THESIS ON INFORMATICS AND SYSTEM ENGINEERING C76

Formal Verification and Error Correction on High-Level Decision Diagrams

ANTON KARPUTKIN

TUT PRESS

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

This dissertation was accepted for the defense of the degree of Doctor of Philosophy in Computer and Systems Engineering on 28 June 2012.

Supervisors: Prof. Raimund-Johannes Ubar, D. Sc.
Chair of Computer Engineering and Diagnostics, Dept. of Computer Engineering,
Tallinn University of Technology, Tallinn, Estonia
Mati Tombak, PhD
Chair of Computer Science Foundations, Dept. of Computer Science,
Tallinn University of Technology, Tallinn, Estonia
Prof. Jaan Raik, PhD
Chair of Computer Engineering and Diagnostics, Dept. of Computer Engineering,
Tallinn University of Technology, Tallinn, Estonia

Opponents: Prof. Roderick Bloem, PhD
Inst. for Applied Information Processing and Communications,
Graz University of Technology

Prof. Radomir Stanković, PhD
Dept. of Computer Science,
University of Niš, Niš, Serbia

Defense of the thesis: 23 August 2012

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

Anton Karputkin

ISSN 1406-4731
ISBN 978-9949-23-333-5 (publication)
ISBN 978-9949-23-334-2 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C76

Formaalne verifitseerimine ja vigade parandamine kõrgtasemelistel otsustusdiagrammidel

ANTON KARPUTKIN

TTÜ
KIRJASTUS

to my wife

CONTENTS

ABSTRACT

This dissertation explores the theory of High-Level Decision Diagrams (HLDD) in application to formal verification and design error correction. We start with methods for synthesizing the diagrams for representing digital systems at higher behavioral, functional or register-transfer levels.

After that we show how the synthesized HLDDs can be used for high-level verification of digital systems. For this purpose, the HLDD model is appended by characteristic polynomials that canonically describe the graph structure of a diagram. These polynomials can be used for proving the equivalence between two HLDDs which have the same functionality but may have different structures.

To cope with the complexity of verification problem, a novel method for probabilistic equivalence checking of digital systems is developed, which is based on extending the function domain and testing the polynomial values at random vectors from this extended portion of the new domain. It is shown that probability of getting the same results for different polynomials is very small.

As soon as an error has been detected by the proposed approach, it must be localized and fixed. The characteristic polynomial-based method is developed further to be applied to automated correction of design errors. We show how realistic design errors can be represented by the redirection-based fault model. The theoretical basis of the approach is presented with the key advantages being the ability to handle multiple errors as well as the fact that the error correction is not restricted by the input stimuli.

ANNOTATSIOON

Käesolevas väitekirjas uuritakse kõrgtaseme otsustusdiagrammide (KTOD) teooriat ja selle rakendamise võimalusi formaalse verifitseerimise ja vigade automaatse parandamise valdkonnas. Alustame meetoditega, mis võimaldavad diagramme sünteesida käitumis-, funktsioonaalse või, register-siirde taseme digitaalsüsteemide kirjeldustest.

Seejärel näitame, kuidas sünteesitud KTOD-i rakendada süsteemide formaalseks verifitseerimiseks. Seda eesmärki silmas pidades on KTOD mudelit täiendatud karakteristlike polünoomide mõistega. Need polünoomid võimaldavad kanooniliselt kirjeldada diagrammi graafi struktuuri ja neid saab rakendada kahe diagrammi, mis täidavad sama funktsiooni, aga millel on erinevad sisemised struktuurid, ekvivalentsuse tõestamiseks.

Verifitseerimisprobleemi keerukusega toime tulemaks on arendatud uudne tõenäosuslik ekvivalentsuskontrolli meetod, mille põhiideedeks on funktsiooni määramispiirkonna laiendamine ning polünoomide väärtuste võrdlemine juhuslikes punktides, mis on võetud määramispiirkonna laiendatud osast. On näidatud, et samade tulemuste saamise tõenäosus erinevate polünoomide puhul on väga väike.

Niipea kui viga on tuvastatud pakutud lähenemisega, tuleb ta lokaliseerida ja parandada. Karakteristlikel polünoomidel põhinev meetod on arendatud edasi automaatse vigade parandamise ülesande lahendamiseks. Näitame kuidas realistlike disainivigu saab esitada ümbersuunamisel põhineva vigade mudeli abil. Lähenemisel on järmised eelised: on võimalik parandada mitu vigu korraga ning parandamine ei ole sisendvektoritega piiratud.

