple ZDB persistence also gives us basic scalability, for we can essentially process any number of library/design units very efficiently as long as individual units fit in RAM.

Unfortunately, with automatically generated flat HDL net-list dumps this assumption does not hold. Such dumps can contain millions of concurrent statements in one single unit causing the parse pass to run out of memory. To solve this, an additional level of ZDB indirection has been introduced: for architectures, we store each concurrent statement separately using ZDB and keep just a list of *DBIDs* (simple long integers) in memory. That way, zamiaCAD can deal with arbitrarily large net-list dumps very efficiently.

Besides these scalability precautions, in the rest of it the AST parser is a standard top-down parser generated by JavaCC parser generator [99], but having the standard JJTree tree builder [99] replaced with zamiaCAD's own hand-coded scalable tree constructing procedure injected directly into the generated parser.

4.2.1 Applications of AST model

Because of the lack of semantic information, AST applications cannot be very complex. Generally AST applications in RTL design flow may include editor support such as folding and outline view (e.g. zamiaCAD relies for these two applications on the AST model), identifier completion proposals, simple identifier occurrence tracing by scoping rules within the file (local tracing), local identifier refactorings and simple signal value annotations. However, any task (e.g. static analysis, simulation) that needs identifier resolution, including type information, can be performed only on properly elaborated design models such as IG.

4.3 IG model

The process of design elaboration in zamiaCAD results in its representation by an Instantiation Graph (IG).

Definition 3: The *Instantiation Graph* is a data structure represented by a densely connected graph of semantically resolved objects representing elements of the hardware design.

IG is generated by applying semantic rules to the AST to resolve all names (identifiers) and replace them with references to the actual objects. While consistency and rule checks (e.g. type checks) are also applied during this process, the main purpose here is name resolution. The names can reference objects located in outer scopes, packages or other units and therefore IG partitioning prior storing to a database should be performed with care. This is done as described below.

Just like AST objects, IG objects are persisted in ZDB. However, due to IG being a true graph with many cross-references between items, storing IG structures takes considerably more effort than AST sub-trees do. Fortunately, since name resolution is the source of all these interconnections, it makes it a natural point to start cutting the graph to pieces exactly there when partitioning IG for storage. So,

anything which can be denoted by names needs to be stored as a separate ZDB object and (except for transient fields) is never allowed to be referenced directly, but through the object's DBID only. This is mainly true for two kinds of objects: *IGObjects* which denote VHDL objects (signals, variables, constants and files) and *IGTypes* which denote types. Besides that, we also apply the same approach we used in AST to handle large dumped net-lists: *IGStructures* do not reference concurrent statements directly, but store them as individual ZDB objects and only keep their DBIDs (which are simple long integers).

Currently, IG is modeled closely after the ideas outlined by the VHDL language reference manual [17] when it comes to elaboration. However, IG is designed to be language-agnostic, so it should be possible to extend it to support e.g. Verilog [100] or AHDL [101] by simply adding missing classes and attributes to cover concepts specific to those languages. Thus, the main idea behind IG is to abstract away the *syntax* and to map constructs with equivalent hardware *semantics* to the same IG classes regardless of the language the source code was written in, while maintaining proper back-annotations to the source code. Facilitating factors which make it easier to introduce new HDLs into IG are:

- already available persistence and scalability mechanisms of ZDB;
- knowing well the places of required ZDB indirection (same "cut-through" points in the IG graph for all possible RTL languages);
- most of the RTL hardware related constructs are already expressed in IG due to virtually full VHDL support;
- the source code of zamiaCAD contains a working example of how elaboration is implemented for VHDL, e.g. how name resolution is done etc.

Instead, most of the conceptual difference (if any) is expected to appear in the semantics of the notions defined by the new languages, and hence will rather be a bother of the simulator, synthesizer, etc. and not of IG or elaboration engine.

4.3.1 Modules in IG

Elaborated units are called *modules* in IG. Apart from library packages, they form the topmost hierarchical level in IG. Figure 4.5 shows what the high-level IG data structures look like for our example from Figure 4.3. Note that it is impossible to show a complete IG even for a very basic design, because IGs have far too many nodes and edges. We will therefore have to look at small, hand-selected and simplified portions of IG. IG does not separate between entity (interface) and architecture (body). Also, modules do not have generics — instead, for each distinct set of generics, which is used anywhere in the design, a new *IGModule* is generated. The set of actual generics (a set of static values) is stored inside the module and is also used to compute a signature (a simple string) for the IGModule. Therefore it is possible to quickly locate it via a ZDB index during elaboration or reference it by *IGInstantiations*.

The actual body of the module is to be found in an *IGStructure* object. Since modules can be hierarchical (e.g. when the original VHDL architecture uses blocks or generate statements), IGStructures can contain IGStructures in a recursive way. Hence, IGStructure is one of only three possible statements to be found in an IGStructure:

- IGStructure
- IGProcess
- IGInstantiation

All other kinds of concurrent statements found in VHDL are mapped to equivalent *IGProcess* statements in IG. In our example we have two concurrent signal assignment statements which have been converted into two equivalent IGProcess statements.

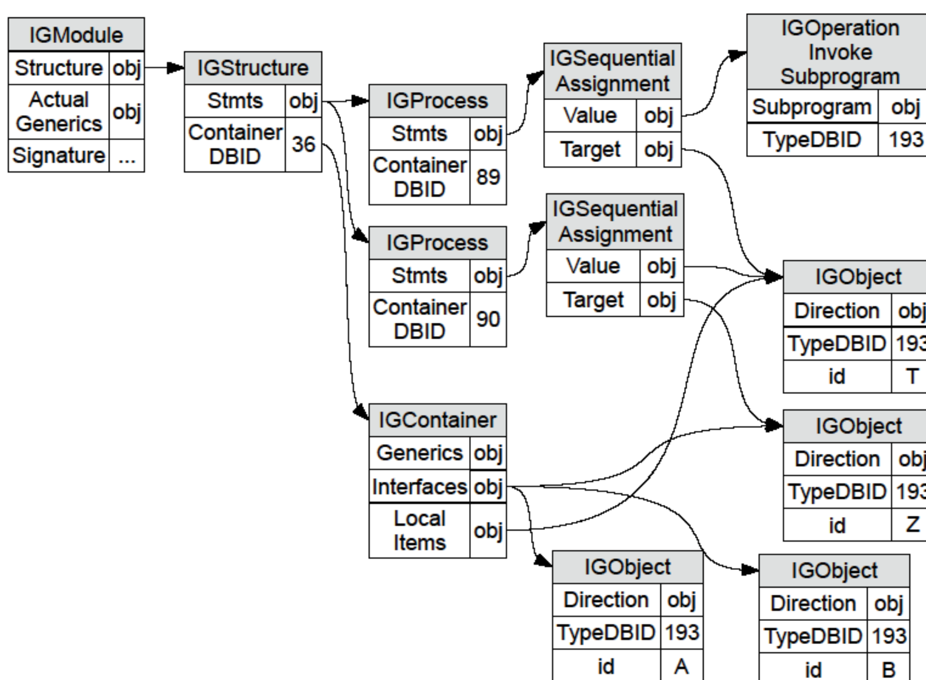


Figure 4.5 A Module in IG

Besides the concurrent statements, IGStructures also contain information about all (VHDL) objects declared in their scope in an *IGContainer* object. IGContainers can be regarded as namespaces — they contain references to all named objects in a specific namespace. In our case, we find four different objects here: *A*, *B* and *Z* which are interfaces declared in the entity, and *T* which is a local signal. In a similar manner, library packages, represented in IG by *IGPackage* objects, consist of a single (possibly hierarchical) IGContainer which holds references to all the named objects they provide. Packages are located via a global index (scope) called *GlobalPackageContext* which is stored as a named object in ZDB.

4.3.2 Objects, types and expressions in IG

Expressions illustrate best the difference between AST and IG. In IG, all names have been resolved so they can be replaced by a reference to the specific object which was denoted by them.

Figure 4.5 shows what the concurrent assignments found in our example from Figure 4.3 look like in IG. The two IGProcess data structures correspond to the two statements as discussed in the previous section. Also where in AST we have previously used *T* in two different places, we now have only one *IGObject T* with two links to it (plus one for the declaration). Let's concentrate now on the first assignment with an operator in it:

```
T <= A xor B;
```

In IG, this is turned into an *IGSequentialAssignment* statement inside a process. The target *T* of the expression is the only VHDL object directly referenced by this sequential assignment. The other two objects are hidden behind a logical operation expression. In IG such expressions are actually translated into subprogram calls, which is due to the fact that operators in VHDL can be overloaded and therefore need to be resolved just like any other name. So, in IG we have a direct reference to the specific subprogram which is to be invoked to perform the correct operation.

Likewise, names denoting types are resolved and replaced by references to the actual objects representing those types. Figure 4.6 shows the corresponding IG sub-graph for our example. The *BIT* type from the package named *STANDARD* (which is imported implicitly in any VHDL unit) has been resolved to an *IGTypeStatic* object, which is a subclass of the more generic *IGType* class that can also represent arrays not having static boundaries (can be used in subprograms). The *BIT* type is comprised of the two enumeration literals *0* and *1* represented by *IGStaticValue* objects. Such objects are used in IG to represent any kind of static (constant) value.

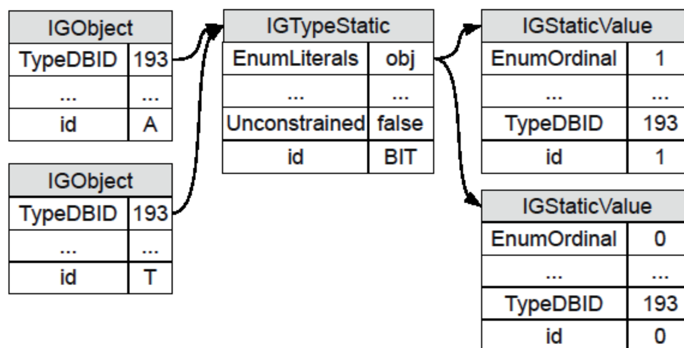


Figure 4.6 Types in IG

4.4 Application of IG model to verification and debug

IG model applications are much more powerful than those of AST. Any static analysis task that needs identifier resolution (including type information) becomes possible with IG. Such tasks include tracing of parts of signals, precise cross-module (global) tracing, tracing through *generate*-statements, correct resolution of overloaded subprograms, global refactorings, advanced signal value annotations (e.g. annotating only one bit of a vector), computing expressions on the fly, filtering a design with static slices. Besides static tasks, IG allows different visualizations. E.g. IG can be mapped to semantically equivalent RTL graphs using RTL elements. In the resulting RTL graph, the dataflow is represented as an interconnection of logic and arithmetic modules with automatically inferred (where appropriate) memory elements. IG also allows dynamic tasks such as simulation.

In the following two sub-sections we will describe the tasks of signal tracing and simulation. The task of debug will be discussed in a separate section 4.6.

4.4.1 Static analysis

Static analysis tasks are there to boost designer's productivity during design and particularly during debug. For our experiments, we have considered two such static tasks: signal reference search (drivers/readers) over the whole design, and computation of a static slice for a given signal. Both tasks can be carried out on the IG model directly in almost no time and thus do not require any other additional data structures. The main prerequisite for fulfilling static tasks is the fully elaborated design model, which IG inherently is.

To the best of our knowledge there are no other tools capable of neither computing static slices nor helping a designer to grasp all the references of a signal he is interested in (i.e. drivers and readers). That means, so far designers have had no CAD tools to help them during debug when trying to trace down the source of an error, and also during design exploration when trying to find where a certain signal obtains its value from and which parts of the design it influences. ModelSim does allow tracing signals dynamically, but the final result completely depends on the stimuli quality, which makes such a dynamic analysis just not as comprehensive as static analysis is. We therefore only show whether it is possible to perform the two mentioned tasks with IG model on large designs or not — and this way demonstrate the usability and effectiveness of IG model.

We have used the following designs as benchmarks:

- **B19** - a design from ITC'99 benchmarks family [70] combining several smaller circuits from it (a total of 200 000 gates).
- **Plasma** - an open-source MIPS processor benchmark [102] from the OpenCores.org website.

- **L3-SoC** - a SoC built by interconnecting a wide set of IP cores from Gaisler Research Library (GRLIB). It is available from [92].
- **L3-1** to **L3-3584** - the original unchanged LEON3 CPU core design [93] configured to contain *n* instantiations with *m* CPUs as AMBA masters. *L3-64* is 64 instantiations with a single CPU on AMBA bus (64 core design). *L3-896* is 64 instantiations with 14 CPUs on AMBA bus (896 core design). *L3-3584* is 256 instantiations with 14 CPUs on AMBA bus (3584 core design). The tools capable of detecting identical modules inside the design will avoid duplications and thus save memory resources.

Table 4.1 and Table 4.2 present results for the two static analysis tasks: global signal reference search and static slice extraction. For the reference search, the experiment consisted of picking some signal widely used across the design and finding all the usages of this signal. For this task, in case of *B19* design, AST-based tools would find only 39 occurrences of identifier *CLOCK* in the design file due to ignoring multiple instances of components, as opposed to 121 actual references found in the elaborated design by zamiaCAD. In case of the *L3-1* and *L3-3584* designs the precise global signal tracing (tracing through *generate*-statements) also results in a larger and more correct amount of actual references. Note that it only takes about 20 seconds to scan through the largest SoC from our set — that is, to traverse the whole big IG graph without any indices and optimizations. The reference search task is thus established doable.

Table 4.1 Reference search

Design	Signal	References (#)	Time (ms)
B19	CLOCK	121	22
L3-SoC	IRQO	71	31
L3-1	CLK	6	7
	RST	5	8
L3-3584	CLK	1281	20203
	RST	1025	23241

Table 4.2 Static slices

Design	Signal	References (#/%)	Lines (#/%)	T (s)
B19	BS_{read}	56/9	315/23	0.4
L3-SoC	ATA_DMACK_{drv}	1984/51	5846/2	26.8
L3-1	ERX_DV_{read}	2746/53	4269/1	19.0
	$ERX_DV_{lim=3}$	58/1	288/0.1	2.2
L3-3584	ERX_DV_{read}	3001/0.05	4269/1	103.4
	$ERX_DV_{lim=3}$	313/0.006	288/0.1	69.1

A static slice represents a subset of those signals/variables and their assignments/querying that affect the signal of interest. Slices can be optionally filtered to only readers or only drivers of this signal, and can be limited by depth of the dependency. In other words, the readers of the signal constitute its forward slice (a.k.a. cone of influence), while its drivers constitute its backward slices (a.k.a. cone of dependence). Table 4.2 shows the size of static slices for signals *BS*, *ATA_DMACK* and *ERX_DV*. We define the size of a slice both in terms of signal/variable references and code lines. Percentage shows what portion of the design the slice constitutes. *B19* has 56 reference signals and variables (appearing on 315 lines of code) that are dependent through signal assignments on signal *BS*. The dependency cone for signal *ATA_DMACK* contains 1984 or 51% of the signals and variables in design *L3-SoC*. There were also extracted static slices for benchmarks *L3-1* and *L3-3584* containing all readers and readers within dependency depth limited to 3 for signal *ERX_DV*. In case of the *L3-3584* design the static slice involves readers in all instances of the processor core. When the depth of the slice is limited to a small number of levels, the obtained portion of the design (the percentage in Table 4.2) is rather small. The slices virtually filter a design with respect to some signal. In such a way, slices help designers to concentrate on the important code areas during debug. And their computation with IG model has been shown to be doable.

All experiments were performed on an Intel® Pentium® Dual CPU 2.2GHz machine with 3.8GB of RAM running under Linux OS distribution CentOS 6.0. This roughly corresponds to a typical hardware engineer's workstation.

4.4.2 Simulation

In this section we describe the way IG model can be simulated and the models required for that. To simulate IG model, we need to make each IG object know how to interpret itself and have a simple simulator manage the overall process, in particular, by introducing the concept of time into the simulation procedure.

One way to do that is to add the required execution semantics right into IG structures (the widely used *interpreter* design pattern [103]). However, IG would then be polluted with execution-related data such as execution context, signal values, future waveform values. Instead, to keep IG model lightweight (as any general-purpose model should ideally be) we have extracted all the execution semantics out of IG into a *separate, stand-alone interpreter*. This way, a possibility is also reserved to optimize the simulation process independently from IG.

Each key IG structure thus generates its own portion of executable code (a list of executable objects) and simulator runs this generic code in the same way compiled programs are run on general purpose processors. A sequence diagram and a generalized IG simulation flow are depicted in Figure 4.7 and Figure 4.8. Inside the simulator, most of the work is delegated to the very same *interpreter* previously used to compute static values on the fly during elaboration. And the code generated

by IG is in fact the *interpreter code* (typically generated for each *IGProcess*). Simulation thus means executing interpreter code by the interpreter and producing static values as a result of this code evaluation. Obtained values are then stored hierarchically inside *interpreter contexts*. The only aspect left to be covered explicitly by the simulator is the concurrent timing model for signals, which in case of VHDL is the delta-delay timing model complemented by signal drivers.

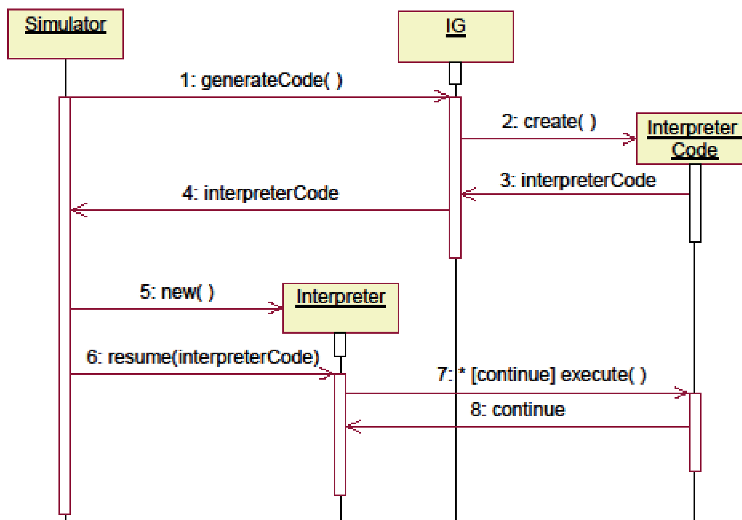


Figure 4.7 Simulator sequence diagram

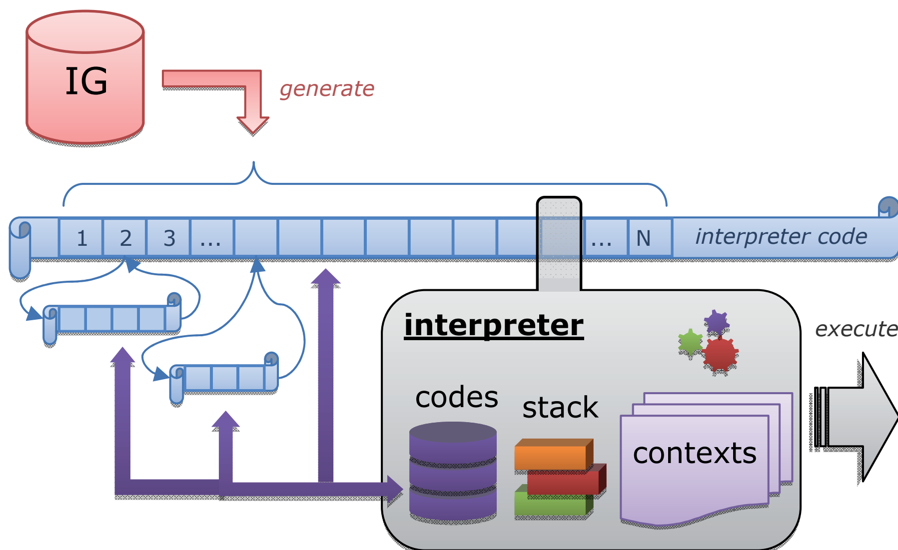


Figure 4.8 Generalized IG simulation flow

The *interpreter* (Figure 4.8) is implemented as a stack machine. It maintains different interpreter *contexts* (variable contexts), a *call stack* (interpreter codes) and

a *program counter* (PC). When traversing the interpreter code, interpreter uses PC to keep track of the statement being currently executed. In case the statement calls another interpreter code (e.g. of a function; the short codes in Figure 4.8), the current code and its PC get stored into the call stack for further resuming, and the new code is loaded and PC is reset. After the new code execution has completed, the stacked code is resumed by popping it out of the call stack and restoring the PC to the previously saved value.

The *interpreter code*, which is a list of executable statements, represents the execution semantics of the given HDL. Its task is to compute new values for signals and variables. It takes current values from signal and variable *drivers* (organized hierarchically into interpreter contexts), processes them using a variety of value-producing executable statements, and writes new values back to drivers by scheduling values for assignment using the same drivers. Intermediate results of value computation (drivers, types, static values, literals) are passed from one executable statement to another through a *value stack* of the interpreter.

Drivers are also used to access individual elements/ranges of complex signals or to write to parts of signals. In the following VHDL code fragment:

```
tar (7 downto 4) <= temp (15 downto 12);
```

the target will be represented by a range driver mapped to the leftmost 4 elements of the original driver of *tar* if *tar* it is a (7 downto 0) array. The mapped driver and the assigned value will be pushed to stack by corresponding interpreter statements and will be reunited by a statement which pops the two objects off the stack and schedules a value change of *tar* signal using the mapped range driver.

Execution of any given interpreter code is finally terminated with a *wait* executable statement, which passes control to other interpreters that evaluate the rest of IGProcesses. Also, all executable statements that constitute an arbitrary interpreter code are properly back-annotated to the original source code they were generated from by the IG structures. This allows measuring HDL code coverage.

4.5 Scalability assessment

In this section we present the results for scalability experiments [91][co-auth] on IG model implemented in zamiaCAD. We have taken for comparison the leading commercial EDA tools which all target different tasks, but all have their own elaboration engine inside. To compare these engines only, we have run each tool up to the point where only the elaboration step had been performed. For example, for simulators it is the point right after the simulation has started with 0-cycle typically executed.

To estimate scalability of IG model, we have used as a reference widely known commercial state-of-the-art tools for synthesis (T1), for design elaboration front-

end (T2), a commercial and an open-source tool for simulation (T3 and T4) and an advanced code entry tool T5. All of the tools create their own proprietary elaborated design representation models dedicated for the targeted task. We show that only proposed IG model can easily handle designs of the given complexity, while the other models fail to do so.

We have used the same design benchmarks as with static analysis experiments in 4.4.1. All the designs share the common property of being publicly available. It is important to mention, though, that during its development zamiaCAD and its IG model have been constantly tested for scalability issues against state-of-the-art IBM SoCs which typically contained tens of millions of VHDL code lines. But since those designs could not be made publicly available to other researchers, we have come up with a set of open designs of similar size and complexity.

Table 4.3 Elaboration time (sec)

Design	Size	T1	T2	T3	T4	T5	zC
B19	834	7	1	5	1	4	3
Plasma	1068	3	1	9	2	7	9
L3-SoC	38107	83	5	135	104	918	183
L3-1	27409	87	6	82	187	694	137
L3-64	27409	OOM[1]	23	87	OOM[7]	693	162
L3-896	27409	OOM[2]	97	OOM[5]	OOM[8]	695	223
L3-3584	27409	OOM[3]	OOM[4]	OOM[6]	OOM[9]	692	405

Table 4.3 presents the elaboration time for the previously described large benchmarks. The second column presents the size of the designs in terms of VHDL concurrent and sequential statements (a more accurate metric than lines of code).

Here OOM^k means a reference tool has run out of memory or crashed for the given design. $OOM^{1\text{-}3}$ - T1 has run out of memory in 448, 860 and 2340 seconds, respectively. OOM^4 - T2 has crashed on 208th instantiation with 14 CPU cores (2912th of the 3584 cores). $OOM^{5\text{-}6}$ - T3 has halted after 3170 and 949 seconds, respectively, and kept waiting infinitely for additional memory resources. $OOM^{7\text{-}9}$ - T4 has run out of memory in 16, 8 and 8 seconds, respectively.

In case of zamiaCAD, the elaboration time measured includes indexing, parsing, IG elaboration and ZDB commit (flushing ZDB contents to disk). For instance, the total time spent for elaboration of *L3-1* and *L3-3584* designs can be split, respectively, to (137.44s total = 1.44s parsing + 3.57s indexing + 95.19s IG build + 37.24s ZDB commit) and (405.16s total = 1.16s parsing + 5.06s indexing + 363.73s IG build + 35.21s ZDB commit).

Table 4.4 presents values for memory allocation both in RAM and on hard disk drive for handling the internal models during the elaboration process (see Table

4.3). With this experiment we show that, given a reasonable amount of RAM, zamiaCAD models scale well with the design size.

Most of the compared tools were rather upset about the idea of dealing with such large designs. These tools either crashed with the out of memory exception or just kept sitting in the background waiting for additional memory resources. Apart from zamiaCAD, only VHDL editor T5 survived the test. However, the VHDL editor only builds an AST and does not elaborate the design (can be derived from the memory requirements which do not change with the growing number of cores in the design). VHDL editor also does not use hard disk (model size on disk is constantly 0), which is exactly the reason why it does not accept net-lists — it simply can't. Hence we can draw our *first conclusion* here: without utilizing the hard drive it is not possible to handle large designs (e.g. net-lists) already on the stage of AST. Alternatively, even if you use hard drive as the rest of the tools are trying to do, you still have to do it in a scalable manner, not hoping that a good-old HashMap will be enough for EDA data structures, large as they typically are. That is our *second important conclusion* drawn.

Table 4.4 Internal model size in RAM / on disk (MB)

Design	T1	T2	T3	T4	T5	zC
B19	120/0.48	17/0	5/0.32	1/2	254/0	248/0.9
Plasma	70/1.3	15/0	5.3/0.68	3/2	280/0	269/5
L3-SoC	310/32	117/0	39/22	45/47	693/0	1337/114
L3-1	268/35	107	72/14	45/66	613/0	1332/83
L3-64	OOM[1]	332	444/14	OOM[7]	610/0	1360/134
L3-896	OOM[2]	1205	OOM[5]	OOM[8]	619/0	1465/236
L3-3584	OOM[3]	OOM[4]	OOM[6]	OOM[9]	616/0	1488/584

Concerning the memory requirements, both T5 and zamiaCAD were given 1424MB of RAM memory. As to the actual memory consumption shown in Table 4.4, the VHDL editor keeps it on the same level for designs with different number of LEON3 cores, because it does not go beyond the parsing process and thus knows nothing about multiple CPU instances. On the contrary, with large designs zamiaCAD tends to use all the available memory and thus with the growing number of cores the memory consumption grows too, although insignificantly.

Generally, in case of memory shortage zamiaCAD can scale down to the available memory resources and is still able to do the job while sacrificing the speed — and vice versa, it can often do the same job faster given a larger amount of memory. The reason why the task gets completed a bit faster is because of the less intense writing to disk, which is also why the difference is not that large for the bigger designs where the elaboration time prevails over the commit time. For example, the saving in design elaboration time for the case when memory

resources in zamiaCAD were extended 1.88 times (from 1.4GB to 2.7GB) is between 1.1 and 1.4 times, with the smallest increase (1.1x) coming from the largest design. As expected, this indicates that just adding memory does not help in solving a time-consuming (performance intensive) elaboration task. What can help in this case, however, is multithreading inside the elaboration engine which is disabled in zamiaCAD at the moment of this writing, for it needs to be properly finished. Still, this means that even those 6 minutes required for the elaboration of a SoC with 3584 LEON3 CPU cores, already short as they are, can be considerably reduced even in case of laptops, which nowadays often contain 8 processing cores and are expected to contain even more of them in future.

Another important aspect is the way design units are managed inside the tool. To remain responsive and usable, the VHDL editor has to be provided with the exact list of VHDL files used in the given design. Otherwise the editor halts trying to parse everything in the imported library, which takes it about 2 hours in case of the full GRLIB (127K vs. 27K concurrent and sequential statements). To avoid this pitfall, zamiaCAD features a very fast *multithreaded indexer* which allows large legacy libraries to be scanned in almost no time (in order to learn in what file every possible design unit can be found). zamiaCAD's indexer thus saves the parser from processing dead uninitialized code by only feeding it with the files actually used in the design. Also providing the VHDL editor with the exact list of files in case of a large project is highly prone to errors as is the case with any manual work.

As a result, zamiaCAD's scaling efficiency becomes close to the AST-level-only tools, while the ability to fully elaborate the given large designs is preserved.

All experiments were performed on an Intel® Pentium® Dual CPU 2.2GHz machine with 3.8GB of RAM running under Linux OS distribution CentOS 6.0. This roughly corresponds to a typical hardware engineer's workstation.

4.6 Debug

The automated debugger described in this section serves as an example of how different hardware models (IG and interpreter models) can be combined together to gain powerful synergic effect.

Most of verification approaches only concentrate on detecting the presence of a bug. Designers are thus provided with large counter-examples with lots of information which still miss the right information to unambiguously locate the bug. The subsequent manual debug is very time consuming and requires automation.

For automated bug localization, simulation-based [104], [105], [106], [107], [108], [109], [67][co-auth], [110] and formal approaches [82], [111], [112], [8] are known. The formal ones give a high grade of confidence in the results, but are susceptible to design complexity. In this section we rely on a simulation-based approach extended with static analysis.

Most of automated error localization solutions target software development domain [104], [105], [106], [107], [108], [109]. The error localization approaches for hardware designs, e.g. [110], [8] and its extension [113], consider unrolling sequential circuits at the gate-level. Thus they support and provide empirical results for small designs only (medium and small size iscas85 and iscas89 benchmarks). These approaches also assume existence of a reference golden model never available in hardware design (versus testing). Similar to the approach presented in this section, in 3.2.4 bugs at RTL are localized using a statistical approach combined with dynamic slicing [114]. However, the method proposed here is HDL centric, whereas in 3.2.4 localization is performed on HLDD model created from VHDL, therefore losing direct correspondence to the HDL code. Furthermore, the localization approach of current section takes advantage of different code coverage items, which significantly refine the localization, in particular for bugs in conditions. To the best of our knowledge this is the first approach that applies code coverage metrics for refining the bug locations.

The main contribution of this section is thus a new approach for automated localization of design errors (bugs) in processor designs at RTL. It is based on statistical analysis of HDL code items (statements, branches and conditions) of the design from a corresponding dynamic slice.

The core advantages of the approach are:

- Support for very large industrial designs due to the scalability of zamiaCAD elaboration.
- The approach is fault-model free. There is no need to explicitly enumerate the bug types.
- Accurate localization due to slicing and use of different code coverage metrics.
- Can be executed on the functional test. No need for separate diagnostic test generation.
- Supports localization of multiple bugs.

We also provide a case study on a real industrial processor ROBSY [115], [116] supplied with a list of documented bug cases and the original functional test. The approach is implemented and evaluated as a part of the open source RTL design framework zamiaCAD.

4.6.1 Bug localization with zamiaCAD

The proposed approach [117][co-auth] for design error localization assumes that design verification has been performed and an erroneous behavior has been detected (e.g. at observable outputs, by assertion violations, etc.). The approach is based on two main iterative phases: dynamic slicing and statistical suspiciousness ranking of the HDL statements in the design. The dynamic slicing reduces the debugging analysis to all the statements that actually affect the design's faulty

behavior for a given stimuli. Then, the suspiciousness ranking assigns a suspiciousness score to each statement present in the dynamic slice. Intuitively, if a statement occurs very frequently in executions revealing the error, it is very likely to contain a bug. To reveal artifacts more accurately, the suspiciousness ranking is performed also hierarchically for the branches and conditions that the ranked statements may have.

All individual steps of the proposed approach are easily doable on zamiaCAD scalable internal models. Dynamic slicing employs both the IG model and its executable layer (the interpreter model) right out of the box — that is without any changes made to them. Statistical suspiciousness ranking is performed on interpreter model and only requires some minor changes to specify which code items are to be ranked during design simulation. The scalable nature of the models employed allows the method to be applied to real life industrial designs, while the scripting interface of zamiaCAD allows the whole approach to be fully automated.

Consider the following design example in Figure 4.9. It presents a VHDL implementation of a signal chopper design *chopper* that is a modified motivational example from [26]. The *chopper* design has 3 processes calculating 4 outputs representing different chops for the input signal SRC based on the design configuration by inputs INV and DUP. It is assumed that the design has 5 individual tests T1-T5 of varied length each keeping the values of INV and DUV constant while flipping the value of the SRC input and having appropriate behavior of the clock and reset signals (CLK, CLKN, RST). The design has a bug introduced on line 28 where instead of correct assignment `F0 <= FF;` the design has a buggy assignment `F0 <= not FF;`. Tests T1, T3 and T4 are able to detect the bug and are referred to as *failing tests*, while tests T2 and T5 pass despite the presence of the bug and are referred to as *passing tests*. The faulty behavior of the design caused by the failing tests is observed at output TAR_f (assigned at line 46).

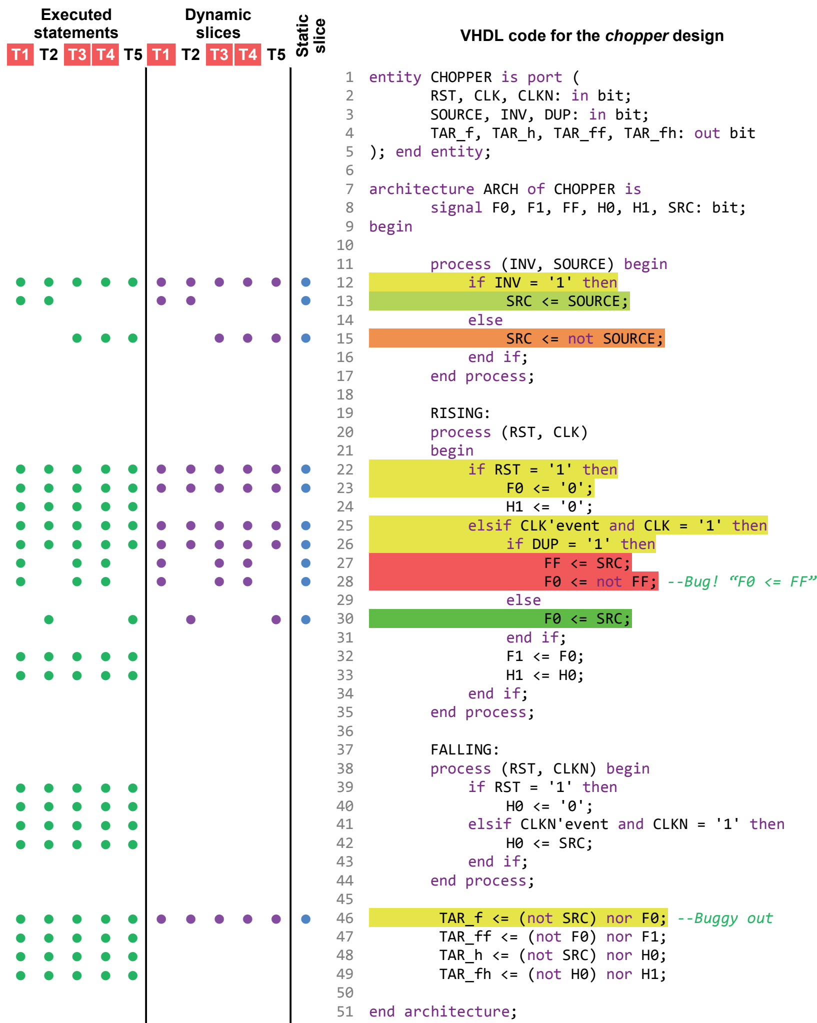


Figure 4.9 Simplified bug localization in a toy design

4.6.1.1 Static slicing

The basis of dynamic slicing is static slicing [85]. The presence of concurrent constructs in HDLs versus sequential software domain languages makes static slice computation considerably more complicated [26]. Various approaches dedicated for this task rely on computed models such as control flow graphs and others. zamiaCAD exploits for this purpose its IG model. Given the IG model it is possible to perform a signal references search through its assignments, both backward to find the dependencies and forward to find other signals and variables influenced by the signal. The resulting *reference graph* has signals and variables in its nodes and

the dependencies are expressed by directed edges. It may contain cyclic dependencies and may be very large, especially if the search was initiated from primary inputs/outputs of the design. It is possible to limit such search by constraining the depth of the graph. An example dependency graph computed for the *chopper* design's output *TAR_f* is shown in Figure 4.10.

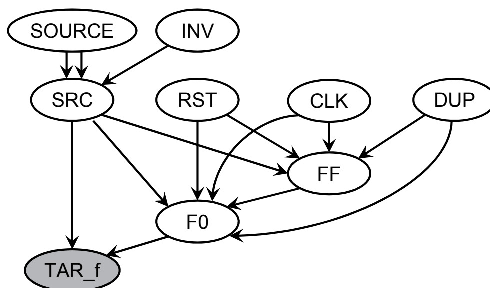


Figure 4.10 Through-signal-assignment backward reference graph on signal TAR_f in the *chopper* design

Given the through-signal-assignment reference graph, the HDL statements representing the signals and variables in its nodes are collected into a set. The resulting set represents a *static slice* on the given signal. However, our approach for static slice computation does not consider the order of HDL statements and can therefore be slightly too optimistic, i.e. it can potentially include into the static slice some statements that do not represent dependencies influencing the signal of interest. This can only be observed for certain combinations of variable (versus signal) assignments which are a rare case in practical HDL descriptions.

The column “Static slice” in Figure 4.9 marks VHDL statements of a static slice on the TAR_f output by dots (blue in color prints). Static slicing allows having a design filter that eliminates from the analysis space those design parts that do not influence the signal of interest. As a result in the *chopper* design example the entire process FALLING and a large part of other statements were excluded from the future analysis.

4.6.1.2Suspiciousness ranking based on statement and branch coverage metrics

The proposed statistical suspiciousness ranking procedure is based on design simulation by a *diagnostic test*. A diagnostic test is required to contain a set of independent sub-tests (e.g. separated by design reset) where both failing and passing tests are present. The quality of statistical ranking depends on the quality of diagnostic test [118][co-auth]. *Functional tests* for processors are particularly suitable as diagnostic tests because they are divided into separate sub-tests for processor instructions, so that each such sub-test can be executed independently.

The column "Executed statements" in Figure 4.9 marks the VHDL statements executed during design simulation with each of the 5 tests by dots (green in color print). A fraction of executed statements can be excluded from further analysis by applying a static slice filter on the output signal where the faulty behavior was observed. This approach allows obtaining a *dynamic slice* of the design on this signal. The column "Dynamic slices" in Figure 4.9 marks the VHDL statements taking part in the dynamic slices of the tests by dots (purple in color print). Thus the analysis space for the current example was reduced 2.2 times (42 versus 92 statement executions by the sub-tests).

The statistical *suspiciousness score* for ranking of the HDL code item i is calculated by the following Formula (1):

$$S(i) = \frac{\frac{Failed(i)}{TotalFailed}}{\frac{Passed(i)}{TotalPassed} + \frac{Failed(i)}{TotalFailed}} \qquad (1)$$

Where *S(i)* is the suspiciousness score value of the code item i, *Passed(i)* and *Failed(i)* are the counts of passing and failing tests that covered the item i in a dynamic slice, while $TotalPassed$ and $TotalFailed$ are the total numbers of the passing and failing tests in the complete diagnostic test, respectively.

Further zamiaCAD environment visualizes the score *S(i)* by 4-bit colors that can be interpreted as follows:

- **1** - is colored by extreme red and is interpreted as highly suspicious HDL code item to contain or to lead to the bug;
- **0** - is colored by extreme green and is interpreted as an HDL code item above suspicion;
- $S_{threshold}$ - is colored by yellow and is interpreted as an item that was not emphasized by the analysis;

Here $0 < S_{threshold} < 1$ is the *suspiciousness threshold* specified by the designer and is by default equal to 0.5. The items having score S values between 0, $S_{threshold}$ and 1 are colored by greenish and reddish yellow tones respectively and represent different levels of suspiciousness. Items without scores (and therefore not colored) were either eliminated from the analysis by the static slice filter or not covered by the diagnostic test.

An example of applying the proposed suspiciousness ranking to the *chopper* design is demonstrated in Figure 4.9. Here the assignment statements on lines 27 and 28 were calculated as the most suspicious (score $S = 1$) and were colored red, the assignment on line 30 was calculated to be above suspicion (score $S = 0$) and colored green, while statements on lines 13 and 15 have scores S 0.4 and 0.6 correspondingly and therefore are colored greenish yellow and orange.

4.6.1.3 Hierarchical analysis based on condition coverage

As it will be demonstrated in the next section, the ROBSY processor case study has emphasized an important general category of design errors that are difficult to localize. These are bugs in complex condition expressions of conditional statements. E.g. Bug 1 in this case study is an erroneous comparison of one of the 35 conditions in a conditional assignment *when* of the ALU module. Localization of such bugs is assisted by suspiciousness ranking of condition items.

We propose to hierarchically rank condition items of the selected (e.g. by a rank threshold) suspicious branch items that belong to suspicious statements. Formula (1) is applied for this purpose considering for *i* branch and condition items instead of statements. In the proposed approach the *branch items* are separate evaluations into 'true' or 'false' of conditional statements branches and the *condition items* are separate evaluations into 'true' or 'false' of logical operators (i.e. *OR*, *AND*, *XOR*, *NOR*, etc.) and relational operators (i.e. >, =, ≤, etc.). A detailed example for hierarchical conditions ranking and its application for bug localization is demonstrated in section 4.6.3.

4.6.2 Case study

As a case study, the proposed approach was evaluated [117][co-auth] by debugging an industrial processor developed as a part of the ROBSY (Reconfigurable On Board self test SYstem) project. This custom processor follows a new test approach [115], [116] to improve the fault coverage and reduce the test time of Printed Circuit Boards (PCBs) during the manufacturing process, and it is developed in cooperation with a major vendor of PCB testing equipment. The ROBSY processor is classified as a single instruction single data (SISD) processor with separated program and data buses (Harvard architecture). The processor has many of the properties of a reduced instruction set computer (RISC), and uses the Wishbone protocol (WB) for the I/O transactions. The current implementation of the processor core contains 17K lines of VHDL code. There are 481 direct signal assignment statements, 413 branches and 1573 conditions.

4.6.2.1 ROBSY processor: functional test

To verify the correct functionality of the Instruction Set Architecture (ISA), a functional test was developed during the implementation phase of ROBSY project. The functional test consists of a test program written in assembler, executed in a predefined order to test all the instructions supported by the processor. The test program is divided into sub-tests, where each sub-test is in charge of testing a specific instruction and setting register R1 to a specific value that acts as a sub-test label (error code). During the sub-test execution, it is evaluated if the values obtained in the primary registers, flags, etc. are as expected. In Figure 4.11 we have an example of a sub-test corresponding to the compare (CMP) instruction.

In case of unexpected value, the processor goes to the code section labeled with "fail", the execution is aborted and the error code of the failed sub-test is written to a dedicated register.

```
; check CMP with flags (register content unsigned)
MOV   R1, 01; -- error code 01---
MOV   R2, A3;
CMP   R2, 05;
JZ    fail;    if R2 equal 05 (jump zero)
JC    fail;    if R2 < 05     (jump carry)
CMP   R2, A3;
JNZ   fail;    if R2 not equal 05 (jump not zero)
JC    fail;    if R2 < 05
MOV   R3, A4;
CMP   R2, R3;
JNC   fail;    if R2 > R3 (jump not carry)
```

Figure 4.11 ROBSY processor test program sub-test for the CMP instruction

4.6.2.2 Set of documented design errors

The ROBSY design team has documented a set of their VHDL coding bugs that have the following nature:

- **Bug 1** A *wrong register* is used as one of the operands in a very long conditional expression (35 operators) inside a conditional signal assignment. Possibly, due to a copy-paste error.
- **Bug 2** An entire *conditional sub-expression* (3 operators) resides in the wrong branch of a conditional signal assignment, which contains 9 branches in total.
- **Bug 3** Both, a *missing branch* and a *missing driver* in a short conditional signal assignment.
- **Bug 4** A *wrong enumeration constant* is used in a comparison operation inside a conditional signal assignment.
- **Bug 5** A *wrong driver* is used in a conditional signal assignment. More specifically, register *R* is not updated with its newly computed value typically stored in *R_next* or *R_new* signal. Instead, the same register *R* is used as a driver, which indicates an obvious copy-paste error.
- **Bug 6** A *missing conditional sub-expression* (3 operators out of 6 required ones) in one of the 4 branches of a conditional signal assignment.
- **Bug 7** *One bit* of a register is always and unconditionally set to 0. The whole code line to blame is unnecessary and hence incorrect.

4.6.3 Details of automated bug localization

This section presents experimental results for the proposed design errors localization approach evaluation on the industrial processor ROBSY. For the purpose of the proposed approach the original functional test (i.e. an Assembler program) was split into 28 independent sub-tests, each targeting a separate instruction. Each of the 7 buggy versions of the processor was simulated with the

resulted diagnostic test. The ratio of failing vs passing sub-tests for the bugs in the case-study was the following: *Bug 1*: 4 vs 24; *Bug 2*: 2 vs 26; *Bug 3*: 2 vs 26; *Bug 4*: 1 vs 27; *Bug 5*: 2 vs 26; *Bug 6*: 1 vs 27; *Bug 7*: 1 vs 27.

#	Stm. score	Bran. score	Cond. score	Line	Source code lines
	alu.vhd				
6	0.51			88	
5	0.55			104	
2	0.67			108	
1	0.80			110	
5	0.55			116	
5	0.55			127	
				...	
3	0.64	0.64		260	svFlag_new(0) <= '1' when afClass=cfClass_1
			0.64^{1}_{T}	261	and ((svOp_mux(cnD_w)=REG_SOURCE_DEST_IN(cnD_w)--add case
				262	and svOp_mux(cnD_w)/=svRes(cnD_w)
				263	and ((aCmd=cvCmd_ADD_R_R and c_en_ADD_R_R)
				264	or (aCmd=cvCmd_ADD_R_IMM and c_en_ADD_R_IMM)))
			0.51^{1}_{T}	265	or (svOp_mux(cnD_w)/=REG_SOURCE_DEST_IN(cnD_w)--sub case
					-- Bug: correct compar. between REG_SOURCE_DEST_IN and svRes
				266	and svOp_mux(cnD_w)/=svRes(cnD_w)
			0.69^{1}_{T}	267	and ((aCmd=cvCmd_SUB_R_R and c_en_SUB_R_R)
			0.55^{2}_{T}		
			0.75^{2}_{T}	268	or (aCmd=cvCmd_SUB_R_IMM and c_en_SUB_R_IMM)
			0.57^{1}_{T}	269	or (aCmd=cvCmd_CMP_R_R and c_en_CMP_R_R)
			0.55^{2}_{T}		
			0.51^{1}_{T}	270	or (aCmd=cvCmd_CMP_R_IMM and c_en_CMP_R_IMM)))
				271	or (REG_SOURCE_DEST_IN(cnD_w)/=svRes(cnD_w)--shift cases
			0.55^{1}_{T}	272	and ((aCmd=cvCmd_SHL_R and c_en_SHL_R)
			0.55^{2}_{T}	273	or (aCmd=cvCmd_SHR_R and c_en_SHR_R))))
5	0.55	0.55		274	else '0' when afClass=cfClass_1 --overflow reset
				275	and ((aCmd=cvCmd_ADD_R_R and c_en_ADD_R_R)
				276	or (aCmd=cvCmd_ADD_R_IMM and c_en_ADD_R_IMM)
				277	or (aCmd=cvCmd_SUB_R_R and c_en_SUB_R_R)
				278	or (aCmd=cvCmd_SUB_R_IMM and c_en_SUB_R_IMM)
				279	or (aCmd=cvCmd_CMP_R_R and c_en_CMP_R_R)
				280	or (aCmd=cvCmd_CMP_R_IMM and c_en_CMP_R_IMM)
				281	or (aCmd=cvCmd_SHL_R and c_en_SHL_R)
				282	or (aCmd=cvCmd_SHR_R and c_en_SHR_R))
NA	0.50			283	else svFlag(0);
	data_interface_mod.vhd				
2	0.67			155	
2	0.67			158	
	gprs_mod.vhd				
4	0.60			97	
	state_machine.vhd				
4	0.60			100	
4	0.60			123	
4	0.60			168	

Figure 4.12 Details of automated localization of *Bug 1* in the ROBSY processor

Figure 4.12 demonstrates the proposed hierarchical localization of *Bug 1*. The grey areas denote that some detailed information was omitted from the figure. First the dynamic slices (intersection of executed statements with the static slice on an observable faulty output) were generated for all of the test cases and the statistical suspiciousness ranking was performed. This analysis resulted in 14 statement candidates (out of the initial total 481 assignment statements) whose suspiciousness score S was above the default suspiciousness threshold $S_{threshold} = 0.5$. The figure

shows in the second column *Stm. score* the scores for these 14 suspicious statements, and in the first column their rank # based on the score (6 ranks in total). Most of the statements with high scores were found in the ALU processor module (file *alu.vhd*). The figure demonstrates a part of the actual VHDL code for the conditional assignment of the overflow flag signal *svFlag_new(0)*. Bug 1 is located in the condition expression at line 266 (correct comparison had to be made between signals *svRes(cnD_w)* and *REG_SOURCE_DEST_IN* instead of *svOp_mux(cnD_w)*). This complex conditional assignment (lines 260-283) contains 3 individual assignments at lines 260, 274 and 283. The first two assignments have 3rd and 5th ranks while the last one has the "yellow" score $S = 0.5$ and is filtered out together with other statements with scores 0.5 and less. The automated localization iteratively advises the designer to consider as bug location candidates the statements with the highest ranks starting with the one at line 110, followed by statements at line 108 in *alu.vhd* and lines 155 and 158 in *data_interface_mod.vhd* (complemented with hierarchical analysis of the corresponding branches and conditions). Further it will advise the designer the statement at line 260 in *alu.vhd* with the next rank 3 and score value 0.64. Then it will proceed with score computation of its branch on the same line ($S_{b260} = 0.64$ in column *Bran.score*). The suspiciousness scores of separate condition evaluations to 'true' and 'false' related to this branch artifact are also calculated. The ones that have score $S > 0.5$ are specified in column *Cond.score*. One of the highest scores here has the logical *and* at line 267. One of its operands is actually the incorrect signal comparison documented as *Bug 1*.

Table 4.5 Statistics of the proposed bug localization approach for the ROBSY processor bugs

The proposed automated localization						**Manual debug**
Bug name	Statements cand. / %	Conditions cand.	Localized stm. rank	Localization result	Time (min)	Time
Bug 1	14 / 2.9%	16 of 35	3	Automatic	2	4 hours
Bug 2	7 / 1.4%	13 of 43	1	Automatic	2	2 hours
Bug 3	20 / 4.0%	2 of 7	3	Automatic	2	4 hours
Bug 4	6 / 1.2%	N/A	(1) + ext.slc.	Semi-automatic	2 (+5)	4 hours
Bug 5	11 / 2.3%	4 of 11	1	Automatic	2	2 hours
Bug 6	8 / 1.7%	N/A	(1) + ext.slc.	Semi-automatic	2 (+10)	5 hours
Bug 7	21 / 4.3%	N/A	(1) + ext.slc.	Semi-automatic	2 (+1)	1 hours

Table 4.5 demonstrates the statistics of applying the proposed bug localization approach to all of the 7 bugs. The second column in the table shows how many statements were proposed as bug location candidates and also demonstrates these numbers in percentage of the total number of statements which was 481. As supplementary debug data, the hierarchical analysis has selected suspicious conditions out of conditions of the candidate statements from the second column. Their percentage varies for different bugs from 20.3% to 39.8% (e.g. 72 conditions with suspiciousness score above the threshold out of all 354 conditions of the statements selected as *Bug 5* location candidates). The third column demonstrates the number of selected conditions candidates of the total number of conditions of the located statement. The fourth column shows the rank of the statement actually containing the bug. The diagnostic test was sufficient to automatically localize 4 of the 7 bugs (i.e. *Bugs 1*, *2*, *3* and *5*). Localization of the remaining three bugs was semi-automatic and required manual interaction. In these cases, the bug locations were traced in zamiaCAD by the through-signal-assignment reference search (also used for static slice computation) with limited depth initiated from the signals involved in the highly ranked assignment statements. *Bugs 4*, *6* and *7* were present within an extra static slice on the signal from the statement with the highest rank (reference search depth was 1 assignment for the three bugs and has introduced 21, 13 and 10 additional bug candidates respectively). Automation of this process is conceptually possible and is planned as a future work.