LIST OF JOURNAL PUBLICATIONS

1. Viilukas, T., Karputkin, A., Raik, J., Jenihhin, M., Ubar, R., Fujiwara, H.; **Identifying Untestable Faults in Sequential Circuits Using Test Path Constraints.** *Journal of Electronic Testing Theory and Applications, [Accepted for publication].*

2. Karputkin, A., Ubar, R., Raik, J., Tombak, M.; **Canonical representations of high-level decision diagrams.** *Estonian Journal of Engineering*, Volume 16, Issue 1, Tallinn, Estonia, 2010, pp. 39-55.

LIST OF CONFERENCE PRESENTATIONS

1. Karputkin, A., Ubar, R., Tombak, M. and Raik, J.; **Probabilistic Equivalence Checking Based on High-Level Decision Diagrams.**; 14th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, Cottbus, Germany, April 13-15, 2011, IEEE Computer Society Press, 2011, pp. 423-428.

2. Karputkin, A., Ubar, R., Tombak, M., Raik, J.; **Interactive Presentation Abstract: Automated Correction of Design Errors by Edge Redirection on High-Level Decision Diagrams**; 16th IEEE International High Level Design Validation and Test Workshop, Napa Valley, CA, USA, Nov. 9-11, 2011. IEEE Computer Society, 2011, p. 83.

3. Karputkin, A., Ubar, R., Tombak, M., Raik, J.; **Automated Correction of Design Errors by Edge Redirection on High-Level Decision Diagrams**; 13th International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, Mar. 19-21, 2012, pp. 686 - 693.

ACKNOWLEDGEMENTS

I WOULD LIKE TO EXPRESS MY GRATITUDE to everybody who helped me during my PhD studies and without whom this work would not appear.

In particular, I would like to thank Dr. Mati Tombak, who, being my bachelor's thesis supervisor in the University of Tartu, introduced me this research domain; my Master's thesis supervisor, Dr. Ahti Peder, who encouraged me to move to Tallinn University of Technology joining the group of Prof. Raimund Ubar, where I could more deeply study this field; Prof. Raimund Ubar and Dr. Jaan Raik for supporting me in my first steps in the engineering domain, which I, with my purely theoretic background, had absolutely no clue about.

I would like to show appreciation to all my supervisors for their wise advices and support; for interesting discussions that helped me to develop the theory and overcome all obstacles that appeared on the way to this thesis.

Furthermore, I would like to acknowledge the organizations that have supported my PhD studies: Tallinn University of Technology, National Graduate School in Information and Communication Technologies (IKTDK), EU's FP7 collaborative research project DIAMOND, European Regional Development Fund through the Centre for Integrated Electronic Systems and Biomedical Engineering (CEBE) and Estonian IT Foundation (EITSA).

Finally, I would like to thank my family for all the care and patience – especially my wife Jekaterina. Thank you!

Anton Karputkin
Tallinn, June 2012

ACRONYMS

HLDD	High-Level Decision Diagram
BDD	Binary Decision Diagram
*BMD	Multiplicative Binary Moment Diagram
EVBDD	Edge-Valued Binary Decision Diagram
MDD	Multiple-valued Decision Diagram
SMT	Satisfiability Modulo Theory
CDCL	Conflict-Driven Clause Learning
SSBDD	Structurally Synthesized Binary Decision Diagram
ROBDD	Reduced Ordered Binary Decision Diagram
CNF	Conjunctive Normal Form
FSM	Finite-State Machine
EFSM	Extended Finite-State Machine
RTL	Register-Transfer Level
EDA	Electronic Design Automation
TLM	Transaction Level Modeling
HDL	Hardware Description Language
NNF	Negation Normal Form
VLSI	Very-large-scale integration

THESIS

1

INTRODUCTION

In this introductory chapter we give a brief overview of the domain addressed by the current thesis. We start with the motivation for this work, followed by the problem formulation and the outline of main contributions. The last section describes the organization of the thesis.

1.1 MOTIVATION

Nowadays we all are surrounded by digital devices. They are everywhere: at home, at work, in our cars and our pockets. Wi-Fi hotspots, cell towers and even space satellites are granting us Internet access even in most remote places of our planet. Soon the cars will not need a driver [90]. Mobile phones are evolved into a complex platform that replaced a dozen independent devices we used to have just 15 years ago. People quickly got accustomed to such rapid progress and now some of us even cannon imagine their lives without checking Facebook updates every half an hour. All this is possible due to correct functioning of hundreds and thousands of different microelectronic devices, and behind this correctness there is a hard work of hardware designers, verification and test engineers.