The last two columns in Table 4.5 compare the time required for bugs localization by the proposed automated localization approach and conventional manual debug process used by designers before. The time values for the manual process are reported by designers based on their real experience with locating these bugs using a fast commercial simulator. The reported time for the automated debug approach (2 min) is mainly spent for separate simulation of the 28 sub-tests and is still pessimistic due to the relatively slow current implementation of the VHDL simulator in the zamiaCAD framework. The framework also supports interaction with external simulators and importing waveform VCD dump files, whose application would allow reducing the runtimes of the automatic approach by a factor of ten. The additional time in brackets reported for *Bugs 4*, *6* and *7* reflects the manual interaction part in the semi-automatic localization process.

To the best of our knowledge none of the state-of-the-art automated hardware design error localization approaches are capable to handle industrial size RTL designs such as ROBSY. Therefore direct comparison to other than manual approaches was not possible for this empirical study.

Figure 4.13 presents a screenshot of the automated debugger implemented in zamiaCAD [92]. All bug candidates are listed in the *Markers* tab and also highlighted in the VHDL code with different colors according to their suspiciousness score.

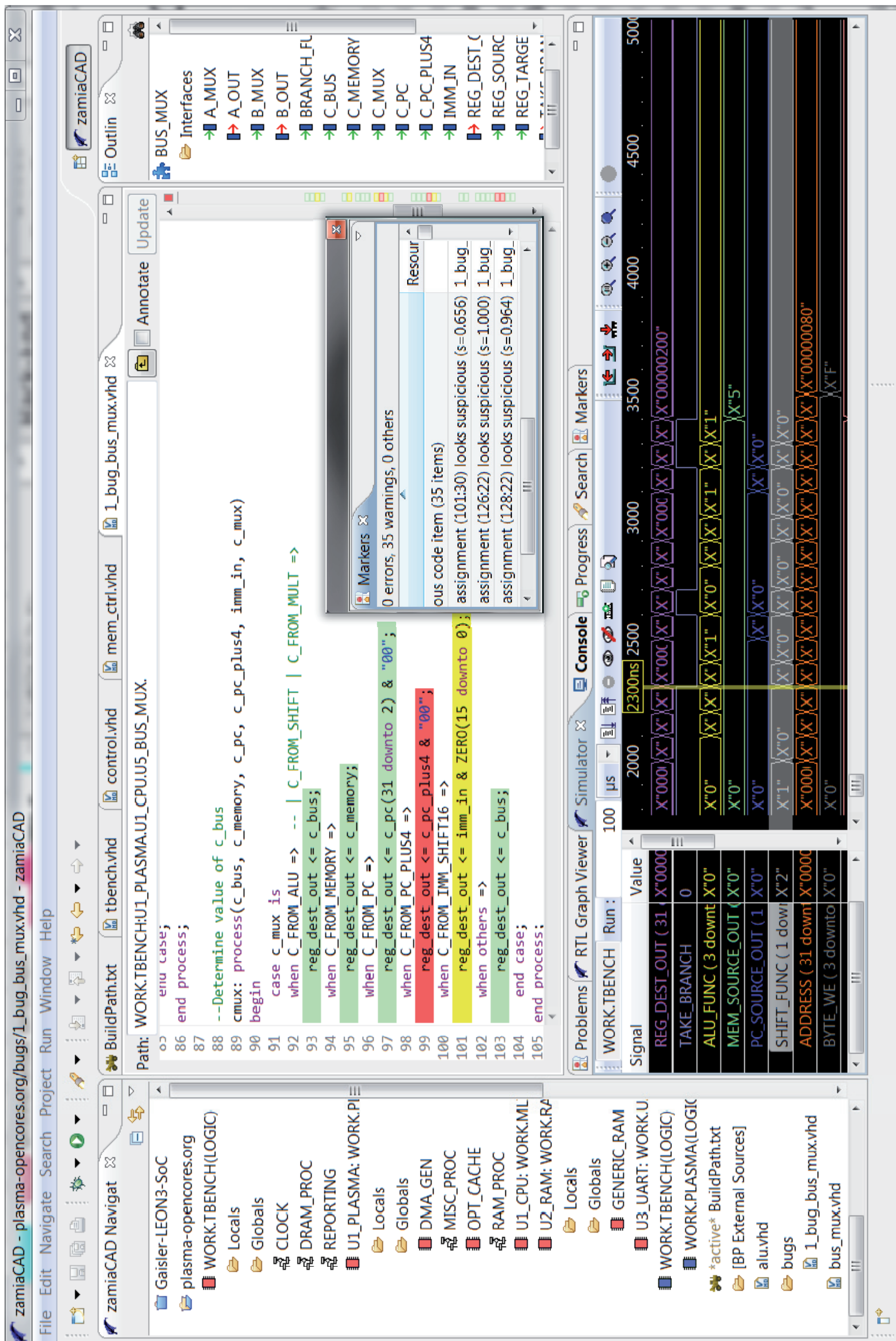


Figure 4.13 Automated debugger in zamiaCAD

4.7 Summary

This chapter has introduced a scalable IG model, designed to represent large industrial circuits, and its application to the task of debug.

First, details of comprehensive HDL elaboration into IG model were described with a strong accent on persistence and scalability. It was shown that IG model's implementation inside an open source framework zamiaCAD is capable of handling very large circuits and is therefore suitable for building experimental environments for scientific research on designs of tomorrow's scale.

Second, an approach for automated localization of design errors was presented. The novelty of the approach is that it successfully and in a scalable manner applies static slicing for analysis space reduction to realistic-size industrial designs and considers different coverage metrics for refining the bug localization. The approach is fault-model free and supports localization of multiple bugs. The original functional test can be used as a diagnostic test and is sufficient for the approach.

Chapter 5
CONCLUSIONS

This thesis has presented two hardware design representation models for verification and debug.

HLDD model is efficient for fast assertion checking, accurate code coverage measurement, mutation testing and automated debug, and finds its application in academic research. ApricotCAD combines all HLDD tools into a single flow.

IG model is instead very suitable for industrial use and is capable of representing very large hardware designs in an HDL-agnostic way. It serves as a scalable basis for simulation, static analysis, automated debug and other design tasks, and has attracted interest of several industrial companies in the field of hardware design and verification. zamiaCAD framework implements IG model and combines all IG-based tools into a single synergic flow that can also be used in research. zamiaCAD project has received IBM Faculty Award 2011/2012.

5.1 Contributions

The following objectives were reached in this thesis:

- To *improve verification quality*, we have proposed using HLDD model of HDL descriptions to accurately measure code coverage for the given test set. We have shown that HLDD-based coverage metrics are closer to path coverage and are more stringent than classical HDL-based metrics. E.g. for 100% HDL statement and branch coverages, the HLDD-based node coverage was only 86% and edge coverage only 78%. Therefore HLDD-based metrics provide a more stringent assessment of a test set with respect to its ability to

catch bugs and suggest exact places for potential test improvement. Consequently they give a better measure of confidence in the quality of hardware designs. The time overhead of HLDD-based coverage measurement is also much smaller: 1-4% versus 28-78% for a commercial state-of-the-art simulator.

- We have also proposed a way of modeling widely used PSL assertions with THLDDs. This allows fast design property checking during simulation (around 2 times faster than a state-of-the-art commercial simulation tool) and gives possibility to measure assertion coverage similarly to how design code coverage is measured.

- We have also arranged all our verification techniques into a *single flow implemented as ApricotCAD tool* to facilitate unified design verification and scientific research. Given an HDL description to be verified, the tool allows the designer to automatically generate behavioral HLDD model for the given description, simulate it with a test set and measure the code coverage of this test while being confident the coverage metrics is stringent enough to catch operation-critical design bugs. In case of insufficient test set, structural HLDDs can be synthesized by the tool, which allow the RTL ATPG DECIDER to automatically generate a better test set. In addition, verification assertions can always be co-simulated with this test set to make sure important high-level properties of the design meet its specification. This is facilitated by automatic generation of THLDD property descriptions from PSL verification properties and by seamless and fast co-simulation of THLDDs and HLDDs. If some tests lead to a property failure, the built-in automated debugger allows tracing the source of the error. It provides a list of bug candidates to the designer for deeper consideration and intuitively highlights the appropriate code regions.

- We have proposed a methodology for comprehensive *HDL code elaboration into a scalable IG model* and presented an *open source platform zamiaCAD* that implements the IG model and targets design and verification. Performed scalability experiments reveal that both the model and the platform can support very large SoCs such as one with 3584 LEON3 CPU cores inside, while many state-of-the-art commercial tools fail to do so. IG model is also shown useful as an intermediate design representation.

- We have proposed a *scalable and efficient method for automated bug localization* in large concurrent designs of today's scale. The method filters a design with dynamic slicing and uses statistical suspiciousness ranking of fine-grained HDL code items to hint the designer for the most probable bug locations. Fine-grained items allow the method to effectively locate bugs in individual conditions of complex conditional statements. The method

considerably reduces the search space for possible bug locations and in our experiments only leaves for designer's consideration from 1.2% to 4.3% of all the design statements. The method also proved efficient in locating real documented design bugs. The automated method is implemented in zamiaCAD and was tested on different processor designs.

- Fast and seamless integration of the scalable debug approach into *zamiaCAD framework* also indicates the successful achievement of our last goal. We have contributed to the development of a highly scalable free platform for research and industrial use in the field of hardware design and verification. It supports huge industrial designs and provides for easy automation through scripting and for convenient integration with other tools.

5.2 Future work

As mentioned above, assertion coverage measurement on THLDDs should be investigated to further raise the quality of verification. Also, the VHDL subset supported by HLDD generation method is not complete and should be extended to support loops, generate-statements, aggregates etc.

In zamiaCAD, its ability to handle large designs should further be exploited. Different analysis and automation tasks could be implemented to further improve design and verification capabilities of designers without compromising the scalability property. Efficient and scalable tracing of source-less/sink-less signals would allow elimination of bugs caused by careless component reuse which is an established practice today. Tracing individual bits and array elements requires a synthesis step similar to HLDD generation and is useful for better design understanding and could also be useful for automated verification tasks. Automated synthesis from behavioral RTL VHDL descriptions to structural would allow visualization of the latter and improve understanding of design's underlying structure. The analysis process (opposite to synthesis) implies extraction of higher-level concepts. E.g. generation of SystemC algorithmic descriptions from RTL VHDL code could solve the bottleneck of slow legacy components simulation. Semantic information contained in IG would be a good starting point for such an analysis, if at all possible. Also, advancing from incremental elaboration to incremental simulation would further improve the speed of manual debug and apparently allow automated HDL-based bug correction.

Integration of ApricotCAD functionality into zamiaCAD is also a promising step. It would allow the inherently HDL-based zamiaCAD to stringently measure code coverage and assertion coverage at RTL. At the same time, due to its extensive support of the VHDL standard, zamiaCAD could allow more scalable HLDD generation than is currently achieved with the ApricotCAD's own converter. Also, zamiaCAD could produce HLDDs for a large design, which could

then be used to analyze e.g. hardware aging with the help of structurally synthesized BDDs. This information about aging could then be passed back to IG model and back-annotated to the HDL code so that the designer would know which RTL code is expected to cause premature aging of the given hardware. Moreover, since fine-grained handling of array and record elements by IG implies building HLDD-like structures in zamiaCAD, the existing expertise of HLDD construction could be reused for this purpose and is already being reused, in fact.

To conclude, this thesis revealed not only the advantages of applying HLDD and IG models to verification and debug, but also their potential for future research. The possibilities are abundant, and both the industry and academia can benefit from advancing and utilizing these two models and their corresponding implementations — ApricotCAD and zamiaCAD.

References

[1] **Tasiran, S. and Keutzer, K.** "Coverage metrics for functional validation of hardware designs". – *Design & Test of Computers*, Vol. 18, 2001, pp. 36-45.

[2] **International Technology Roadmap for Semiconductors.** ITRS 2010 update. [Online] 2010. www.itrs.net.

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

[4] **RTCA DO-178B / EUROCAE ED-12B.** "Software Considerations in Airborne Systems and Equipment Certification". – *RTCA Inc.*, Washington, 1992.

[5] **Lions, J. L.** "ARIANE 5 Flight 501 Failure". – *Report by the Inquiry Board*, Paris, France, 1996.

[6] **Intel Corp.** "FDIV Replacement Program". – *White paper*, 1994.

[7] **Leveson, Nancy.** Medical Devices: The Therac-25. [Online] 1995. http://sunnyday.mit.edu/papers/therac.pdf.

[8] **Wotawa, Franz.** "Debugging VHDL Designs: Introducing Multiple Models and First Empirical Results". – *Applied Intelligence*, Vol. 21 (2), 2004, pp. 159-172.

[9] **Peischl, B., Soomro, S. and Wotawa, F.** "Lightweight Fault Localization with Abstract Dependences". – *Proc. of International Conference on Engineering of Intelligent Systems (ICEIS '06)*, 2006, pp. 300-306.

[10] **Jasper Design Automation, Inc.** [Online] http://www.jasper-da.com/.

[11] **Mentor Graphics, Inc.** ModelSim. [Online] 2012. http://www.mentor.com.

[12] **Gajski, D. D.** *Principles of Digital Design.* s.l. : Prentice Hall, 1997.

[13] **Calypto Design Systems, Inc.** Catapult: Product Family Overview. [Online] http://calypto.com/en/products/catapult/overview/.

[14] **Bluespec, Inc.** BlueSpec Compiler. *High-Level Synthesis Tools.* [Online] http://www.bluespec.com/high-level-synthesis-tools.html.

[15] **Synopsys, Inc.** Synopsis Synphony C Compiler. [Online] http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx.

[16] **Rose, A., Swan, S., Pierce, J. and et al.** "Transaction Level Modeling in SystemC". – *Open SystemC Initiative*, 2005.

[17] **IEEE Commission.** "1076-2002 IEEE Standard VHDL Language Reference Manual". – *IEEE Std 1076-2002*, 2002.

[18] **IEEE-Commission.** "1647-2011 IEEE Standard for the Functional Verification Language e". – *IEEE Std 1647-2011*, 2011.

[19] **IEEE-Commission.** "IEEE standard for Property Specification Language (PSL)". – *IEEE Std 1850-2005*, 2005.

[20] **Jenihhin, M.** "Simulation-Based Hardware Verification with High-Level Decision Diagrams". – *PhD Thesis*, Tallinn, 2008.

[21] **Synopsys, Inc.** SpringSoft - Verdi Automated Debug System. [Online] http://www.springsoft.com/products/debug-automation/verdi.

[22] Open Source VHDL Verification Methodology. [Online] http://osvvm.org.

[23] **Tadashi, K.** Veditor. [Online] 2012. http://veditor.sf.net.

[24] **Sigasi nv.** Sigasi. [Online] 2012. http://www.sigasi.com.

[25] **Simplifide.** SimplifIDE. [Online] 2012. http://simplifide.com.

[26] **Clarke, E. M., Fujita, M., Rajan, S. P., Reps, T., Shankar, S. and Teitelbaum, T.** *Program slicing of hardware description languages.* 1999.

[27] **Mentor Graphics, Inc.** HDL Designer. [Online] 2012. http://www.mentor.com.

[28] **Synopsys, Inc.** Design Compiler. [Online] 2012. http://www.synopsys.com.

[29] **Verific Design Automation, Inc.** Verific Design Automation. [Online] 2012. http://www.verific.com.

[30] **Gingold T.** GHDL. A VHDL compiler. [Online] 2012. http://ghdl.free.fr.

[31] **Rudell, R. L.** *Multiple-valued logic minimization for pla synthesis.* Berkeley : Technical Report UCB/ERL M86/65, EECS Department, University of California, 1986.

[32] **Stok, L., Kung, D. S., Brand, D., Drumm, A. D., Sullivan, A. J. and et al.** "Booledozer: Logic synthesis for asics". 40(4), s.l. : IBM Journal of Research and Development, 1996, pp. 407-430.

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

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

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

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

[37] **Ubar, R.** "Test Generation for Digital Circuits Using Alternative Graphs". – *Proc. of Tallinn Technical University*, Vol. 409, Tallinn, Estonia, 1976, pp. 75-81 (in Russian).

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

[39] **Chaiyakul, V. and Gajski, D. D.** "Assignment Decision Diagram for High-Level Synthesis". – *Technical Report*, 1992.

[40] **Stellenberg, D. S. and Karchmer, D.** *Using assignment decision diagrams with control nodes for sequential review during behavioral simulation. US6697773 B1* USA, February 24, 2004. Grant.

[41] **Baranov, S.** *Logic and System Design of Digital Systems.* s.l. : TUT Press, 2008.

[42] **Jenihhin, M., Baranov, S., Raik, J. and Tihhomirov, V.** "PSL assertion checkers synthesis with ASM based HLS tool ABELITE". – *13th Latin American Test Workshop (LATW)*, 2012, pp. 1-6.

[43] **Gajski, D., Dutt, N., Allen, C. and Steve, Y.** "High-level synthesis: introduction to chip and system design". – *Kluwer Academic Publishers*, 1992.

[44] **Cheng, K. and Krishnakumar, A.** "Automatic Generation of Functional Vectors Using The Extended Finite State Machine Model". – *ACM Trans. on Design Automation of Electronic Systems*, Vol. 1 (1), 1996, pp. 57-79.

[45] **Peterson, J. L.** "Petri Net Theory and the Modeling of Systems". – *Prentice Hall*, Englewood Cliffs, New Jersey, 1981.

[46] **Karlsson, D.** "Verification of Component-Based Embedded System Designs". – *PhD Thesis*, Linköping, Sweden, 2006.

[47] **Chepurov, A., Di Guglielmo, G., Fummi, F., Pravadelli, G., Raik, J., Ubar, R. and Viilukas, T.** "Automatic Generation of EFSMs and HLDDs for Functional ATPG". – *11th Biennial Baltic Electronics Conference (BEC'08)*, 2008, pp. 143-146.

[48] **Ubar, R.** "Test Generation for Digital Systems on the Vector Alternative Graph Model". – *Proc. of the 13th Symposium on Fault Tolerant Computing*, Milan, Italy, 1983, pp. 374-377.

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

[50] **Ubar, R., Morawiec, A. and Raik, J.** "Cycle-based Simulation with Decision Diagrams". – *Proceedings of the DATE Conference*, 1999, pp. 454-458.

[51] **Raik, J.** "Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams". – *PhD thesis*, 2001.

[52] **Chepurov, A.** "Interface between VHDL and High-level Decision Diagram model descriptions". – *Master thesis*, Tallinn, 2008.

[53] **OpenCores.** UART16750. [Online] May 2013. http://opencores.org/project,uart16750.

[54] **Viilukas, T., Raik, J., Jenihhin, M., Ubar, R. and Krivenko, A.** "Constraint-based test pattern generation at the Register-Transfer Level". – *13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2010.

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

[56] **Jenihhin, M., Raik, J., Chepurov, A. and Ubar, R.** "Assertion Checking with PSL and High-Level Decision Diagrams". – *Proc. of the IEEE 8th Workshop on RTL and High Level Testing (WRTLT'07)*, Beijing, China, 2007, pp. 1-5.

[57] **Raik, J., Jenihhin, M., Chepurov, A., Reinsalu, U. and Ubar, R.** "APRICOT: a Framework for Teaching Digital Systems Verification". – *19th EAEEIE Annual Conference*, 2008, pp. 172-177.

[58] **Minakova, K., Reinsalu, U., Chepurov, A., Raik, J., Jenihhin, M., Ubar, R. and Ellervee, P.** "High-Level Decision Diagram Manipulations for Code Coverage Analysis". – *11th Baltic Electronics Conference*, 2008, pp. 207-210.

[59] **Raik, J., Ubar, R., Jenihhin, M. and Chepurov, A.** "PSL Assertion Checking with Temporally Extended High-Level Decision Diagrams". – *9th IEEE Latin American Test Workshop*, Puebla, Mexico, 2008, pp. 49 - 54.

[60] **Jenihhin, M., Raik, J., Chepurov, A. and Ubar, R.** "PSL Assertion Checking Using Temporally Extended High-Level Decision Diagrams". – *Journal of Electronic Testing: Theory and Applications (JETTA)*, Vol. 25 (6), 2009, pp. 289-300.

[61] **Jenihhin, M., Raik, J., Chepurov, A. and Ubar, R.** "Temporally Extended High-Level Decision Diagrams for PSL Assertions Simulation". – *Proc. of the 13th IEEE European Test Symposium*, Los Alamitos, USA, 2008, pp. 61 - 68.

[62] **Jenihhin, M., Raik, J., Chepurov, A., Reinsalu, U. and Ubar, R.** "Code Coverage Analysis for Concurrent Programming Languages Using High-Level Decision Diagrams". – *12th European Workshop on Dependable Computing (EWDC 2009)*, 2009.

[63] **Jenihhin, M., Raik, J., Chepurov, A., Reinsalu, U. and Ubar, R.** "High-Level Decision Diagrams based Coverage Metrics for Verification and Test". – *10th Latin American Test Workshop (LATW '09)*, 2009.

[64] **Jenihhin, M., Raik, J., Chepurov, A. and Ubar, R.** "Simulation-based Verification with APRICOT Framework using High-Level Decision Diagrams". – *East-West Design & Test Symposium*, 2009, pp. 13-16.

[65] **Jenihhin, M., Raik, J., Chepurov, A. and Ubar, R.** "Application of High-Level Decision Diagrams for Simulation-Based Verification tasks". – *Estonian Journal of Engineering*, Vol. 16 (1), 2010, pp. 56 - 77.

[66] **Hantson, H., Raik, J., Jenihhin, M., Chepurov, A., Ubar, R., Di Guglielmo, G. and Fummi, F.** "Mutation Analysis with High-Level Decision Diagrams". – *11th Latin American Test Workshop (LATW '10)*, 2010.

[67] **Raik, J., Repinski, U., Ubar, R., Jenihhin, M. and Chepurov, A.** "High-Level Design Error Diagnosis Using Backtrace on Decision Diagrams". – *NORCHIP '10*, 2010.

[68] **Raik, J., Repinski, U., Jenihhin, M. and Chepurov, A.** "High-Level Decision Diagram Simulation for Diagnosis and Soft-Error Analysis". *Design and Test Technology for Dependable Systems-on-Chip.* Hershey, New York : IGI Global, 2011, pp. 294-309.

[69] **Raik, J., Reinsalu, U., Ubar, R., Jenihhin, M. and Ellervee, P.** "Fast Code Coverage analysis using High-Level Decision Diagrams". – *DDECS'08*, 2008, pp. 1-6.

[70] ITC99 Benchmark Home Page. [Online] http://www.cerc.utexas.edu/itc99-benchmarks/bench.html.

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

[72] **Penry, David A. and August, David I.** "Optimizations for a Simulator Construction System Supporting Reusable Components". – *Proceedings of the 40th Design Automation Conference (DAC)*, 2003.

[73] **Piziali, A.** "Functional Verification Coverage Measurement and Analysis". – *Springer*, 2008.

[74] ""Code Coverage Analysis" and "Minimum Acceptable Code Coverage"". [Online] http://www.bullseye.com.

[75] **Accellera.** "Property Specification Language Reference Manual". – , Vol. 1.1, 2004.

[76] PROSYD: Property-Based System Design. *FP6 funded STREP.* [Online] 2004. http://www.prosyd.org/.

[77] **Lam, W. K.** "Hardware Design Verification: Simulation and Formal Method-Based Approaches". – *Pearson Education Inc.*, Upper Saddle River, NJ, 2005.

[78] **Ali, M. F., Safarpour, S., Veneris, A., Abadir, M. S. and Drechsler, R.** "Post-verification debugging of hierarchical designs". – *In Proceedings of ICCAD*, 2005, pp. 871-876.

[79] **Wahba, A. and Borrione, D.** "Design error diagnosis in sequential circuits". – *Lecture Notes In Computer Science*, Vol. 987, Springer, 1995, pp. 171-188.

[80] **Madre, J. C., Coudert, O. and Billon, J. P.** "Automating the Diagnosis and the Rectification of Design Errors with PRIAM". – *Proc. of ICCAD*, 1989, pp. 30-33.

[81] **Abadir, M. S., Ferguson, J. and Kirkland, T. E.** "Logic design verification via test generation". – *IEEE Transactions on Computer-Aided Design*, Vol. 7 (1), 1988.

[82] **Smith, A., Veneris, A. and Viglas, A.** "Design Diagnosis Using Boolean Satisfiability". – *Proc. Asia and South Pacific Design Automation Conference (ASPDAC)*, 2004, pp. 218-223.

[83] **Fey, G., Staber, S., Bloem, R. and Drechsler, R.** "Automatic Fault Localization for Property Checking". – *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 27 (6), 2008, pp. 1138-1149.

[84] **Chang, K. and et al.** "Automatic Error Diagnosis and Correction for RTL Designs". – *Proc. High-Level Design and Validation Workshop (HLDVT)*, Irvine, CA, 2007.

[85] **Weiser, M.** "Program slicing". – *In Proceedings of the 5th International Conference on Software Engineering*, Washington, DC, 1981, pp. 439-449.

[86] **Abramovici, M., Menon, P. R. and Miller, D. T.** "Critical path tracing - an alternative to fault simulation". – *In Proceedings of the 20th Design Automation Conference*, Miami Beach, FL, 1983, pp. 214-220.

[87] **Offutt, A. J., Rothermel, G. and Zapf, C.** "An experimental evaluation of selective mutation". – *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, MD, 1993, pp. 100-107.

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

[89] **ESD Group.** *HIFSuite.* [Online] www.edalab.it/HIFSuite.

[90] **Tshepurov, A., Jenihhin, M. and Raik, J.** "Simulator for ZamiaCAD Integrated Hardware Design Environment". – *In University Booth section of Design, Automation and Test in Europe (DATE'10)*, Dresden, Germany, 2010, pp. 1-2.

[91] **Tšepurov, A., Bartsch, G., Dorsch, R., Jenihhin, M., Raik, J. and Tihhomirov, V.** "A Scalable Model Based RTL Framework zamiaCAD for Static Analysis". – *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC '12)*, Santa Cruz, USA, 2012, pp. 181-186.

[92] zamiaCAD Framework website. [Online] http://zamiacad.sf.net/.

[93] **Aeroflex Gaisler AB.** LEON3 SPARC V8 Processor IP core, GRLIB IP library, v.1.1.0-b4108. [Online] 2012. http://www.gaisler.com.

[94] **Breuer, M. and MacDougall, M.** "Digital system design automation: languages, simulation & data base". – *Digital system design series. Computer Science Press*, 1975.

[95] **Silberschatz, A., Korth, H. F. and Sudarshan, S.** "Database System Concepts, 3rd Edition". – *McGraw-Hill Book Company*, 1997.

[96] **Sun Microsystems, Inc.** "Java Object Serialization Specification". – *Revision 1.4.4*, 2001.

[97] **Fagin, R., Nievergelt, J., Pippenger, N. and Strong, H. R.** "Extendible hashing - a fast access method for dynamic files". – *ACM Trans. Database Syst.*, Vol. 4, 1979, pp. 315-344.

[98] **Stonebraker, M. and Rowe, L. A.** "The design of postgres". – *SIGMOD Rec.*, Vol. 15(2), 1986, pp. 340-355.

[99] Java Compiler Compiler (JavaCC) - The Java Parser Generator. [Online] http://javacc.java.net/.

[100] **IEEE-Commission.** "1364-2005 IEEE standard for verilog hardware description language". – *IEEE Std 1364-2005*, 2006.

[101] **Scarpino, F.** "VHDL and AHDL Digital System Implementation". – *Prentice Hall PTR*, 1997.

[102] **OpenCores.org.** Plasma - most MIPS opcodes. [Online] May 2013. http://opencores.org/project,plasma.

[103] **Gamma, E., Helm, R., Johnson, R. and Vlissides, J.** "Design Patterns: Elements of Reusable Object-Oriented Software". – *Addison-Wesley*, 1994.

[104] **Liblit, B., Naik, M., Zheng, A. X., Aiken, A. and Jordan, M. I.** "Scalable statistical bug isolation". – *ACM SIGPLAN Notices*, Vol. 40(6), 2005, pp. 15-26.

[105] **Liu, G., Fei, L., Yan, X., Han, J. and Midkiff, S. P.** "Statistical debugging: A hypothesis testing-based approach". – *IEEE Trans. on Software Engineering*, Vol. 32(10), 2006, pp. 831-848.

[106] **Wong, W. E., Debroy, V. and Choi, B.** "A family of code coverage-based heuristics for effective fault localization". – *Journal of Systems and Software*, Vol. 83(2), 2010, pp. 188-208.

[107] **Wong, W. E. and Qi, Y.** "BP neural network-based effective fault localization". – *International Journal of Software Engineering and Knowledge Engineering*, Vol. 19(4), 2009, pp. 573-597.

[108] **Cleve, H. and Zeller, A.** "Locating causes of program failures". – *Proc. Int. Conf. on Software Engineering*, 2005, pp. 342-351.

[109] **Jones, J. A. and Harrold, M. J.** "Empirical evaluation of the Tarantula automatic fault-localization technique". – *Proc. Int. Conf. on Automated Software Engineering*, 2005, pp. 273-283.

[110] **Veneris, A. and Hajj, I. N.** "Design error diagnosis and correction via test vector simulation". – *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 18(12), 1999, pp. 1803-1816.

[111] **Staber, S., Jobstmann, B. and Bloem, R.** "Finding and fixing faults". – *Proc. CHARME*, 2005, pp. 35-49.

[112] **Konighofer, R. and Bloem, R.** "Automated error localization and correction for imperative programs". – *Proc. of Formal Methods in Computer Aided Design*, 2011, pp. 91-100.

[113] **Peischl, B. and Wotawa, F.** "Automated Source-Level Error Localization in Hardware Designs". – *Design&Test of Computers*, Vol. 23(1), 2006, pp. 8-19.

[114] **Korel, B. and Laski, J.** "Dynamic program slicing". – *Information Processing Letters*, Vol. 29(3), 1988, pp. 155-163.

[115] **Meza Escobar, J. H., Sachsse, J., Ostendorff, S. and Wuttke, H. D.** "Automatic generation of an FPGA based embedded test system for printed circuit board testing". – *Proc. LATW2012*, Quito, Ecuador, 2012, pp. 75-80.

[116] **Sachsse, J., Ostendorff, S., Wuttke, H. D. and Meza Escobar, J. H.** "Architecture of an adaptive Test System built on FPGA". – *Proc. ARCS 2011 - Architecture of Computing Systems*, Vol. LNCS 6566, Como, Italy, 2011, pp. 86-97.

[117] **Tšepurov, A., Tihhomirov, V., Jenihhin, M., Raik, J., Bartsch, G., Meza Escobar, J. H. and Wuttke, H.-D.** "Localization of Bugs in Processor Designs Using zamiaCAD Framework". – *13th International Workshop on Microprocessor Test and Verification (MTV 2012) Common Challenges and Solutions*, Austin, USA, 2012, pp. 1-6.

[118] **Tihhomirov, V., Tšepurov, A., Jenihhin, M., Raik, J. and Ubar, R.** "Assessment of Diagnostic Test for Automated Bug Localization". – *14th Latin American Test Workshop (LATW'13)*, Cordoba, Argentina, 2013, pp. 1-6.

Curriculum Vitae

Personal Data

Name	Anton Tšepurov
Date of birth	01.01.1985
Place of birth	Tartu, Estonia
Citizenship	Estonian

Contact Data

E-mail	anton.chepurov@gmail.com

Education

2008 – …	Ph.D. student in Computer and Systems Engineering, Tallinn University of Technology (TUT)
2006 – 2008	M.Sc. (cum laude) in Computer and Systems Engineering, TUT
2003 – 2006	B.Sc. in Computer and Systems Engineering, TUT
1991 – 2003	Secondary Education, Tartu Annelinna High school

Carrier

2007 – …	Researcher at Department of Computer Engineering, TUT
2005 – 2005	Software developer at Hansabank AS

Awards

2008 – 2011 "Tiger University" scholarships (4x) for ICT PhD students, Estonian Information Technology Foundation (EITSA)

Research topics

Test, verification and debug of digital designs; EDA tools development

Elulookirjeldus

Isikuandmed

Nimi	Anton Tšepurov
Sünniaeg	01.01.1985
Sünnikoht	Tartu, Eesti
Kodakondsus	Eesti

Kontaktandmed

E-post	anton.chepurov@gmail.com

Hariduskäik

2008 – …	doktoriõpe, info- ja kommunikatsioonitehnoloogia õppekava, Tallinna Tehnikaülikool (TTÜ)
2006 – 2008	tehnikateaduse magistri kraad (*cum laude*), arvuti- ja süsteemitehnika õppekava, TTÜ
2003 – 2006	tehnikateaduse bakalaureuse kraad, arvuti- ja süsteemitehnika õppekava, TTÜ
1991 – 2003	keskharidus, Tartu Annelinna Gümnaasium

Teenistuskäik

2007 – …	Noorem teadur, Arvutitehnika instituut, TTÜ
2005 – 2005	Tarkvaraarendaja, Hansapank AS

Teaduspreemiad

2008 – 2011 "Tiigriülikooli" stipendiumid (4 tk.) IKT doktorantidele, Eesti Infotehnoloogia Sihtasutus (EITSA)

Teadustegevus

Digitaaldisainide testimine, verifitseerimine ja silumine; CAD tarkvara arendamine

Appendix

Research paper I

A. Tšepurov, V. Tihhomirov, M. Jenihhin, J. Raik, G. Bartsch, J.-H. Meza Escobar, H.-D. Wuttke, "Localization of Bugs in Processor Designs Using zamiaCAD Framework", *Microprocessor Test and Verification Conference (MTV'12)*, December 10–12, 2012, Austin, Texas, USA.

The author's contributions are: participation in the development of the IG-based debug method; development of zamiaCAD infrastructure as an experimental setup; evaluation of different statistical bug ranking strategies; implementation of the automated debugger in zamiaCAD based on the proposed approach; drafting the case study section of the article.

Localization of Bugs in Processor Designs Using *zamiaCAD* Framework

Anton Tšepurov[1], Valentin Tihhomirov[1], Maksim Jenihhin[1], Jaan Raik[1],
Günter Bartsch[2], Jorge Hernan Meza Escobar[3], Heinz-Dietrich Wuttke[3]
[1]*Dept. of Computer Engineering, Tallinn UT, Tallinn, Estonia, Email: {anchep | valentin | maksim | jaan} @ati.ttu.ee*
[2]*zamiaCAD project founder, Stuttgart, Germany, Email: guenter@zamia.org*
[3]*ICS Group, Ilmenau UT, Ilmenau, Germany, Email: {jorge-hernan.meza-escobar | dieter.wuttke}@tu-ilmenau.de*

***Abstract*—This paper proposes an approach to automatic localization of design errors (bugs) in processor designs based on combining statistical analysis of dynamically covered VHDL code items and static slicing. The approach considers coverage of different VHDL code items including statements, branches and conditions during processor simulation which together contribute to accurate localization of bugs. The accuracy of analysis is further improved by applying a static slicing based filter calculated by means of reference graph generation using a through-signal-assignment search from the semantically resolved elaborated models of processor designs. The localization approach has been integrated to highly scalable zamiaCAD RTL design framework. The efficiency of the proposed approach is demonstrated by applying it to debugging of an industrial processor ROBSY designed for FPGA-based test systems. The experimental results evaluate the approach for a set of real documented bug cases and the original functional test.**

***Keywords*-automated debug; design error localization; processor design; VHDL; electronic design automation**

I. Introduction

Growing complexity of VLSI System-on-a-Chip (SoC) and processor designs leads to increasing design costs. The ITRS Roadmap [1] lists design error verification as one of the most expensive tasks in the design process. There exist several verification approaches that are mostly concentrated on identifying the occurrences of bugs. They provide feedback to the designers in form of counter-examples. On one hand, as the design complexity grows, the designer is faced with too much information contained in the large counter-examples. On the other hand, there is not enough information in order to unambiguously locate the bug. Therefore, the manual bug localization activity is very time consuming and there is a need for automated localization approaches.

For automated design error localization, simulation-based [2–9] and formal approaches [10–13] are known. It is widely acknowledged that simulation-based techniques scale well with the design sizes, but are not exhaustive, while formal techniques provide a high grade of confidence in the results, but are susceptible to the design complexity. In this paper, we rely on a simulation-based approach supported by static analysis.

Most of the works on automated error localization propose solutions for software development domain [2–7]. The error localization approaches proposed for hardware designs, e.g. [9], [13] and its extension [14], are mostly considering unrolling sequential circuits at the gate-level. Thus they support and provide empirical results for small designs only (medium and small size iscas85 and iscas89 benchmarks). These approaches also asume existence of a reference golden model. Similar to the approach presented in this paper, in [8] bugs at the Register-Transfer Level (RTL) are localized based on a statistical approach combined with dynamic slicing [15]. However, the method proposed in this paper is Hardware Description Language (HDL) centric, whereas [8] is performing localization on a High-Level Decision Diagram (HLDD) model created from VHDL, therefore losing direct correspondence to the HDL code. Furthermore, we have developed a localization approach that takes advantage of different code coverage items, which significantly refine the localization, in particular for bugs in the conditions. To the best of our knowledge this is the first approach that applies code coverage metrics for refining the bug locations.

Several approaches to statistical simulation based bug localization in software and hardware are refined with dynamic slicing [15] allowing to increase the resolution of the localization. The basis of dynamic slicing is static slicing [16]. The presence of concurrent constructs such as the ones found in HDLs makes static slice considerably more complicated [17] and exploits computed design models such as control flow graphs and others. In the proposed approach we rely on static slice computation based on reference graph generation using a through-signal-assignment search from the zamiaCAD semantically elaborated models of processor designs. This allows applying static slicing in a practical and scalable manner for realistic-size industrial designs.

The main contribution of this paper is a new approach for automated localization of design errors (bugs) in processor designs at RTL. It is based on statistical analysis of HDL code items (statements, branches and conditions) of the design from a corresponding dynamic slice.

The core advantages of the approach are:

- Support for very large industrial designs due to the scalability of zamiaCAD elaboration.
- The approach is fault-model free. There is no need to explicitly enumerate the bug types.
- Accurate localization due to slicing and use of different

code coverage metrics.

- Can be executed on the functional test. No need for separate diagnostic test generation.
- Supports localization of multiple bugs.

As a result the approach singles out a small set of the bug location candidates, i.e. HDL statements. Where applicable the analysis result is refined by artifact candidates in branches and single conditions.

The power of localization of the statistical approach proposed is dependent on the number of test sequences. Therefore, the functional tests developed for processors where test for each instruction can be singled out are highly suitable as diagnostic tests providing necessary statistics for localization. In this paper, we present a case study on a real industrial processor ROBSY [18, 19] supplied with a list of documented bug cases and the original functional test. The approach is implemented and evaluated as a part of an open-source RTL design framework zamiaCAD [20].

The rest of the paper is organized as follows. Section 2 gives a brief introduction to the zamiaCAD environment. Section 3 presents details of the proposed approach for automated design bug localization for processors. Section 4 introduces a case-study based on an industrial processor design ROBSY. Experimental results are demonstrated in Section 5 and Section 6 concludes the paper.

II. zamiaCAD Framework Overview

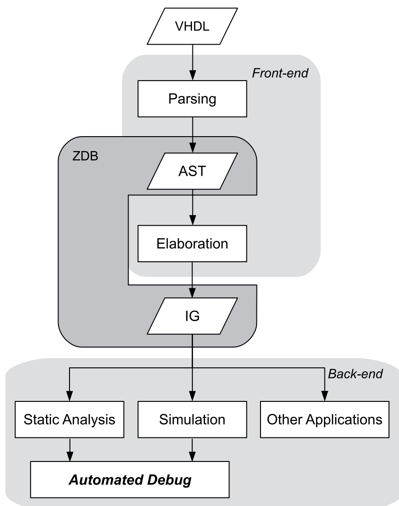


Figure 1. Simplified zamiaCAD Flow

The basis of the proposed approach is a scalable model based open-source framework zamiaCAD [20]. The framework addresses RTL design, verification and analysis. Currently the front-end fully supports VHDL. On the back-end side zamiaCAD supports design entry, navigation and analysis and has an Eclipse plug-in based user interface. It also contains a built-in VHDL simulator supporting verification and debug flows and it has stubs for synthesis and for visualization of the design structure. Figure 1 demonstrates a simplified flow of the framework.

An object database ZDB (zamiaCAD Data Base), which has been custom-designed and highly optimized for scalability and performance is used for zamiaCAD applications. The database is HDL independent and is able to accommodate extremely large designs. Full elaboration in zamiaCAD semantically resolves the Abstract Syntax Tree (AST) generated by the parser and results in a set of scalable Instantiation Graph (IG) data structures, stored in ZDB.

Definition. The *Instantiation Graph* is a data structure represented by a densely connected graph of semantically resolved objects representing elements of the hardware design.

IG is the base for the zamiaCAD applications. To handle designs which do not fit in memory, ZDB with an elaborated design in it is automatically and efficiently persisted to disk, thus saving processing time during later sessions. As demonstrated in [21] the framework is capable of handling very large industrial multi-core designs (tens of millions of VHDL code lines, e.g a SoC made of more than 3500 Leon3 [22] processor cores).

III. Bug Localization with zamiaCAD

The proposed approach for design error localization assumes that design verification has been performed and an erroneous behavior has been detected (e.g. at observable outputs, by assertion violations, etc.). The approach is based on two main iterative phases: dynamic slicing and statistical suspiciousness ranking of the HDL statements in the design. The dynamic slicing reduces the debugging analysis to all the statements that actually affect the design's faulty behavior for a given stimuli. Then, the suspiciousness ranking assigns a suspiciousness score to each statement present in the dynamic slice. Intuitively, if a statement occurs very frequently in executions revealing the error, it is very likely to contain a bug. To reveal artifacts more accurately, the suspiciousness ranking is performed also hierarchically for the branches and conditions that the ranked statements may have.

Consider the following design example in Figure 2. It presents a VHDL implementation of a signal chopper design *chopper* that is a modified motivational example from [17]. The *chopper* design has 3 processes calculating 4 outputs representing different chops for the input signal SRC based on the design configuration by inputs INV and DUP. It is assumed that the design has 5 individual tests T1-T5 of varied length each keeping the values of INV and DUV constant while flipping the value of the SRC input and having appropriate behavior of the clock and reset signals (CLK, CLKN, RST). The design has a bug introduced on

line 28 where instead of correct assignment `F0 <= FF;` the design has a buggy assignment `F0 <= not FF;`. Tests T1, T3 and T4 are able to detect the bug and are referred to as *failing tests*, while tests T2 and T5 pass despite the presence of the bug and are referred to as *passing tests*, respectively. The faulty behavior of the design caused by the failing tests is observed at output TAR_f (assigned at line 46).

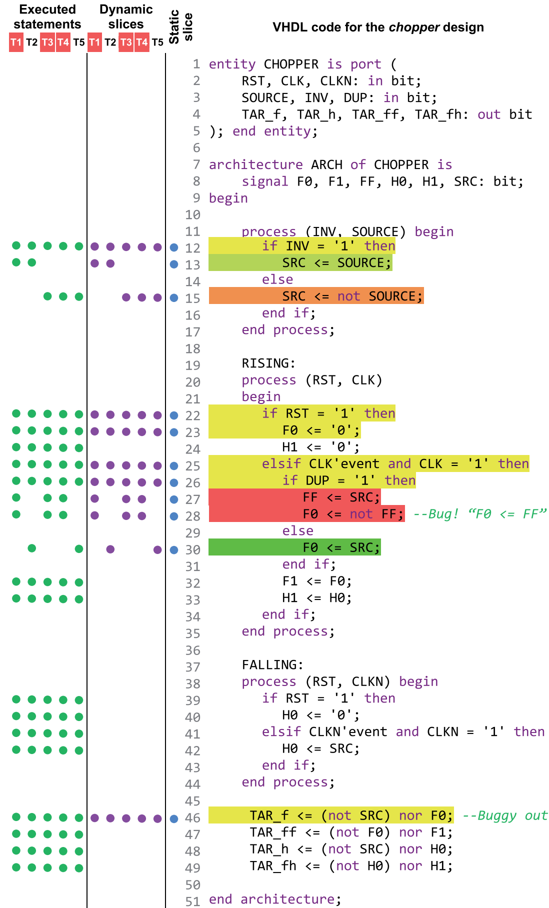


Figure 2. Simplified bug localization in a toy design

A. Static slicing

The presence of concurrent constructs such as the ones found in HDLs versus sequential software domain languages makes static slice computation considerably more complicated [17]. Various approaches dedicated for this task rely on computed models such as control flow graphs and others. zamiaCAD exploits for this purpose its elaborated model refered to as instantiation graphs [21]. Given the IG model it is possible to perform a signal references search through its assignments, both backward to find the dependencies and forward to find other signals and variables influenced by the signal. The resulting *reference graphs* has the signals and variables in its nodes and the dependencies are expressed by directed edges. It may contain cyclic dependencies and may be very large, especially if the search was initiated from primary inputs/outputs of the design. It is possible to limit such search by constraining the depth of the graph. An example dependency graph computed for the *chopper* design's output TAR_f is shown in Figure 3.

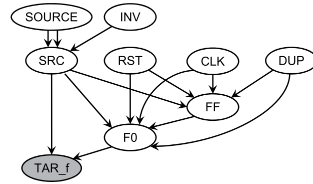


Figure 3. Through-signal-assignment backward reference graph on signal TAR_f in the *chopper* design

Given the through-signal-assignment reference graph, the HDL statements representing the signal and variable dependencies in its edges are collected into a set. The resulting set represents a *static slice* on the signal of interest. However our approach for static slice computation does not consider the order of HDL statements and can therefore be slightly too optimistic i.e. it can potentially include into the static slice some statements that do not represent dependencies influencing the signal of interest. It can be observed only for certain combinations of variable (versus signal) assignments which are a rare case in practical HDL descriptions.

The column Static Slice in Figure 2 marks VHDL statements of a static slice on the TAR_f output by dots (blue in color prints). Static slicing allows having a design ”filter” eliminating from the analysis space the design parts that do not influence the signal of interest. As a result in the *chopper* design example the entire process FALLING and a large part of other statements were excluded from the future analysis.

B. Suspiciousness Ranking Based on Statement/Branch Coverage Metrics

The statistical suspiciousness ranking procedure proposed in this paper is based on design simulation by a *diagnostic test*. A requirement for the diagnostic test is that it has to contain a set of independent sub-tests (e.g separated by design reset) where both failing and passing tests are represented. The quality of the statistical ranking is highly dependent on the quality of the diagnostic test. Functional tests for processors are suitable as diagnostic tests because they are divided into separate sub-tests for processor instructions, so that each such sub-test can be executed independently.

The column Executed Statements in Figure 2 marks the VHDL statements executed during design simulation with each of the 5 tests by dots (green in color prints). A fraction of the set of executed statements can be excluded from the further analysis by applying a static slice filter on an output signal where the faulty behavior was observed. This approach allows obtaining a *dynamic slice* of the design on this signal. The column Dynamic Slices in Figure 2 marks the VHDL statements taking part in the dynamic slices of the tests by dots (purple in color prints). Thus the analysis space for the current example was reduced by 2.2 times (42 versus 92 statement executions by the sub-tests).

The statistical *suspiciousness score* for ranking of the HDL code item i is calculated by the following Formula (1):

$$S(i) = \frac{\frac{Failed}{TotalFailed}}{\frac{Passed}{TotalPassed} + \frac{Failed}{TotalFailed}} \quad (1)$$

Where $S(i)$ is the suspiciousness score value of the code item i, *Passed* and *Failed* are counts of passing and failing tests covered the item i in the dynamic slice, while *TotalPassed* and *TotalFailed* are total numbers of the passing and failing tests in the complete diagnostic test, respectively.

Further zamiaCAD environment visualizes the score $S(i)$ by 4-bit colors that can be interpreted as follows:

- **1** - is colored by extreme red and is interpreted as highly suspicious HDL code item to contain or to lead to the bug
- **0** - is colored by extreme green and is interpreted as an HDL code item above suspicion
- $S_{threshold}$ - is colored by yellow and is interpreted as an item that was not emphasized by the analysis

Here $0<S_{threshold}<1$ is the *suspiciousness threshold* specified by the designer and is by default equal to 0.5. The items having score S values in-between 0, $S_{threshold}$ and 1 are colored by greenish and reddish yellow tones respectively and represent different levels of suspiciousness. Items without scores (and therefore not colored) were either eliminated from the analysis by the static slice filter or not covered by the diagnostic test.

An example of applying the proposed suspiciousness ranking to the *chopper* design is demonstrated in Figure 2. Here the assignment statements on lines 27 and 28 were calculated as the most suspicious (score S = 1) and were colored red, the assignment on line 30 was calculated to be above suspicion (score S = 0) and colored green, while statements on lines 13 and 15 have scores S 0.4 and 0.6 correspondingly and therefore are colored greenish yellow and orange.

C. Hierarchical Analysis Based on Condition Coverage

As it will be demonstrated in the next section, the ROBSY processor case study has emphasized an important general category of design errors that are difficult to localize. They are bugs in complex condition expressions of conditional statements. E.g. Bug 1 in this case study is an erroneous comparison of one of the 35 conditions in a conditional assignment *when* of the ALU module. Localization of such bugs is assisted by suspiciousness ranking of condition items.

We propose to hierarchically rank condition items of the selected (e.g. by a rank threshold) suspicious branch items that belong to suspicious statements. Formula (1) is applied for this purpose considering for i branch and condition items instead of statements. In the proposed approach the *branch items* are separate evaluations into 'true' or 'false' of conditional statements branches and the *condition items* are separate evaluations into 'true' or 'false' of logical operators (i.e. OR, AND, XOR, NOR, etc.) and relational operators (i.e. $>$, $=$, $\leq$, etc.). A detailed example for hierarchical conditions ranking and its application for bug localization is demonstrated in Section 5.

IV. Case Study: ROBSY Processor Design

As a case study, the proposed approach was evaluated by debugging an industrial processor developed as a part of the ROBSY (Reconfigurable On Board self test SYstem) project. This custom processor follows a new test approach [18, 19] to improve the fault coverage and reduce the test time of Printed Circuit Boards (PCBs) during the manufacturing process, and it is developed in cooperation with a major vendor of PCB testing equipment. The ROBSY processor is classified as a single instruction single data (SISD) processor with separated program and data buses (Harvard architecture). The processor has many of the properties of a reduced instruction set computer (RISC), and uses the Wishbone protocol (WB) for the I/O transactions. The current implementation of the processor core contains 17K lines of VHDL code. There are 481 direct signal assignment statements, 413 branches and 1573 conditions.

A. ROBSY Processor: Functional Test

To verify the correct functionality of the Instruction Set Architecture (ISA), a functional test was developed. The functional test consists of a test program written in assembler, executed in a predefined order to test all the instructions supported by the processor. The test program is divided into sub-tests, where each sub-test is in charge of testing a specific instruction and setting register R1 to a specific value that acts as a sub-test label (error code). During the sub-test execution, it is evaluated if the values obtained in the registers, flags, etc., are as expected. In Figure 4 we have an example of a sub-test corresponding to the compare (CMP) instruction.

In the case of an unexpected value, the processor goes to the code section labeled with "fail", the execution is aborted

and the error code of the failed sub-test is written to a dedicated register.

```
; check CMP with flags (register content unsigned)
MOV   R1, 01;  -- error code 01---
MOV   R2, A3;
CMP   R2, 05;
JZ    fail;    if R2 equal 05 (jump zero)
JC    fail;    if R2 < 05     (jump carry)
CMP   R2, A3;
JNZ   fail;    if R2 not equal 05 (jump not zero)
JC    fail;    if R2 < 05
MOV   R3, A4;
CMP   R2, R3;
JNC   fail;    if R2 > R3 (jump not carry)
```

Figure 4. ROBSY processor test program sub-test for the CMP instruction

B. Set of Documented Design Errors

The ROBSY design team has documented a set of their VHDL coding bugs that have the following nature:

- **Bug 1** A *wrong register* is used as one of the operands in a very long conditional expression (35 operators) inside a conditional signal assignment. Possibly, due to a copy-paste error.
- **Bug 2** An entire *conditional sub-expression* (3 operators) resides in the wrong branch of a conditional signal assignment, which contains 9 branches in total.
- **Bug 3** Both, a *missing branch* and a *missing driver* in a short conditional signal assignment.
- **Bug 4** A *wrong enumeration constant* is used in a comparison operation inside a conditional signal assignment.
- **Bug 5** A *wrong driver* is used in a conditional signal assignment. More specifically, register *R* is not updated with its newly computed value typically stored in *R_next* or *R_new* signal. Instead, the same register *R* is used as a driver, which indicates an obvious copy-paste error.
- **Bug 6** A *missing conditional sub-expression* (3 operators out of 6 required ones) in one of the 4 branches of a conditional signal assignment.
- **Bug 7** *One bit* of a register is always and unconditionally set to 0. The whole code line to blame is unnecessary and hence incorrect.