Such a hard work is a must; the cost of an error is always high in the industry. It cannot be fixed by applying a patch, a hotfix or something like that, as software developers are used to do. There are only two options: replace the malfunctioning component or live with it. One can remember some cases, when companies were obligated to withdraw already produced and sold devices due to bugs, like it happened in 1994 with Pentium FDIV bug [52], an error in the floating point division implementation of Intel P5 Pentium CPU, that cost Intel approximately half a billion dollars [49].

In the more critical fields of application, like the aerospace industry, money loss is even not the hardest consequence of bugs. Errors can make years of work of thousands people useless, like the unit conversion problem of the Mars Climate Orbiter [95], where the flight system calculated trajectory using the metric

system while the entered data were in English units. As a result, four years of design and nine months of waiting while the Orbiter reaches the Mars were lost.

Thus, hardware bugs must be located and fixed at any cost before the device or component production, and this makes the situation extremely challenging. On the other hand, companies wish to release their products as quickly as possible and, in less critical areas than aerospace or medical industries, manufacturers cannot afford to spend a lot of time and money on testing and debugging, otherwise they will lose the competition with the concurring companies. Hence, we always need verification and error correction solutions that perform in reasonable time to guarantee the product reliability at a competitive price.

1.2 PROBLEM FORMULATION

Hardware design engineers are just humans and, as humans, they always will make mistakes, no matter how advanced design methodologies we will use. Abstraction layers go higher, tools become more and more sophisticated, but bugs still remain. We cannot avoid them at the design stage, but we can find them and fix later, and this stage is called *verification*. One can clearly see its important role from the previous section.

The complexity of digital nanoelectronics designs has reached a level where it is an immense challenge to guarantee their functional correctness. According to statistics, around 70% of the project development cycle is devoted to design verification. Sometimes, in project teams there are twice more verification engineers than design engineers (usual ratio is one to one) [65].

At present, there is a variety of methods they can utilize. Most of them can be divided into two large groups: *simulation-based* and *formal* methods. We discuss them in more detail in the next chapter and here we emphasize the common disadvantage of such methods. They only detect the presence of an error, often providing some sort of counterexample, describing the case, when the designed circuit's behavior is incorrect. But the true goal is to find the root cause of the error and fix it. Such counterexamples contain too much information for a designer to handle but still too little information to identify this root cause. At the moment, this is mostly tedious manual work.

The aim of the current thesis is to make the process of fixing bugs more automated. At present, there are only two notable method classes in this field, that work at higher abstraction levels: *resynthesis* [21] and *error matching* [34]. We discuss them in the next chapter as well. Neither of them can fix all the errors. The method presented here is also unable to do this. Probably, this task is impossible to solve, at least in reasonable time. However, each error, fixed by such

incomplete methods, saves hours of designer's valuable time and makes devices cheaper.

1.3 THESIS CONTRIBUTION

The main contributions of this thesis are summarized as follows:
As the thesis addresses related, but different problems, we can divide the contributions of it into three groups.

- ***General contributions to the theory of HLDDs***.
  - A formal definition of HLDD as a data structure for modeling digital system at higher levels is given.
  - Characteristic polynomials are applied to canonically represent the non-terminal part of HLDDs.
  - An algorithm for normalizing different variable sets is given.
  - An algorithm of removing miscomparing states produced by auxiliary variables.
- ***Contributions to equivalence checking methodology***.
  - An algorithm of computing characteristic polynomial values at random vectors on the given diagram, with speed-up achieved due to the use of binomial coefficients.
  - A new measure of collision probability for this algorithm, more accurate in the general case than Schwartz-Zippel theorem usually applied for similar tasks.
- ***Contributions to the problem of automated error correction***.
  - A novel HLDD-based error model.
  - A new algorithm for fixing edge-related errors on HLDDs by manipulating characteristic polynomial values.

1.4 THESIS STRUCTURE

The thesis contains 7 chapters.

Chapter 2 provides background information on verification and error correction and makes a review of state-of-the-art in the area addressed by the thesis. It also contains a survey of related works for each of the topics.

Chapter 3 consists of two parts. In the first part we continue to provide background information, but this time we focus on various binary and word-level diagram types. In the second part of the chapter we present HLDDs and methods of translating hardware designs into the sets of HLDDs.

In Chapter 4 we study how to describe functions, represented by HLDDs, canonically. We define characteristic polynomials, with aid of which we will restore the hidden canonicity of non-terminal part of HLDDs and present algorithms of generating these polynomials and dealing with some side problems.