V. Experimental results

This section presents experimental results for the proposed design errors localization approach evaluation on the industrial processor ROBSY. For the purpose of the proposed approach the original functional test (i.e. an Assembler program) was split into 28 independent sub-tests, each targeting a separate instruction. Each of the 7 buggy versions of the processor was simulated with the resulted diagnostic test. The ratio of failing vs passing sub-tests for the bugs in the case-study was the following: *Bug 1*: 4 vs 24; *Bug 2*: 2 vs 26; *Bug 3*: 2 vs 26; *Bug 4*: 1 vs 27; *Bug 5*: 2 vs 26; *Bug 6*: 1 vs 27; *Bug 7*: 1 vs 27.

Figure 5 demonstrates the proposed hierarchical localization of *Bug 1*. The grey areas denote that some detailed

#	Stm. score	Bran. score	Cond. score	Line	Source code lines
	alu.vhd				
6	0.51			88	
5	0.55			104	
2	0.67			108	
1	0.80			110	
5	0.55			116	
5	0.55			127	
				...	
3	0.64	0.64		260	`svFlag_new(0) <= '1'  when afClass=cfClass_1`
			0.64^1_T	261	`and ((svOp_mux(cnD_w)=REG_SOURCE_DEST_IN(cnD_w)--add case`
				262	`and svOp_mux(cnD_w)/=svRes(cnD_w)`
				263	`and ((aCmd=cvCmd_ADD_R_R and c_en_ADD_R_R)`
				264	`or (aCmd=cvCmd_ADD_R_IMM and c_en_ADD_R_IMM)))`
			0.51^1_T	265	`or (svOp_mux(cnD_w)/=REG_SOURCE_DEST_IN(cnD_w)--sub case`
					`-- Bug: correct compar. between REG_SOURCE_DEST_IN and svRes`
				266	`and svOp_mux(cnD_w)/=svRes(cnD_w)`
			0.69^1_T	267	`and ((aCmd=cvCmd_SUB_R_R and c_en_SUB_R_R)`
			0.55^2_T		
			0.75^2_T	268	`or (aCmd=cvCmd_SUB_R_IMM and c_en_SUB_R_IMM)`
			0.57^1_T	269	`or (aCmd=cvCmd_CMP_R_R and c_en_CMP_R_R)`
			0.55^2_T		
			0.51^1_T	270	`or (aCmd=cvCmd_CMP_R_IMM and c_en_CMP_R_IMM)))`
				271	`or (REG_SOURCE_DEST_IN(cnD_w)/=svRes(cnD_w)--shift cases`
			0.55^1_T	272	`and ((aCmd=cvCmd_SHL_R and c_en_SHL_R)`
			0.55^2_T	273	`or (aCmd=cvCmd_SHR_R and c_en_SHR_R))))`
5	0.55	0.55		274	`else '0'       when afClass=cfClass_1     --overflow reset`
				275	`and ((aCmd=cvCmd_ADD_R_R and c_en_ADD_R_R)`
				276	`or (aCmd=cvCmd_ADD_R_IMM  and c_en_ADD_R_IMM)`
				277	`or (aCmd=cvCmd_SUB_R_R and c_en_SUB_R_R)`
				278	`or (aCmd=cvCmd_SUB_R_IMM and c_en_SUB_R_IMM)`
				279	`or (aCmd=cvCmd_CMP_R_R and c_en_CMP_R_R)`
				280	`or (aCmd=cvCmd_CMP_R_IMM and c_en_CMP_R_IMM)`
				281	`or (aCmd=cvCmd_SHL_R and c_en_SHL_R)`
				282	`or (aCmd=cvCmd_SHR_R and c_en_SHR_R))`
NA	0.50			283	`else          svFlag(0);`
	data_interface_mod.vhd				
2	0.67			155	
2	0.67			158	
	gprs_mod.vhd				
4	0.60			97	
	state_machine.vhd				
4	0.60			100	
4	0.60			123	
4	0.60			168	

Figure 5. Details of automated localization of *Bug 1* in the ROBSY processor

information was omitted from the figure. First the dynamic slices (intersection of executed statements with the static slice on an observable faulty output) were generated for all of the test cases and the statistical suspiciousness ranking was performed. This analysis resulted in 14 statement candidates (out of the initial total 481 assignment statements) whose suspiciousness score S was above the default suspiciousness threshold $S_{threshold}$=0.5. The figure shows in the second column *Stm. score* the scores for these 14 suspicious statements, and in the first column their rank # based on the score (6 ranks in total). Most of the statements with high scores were found in the ALU processor module (file `alu.vhd`). The figure demonstrates a part of the actual VHDL code for the conditional assignment of the overflow flag signal `svFlag_new(0)`. Bug 1 is located in the condition expression at line 266 (correct comparison had to be made between signals `svRes(cnD_w)` and `REG_SOURCE_DEST_IN` instead of `svOp_mux(cnD_w)`). This complex conditional assignment (lines 260-283) contains 3 individual assignments at lines 260, 274 and 283. The first two assignments have 3rd and 5th ranks while the last one has the "yellow" score S=0.5 and is filtered out together with other statements with scores 0.5 and less. The automated localization iteratively advises the designer to consider as bug location candidates the statements with the highest ranks starting with the one at line 110, followed

by statements at line 108 in `alu.vhd` and lines 155 and 158 in `data_interface_mod.vhd` (complemented with hierarchical analysis of the corresponding branches and conditions). Further it will advise the designer the statement at line 260 in `alu.vhd` with the next rank 3 and score value 0.64. Then it will proceed with score computation of its branch on the same line (S_{b260}=0.64 in column *Bran.score*). The suspiciousness scores of separate condition evaluations to 'true' and 'false' related to this branch artifact are also calculated. The ones that have score S>0.5 are specieved in column *Cond.score*. One of the highest scores here has the logical *and* at line 267. One of its operands is actually the incorrect signal comparison documented as *Bug 1*.

Table I
STATISTICS OF THE PROPOSED BUG LOCALIZATION APPROACH FOR THE ROBSY PROCESSOR BUGS

The proposed automated localization						Manual debug
Bug name	Statements cand. / %	Conditions cand.	Localized stm. rank	Localization result	Time (min)	Time
Bug 1	14 / 2.9%	16 of 35	3	Automatic	2	4 hours
Bug 2	7 /1.4%	13 of 43	1	Automatic	2	2 hours
Bug 3	20 / 4.0%	2 of 7	3	Automatic	2	4 hours
Bug 4	6 /1.2%	N/A	(1) + ext.slc.	Semi-automatic	2 (+5)	4 hours
Bug 5	11 / 2.3%	4 of 11	1	Automatic	2	2 hours
Bug 6	8 / 1.7%	N/A	(1) + ext.slc.	Semi-automatic	2 (+10)	5 hours
Bug 7	21 / 4.3%	N/A	(1) + ext.slc.	Semi-automatic	2 (+1)	1 hour

Table I demonstrates the statistics of applying the proposed bug localization approach to all of the 7 bugs. The second column in the table shows how many statements were proposed as bug location candidates and also demonstrates these numbers in percentage of the total number of the corresponding items which was 481. As supplementary debug data, the hierarchical analysis has selected suspicious conditions out of conditions of the candidate statements from the second column. Their percentage varies for different bugs from 20.3% to 39.8% (e.g. 72 conditions with suspiciousness score above the threshold out of all 354 conditions of the statement selected for *Bug 5* location candidates). The third column demonstrates the number of selected conditions candidates of the total number of conditions of the located statement. The fourth column shows the rank of the statement actually containing the bug. The diagnostic test was sufficient to automatically localize 4 of the 7 bugs (i.e. *Bugs 1*, *2*, *3* and *5*). Localization of the remaining three bugs was semi-automatic and required manual interaction. In these cases, the bug locations were traced in zamiaCAD by the through-signal-assignment reference search (also used for static slice computation) with limited depth initiated from the signals involved in the highly ranked assignment statements. *Bugs 4*, *6* and *7* were present within extra static slices on the signal from the corresponding statements with the highest rank (reference search depth was 1 assignment for the three bugs and has introduced 21, 13 and 10 additional bug candidates respectively). Automation of this process is conceptually possible and is planned as a future work.

The last two columns in Table I compare the time required for bugs localization by the proposed automated localization approach and conventional manual debug process used by designers before. The time values for the manual process are reported by designers based on their real experience with locating these bugs using a fast commercial simulator. The reported time for automated debug approach (2 min) is mainly spent for separate simulation of the 28 sub-tests and is still pessimistic due to the relatively slow current implementation of the VHDL simulator in the zamiaCAD framework. The framework also supports interaction with external simulators and waveform VCD dump files imporing whose application would allow reducing the runtimes of the automatic approach by a factor of ten. The additional time in brackets reported for *Bugs 4*, *6* and *7* reflects the manual interaction part in the semi-automatic localization process.

To the best of our knowledge none of the state-of-the-art automated hardware design error localization approaches are capable to handle industrial size RTL designs such as ROBSY. Therefore direct comparison to other than manual approaches was not possible for this empirical study.

VI. CONCLUSIONS

The paper presents an approach to automatic localization of design errors (bugs) in processor designs. The approach is based on two main iterative phases: dynamic slicing and statistical suspiciousness ranking of the HDL statements in the design. The dynamic slicing reduces the debugging analysis to all the statements that actually affect the design's faulty behavior for a given stimuli. Then, the suspiciousness ranking assigns a suspiciousness score to each statement present in the dynamic slice.

The novelty of the approach is that it successfully in a scalable manner applies static slicing for analysis space reduction to realistic-size industrial designs and considers different coverage metrics for refining the bug localization. The approach is fault-model free and supports localization of multiple bugs. The original functional tests of processor designs can be used as a diagnostic test and is sufficient for the approach. However, quality diagnostic test can further increase the localization accuracy.

VII. ACKNOWLEDGMENTS

The work has been supported in part by EC project FP7-ICT-2009-4-248613 DIAMOND, by EU through the European Regional Development Fund (ERDF, EFRE) and by Estonian SF grants 8478 and 9429. The ROBSY project is funded by the Thuringian Government and sponsored by European Commission (ESF) and the TAB - Thueringer Aufbaubank under the grant number 2010 VF 0028.

REFERENCES

[1] SIA. International Technology Roadmap for Semiconductors, Design, 2011 edition: http://www.itrs.net/Links/2011ITRS/Home2011.htm.

[2] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, Scalable statistical bug isolation, ACM SIGPLAN Notices, vol. 40, no. 6, pp. 15-26, 2005.

[3] G. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, Statistical debugging: A hypothesis testing-based approach,? IEEE Trans. on Software Engineering, vol. 32, no. 10, pp. 831-848, 2006.

[4] W. E. Wong, V. Debroy, and B. Choi, A family of code coverage-based heuristics for effective fault localization,? Journal of Systems and Software, vol.83, no.2, pp.188-208, 2010.

[5] W. E. Wong and Y. Qi, BP neural network-based effective fault localization,? International J. of Software Engineering and Knowledge Engineering, vol. 19, no. 4, pp. 573-597, 2009.

[6] H. Cleve and A. Zeller, Locating causes of program failures,? Proc. Int. Conf. on Software Engineering, pp. 342-351, 2005.

[7] J. A. Jones and M. J. Harrold Empirical evaluation of the Tarantula automatic fault-localization technique,? Proc. Int. Conf. on Automated Software Engineering, pp. 273-283, 2005.

[8] J. Raik, U. Repinski, R. Ubar, M. Jenihhin, A. Chepurov. High-Level Design Error Diagnosis Using Backtrace on Decision Diagrams, Proc. Norchip, pp. 1–4, 2010.

[9] A. Veneris, I.N. Hajj, Design error diagnosis and correction via test vector simulation, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol.18, no.12, pp.1803-1816, 1999.

[10] A. Smith, A. Veneris and A. Viglas. Design Diagnosis Using Boolean Satisfiability, Proc. Asia and South Pacific Design Automation Conference (ASPDAC), pp. 218–223, 2004.

[11] S. Staber, B. Jobstmann, and R. Bloem, Finding and fixing faults, Proc. CHARME, pp. 35-49, 2005.

[12] R. Konighofer and R. Bloem, Automated error localization and correction for imperative programs, Proc. of Formal Methods in Computer Aided Design, pp. 91-100, 2011.

[13] F. Wotawa, Debugging VHDL Designs: Introducing Multiple Models and First Empirical Results, Applied Intelligence 21, 2, 159-172, 2004.

[14] B. Peischl and F. Wotawa, Automated Source-Level Error Localization in Hardware Designs, Design&Test of Computers, 23, 1, pp. 8-19, 2006.

[15] B. Korel and J. Laski, Dynamic program slicing. Information Processing Letters, vol. 29, no. 3, 1988, pp. 155-163.

[16] M. Weiser. Program slicing. IEEE Transactions on Software Engineering, 10(4):352-357, 1984.

[17] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing for VHDL. In Charme99, Bad Herrenalb, Germany, September 1999.

[18] J.H. Meza Escobar, J. Sachsse, S. Ostendorff, and H. D. Wuttke, Automatic generation of an FPGA based embedded test system for printed circuit board testing, Proc. LATW2012, Quito, Ecuador, 2012, pp. 75-80.

[19] J. Sachsse, S. Ostendorff, H. D. Wuttke, and J. H. Meza Escobar, Architecture of an adaptive Test System built on FPGA, Proc. ARCS 2011 - Architecture of Computing Systems, Como, Italy, 2011, vol. LNCS 6566, pp. 86-97.

[20] zamiaCAD Framework website. http://zamiacad.sf.net/, 2012.

[21] A. Tšepurov, G. Bartsch, R. Dorsch, M. Jenihhin, J. Raik, V. Tihhomirov, A Scalable Model Based RTL Framework zamiaCAD for Static Analysis, Proc. IEEE VLSI-SOC, October 7-10, 2012. Santa Cruz, USA, pp. 1-6.

[22] Aeroflex Gaisler AB. LEON3 SPARC V8 Processor IP core, GRLIB IP library, v.1.1.0-b4108. http://www.gaisler.com/.

Research paper II

A. Tšepurov, G. Bartsch, R. Dorsch, M. Jenihhin, J. Raik, V. Tihhomirov, "A Scalable Model Based RTL Framework zamiaCAD for Static Analysis". *IFIP/IEEE 20th International Conference on VLSI and System-on-Chip (VLSI-SoC)*, October 7-10, 2012, Santa Cruz, USA.

The author's contributions are: investigating and writing the related work section of the article, drafting other sections; participating in describing the internal structure of IG model in the article; setting up the experimental environment, creating large systems-on-chip (SoCs) based on LEON3 CPU, performing experiments on the created SoCs using different tools and evaluating the tools against zamiaCAD; presentation of the article at VLSI-SoC conference in Santa Cruz, USA (Silicon Valley).

A Scalable Model Based RTL Framework *zamiaCAD* for Static Analysis

Anton Tšepurov[1], Günter Bartsch[2], Rainer Dorsch[3], Maksim Jenihhin[1], Jaan Raik[1], Valentin Tihhomirov[1]
[1]Dept. of Computer Engineering, Tallinn UT, Tallinn, Estonia, Email: {anchep | maksim | jaan | valentin} @ati.ttu.ee
[2]zamiaCAD project founder, Stuttgart, Germany, Email: guenter@zamia.org
[3]IBM Research and Development GmbH, Böblingen, Germany, Email: rdorsch@ieee.org

***Abstract*—As of today, RTL still remains the primary abstraction level for VLSI SoC design entry and state-of-the-art design flows need to cope with designs of enormous size, and thus, to scale well. This paper presents an open-source framework zamiaCAD based on a scalable model that includes both, a comprehensive elaboration front-end for RTL design and design processing back-end flows. The persistence and scalability are guaranteed by a custom-designed and highly optimized object database. As an HDL-centric framework it follows the concept of non-intrusiveness. In this paper, we discuss in detail the concepts of design elaboration into the scalable design model and present an evaluation of the model for static analysis as one of the back-end applications. Experimental results on very large designs show that zamiaCAD compares favorable to other frameworks with respect to the scalability aspects.**

***Index Terms*—RTL, static analysis, scalability, VHDL**

I. Introduction

The ITRS roadmap predicts that the technology scaling for semiconductor chips continues at least until the year 2020 [10]. Furthermore, it is expected that Register Transfer Level (RTL) remains the dominant level of abstraction for design entry in the next years. There are numerous RTL chip design flows implemented in commercial frameworks in use today involving front-end and back-end tools. The front-end consists mainly of design parsing and design elaboration and storing the result in a database. These steps are typically language dependent. The back-end applications read the elaborated design from the database. They might be targeted towards design analysis, design synthesis, and optimization or simulation. The algorithms used in the front-ends and back-ends are often based on methods developed years ago (e.g. [13], [19]) when scalability issues were less of a concern than they are now.

The enormous size of today's RTL designs for VLSI SoCs together with the predicted technology scaling brings to the front the scalability property of the model. Since satisfying the scalability property is a non-trivial task, such models are usually kept in-house and are as proprietary as the tools built upon them.

Design analysis as a typical back-end application for design exploration, verification and debug allows the designer to effectively manage design functionality and structure. It is possible to perform several tasks such as signal tracing, data flow, and schematics view to some extent dynamically (e.g [12]) relying on the testbench quality. However, proper design analysis for these and a set of other analysis tasks (e.g hierarchy exploration, precise global signal tracing, type check, sink-less or source-less signals analysis) can be performed only statically [6] and requires generation of a proper computed design model.

The proposed framework puts emphasis on scalability, accessibility to the research and engineering communities, and non-intrusiveness (HDL-centric). The latter is achieved by having a Hardware Description Language (HDL) as the only data format and requiring almost no configuration for pre-existing design projects. Functionally, the framework addresses advanced hardware RTL design, verification and analysis. Currently, zamiaCAD supports IEEE 1076-1993 VHDL plus many extensions of the later standard VHDL 2008 in the front-end. The front-end may be extended to other languages. On the back-end side zamiaCAD supports design entry and comprehensive analysis. It includes an Eclipse IDE plugin based Graphical User Interface (GUI) and contains a built-in VHDL simulator and work-in-progress stubs for synthesis. It handles very large open source (e.g. Leon3 [4] based) and industrial designs. Altogether zamiaCAD contains currently more than 100,000 lines of Java code (excluding generated parts).

This is the first paper describing zamiaCAD. We present the core components of zamiaCAD with a strong focus on scalability aspects and omit the description of the end user features. The contributions of the paper may be summarized as follows. First, it introduces the concepts of a new scalable computed design model named IG (Instantiation Graph model). The persistence and scalability of the model are guaranteed by a custom designed object database referred to as ZDB that is highly optimized for performance. Second, it presents the details of the comprehensive RTL design elaboration front-end. Third, it evaluates the model for static analysis as one of the back-end applications. The experimental results use as a reference widely used commercial state-of-the-art tools and consider for benchmarks very large RTL designs including a SoC with multiple thousands of Leon3 CPU cores [4].

The rest of the paper is organized as follows. Section 2 discusses related works and the capabilities of existing RTL design flow frameworks relying on their design models. Section 3 briefly introduces the general zamiaCAD flow. Section 4 describes the front-end steps preceding the elaboration process and discusses its limitations. Section 5 presents the concepts of the scalable model and reveals the details of the elaboration

process. Section 6 discusses the persistence and scalability instruments. Section 7 presents the experimental results and Section 8 concludes the paper.

II. Related Work

HDL code entry assistance (HDL front-end) could be regarded as the easiest hardware design task. It only requires a parser and a parse tree as an underlying model. Tools like Veditor [23], Sigasi [15], Simplifide [17] propose good Eclipse-based solutions for this and provide code entry, syntax highlighting, code outline and content assist for VHDL and Verilog. In addition, Simplifide offers hardware specific refactoring features and a coarse-grained design hierarchy view.

When it comes to navigation and legacy code exploration, tools can no longer rely on a pure parse tree. It alone will allow neither precise navigation (e.g. to find the correct declaration of an overloaded function), nor a decent design comprehension (no elaborated chip hierarchy and other visual representations), nor global signal tracing etc. For such tasks, an elaborated model is required. To grasp the design hierarchy at once and to speed-up the code entry, HDL Designer [11] from Mentor Graphics offers a sophisticated solution for both initial automatic code generation and further visualization and analysis thereof. However, the code gets generated out of non-standardized tables and graphics, which brings with it all the problems and overheads of invasive approaches, e.g. forces the whole team to use the same tool and only allows a minimal fine-tuning of the generated code thereafter.

The majority of EDA tasks require an elaborated model of the design. For instance Design Compiler [22] by Synopsys, or Verdi Automated Debug System [18] do create such models as an interim step for their further backend processing that are synthesis and verification, respectively. Verific Parser Platform [24] by itself is a naked elaboration engine for both VHDL and Verilog. Verific elaborates designs into a common language-independent HDL database, which is further used as a basis by several commercial hardware design tools.

Finally, when targeting the task of simulation, elaborated HDL models have another trait easily observed in such tools as ModelSim [12] and ghdl [9]. Even though the initial compilation/parse of the design is often fast, one has to start the simulator to launch the elaboration and produce the elaborated model for simulation. In practice, this simulation start takes not only a long time to complete, but it often fails by running out of either memory or time in case of large state-of-the-art designs. It confirms the importance of model scalability. The experimental results in Section VII show that zamiaCAD compares favorable to the other design frameworks in the elaboration process. It even comes close to tools which generate parse trees only.

III. zamiaCAD framework overview

Figure 1 shows a simplified overview of the zamiaCAD framework flow. First, as the front-end part, the HDL code is parsed which results in a (language specific) Abstract Syntax Tree (AST). It is then elaborated into the Instantiation Graph

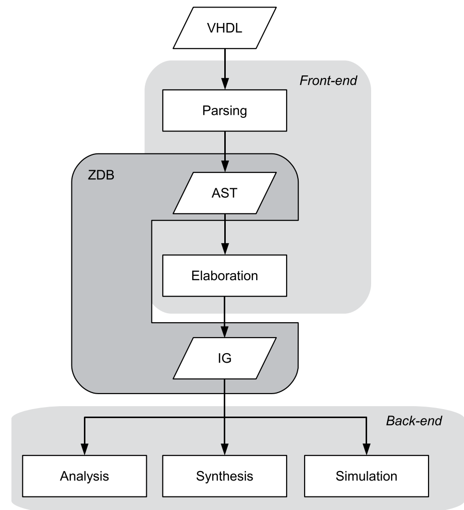


Fig. 1. Simplified zamiaCAD Flow

(IG) model. During the elaboration, all semantic rules as defined by the source HDL are applied and, most importantly, identifiers (names) are being resolved and replaced by references to the objects they denote. Also, semantic rules are applied and checked, such as syntax checks, scoping rules, static array boundaries etc.

The final IG is a very fundamental data structure in zamiaCAD as it is used as the basis for all back-end applications offered by the framework such as analysis (including static analysis focused in this paper), RTL synthesis and simulation.

The persistence and scalability of both AST and IG models are guaranteed by a custom-designed and highly optimized object database zamiaCAD Database (ZDB). In the next three sections we will discuss the models and the database in detail.

IV. Abstract Syntax Tree model and its limitations

The AST model in the zamiaCAD flow serves mainly as an intermediate step in the IG elaboration process. Its only particularity with respect to traditional AST approaches is its scalable handling by ZDB (see Section 6). This allows the model to accommodate large netlists, weakly supported by proprietary tools that rely on AST-like design models. This is particularly important, because today it is a common practice in industry to use netlists for certain cores in a design.

The AST generated by zamiaCAD's parsers is a language-specific, traditional tree structure closely modeled after the grammar productions as specified by the respective HDL language reference manual. It is closely correlated to the source code processed by the parser as there have been very few semantic rules applied to it yet. The AST is a true tree data structure which is due to the context-free nature of the grammars used.

We will illustrate this by looking at the AST generated from a very simple VHDL example given in Figure 2. Figure 3

```
entity foo is
  port (A, B : in  bit;
        Z    : out bit);
end;

architecture rtl of foo is
  signal t : bit;
begin
  T <= A xor B;
  Z <= T;
end;
```

Fig. 2. VHDL Source Code Example

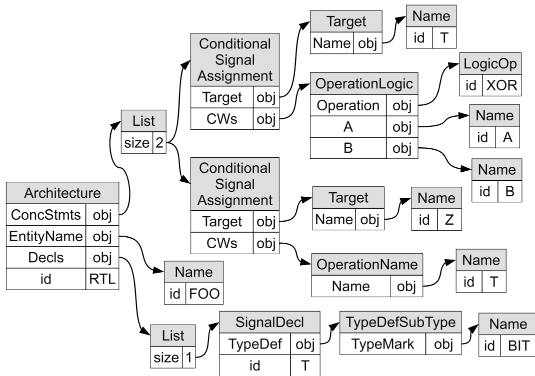


Fig. 3. Abstract Syntax Tree

shows a slightly filtered and simplified AST for the architecture part.

It is important to notice that there is no contextual information present in this data structure. Instead, any VHDL object is still denoted by names (which are, in this example, simple identifiers but could be more complex). If we look at the way the signal declaration is represented here, this is very obvious: the signal *T* is declared as a subtype of *BIT*, but we have no information what the *BIT* type really is.

The same problem can be observed when looking at the concurrent signal assignments — the relevant signals are denoted by names which have not been resolved yet since that would require semantic knowledge which a parser doesn't have.

Generally AST application in RTL design flow may include editor support such as folding and outline view (e.g. zamiaCAD relies for these two applications on the AST model), identifier completion proposals, simple identifier occurrence tracing by scoping rules within the file (local tracing) and local identifier refactorings, and simple signal value annotations.

However, any static analysis tasks that need identifier resolution (including type information) can be performed only on properly elaborated design model such as IG. These tasks include tracing of parts of signals, precise global tracing, overloaded subprograms, tracing through generate-statements, global refactorings/traces, advanced signal value annotations (e.g. annotating only one bit of vector, computing expressions on the fly). The concepts of this model are described in the next section.

V. Instantiation Graph model

The process of design elaboration in the zamiaCAD results in its representation by Instantiation Graph (IG).

Definition. The *Instantiation Graph* is a data structure represented by a densely connected graph of semantically resolved objects representing elements of the hardware design.

IG is generated by applying semantic rules to the AST to resolve all names (identifiers) and replace them with references to the actual objects. While consistency and rule checks (e.g. type checks) are also applied during this process, the main purpose here is name resolution. The names can reference objects located in outer scopes, packages or other units and therefore its partitioning prior storing to a database (see Section 6) should be performed with care.

Currently, IG is modeled closely after the ideas outlined by the VHDL language reference manual [3] when it comes to elaboration. However, IG is designed to become language-agnostic, so it should be possible to extend it to support e.g. Verilog [2] or AHDL [14] by adding classes and attributes to cover concepts specific to those languages.

A. Modules in IG

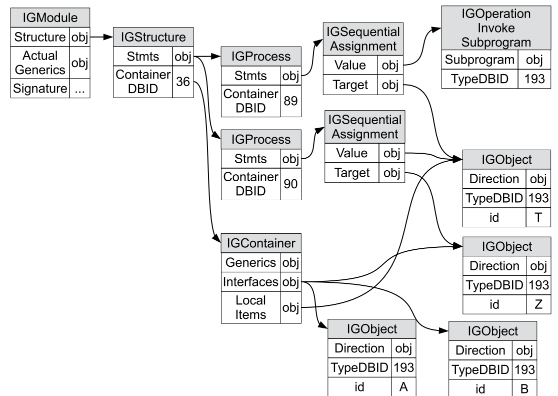


Fig. 4. A Module in IG

Elaborated units are called *modules* in IG. Apart from packages, they form the topmost hierarchical level in IG. Figure 4 shows what the highlevel IG data structures look like for our example from Figure 2. Note that it is impossible to show a complete IG even for a very basic design, because IGs have far too many nodes and edges. We will therefore have to look at small, hand-selected, and simplified portions of IGs. IG does not separate between entity (interface) and architecture (body). Also, modules don't have generics — instead, for each distinct set of generics, which is used anywhere in the design, a new `IGModule` is generated. The set of actual generics (a set of static values) is stored inside the module and is also used to compute a signature (a simple string) for the IGModule.

Therefore it is possible to quickly locate it via a ZDB index during elaboration or reference it by `IGInstantiations`.

The actual body of the module is to be found in an `IGStructure` object. Since modules can be hierarchical (e.g. when the original VHDL architecture uses blocks or generate statements), IGStructures can contain IGStructures in a recursive way. Hence, IGStructure is one of only three possible statements to be found in an IGStructure:

- IGStructure
- IGProcess
- IGInstantiation

All other kinds of concurrent statements found in VHDL are mapped to equivalent `IGProcess` statements in IG. In our example we have two concurrent signal assignment statements, which have been converted into two equivalent IGProcess statements.

Besides the concurrent statements, IGStructures also contain information about all (VHDL) objects declared in their scope in an `IGContainer` object. `IGContainers` can be regarded as namespaces — they contain references to all named objects in a specific namespace. In our case, we find four different objects here: *A*, *B* and *Z* which are interfaces declared in the entity, and *T* which is a local signal. In a similar manner, packages, represented in IG by `IGPackage` objects, consist of a single (possibly hierarchical) IGContainer which holds references to all the named objects they provide.

B. Objects, Types and Expressions in IG

Expressions illustrate best the big difference between AST and IG: in IG, all names have been resolved so they can be replaced by a reference to the specific object which was denoted by them.

Figure 4 shows what the concurrent assignments found in our example from Figure 2 look like in IG. The two `IGProcess` data structures correspond to the two statements as discussed in the previous section. Let's concentrate on the first assignment with an operator in it:

```
T <= A xor B;
```

In IG, this is turned into an `IGSequentialAssignment` statement inside a process. The target *T* of the expression is the only VHDL object directly referenced by this sequential assignment. The other 2 objects are hidden behind a logical operation expression. In IG, such expressions are actually translated into subprogram calls — which is due to the fact that operators in VHDL can be overloaded and therefore need to be resolved just like any other name. So, in IG, we have a direct reference to the specific subprogram which is to be invoked here to perform the correct operation.

Likewise, names denoting types are resolved and replaced by references to the objects representing those types. Figure 5 shows the corresponding IG subgraph for our example. The *BIT* type from the package named *STANDARD* (which is imported implicitly in any VHDL unit) has been resolved to a `IGTypeStatic` object, which is a subclass of the more generic `IGType` class that can also represent arrays not having

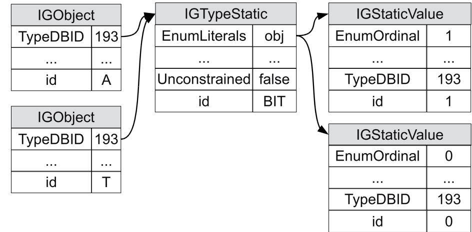


Fig. 5. Types in IG

static boundaries, which can be used in subprograms. We can also see the two enum literals *0* and *1* which are represented by `IGStaticValue` objects — these objects are used in IG to represent any kind of static (constant) value.

VI. Persistence and Scalability

Especially for today's industrial designs, the data structures tend to get very large, often larger than the available RAM on a typical hardware engineer's personal workstation. While many will argue, that the affordable amount of RAM per workstation will increase exponentially with time, unfortunately the same holds true for the increase in size of the designs being processed so in effect we will always face the scalability issue.

Quite naturally, the solution is to hold only small parts of these data structures in memory at any given point in time while keeping the whole data structure on disk [5]. Of course, we have to make sure this is done in a quick and transparent way such that users won't even notice any effects apart from being able to handle large designs on modest hardware. Using disk space to store these data structures also solves the problem of persisting them since disk memory is persistent.

Persistence is very desirable as parsing and elaboration can take a lot of time, especially for large projects so users will want to keep the results from one zamiaCAD session to the next. zamiaCAD exploits principals of incremental elaboration. If only a part of the design has been modified, then only these updated design units will have to be elaborated to IG.

In zamiaCAD a custom designed, highly optimized (for EDA purposes) object database [16] called ZDB is used as the underlying mechanism for persisting data on disk. Figure 6 gives a slightly simplified overview of ZDB's internal structure. At its core, ZDB uses Java's built-in object serialization features [21] to read and write objects from/to disk. Each object stored in ZDB gets assigned a numeric identifier, called *DBID*, to it which can later be used to retrieve the object. ZDB offers various indexing features that allow strings to be used to denote objects making the underlying *DBIDs* transparent for any code that uses ZDB.

As already mentioned, ZDB is highly tuned for performance. Generic performance features include the use of aggressive caching of objects in memory including a last-recently used cache eviction strategy as well as the use of standard extensible hash maps [8] for maintaining potentially large indices on disk. Other optimizations are specific to the EDA use case: typically, we want to build large data structures

very quickly and want to be able to traverse and index them as efficiently as possible. Deleting and modifying data on the other hand happen much less frequently. Therefore, ZDB doesn't really implement these last two operations, but only mimics them: object deletion will only result in it being removed from indices so it becomes inaccessible, but it will continue to be stored on disk. Object modification is not supported at all but will require the application using ZDB to delete the old object from ZDB and store it again — a technique widely used in enterprise databases (e.g. *vacuumdb* utility in Postgres [20]).

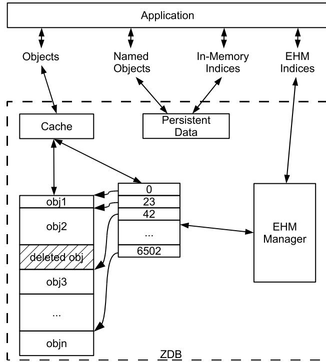


Fig. 6. ZDB

A. Persistence and Scalability in AST

All AST nodes are persisted using ZDB. Due to the pure tree nature of the AST, persistence can be done simply by storing individual library/design units as ZDB objects while keeping track of them using indices so they can be found quickly during elaboration. This kind of simple ZDB persistence also gives us basic scalability, we can basically process any number of units very efficiently as long as individual units fit in RAM.

Unfortunately, with automatically generated flat HDL netlist dumps, this assumption doesn't hold. Such dumps can contain millions of concurrent statements in one single unit causing the parse pass to run out of memory. To solve this, an additional level of ZDB indirection has been introduced: for architectures, we store each concurrent statement separately using ZDB and keep just a list of *DBIDs* (simple long integers) in memory. That way, zamiaCAD can deal with arbitrarily large netlist dumps very efficiently.

B. Persistence and Scalability in IG

Just like AST objects, IG objects are persisted in ZDB. However, due to IG being a true graph with many cross-references between items, storing IG structures takes considerably more effort than AST sub-trees do. Since name resolution is the source of all these interconnections, it is a natural point for partitioning IG for storage. So, anything which can be denoted by names needs to be stored as a separate ZDB object and (except for transient fields) is never allowed to be referenced directly, but through the object's DBID.

VII. Experimental Results

In this section we present the experimental results for the proposed scalable IG model implemented in zamiaCAD. To estimate scalability of the model, we use as a reference widely known commercial state-of-the-art tools for synthesis (T1), for design elaboration front-end (T2), a commercial and an open-source tool for simulation (T3 and T4) and an advanced code entry tool T5. All of the tools create their own proprietary computed design representation models dedicated for the targeted task. We show that only proposed IG model can easily handle designs of the given complexity, while the other models fail to do so.

We have used the following designs as benchmarks:

- **B19** - a design from ITC'99 benchmarks family [7] combining several smaller circuits from it.
- **Plasma** - an open-source processor benchmark from the OpenCores.org website.
- **L3-SoC** - a SoC built by interconnecting a wide set of IP cores from GRLIB. It is available from [1].
- **L3-1** to **L3-3584** - the original unchanged Leon3 CPU core design [4] configured to contain *n* instantiations with *m* CPUs as AMBA masters. *L3-64* is 64 instantiations with a single CPU on AMBA bus (64 core design). *L3-896* is 64 instantiations with 14 CPUs on AMBA bus (896 core design). *L3-3584* is 256 instantiations with 14 CPUs on AMBA bus (3584 core design). The tools capable of detecting identical modules inside the design will avoid duplications and thus save memory resources.

TABLE I
ELABORATION TIME (SEC)

Design	Size	T1	T2	T3	T4	T5	zC
B19	834	7	1	5	1	4	3
Plasma	1068	3	1	9	2	7	9
L3-SOC	38107	83	5	135	104	918	183
L3-1	27409	87	6	82	187	694	137
L3-64	27409	OOM^1	23	87	OOM^7	693	162
L3-896	27409	OOM^2	97	OOM^5	OOM^8	695	223
L3-3584	27409	OOM^3	OOM^4	OOM^6	OOM^9	692	405

Table I presents the elaboration time for the described above large benchmarks. The second column presents the size of the designs in terms of VHDL concurrent and sequential statements (a more accurate metric than lines of code).

Here OOM^k means a reference tool has run out of memory or crashed for the given design. OOM^{1-3} - T1 has run out of memory in 448, 860 and 2340 seconds correspondingly. OOM^4 - T2 has crashed on 208th instantiation with 14 CPU cores (2912th of the 3584 cores). OOM^{5-6} - T3 has halted after 3170 and 949 seconds correspondingly and kept infinitely waiting for additional memory resources. OOM^{7-9} - T4 has run out of memory in 16, 8 and 8 seconds correspondingly.

In case of zamiaCAD, the elaboration time measured includes indexing, parsing, IG elaboration, and ZDB commit. For instance, the total time spent for elaboration of *L3-1* and *L3-3584* designs can be split correspondingly (137.44s total = 1.44s parsing + 3.57s indexing + 95.19 IG build + 37.24s ZDB commit) and (405.16s total = 1.16s parsing + 5.06s indexing + 363.73 IG build + 35.21 ZDB commit).

Table II presents values for memory allocation both in RAM and on hard disk drive for handling the internal models during the elaboration process (see Table I). Given a reasonable amount of RAM, zamiaCAD scales well with the design size.

TABLE II
INTERNAL MODEL SIZE IN RAM / ON DISK (MB)

Design	T1	T2	T3	T4	T5	zC
B19	120/0.48	17/0	5/0.32	1/2	254/0	248/0.9
Plasma	70/1.3	15/0	5.3/0.68	3/2	280/0	269/5
L3-SOC	310/32	117/0	39/22	45/47	693/0	1337/114
L3-1	268/35	107	72/14	45/66	613/0	1332/83
L3-64	OOM[1]	332	444/14	OOM[7]	610/0	1360/134
L3-896	OOM[2]	1205	OOM[5]	OOM[8]	619/0	1465/236
L3-3584	OOM[3]	OOM[4]	OOM[6]	OOM[9]	616/0	1488/584

Table III presents results for two static analysis tasks: global signal reference search and static slice extraction. For the first task in case of *B19* design, AST-based tools (e.g. T5) would find only 39 occurrences of identifier *CLOCK* in the design file due to ignoring multiple instances of components, as opposed to 121 actual references found in the elaborated design by zamiaCAD. In case of the *L3-1* and *L3-3584* designs the precise global signal tracing (tracing through *generate*-statements) also results in a larger and more correct amount of actual references.

A static slice represents a subset of those signals/variables and their assignments/querying that affect the signal of interest. Slices can be optionally filtered to readers or drivers of this signal only, and can be limited by depth of the dependency. Table III shows static slices for signals *BS*, *ATA_DMACK* and *ERX_DV*. *B19* has 56 reference signals and variables (appearing on 315 lines of code) that are dependant through signal assignments on signal *BS*. The dependency cone for signal *ATA_DMACK* contains 1984 or 51% of the signals and variables in design *L3-SOC*. There were also extracted static slices for benchmarks L3-1 and *L3-3584* containing all readers and readers within dependency depth limited to 3 for signal *ERX_DV*. In case of the *L3-3584* design the static slice involves readers in all instances of the processor core. Slices help designers to concentrate on the important code areas during debug.

TABLE III
STATIC ANALYSIS

	Reference Search			Static Slices			
Design	Signal	Refs.#	T(ms)	Signal	Refs.#/%	Lines #/%	T(s)
B19	CLOCK	121	22	BS_{read}	56/9	315/23	0.4
L3-SOC	IRQO	71	31	ATA_DMACK_{drv}	1984/51	5846/2	26,8
L3-1	CLK	6	7	ERX_DV_{read}	2746/53	4269/1	19,0
	RST	5	8	$ERX_DV_{lim=3}$	58/1	288/0.1	2,2
L3-3584	CLK	1281	20203	ERX_DV_{read}	3001/0.05	4269/1	103,4
	RST	1025	23241	$ERX_DV_{lim=3}$	313/0.006	288/0.1	69,1

All experiments were performed on an Intel®Pentium®Dual CPU 2.2GHz machine with 3.8GB of RAM running under Linux OS distribution CentOS 6.0.

VIII. CONCLUSIONS

The paper presented an open-source framework zamiaCAD that includes both comprehensive elaboration front-end and design processing back-end flows for RTL design of VLSI SoCs. The concept of design elaboration into the scalable design model called IG was discussed in detail. The aspects of persistence and scalability of the model guaranteed by a custom-designed and highly optimized object database were discussed. In addition, evaluation of the framework for static analysis as one of the back-end applications was carried out.

The experimental results on very large designs show that the zamiaCAD framework compares favorable to other frameworks with respect to the scalability aspects. First, the existing proprietary tools handled very large designs and multiple instantiations of identical cores poorly compared to zamiaCAD. Second, only a few tools generate properly elaborated design models suitable for comprehensive static analysis tasks. Third, the tested proprietary tools seem to rely more heavily on swapping pages of the operating system and, compared to zamiaCAD, less on use of disk storage when the data is persisted with the help of domain knowledge. Fourth, zamiaCAD's scaling efficiency is close to the AST level only tools.

IX. ACKNOWLEDGMENTS

The work has been supported in part by EC project FP7-ICT-2009-4-248613 DIAMOND, EU Regional Development Fund project CEBE and by Estonian SF grants 8478 and 9429.

REFERENCES

[1] zamiaCAD Framework website. http://zamiacad.sf.net/.
[2] IEEE standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.
[3] IEEE standard vhdl language reference manual. *IEEE Std 1076*, 1993.
[4] Aeroflex Gaisler AB. LEON3 SPARC V8 Processor IP core, GRLIB IP library, v.1.1.0-b4108. http://www.gaisler.com/.
[5] M. Breuer and M. MacDougall. *Digital system design automation: languages, simulation & data base*. Digital system design series. Computer Science Press, 1975.
[6] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages, 1999.
[7] F. Corno, M. S. Reorda, and G. Squillero. Rt-level itc 99 benchmarks and first atpg results. pages 44–53, 2000.
[8] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4:315–344, September 1979.
[9] T. Gingold. GHDL. A VHDL compiler, 2012. http://ghdl.free.fr/.
[10] International Technology Roadmap for Semiconductors. ITRS 2010 update. http://www.itrs.net/.
[11] Mentor Graphics Inc. HDL Designer, 2012. http://www.mentor.com/.
[12] Mentor Graphics Inc. ModelSim, 2012. http://www.mentor.com/.
[13] R. L. Rudell. Multiple-valued logic minimization for pla synthesis. Technical Report UCB/ERL M86/65, EECS Department, University of California, Berkeley, 1986.
[14] F. Scarpino. *VHDL and AHDL Digital System Implementation*. Prentice Hall PTR, 1997.
[15] Sigasi nv. Sigasi, 2012. http://www.sigasi.com/.
[16] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, 3rd Edition*. McGraw-Hill Book Company, 1997.
[17] Simplifide. SimplifIDE, 2012. http://simplifide.com/.
[18] SpringSoft Inc. Verdi Automated Debug System, 2012. http://www.springsoft.com/.
[19] L. Stok, D. S. Kung, D. Brand, A. D. Drumm, A. J. Sullivan, L. N. Reddy, N. Hieter, D. J. Geiger, H. H. Chao, and P. J. Osler. Booledozer: Logic synthesis for asics. *IBM Journal of Research and Development*, 40(4):407 –430, july 1996.
[20] M. Stonebraker and L. A. Rowe. The design of postgres. *SIGMOD Rec.*, 15(2):340–355, June 1986.
[21] Sun Microsystems, Inc. Java Object Serialization Specification, Aug. 2001. Revision 1.4.4.
[22] Synopsys Inc. Design Compiler, 2012, http://www.synopsys.com/.
[23] K. Tadashi. Veditor, 2012, http://veditor.sf.net/.
[24] Verific Design Automation, Inc. Benefits of Verifics Parser Platform, May 2010. http://www.verific.com.

Research paper III

M. Jenihhin, J. Raik, A. Chepurov, R. Ubar, "PSL Assertion-Checking Using Temporally Extended High-Level Decision Diagrams". *Journal of Electronic Testing - Theory and Applications (JETTA)*, 25(6), 2009.

The author's contributions are: participation in the development of the methodology for cycle-free Temporally Extended High-Level Decision Diagrams (THLDD) synthesis from PSL properties and in creation of the Primitive Property Graphs library; implementation of the THLDD generator in ApricotCAD tool.

J Electron Test (2009) 25:289–300
DOI 10.1007/s10836-009-5116-4

PSL Assertion Checking Using Temporally Extended High-Level Decision Diagrams

Maksim Jenihhin · Jaan Raik · Anton Chepurov · Raimund Ubar

Received: 30 September 2008 / Accepted: 28 September 2009 / Published online: 13 October 2009
© Springer Science + Business Media, LLC 2009

Abstract This paper proposes a novel method for the simulation-based checking of assertions written in the PSL language. The method uses a system representation model called High-Level Decision Diagrams (HLDDs). Previous works have shown that HLDDs are an efficient model for simulation and convenient for diagnosis and debug. 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 the paper are a methodology for direct conversion of PSL properties to HLDD and modification of the HLDD-based simulator for assertion checking support. Experimental results show the feasibility and efficiency of the proposed approach.

Keywords Verification · PSL · Decision diagrams · Assertions

1 Introduction

Assertions have been found to be beneficial for solving a wide range of tasks in systems design from modeling, verification to manufacturing test [16]. In this paper we consider assertion checking which has been recognized as an efficient technique in many steps of the state-of-the-art digital systems design [13]. Assertions may be implemented both in simulation-based and formal methods. Here, we consider the first case, where assertions play the role of monitors for a particular system behavior during simulation signaling about satisfaction or violation of the property of interest [23]. PSL can also be used for defining environmental constraints or generators [23] and can serve for stimuli filtering or generation. Assertions have their application not only in verification but also in testing [16, 19]. As an example of the latter, in [19] the authors show that formal PSL assertion checking can be efficiently applied to identify untestable faults in sequential circuits.

IEEE Std 1850 Property Specification Language (PSL) [11] is a language that is commonly used for expressing assertions. PSL has been specifically designed to fulfill several general functional specification requirements for hardware, such as ease of use, concise syntax, well-defined formal semantics, expressive power and known efficient underlying algorithms in simulation-based verification. Research on the topic of converting PSL assertions to various design representations such as finite state machines and hardware description languages is gaining in popularity [2, 4, 9, 15]. Probably the most well-known commercial tool for this task is FoCs [10] by IBM. The above-mentioned solutions and the approach proposed in [14] mainly address synthesis of checkers from PSL properties that 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 approach presented in this paper allows avoiding the above limitations. The structure of an HLDD design representation

Responsible Editor: C. Metra

M. Jenihhin (✉) · J. Raik · A. Chepurov · R. Ubar
Department of Computer Engineering,
Tallinn University of Technology,
Raja 15, Tallinn 12618, Estonia
e-mail: maksim@pld.ttu.ee

J. Raik
e-mail: jaan@pld.ttu.ee

A. Chepurov
e-mail: anchep@pld.ttu.ee

R. Ubar
e-mail: raiub@pld.ttu.ee

with the temporal extension proposed in this paper allows straightforward and lossless translation of PSL properties.

In this paper, we present an approach to checking PSL assertions using High-Level Decision Diagrams (HLDDs). Here, the assertions are translated into HLDD graphs and integrated into fast HLDD-based simulation. HLDDs are a convenient model for diagnosis and debug since they provide for easy identification of cause-effect relationships. The feasibility and efficiency of the proposed approach are demonstrated by the experimental results, where the proposed method is compared against a state-of-the-art commercial simulator with PSL assertions support from a major CAD vendor. High-Level Decision Diagrams have been proposed and further developed by the authors in [22]. For more than a decade this model of digital design representation has been successfully applied in design simulation [21] and test generation [17, 18, 20] research areas.

The paper is organized as follows. PSL and its supported subset are discussed in Section 2. Section 3 defines the existing HLDD graph model and describes the HLDD-based simulation process. In Section 4, HLDD monitor generation from VHDL checkers generated by FoCs is explained. Section 5 presents the temporal extension for HLDD model, discusses the hierarchical creation of THLDD representation of PSL properties and the THLDD assertion checking process. Section 6 provides the experimental results, and finally conclusions are drawn.

2 PSL for Assertion Checking

The popularity of assertion-based verification has encouraged a co-operative development of the *Property Specification Language* by the Functional Verification Technical Committee of Accellera. After a process in which donations from a number of sources were evaluated, the Sugar language from IBM was chosen as the basis for PSL. The Language Reference Manual for PSL version 1.1 was released in 2004 [1]. The language became an IEEE 1850 Standard in 2005 [11].

Let us consider an example of a PSL property *reqack* shown in Fig. 1. Its possible timing diagrams are illustrated in Fig. 2. Figure 2a shows that *ack* becomes high after *req* being high resulting in a pass of the property *reqack* on the given simulation trace. A system behavior that activates the property, however, obviously violating it is demonstrated in Fig. 2b. Figure 2c shows the case when the precondition never occurs. In other words, the property *passes vacuously*. A designer is normally not interested in vacuous passes and therefore we say that property *reqack* was not activated in the simulation trace presented in Fig. 2c.

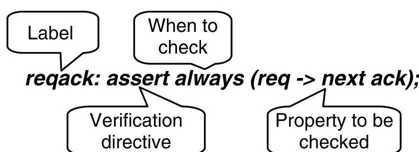


Fig. 1 PSL property *reqack*

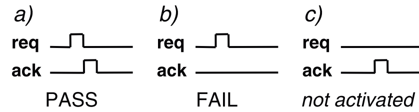


Fig. 2 Timing diagrams for the property *reqack*

For the sake of wider adoption among the design community, PSL is a multi-flavored language, which means that it supports common constructs of VHDL, Verilog, IBM's GDL, SystemVerilog and SystemC [8]. PSL is also a multi-layered language [1]. The layers include:

- *Boolean layer* — the lowest one, consists of Boolean expressions in HDL (e.g. *a &&(b || c)*)
- *Temporal layer* — sequences of Boolean expressions over multiple clock cycles, also supports Sequential Extended Regular Expressions (SERE) (e.g. *{A[*3];B} |-> {C}*)
- *Verification layer* — it provides directives that tell a verification tool what to do with the specified sequences and properties.
- *Modeling layer* — additional helper code to model auxiliary combinational signals, state machines etc. that are not part of the actual design but are required to express the property.

The temporal layer of PSL language has two constituents:

- *Foundation Language* (FL) that is Linear time Temporal Logic (LTL) with embedded SERE
- *Optional Branching Extension* (OBE) that is Computational Tree Logic (CTL)

The latter considers multiple execution paths and models design behavior as execution trees. CTL can only be used in formal verification. Therefore, in this paper we will consider only the FL part of PSL. In fact, only FL, or more precisely its subset known as PSL Simple Subset, is suitable for dynamic assertion checking. This subset is explicitly defined in [1] and loosely speaking it has two requirements for time: to advance monotonically and to be finite, which leads to restrictions on types of operands for several operators. In this paper only several LTL operators without SERE support were implemented. However, as it will be shown later, the support for SERE as well as for any other language constructs can be easily added by an appropriate library extension.

3 High-Level Decision Diagrams

Decision Diagrams (DD) have been used in verification for about two decades. Reduced Ordered Binary Decision Diagrams (BDD) [3] as canonical forms of Boolean functions have their application in equivalence checking and in symbolic model checking. Additionally, a higher abstraction level DD representation, called Assignment Decision Diagrams (ADD) [5], have been successfully applied to both register-transfer level (RTL) verification and test. In this paper we consider a different decision diagram representation, High-Level Decision Diagrams (HLDDs) that unlike ADDs can be viewed as a generalization of BDD. HLDDs can be used for representing different abstraction levels from RTL to TLM (Transaction Level Modeling) and behavioral. HLDDs have proven to be an efficient model for simulation and diagnosis since they provide for a fast evaluation by graph traversal and for easy identification of cause-effect relationships [21].

3.1 HLDD Model Definition

High-Level Decision Diagrams (HLDD) are graph representations of discrete functions. 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.

Definition 1 A HLDD representing a discrete function $y=f(x)$ is a directed non-cyclic labeled graph that can be defined as a quadruple $G_y=(M,E,Z,\Gamma)$, 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 labeling the nodes*, and Γ is a function on E. The function $Z(m_i)$ returns the variable x_k, which is labeling node m_i. Each node of a HLDD is labeled by a variable. In special cases, nodes can be labeled by constants or algebraic expressions. An edge $e\in E$ of a HLDD is an ordered pair $e=(m_1,m_2)\in E^2$, where E^2 is the set of all the possible ordered pairs in set E. 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 $Pm_i = \{\Gamma(e) | e = (m_i, m_j) \in E\}$ is a partition of the set X_k. Figure 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* $M^{term} \in M$.

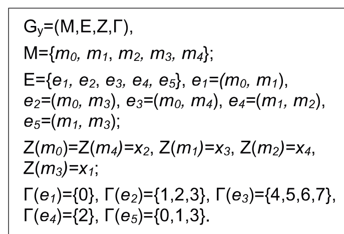
HLDD models can be used for representing digital systems. In such models, the non-terminal nodes correspond to conditions or to control signals, and the terminal nodes represent data operations (functional units). Register transfers and constant assignments are treated as special cases of operations. When representing systems by decision diagram models, in general case, a network of HLDDs rather than a single HLDD is required. During the simulation in HLDD systems, the values of some variables labeling the nodes of a HLDD are calculated by other HLDDs of the system.

Figure 4b presents the HLDD system for the RTL pseudocode shown in Fig. 4a implementing the Greatest Common Divisor (GCD) algorithm. In the Fig. 4a, T and F stand for true and false, respectively. The ε character denotes default edges.

Different from the well-known Reduce Ordered BDD models which have worst-case exponential space requirements, HLDD size scales well with respect to the size of the RTL code. Let n be the number of processes in the RTL code, v be the average number of variables/signals inside a process and c be the average number of conditional branches in a process. In the worst case the number of nodes in the HLDD model will be equal to $n \cdot v \cdot c$. Note, that the complexity of HLDDs is just $O(n)$ with respect to the number of processes in the code.

3.2 Simulation Using 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 $m_i \notin M^{term}$



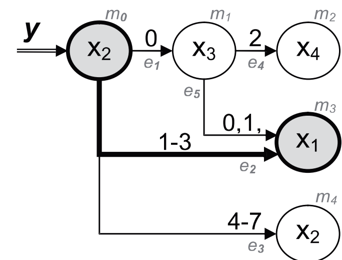


Fig. 3 A high-level decision diagram representing a function $y=f(x_1,x_2,x_3,x_4)$

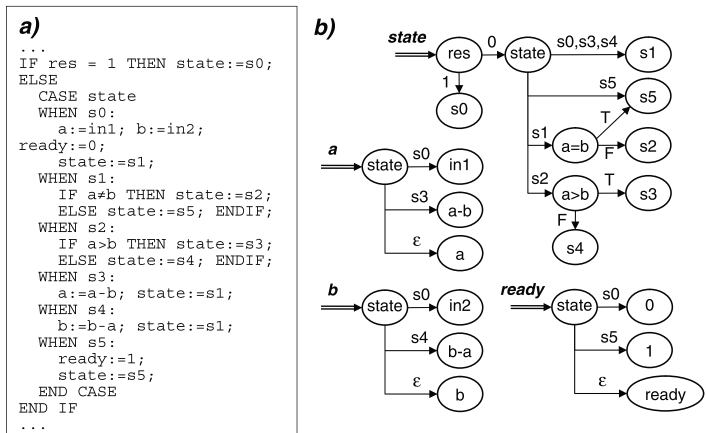


Fig. 4 **a** An RTL pseudocode and b) its HLDD

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 always exists 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 labeling the terminal node of the main activated path.

Let us explain the HLDD simulation process on the decision diagram example presented in Fig. 3. 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 labeled 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 color in Fig. 3).

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

In this work, we apply HLDDs as a graph representation of RTL. There exist other word-level decision diagrams such as multi-terminal DDs (MTDDs) [6], K*BMDs [7] and ADDs [5]. However, in MTDDs the non-terminal 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 described in this paper is not canonicity but ease of simulation. The principal difference between HLDDs and ADDs lies in the fact that ADDs edges are not labeled by activating values. Whereas in HLDDs, the selection of a node activates a path through the diagram, which derives the needed value assignments for variables.

3.3 Advantages of HLDD-Based Modeling

As an example, consider a datapath of a Design Under Verification (DUV) depicted in Fig. 5a and its corresponding HLDD representation shown in Fig. 5b. 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 an input bus, SEL_1, SEL_2, SEL_3 and EN_2 serve as input control variables, and *a, b, c, d* and *e* denote internal buses, respectively. In the HLDD, the control variables SEL_1, SEL_2, SEL_3 and EN_2 are labeling internal decision nodes of the HLDD with their values shown at edges. The terminal nodes are labeled 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 the data manipulation operations of the network. By bold lines and grey nodes, a full activated path in the HLDD is shown from $Z(m_0)=EN_2$ to $Z(m^T \in M^T)=R_1*R_2$, which corresponds to the pattern $EN_2=2$, $SEL_3=3$, and $SEL_2=0$. The activated part of the network at this pattern is denoted by grey boxes.

The main advantage and motivation of using HLDDs compared to the netlists of primitive functions is the increased efficiency of simulation and diagnostic modeling because of direct and compact representation of cause-effect relationships. For example, instead of simulating the control word $SEL_1=0$, $SEL_2=0$, $SEL_3=3$, $EN_2=2$ 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, SEL_3 and SEL_2 on the HLDD and compute a single operation

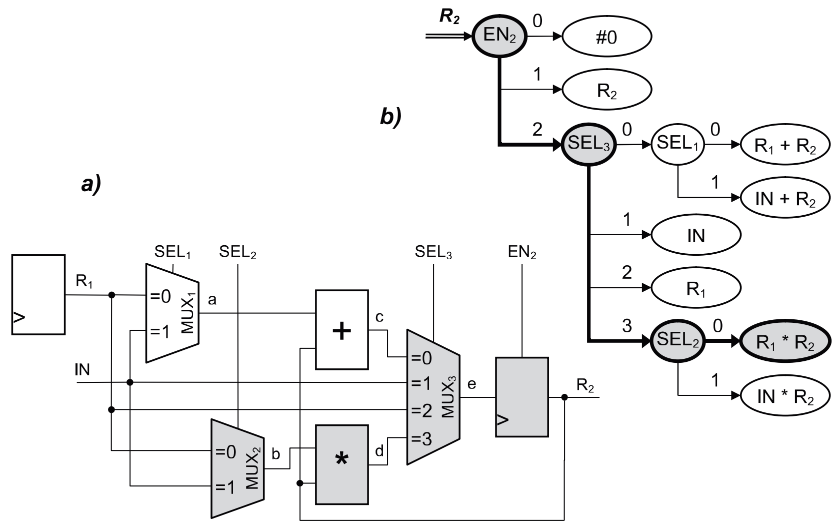


Fig. 5 A datapath of a DUV **a** schematic and **b** HLDD-based representations

$R_2=R_1 * R_2$. In case of detecting an error in R_2 the possible causes can be defined immediately along the simulated path through EN_2, SEL_3 and *y* SEL_2 without any diagnostic analysis inside the corresponding RTL netlist. The activated path provides fault candidates. The further reasoning should be based on analyzing sources of these signals. 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 RT-level region, by exactly locating the cause of the error in this region.

4 HLDD Generation from HDL Checkers

In [14] PSL properties were translated into HLDDs through the VHDL language implying the generation of checkers by IBM's FoCs [10] as an intermediate step. However this experience has revealed particular limitations and inefficiency for HLDD-based assertions creation, which will be discussed below. Moreover, checkers synthesis from PSL properties is mainly efficient 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 the 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 following example. The object for this example is a SERE style PSL assertion *fe_seq*:

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

The precondition of *fe_seq* assertion is the sequence of system behavior when signal *a* is set to '1', followed by a don't-care sequence two 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). Figure 6a 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.

A possible system of HLDD graphs representing the checker is provided in Fig. 7. In the figure we used a notation where trailing quote character after diagram variable denotes one clock cycle delay.

As it can be seen from the Fig. 7, the FoCs-based approach results 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 four cycles time distance and therefore caused four intermediate variables a_1–a_4. The following Section introduces a new kind of HLDD model permitting direct representation of PSL constructs for simulation-based assertion checking.

5 Temporally Extended HLDDs

The topic of this section is an extension to the traditional HLDD model defined in Section 3 with the aim to support temporal logic properties. The extension is referred to as Temporally extended High-Level Decision Diagrams (THLDDs). We present the definition of THLDDs and

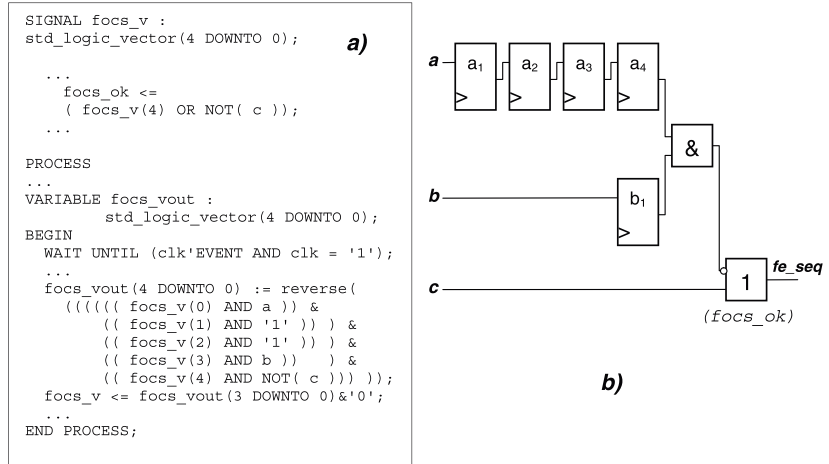


Fig. 6 VHDL checker generated for assertion *fe_seq* by FoCs. **a** code and **b** schematics

propose an algorithm for the hierarchical generation of the model based on a concept of templates for PSL constructs referred to as Primitive Property Graphs (PPGs).

5.1 Basic Definitions

Unlike the traditional HLDD, 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 $x=(x_1, ..., x_m)$ is a vector defined on a finite domain $X=X_1 \times ... \times X_m$ and $T=\{t_1, ..., t_s\}$ 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 labeled graph that can be defined as a sixtuple $G_P=(M,E,T,Z,\Gamma,\Phi)$, 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 labeling the nodes and their domains, 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 labeling variables.

The graph G_P has exactly three terminal nodes $M^{term} \in$ M labeled 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;
- *CHK.* (from CHECKING) — the assertion P has been simulated and it does not fail, nor does it pass non-vacuously. (See Section 5.2 for explanation of vacuity).

The function $\Phi(m_i,t)$ represents the relationship indicating at which time-steps $t \in T$ the node labeling 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 $x_l^{\Delta t_j} \wedge ... \wedge x_l^{\Delta t_k}$ is true, i.e. variable x_l is true at every time-step between t_{curr+j} and t_{curr+k}.

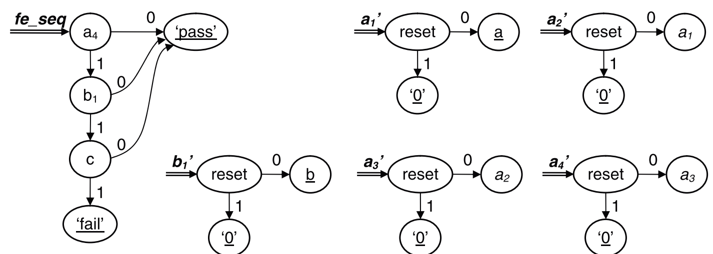


Fig. 7 HLDD representation of the VHDL checker from Fig. 6

- $\Delta t = \exists\{j,...,k\}$, meaning that $x_l^{\Delta t_j} \vee \ldots \vee x_l^{\Delta t_k}$ 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 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 THLDD temporal relationship functions listed above $\Phi(m_i,t)$, which create the variations listed below. For $x_l^{\Delta t}$, where x_l and x_c are node labeling 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 $\Delta t = \forall\{0,..., event(x_c)\text{-}1\}$.
- $\Delta t = \exists\{0,...,event(x_c)\}$, 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 $\Delta t = \exists\{0,...,event(x_c)\text{-}1\}$.
- $\Delta t = event(x_c)$, which means that variable x_l has to be true at the first time-step when x_c becomes true. This is equivalent to the PSL expression $next_event(x_c)x_l$.

For Boolean, i.e. non-temporal variables Δt=0.

Table 1 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 their *derivatives.*

In addition, we introduce 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 5.3 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 THLDD is an extension of HLDD defined in Section 3 as it includes temporal relationships functions. The main purpose of the proposed temporal extension is transferring additional information and directives to the HLDD simulator that will be used for assertions checking.

5.2 Generating THLDDs Using Primitive Property Graphs

The idea of the proposed method relies on the principle of 'divide and conquer'. The method is based on partitioning PSL properties into elementary entities containing one operator only. 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 recursive *hierarchical construction* of the Temporally extended HLDD (THLDD) for a complex property using the PPG Library elements.

Prior to the THLDD 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 FL LTL-style operators. However, by means of the supported operators a large set of properties expressed in PSL can be derived.

Table 1 Temporal relationships in THLDDs

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

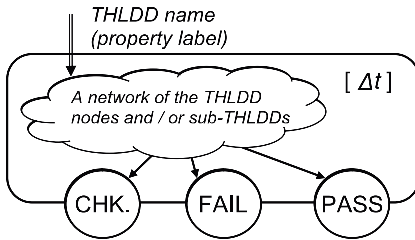


Fig. 8 Standard THLDD interface

A Primitive Property Graph is always a THLDD graph. That means it uses a HLDD model with the temporal extension proposed above and has a standard interface.

The *standard THLDD interface* for all PPGs was introduced in order to support the hierarchy in the recursive construction of a complex property that is described in the next subsection. A PPG has one root node and exactly 3 terminal nodes (CHK., FAIL and PASS), as opposed to an arbitrary number of terminal nodes in a usual HLDD graph. It has also an optional relative time range Δt, which shows when the assertion has to be checked. The standard THLDD interface is shown in Fig. 8.

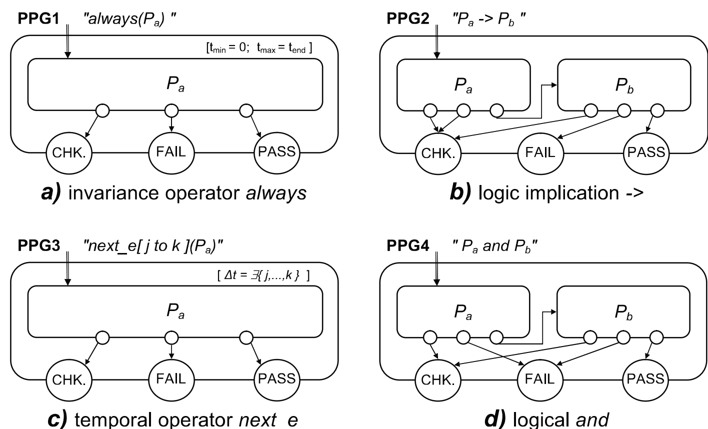
Example PPGs created for 4 PSL operators are shown in Fig. 9. Here the value for a PPG can be evaluated to one of the 3 terminal nodes based on the evaluation of the THLDD sub-graphs (e.g. P_a, P_b) and the intraconnections. The THLDD sub-graphs may also be PPGs as well as Boolean expressions or variables. Complex PSL properties consisting of several operators are represented by a number of PPGs that compose a final complex THLDD graph, as it is explained further in this section. Note, that the logic implication operator '->' in Fig. 9b exits to the terminal node 'CHK.' when the precondition P_a fails (as opposed to the logical *and* PPG in Fig. 9d). This is due to the fact that in assertion checking a verification engineer is not usually interested in vacuous passes of the property.

A *Vacuous pass* occurs if a passing property contains a Boolean expression that, in frames of the given simulation trace, has no effect on the property evaluation. The property has passed not because of meeting all the specified behaviour but only because of the non-fulfilment of logical implication activation conditions. The decision whether to treat vacuous passes as actual satisfactions of properties or not depends on the particular verification tool. The approaches presented in this paper separate vacuous passes from normal passes of a property.

The terminal node 'CHK.' 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. An assertion checking algorithm that is capable of handling such functions is presented in the following subsection.

Complex THLDD properties are hierarchically constructed from elementary graphs of the PPG Library in the following way. At first, the source PSL property is parsed. During the parsing phase the PSL property is partitioned into entities containing one operator only. The hierarchy of operators is determined by the PSL operators precedence specified by the IEEE1850 Standard.

Fig. 9 PPGs for a set of PSL operators



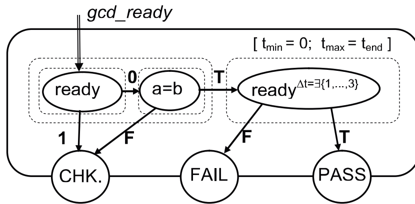


Fig. 10 A THLDD representation of PSL property *gcd_ready*

The construction of complex properties is performed in the top–down manner. The process 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.

Let us consider a sample PSL property *gcd_ready* for the GCD implementation given in Fig. 3.

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

The resulting THLDD graph describing this property is shown in Fig. 10.

The construction of the property *gcd_ready* implies usage of the four PPGs shown in Fig. 9 and a PPG for the logical *not* operator. The nodes in the final THLDD contain pure variables and an HDL expression *(a=b)*.

5.3 HLDD-Based Assertion Checking

The process of HLDD-based assertion checking implies the existing HLDD simulator functionality and its extension for assertion checking presented by Algorithm 1 in Fig. 11. The execution of Algorithm 1 is preceded by simulation of the design, which calculates the *simulation trace*. This trace is a starting point for assertion checking. The latter one takes into account temporal relationships information at the THLD nodes that represent an assertion.

Figure 12 shows an example of time windows for a THLDD graph converted from a two-operator PSL assertion *two_win*:

$$two_win : assert\ always\ (next_a(j\ to\ k)(x))$$

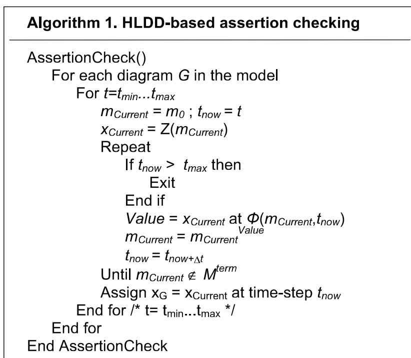


Fig. 11 Algorithm 1. HLDD-based assertion checking

The assertion *two_win* states that *x* should hold between the jth and k^{th} time-step starting from every time-step in the simulation trace.

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 (caused by invariance operators) and several dynamic time windows that can overlap.

A general flow of the HLDD-based assertion checking process is given in Fig. 13. The input data for the first step (simulation) are a HLDD model representation of the design under verification and stimuli. This step results in a 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 information about both the assertions coverage and their validity.

The stored assertion validity data allows further analysis and reasoning of which combinations of stimuli and design states have caused fails and passes of the assertions. This data also implicitly contains information about the monitored assertions coverage (i.e. assertion activity: "*active*" or "*inactive*") by the given stimuli.

The following section shows the feasibility and efficiency of the proposed HLDD-based assertion checking flow.

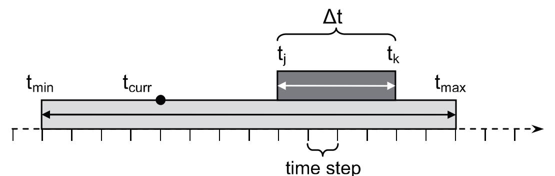


Fig. 12 THLDD time windows in assertion checking

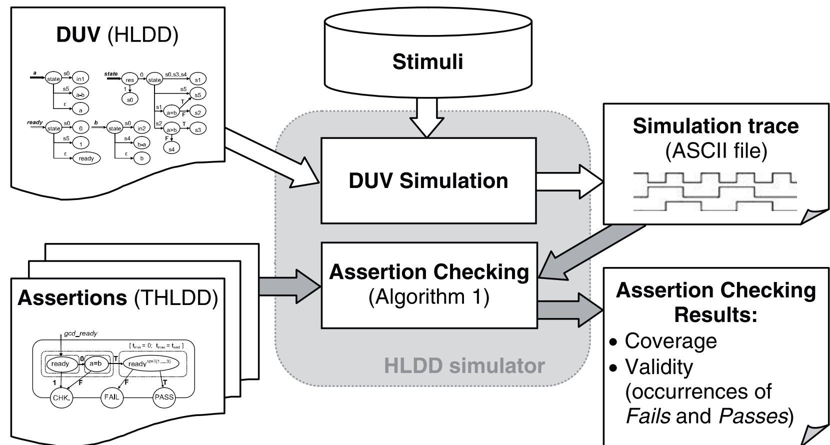


Fig. 13 HLDD-based assertion checking process flow

6 Experimental Results

This section provides experimental results of assertion checking execution times comparison between the THLDD simulator and a commercial tool, which is a state-of-the-art simulator with assertion checking support from a major CAD vendor. The experimental benchmarks are the GCD implementation given in Fig. 3 and three designs from the ITC'99 [12] benchmarks family. The characteristics of the benchmarks are provided in Table 2.

In order to evaluate the feasibility of the proposed THLDD-based assertion-checking method a set of 5 realistic assertions has been created for each benchmark. The assertions were selected on the following basis:

– different types of operators should be included (e.g. Boolean operators, implication, temporal operators, 'until');
– Different outcomes should result (fail, pass, both);
– The failure/pass frequency should vary (frequent, infrequent)

The assertions selected for GCD are the following:

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

The assertions used for the b00, b04, b09 benchmarks had the same temporal complexity as the ones listed for the GCD design. Each assertion has been checking 2–5 signals and besides an invariance operator (*always* or *never*) contained 1–3 LTL temporal operators from Table 1.

Table 2 Benchmark characteristics

Design	Characteristic, number				
	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

Table 3 Execution time comparison

Design	Checking time, s/10^6 stimuli			
	The proposed approach			Commercial tool
	Simulation	Checking	Total	Total
gcd	2.07	4.87	6.94	13.52
b00	3.43	2.95	6.38	13.84
b04	5.47	3.61	9.08	19.23
b09	2.21	4.55	6.76	12.4

SERE have not been used as they are not currently supported. Both simulators were supplied with the same sequences of realistic stimuli providing a good coverage for the assertions.

Table 3 shows assertion checking execution times comparison between our THLDD simulator and the commercial tool. The execution time values in the table are presented in seconds for 10^6 clock cycles of stimuli. The second and third columns show the simulation and assertion checking (Algorithm 1) execution times required for the THLDD simulator. The fourth (highlighted) and the fifth columns are the total execution time taken by the proposed approach and the commercial tool correspondingly. Both tools have shown the identical responses about the assertion satisfactions and violations occurrences.

The conversion of the benchmarks representation from VHDL to HLDD has taken from 219 ms to 260 ms and conversion of the set of 5 assertions for each of the benchmarks from 14 ms to 19 ms, respectively. Please note, that these conversions should be performed only once for each set of DUV and assertions and they are comparable to the commercial CAD tools VHDL compilation times.

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 the state-of-the-art commercial tool.

7 Conclusion

This paper proposed a novel method for checking Property Specification Language (PSL) assertions using a new model representation called Temporally extended High-Level Decision Diagrams (THLDDs). Previous works have shown that HLDDs are an efficient model for simulation and diagnosis since they provide for a fast evaluation by graph traversal and for easy identification of cause-effect relationships. In this paper, the model was extended to support temporal operations inherent in PSL properties and also to directly support assertion checking. We presented a hierarchical approach to generate THLDDs based on a library of Primitive Property Graphs (PPG). Basic algorithms for THLDD based assertion checking were discussed.

As a future work we see the integration of THLDD-based assertion checking methods to design error diagnosis and debug solutions.

Acknowledgments The work has been supported by European Commission Framework Program projects FP6 VERTIGO and FP7 CREDES, by European Union through the European Regional Development Fund, by Estonian Science Foundation grants 7068 and 7483, Enterprise Estonia funded ELIKO Development Center and Estonian Information Technology Foundation (EITSA).

References

1. Accellera, Property specification language reference manual, v1.1, June 9, 2004
2. Boulé M, Zilic Z (2006) Efficient automata-based assertion-checker synthesis of PSL properties. Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT'06)
3. Bryant R (1986) Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers C-35(8):677–691
4. Bustan D, Fisman D, Havlicek J. Automata construction for PSL. The Weizmann Institute of Science, Technical Report MCS05-04
5. Chayakul V, Gajski DD, Ramachandran L (1993) High-level transformations for minimizing syntactic variances. Proc. of the ACM/IEEE Design Automation Conference (DAC), pp. 413–418, June 1993
6. Clarke E, Fujita M, McGeer P, McMillan KL, Yang J, Zhao X (1993) Multi terminal BDDs: an efficient data structure for matrix representation. Proc. of the International Workshop on Logic Synthesis, pp. P6a:1–15
7. Drechsler R, Becker B, Ruppertz S (1996) K*BMDs: a new data structure for verification. Proc. of the European Design & Test Conference, pp. 2–8
8. Eisner C, Fisman D (2006) A practical introduction to PSL. Springer Science
9. Gheorghita S, Grigore R (2005) Constructing checkers from PSL properties. 15th International Conference on Control Systems and Computer Science (CSCS15), 2:757–762
10. IBM AlphaWorks (2007) FoCs Property Checkers Generator ver. 2.04, [www.alphaworks.ibm.com/tech/FoCs]
11. IEEE-Commission, IEEE standard for Property Specification Language (PSL), (2005) IEEE Std 1850–2005
12. ITC99 Benchmark Home Page. URL: http://www.cerc.utexas.edu/itc99-benchmarks/bench.html
13. ITRS Roadmap. URL: http://www.itrs.net
14. Jenihhin M, Raik J, Chepurov A, Ubar R (2007) Assertion checking with PSL and high-level decision diagrams. IEEE Workshop on RTL and High Level Testing (WRTLT'07), October 12–13, 2007
15. Morin-Allory K, Borrione D (2006) Proven correct monitors from PSL specifications, Proceedings of the IEEE/ACM Design, Automation & Test in Europe (DATE)
16. Oddos Y, Morin-Allory K, Borrione D (2007) Prototyping generators for on-line test vector generation based on PSL properties. Proceedings of the 10th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2007), pp 383–388
17. Raik J, Ubar R (2000) Fast test generation for sequential circuits using decision diagrams representations. Journal of Electronic Testing: Theory and Applications, Kluwer
18. Raik J, Nõmmeots T, Ubar R (2005) A new testability calculation method to guide RTL test generation. Journal of Electronic Testing: Theory and Applications 21(1):71–82 Springer Science
19. Raik J, Fujiwara H, Ubar R, Krivenko A (2008) Untestable fault identification in sequential circuits using model-checking, IEEE Asian Test Symposium
20. Raik J, Ubar R, Viilukas T, Jenihhin M (2008) Mixed hierarchical-functional fault models for targeting sequential cores. Elsevier Journal of Systems Architecture 54(3–4):465–477 Elsevier
21. Ubar R, Raik J, Morawiec A (2000) Back-tracing and event-driven techniques in high-level simulation with decision diagrams. The IEEE International Symposium on Circuits and Systems (ISCAS 2000) 1:208–211
22. Ubar R (1996) Test synthesis with alternative graphs. In IEEE Design and Test of Computers, pp 48–57
23. Yuan J, Pixley C, Aziz A (2006) Constraint-based verification. Springer

Maksim Jenihhin received his M.Sc. and Ph.D. degrees in Computer Engineering from Tallinn University of Technology in 2004 and in 2008, respectively. 2004–2007 he was employed as a researcher in ELIKO Technology Development Center, Tallinn. Currently he is employed as a researcher at the Department of Computer Engineering of Tallinn University of Technology. He has co-authored more than 25 scientific papers. His research interests include design manufacturing testing and functional verification.

Jaan Raik received his M.Sc. and Ph.D. degrees in Computer Engineering from Tallinn University of Technology in 1997 and in 2001, respectively, where he currently holds a position of a senior researcher. He is a member of IEEE Computer Society, a member of program committees for several top conferences and has co-authored more than 100 scientific publications. In 2004, he was awarded the national Young Scientist Award. His main research interests include high-level test generation and verification.

Anton Chepurov received his B.S. and M.Sc. degrees in Computer Engineering from Tallinn University of Technology (TUT) in 2006 and in 2008, respectively. Currently he is a Ph.D. student at TUT. His research interests include design manufacturing testing and verification.

Raimund Ubar received his Ph.D. degree in 1971 at the Bauman Technical University in Moscow. He is a professor of Computer Engineering at Tallinn University of Technology. His research interests include computer science, electronics design, design verification, test generation, fault simulation, design-for-testability, fault-tolerance. He has published more than 200 papers and two books. R.Ubar has given seminars or lectures in 25 universities in more than 10 countries. In 1993-1996 he was the Chairman of the Estonian Science Foundation and a member of the Estonian Science Council. He is a Golden Core Member of the IEEE, a member of ACM, SIGDA, Gesellschaft der Informatik (Information Society, Germany), European Test Technology Technical Committee and Estonian Academy of Sciences.

Research paper IV

M. Jenihhin, R. Raik, A. Chepurov, U. Reinsalu, R. Ubar, “High-Level Decision Diagrams based Coverage Metrics for Verification and Test”. *Proceedings of 10th IEEE Latin American Test Workshop (LATW’09)*, 2009.

The author’s contributions are: investigation into different options for conditional statements representation in High-Level Decision Diagrams (HLDD); evaluation of different compactness levels of HLDD model for finding the optimal level for the stringent coverage measurement; implementation of a flexible HLDD interface that allows obtaining different HLDD representations from VHDL descriptions.

High-Level Decision Diagrams based Coverage Metrics for Verification and Test

Maksim Jenihhin, Jaan Raik, Anton Chepurov, Uljana Reinsalu, Raimund Ubar
Department of Computer Engineering, Tallinn University of Technology
E-mail: { maksim|jaan|anchep|uljana|raiub }@pld.ttu.ee

Abstract

The paper proposes High-Level Decision Diagrams (HLDDs) model based structural coverage metrics that are applicable to, both, verification and high-level test. Previous works have shown that HLDDs are an efficient model for simulation and test generation. However, the coverage properties of HLDDs against Hardware Description Languages (HDL) have not been studied in detail before. In this paper we show that the proposed methodology allows more stringent structural coverage analysis than traditional VHDL code coverage. Furthermore, the main new contribution of the paper is a hierarchical approach for condition coverage metric analysis that is based on HLDDs with expansion graphs for conditional nodes. Experiments on ITC99 benchmarks show that up to 14% increase in coverage accuracy can be achieved by the proposed methodology.

1. Introduction

Structural coverage metric (also referred to as code coverage) has its application, both, in functional verification and high-level test. Due to the fact that it is impractical to verify exhaustively all possible inputs and states of a design, the confidence level regarding the quality of the design must be quantified to control the testing effort. The fundamental question is: *when is the design simulated enough*? Structural coverage is a measure of confidence and it is expressed as a percentage of items covered out of all possible items. Different definitions of items give rise to different coverage measures, or coverage metrics.

Over the years, a large variety of code coverage metrics have been proposed, including statement coverage, block coverage, path coverage, branch coverage, expression coverage, transition coverage, sequence coverage, toggle coverage etc [1],[10]. The *statement coverage* metric measures the ratio of code instructions exercised from the entire set of instructions by the program stimuli. *Toggle coverage* shows whether and how much the signals of the design toggle, i.e. how many bits change their state from 0 to 1 or vice versa. In the case of *branch coverage*, we measure the ratio of branches in the control flow graph of the code taken under the set of program stimuli. *Condition coverage* metric ([1],[6]) reports the number of Boolean sub-expressions, separated by logical operators, calculated in conditional statements causing their evaluation to one of the decisions (e.g. 'true' or 'false' values). It differs from the branch coverage, by the fact that in the latter only the final decision determining the branch is taken into account.

Structural coverage has a long history in software testing and only with the emergence of hardware description languages it has been applied to hardware verification and test. Several standards for system quality test such as DO-178B [7] for software and its analog for hardware DO-254 [8] used in airborne systems state that condition coverage along with statement and branch coverages has to be applied in the cases where system failures would cause catastrophic results.

In this paper we introduce a methodology for structural coverage analysis based on HLDD graph model proposed and developed in Tallinn University of Technology. Experiments on ITC99 benchmarks show that up to 14 % increase in coverage accuracy can be achieved by the proposed methodology compared to traditional HDL code coverage. The main new contribution of the paper is a hierarchical approach for condition coverage metric analysis that is based on HLDDs with expansion graphs for conditional nodes. The proposed methodology has significantly lower computational overhead compared to HDL-based approaches and relies on homogeneous verification flow (i.e. one model and one tool).

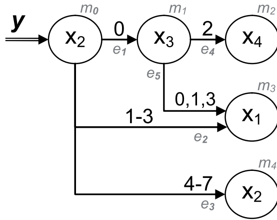
The rest of the paper is organized as follows. Section 2 introduces HLDD graph model and its simulation algorithm. Section 3 defines different representation types for HLDDs based on respective rules. HLDD-based structural coverage analysis implying a novel approach for condition coverage metric analysis is proposed in Section 4. Section 5 demonstrates advantages behind the proposed methodology by an example study and experimental results on ITC99 benchmarks.

978-1-4244-4206-5/09/$25.00 ©2009 IEEE

2. High-Level Decision Diagrams

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



$G_y=(M,E,Z,\Gamma)$,

$M=\{m_0, m_1, m_2, m_3, m_4\}$;

$E=\{e_1, e_2, e_3, e_4, e_5\}$, $e_1=(m_0, m_1)$, $e_2=(m_0, m_3)$, $e_3=(m_0, m_4)$, $e_4=(m_1, m_2)$, $e_5=(m_1, m_3)$;

$Z(m_0)=Z(m_4)=x_2$, $Z(m_1)=x_3$, $Z(m_2)=x_4$, $Z(m_3)=x_1$;

$\Gamma(e_1)=\{0\}$, $\Gamma(e_2)=\{1,2,3\}$, $\Gamma(e_3)=\{4,5,6,7\}$, $\Gamma(e_4)=\{2\}$, $\Gamma(e_5)=\{0,1,3\}$.

Fig.1. A high-level decision diagram representing a function *y= f(x1,x2,x3,x4)*

Definition 1**:** A high-level decision diagram is a directed non-cyclic labeled 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 labeling the nodes*, and Γ is a function on E.

The function $Z(m_i)$ returns the variable x_k, which is labeling node m_i. Each node of a HLDD is labeled by a variable. In special cases, nodes can be labeled by constants or algebraic expressions. An edge $e \in E$ of a HLDD is an ordered pair $e=(m_{pc},m_{sc}) \in E^2$, 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.

Fig. 1 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 Fig. 1). 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 labeling the nodes of a HLDD are calculated by other HLDDs of the system.

2.2 Systems simulation using HLDDs

In our earlier works [4], we have implemented an algorithm supporting, both, Register-Transfer Level (RTL) and behavioral design abstraction levels. *Algorithm 1* (Fig. 2) presents the HLDD based simulation engine for RTL, behavioral and mixed HDL description styles.

```
SimulateHLDD()
  For each diagram G in the model
    mCurrent = m0
    Let xCurrent be the variable labeling mCurrent
    While mCurrent is not a terminal node
        If  xCurrent is clocked or its DD is ranked after G then
            Value = previous time-step value of xCurrent
        Else
            Value = present time-step value of xCurrent
        End if
        For {Γ | Value ∈ Γ(eactive), eactive =( mCurrent, mNext)}
            mCurrent = mNext
        End if
      End while
      Assign xG = xCurrent
  End for
End SimulateHLDD
```

Fig. 2. Algorithm 1. RTL/behavioural simulation on HLDDs

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

3. Representation types of HLDD model

The methodology for HLDD-based coverage analysis proposed in Section 5 implies manipulation techniques on HLDD model aimed to different aspects of the proposed analysis.

In the proposed methodology we 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 presented in [5] and can be generalized as follows:

- *HLDD reduction rule1*: Eliminate all the redundant nodes whose all edges point to an equivalent 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. This type of representation includes a lot of redundancy. They introduce large space requirements and relatively slow simulation times.
- *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. This type of representation combines the advantages of two full-tree representation and minimized representation. The HLDDs of this type are reasonably compact and they allow more stringent coverage measurement than the minimized representation. Furthermore, the average HLDD path length, and therefore the simulation speed, is exactly equal to the more compact minimized representation!
- *Minimized HLDD* is obtained by application of both HLDD reduction rules 1 and 2 to the full tree representation. This representation is the most compact of the three. However, the minimization step may cause loss of coverage measurement accuracy.

Let us consider an example typical design *latw09_ex1*. The functional segment of its RTL VHDL representation is provided in Fig. 3. The parts of the variables' names have the following notations {*V*- an output variable; *cS* - a conditional statement; *D*- a decision; *T*- a terminal node; *C*- a condition; *W*- a value}. The emphasized by bold keywords determine if a line has a statement, a branch or conditions. Figures 4, 5 and 6 present a reduced, minimized, full-tree and HLDD model representations correspondingly.

Stm	Dcn	VHDL code
		...
1		**if** (cS1_C1 **and** cS1_C2)
	1	**then**
2		V1 **<=** V1_T1;
	2	**else**
3		V1 **<=** V1_T2;
		end if;
4		**case** cS2_C is
	3	**when** cS2_C_W1 =>
5		V2 **<=** V2_T1;
	4	**when** cS2_C_W2 =>
6		V2 **<=** V2_T2;
	5	**when** cS2_C_W3 =>
7		V1 **<=** V1_T2;
8		**if**(cS3_C1 **and** ((**not** cS3_C2) **or** cS3_C3))
	6	**then**
9		V2 **<=** V2_T2;
	7	**else**
10		V2 **<=** V2_T3;
		end if;
		end case;
		...

Fig. 3. A segment of the VHDL code of *latw09_ex1* design

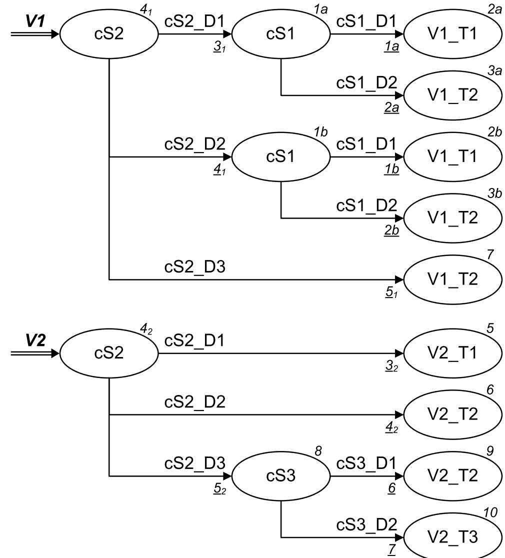


Fig. 4. Reduced HLDD for *latw09_ex1*

4. HLDD-based coverage analysis

4.1 Statement coverage mapping

The initial idea of statement coverage representation on HLDDs was proposed in [2]. This paper demonstrates the details. The statement coverage metric has a straightforward mapping to HLDD-based coverage. It maps directly to the ratio of nodes $m_{Current}$ traversed during the HLDD simulation presented in *Algorithm 1* (Subsection 2.2) to the total number of the HLDD nodes in the DUV's representation. The appropriate type of HLDD representation for the analysis of, both, statement and branch coverage metrics is the *reduced* one. The variations in the analysis caused by different HLDD representation types are discussed in Subsection 5.1.

Please consider the VHDL description of the *latw09_ex1* design provided in Fig. 3. The numbers from the first column (*Stm*) correspond to the lines with 10 statements (both conditional and assignment ones). The 14 HLDD nodes of the two graphs in Fig. 4 correspond to these statements. Covering all nodes in a HLDD model (i.e. full *HLDD node coverage*) corresponds to covering all statements in the respective HDL. However, the opposite is not true. HLDD node coverage is slightly more stringent that HDL statement coverage. Please consider as an example VHDL statements *1*, *2* and *3* and the respective nodes in Fig. 4 *1a*, *1b*, *2a*, *2b*, and *3a*, *3b*. This impact in terms of the stringency is also discussed in Subsection 5.1. At the same time some of the HDL statements have duplicated representation by the HLDD nodes (with subscript indexes) due to the fact that in HLDD-based design representation the diagrams are

normally generated for each data variable separately. As an example please consider VHDL statement *4* and HLDD nodes 4_1, 4_2 in Fig.4.

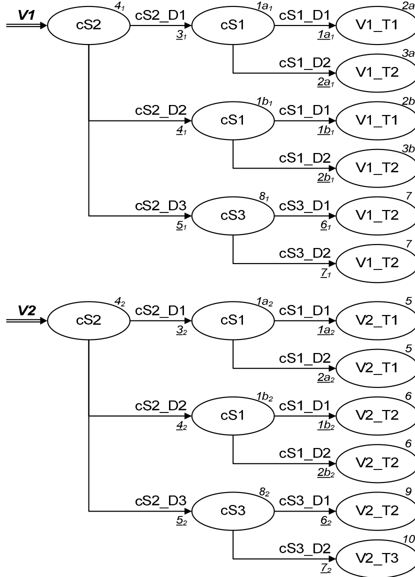


Fig. 5. Full tree HLDD for *latw09_ex1*

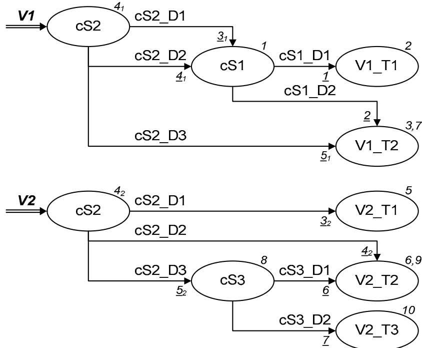


Fig. 6. Minimized HLDD for *latw09_ex1*

4.2 Branch coverage mapping

Similar to the statement coverage, branch coverage also has very clear representation in HLDD model. (Also initially proposed in [2], the details are demonstrated in this paper). It is the ratio of every edge e_{active} activated in the simulation process presented by *Algorithm 1* (Subsection 2.2) to the total number of edges in the corresponding HLDD representation of the DUV.

The numbers (underlined) from the second column (*Dcn*) in Fig. 3 correspond to the lines with 7 branches (i.e. decisions). The 12 HLDD nodes of the two graphs in Fig. 4 correspond to these decisions. Covering all edges in a HLDD model (i.e. full *HLDD edge coverage*) corresponds to covering all branches in the respective HDL. However, similar to the previously discussed statement coverage mapping, here the opposite is also not true and HLDD edge coverage is slightly more stringent that HDL branch coverage.

The VHDL branches (Fig. 3) *1* and *2* are represented in Fig. 4 by respective edges *1a*, *1b* and *2a*, *2b*. The duplicated edges are also emphasized by subscript indexes, (e.g. 3_1,3_2 ; 4_1,4_2 ; 5_1,5_2).

4.3 A hierarchical approach for condition coverage analysis

Condition coverage metric reports all cases 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.

In this section we present a methodology for condition coverage metric HLDD-based. 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 expanded into separate HLDD graphs.

Let us consider the example design *latw09_ex1* provided in Figures 3 and 4. It contains the following 3 conditional statements:

```
cS1: If (C1 and C2)
cS2: { case cS2_C1 is
          when cS2_C1_W1 => cS2_D1;
          when cS2_C1_W2 => cS2_D2;
          when cS2_C1_W3 => cS2_D3;}
cS3: If (C1 and ((not C2) or C3))
```

The HLDD expansion graphs for these conditional statements are provided in Fig. 7. Here the terminal nodes are marked by background colors according to different decisions for better readability.

These 3 expansion graphs can be considered as sub-graphs representing "virtual" variables (because they are not real variables of the *latw09_ex1* VHDL representation) *cS1*, *cS2*, *cS3*. Thus, together with the two HLDD graphs for variables *V1* and *V2* from Fig. 4 these sub-graphs compose design's hierarchical HLDD representation.

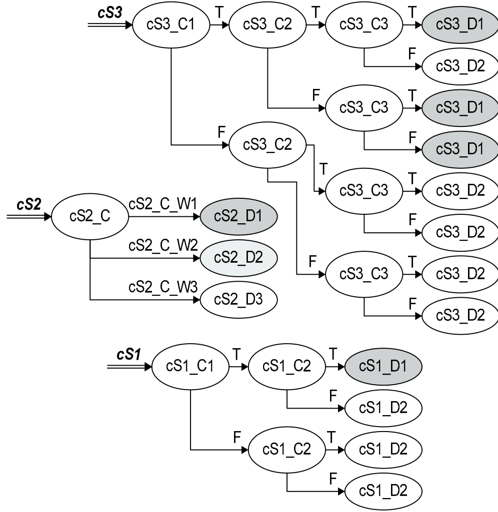


Fig. 7. Expansion graphs for conditional statements of *latw09_ex1*

The full condition coverage metric maps to the full coverage of *terminal nodes* of the conditional statements expansion graphs during the complete hierarchical DD system simulation with the given stimuli. The size of the items' list I_C for this coverage metric is:

$$I_C = \sum_{i=1}^{n_{cS_{IF}}} 2^{n_{c_i}} + \sum_{k=1}^{n_{cS_{CASE}}} n_{c_k}$$

Here $n_{cS_{IF}}$ is the number of *if*-type conditional statements, n_{c_i} is the number of conditions in the ith conditional statement and $n_{cS_{CASE}}$ is the number of case-type conditional statements and n_{c_k} is the number of conditions in the kth conditional statement. (E.g. condition coverage items' list for *latw09_ex1* is $I_C = (2^2 + 2^3) + (3) = 15$.

The main advantage of the proposed approach is low computational overhead. Once the hierarchical HLDD is constructed, the analysis for an every given stimuli set is evaluated in a straightforward manner during HLDD simulation (*Algorithm 1*, Subsection 2.2).

The size of the hierarchical HLDDs with the expanded conditional statements grows with respect to I_C and therefore there is a 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.

5. Experiments

5.1 Example study

This subsection provides an example study for the proposed HLDD-based coverage analysis methodology based on the *latw09_ex1* design. This design has 2 outputs and 6 control input signals (i.e. *cS1_C1*, *cS1_C2*, etc). The first output (*V1*) can be assigned to 2 possible values while the second one (*V2*) has 3 possibilities (i.e. {*V1_T1*, *V1_T2*} and {*V2_T1*, *V2_T2*, *V2_T3*}). The values labeling the terminal nodes in this example are symbolic and their source is neglected.

Let us assume for a simulation experiment a set of stimuli *Stim_1* with the following test vectors presented in the upper part of Table 1. The resulting values for the outputs *V1* and *V2* are also shown in the table. The lower part of the table demonstrates the items of the considered coverage metrics that are covered by a particular vector. For VHDL metrics the items are numbered statements and branches from Fig. 3. The HLDD metrics' items are numbered nodes and edges of the reduced HLDD design representation (Fig. 4). The last row of the table shows covered items of the condition coverage metric that are terminal nodes of the conditional nodes expansion graphs shown in Fig. 7. A subscript suffix (e.g. '$_{T4}$') points to the respective expansion graph's terminal number counting from top to bottom. The framed VHDL items for vectors 3 and 4 may be considered covered by some HDL simulators, but their execution does not influence the final output of the process.

Table 1. A set of stimuli *Stim_1* for *latw09_ex1*

Input / Output	Vector			
	1	2	3	4
cS1_C1	1	0	(0)	(0)
cS1_C2	1	0	(0)	(0)
cS2_C	W1	W2	W3	W3
cS3_C1	-	-	1	1
cS3_C2	-	-	1	1
cS3_C3	-	-	1	0
V1	V1_T1	V1_T2	V1_T2	V1_T2
V2	V2_T1	V2_T2	V2_T2	V2_T3
Metric	**Items covered**			
VHDL stm.	1,2,4,5	1,3,4,6	[1],[3],4,7,8,9	[1],[3],4,7,8,10
VHDL brnc.	1,3	2,4	[2],5,6	[2],5,7
HLDD nodes	4_1,1a,2a,4_2,5	4_1,1b,2b,4_2,6	4_1,7,4_2,8,9	4_1,7,4_2,8,10
HLDD edges	3_1,1a,3_2	4_1,1b,4_2	5_1,5_2,6	5_1,5_2,7
cond. exp. grp. terminals	$cS2_{T1}$, $cS1_{T1}$	$cS2_{T2}$, $cS1_{T4}$	$cS2_{T3}$, $cS3_{T1}$	$cS2_{T3}$, $cS3_{T2}$

Table 2 presents a resulting coverage values obtained for *Stim_1*. Here we consider covered duplicated items as one, i.e. if node 4_1 is covered then node 4_2 is also covered. That is the reason why the values for full-tree and reduced HLDDs are equal. Please consider the fact that HLDD-based nodes and edges coverage values are lower and therefore more stringent than the respective VHDL metrics' values. The remained uncovered HLDD items are

edges <u>2a</u>,<u>1b</u> and nodes *3a*,*2b*. In the minimized HLDD (Fig. 5) this benefit is lost with minimization. However, even with full statement and branch coverages achieved the full condition coverage would require additional test vectors. It adds orthogonal dimension of confidence in terms of stringency and verification/test accuracy.

Table 2. Coverage metrics accuracy comparison for *latw09_ex1* example study

Coverage metric	VHDL, %	HLDD, %		
		minimized	reduced	full-tree
Statement	10/10 = 100	8/8 = 100	11/13 = 85.6	11/13 = 85.6
Branch	7/7 = 100	7/7 = 100	7/9 = 77.8	7/9 = 77.8
Condition	*n/a*	7/15 = 46.7		

5.2 Experimental results

This subsection presents experimental results for four ITC99 benchmarks [9] that evaluate the proposed HLDD-based structural coverage analysis methodology. Table 3 presents the characteristics of the different HLDD representations introduced in Section 3.

Table 3. Characteristics of different HLDD representations

Design	Number of nodes			Number of edges		
	min	*red.*	*f.tree*	*min*	*red.*	*f.tree*
b01	30	57	267	52	54	264
b02	16	26	48	24	24	46
b06	47	116	440	83	111	435
b09	44	69	125	62	64	120

Table 4 shows comparison results of the proposed methodology based on different HLDD representations and coverage analysis achieved by a commercial state-of-the-art HDL simulation tool from a major CAD vendor using the same sets of stimuli.

As it can be seen, the reduced HLDDs with expanded conditional nodes allow equal or more stringent coverage evaluation in comparison to the commercial coverage analysis software. For three designs (*b01*, *b06* and *b09*) more stringent analysis is achieved using HLDDs. The HLDD model allows increasing the coverage accuracy up to 13% more exact statement measurement and 14% branch measurement (*b09* design). In our previous work [2] it was shown that HLDD-based coverage analysis has significantly lower (tens of times) computation (i.e. measurement) time overhead compared to the same commercial simulator.

6. Conclusions

It is important to emphasize that all coverage metrics (i.e. statement, branch, condition or a combination of them) in the proposed methodology are analyzed by a single HLDD simulation tool which relies on HLDD design representation model. Different levels of coverages are distinguished by simply generating a different level of HLDD (i.e. minimized, reduced, or hierarchical with expanded conditional nodes). Experimental results demonstrate feasibility and efficiency of the proposed methodology.

Table 4. Comparison of code coverage analysis results

Design	Stimuli, (vectors)	Statement coverage,(%)			Branch coverage, (%)		
		red.	*min.*	*VHDL*	*red.*	*min.*	*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

Acknowledgments: The work has been supported in part by EC FP6 VERTIGO FP6-2005-IST-5-033709, Enterprise Estonia funded ELIKO Development Centre, EC FP7 CREDES, EU Regional Development Fund project CEBE, Estonian SF grants 7068 and 7483, and Estonian Information Technology Foundation (EITSA).

References

[1] Andrew Piziali, “Functional Verification Coverage Measurement and Analysis”, *Springer*, 2008

[2] J. Raik, U. Reinsalu, R. Ubar, M. Jenihhin, P. Ellervee, “Code Coverage Analysis using High-Level Decision Diagrams”, *DDECS 2008*, April, 2008, pp. 201-206

[3] K. Minakova, U. Reinsalu, A. Chepurov, J. Raik, M. Jenihhin, R.Ubar, P. Ellervee, “High-Level Decision Diagram Manipulations for Code Coverage Analysis”, *BEC 2008, Tallinn,* Estonia, October 2008, pp. 207 - 210

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

[5] R. E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation”, *IEEE Trans. on Computers, Vol. C-35, No. 8*, August 1986, pp. 677-691

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

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

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

[9] 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

33. **Rauno Gordon**. Modelling of Cardiac Dynamics and Intracardiac Bio-impedance. 2007.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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