In Chapter 5 we propose a method for probabilistic equivalence checking based on characteristic polynomials over HLDDs, give an estimation of collision and solve a side problem on comparing the state diagrams as well. The feasibility of this methodology is studied on the experiments with ITC99 benchmarks appended with some industrial designs.

Chapter 6 contains further study on characteristic polynomial properties applying them for solving the task of error localization and correction. An HLDD-based error model is presented there and an algorithm that fixes a subset of errors from that model is also provided. The chapter is finished by the set of experiments on mostly the same benchmarks as were used in the previous chapter.

Chapter 7 concludes the thesis and discusses the directions for further research.

2

BACKGROUND

In this chapter we give a brief overview of state-of-the art verification and error correction methods. In the first section we describe simulation-based verification. After that, we introduce SAT/SMT solvers' architectures. In the next section we provide an overview of some formal methods. In section 2.4 we explain existing error correction solutions: resynthesis and error matching. Finally, the last section summarizes given content.

2.1 SIMULATION-BASED VERIFICATION

The idea of simulation-based verification is pretty simple and clear from its name. Digital system simulation is a special case of the more general discrete event simulation methods, which were initially developed in the 1960s [69]. Discrete event simulation is a method of modeling a system, either real or hypothetical, over time on a computer. A continuous time line is split into discrete instants at which the system changes its state depending on some combination of external stimulus and current state.

In simulation-based verification the underlying system is a digital circuit. Nowadays, it is the most commonly used verification approach. We compare our circuit implementation with the formal specification or a reference design, if we have one. Special software or hardware, called a *test bench* applies input stimuli, executes the designs and compares their results. Input stimuli can be either generated on the fly or read from some external source. The same holds for the reference output. A typical architecture of the simulator is shown in Figure 2.1.

The front end is very much standard for most simulators and is a function only of the input language. The main tasks for the back end are analysis, optimization, and generation of code to simulate the input circuit; simulator's speed depends mostly on how well the back end solves these tasks. Together, the front end and the back end compile the design into an internal representation of the system. Then the simulation engine takes in the compiled design and computes the behavior accordingly. Simulation control allows the user to interact with the

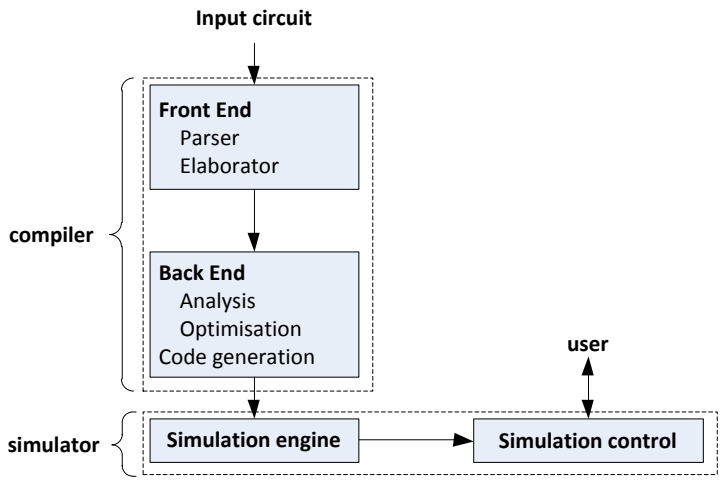

Figure 2.1 – Major components of a simulator

simulator, like setting break points to pause a simulation, examining variable values, etc.

There are two types of simulation engines: *event-driven* and *cycle-based*. The former evaluates a gate whenever one or more of its inputs change values for gate-level simulation. Similarly, at higher levels a block of statements is evaluated whenever the variable values to which the block is sensitive are modified. This value change is called an event. The more events circuit has, the slower is simulation. In the latter type of simulators, the subcircuits are extracted according to clock domains and at each triggering clock edge the corresponding subcircuit is evaluated. In practice, most circuits have a lot of events, thus cycle-based simulators usually run faster. However, they are not applicable when clock domains in the circuit are not well defined. Therefore, event-driven simulators is the only choice, e.g., for asynchronous circuits.

Simulation is a completely general method: any hardware design can be simulated if we have enough time and computing power. The last remark points to one of the main simulation drawbacks. Typically we use one single processor, maybe with 2 or 4 cores to reproduce the hardware running millions of parallel processes. Another drawback is the quality of our verification. It is not feasible to test all possible inputs and states exhaustively for any non-trivial system.

To address this issue three different metrics can be adapted. Each of them estimates the accuracy of simulation with respect to one particular aspect of the design and together they complement each other. *Code coverage* evaluates how much of the implementation code was exercised by simulation. In *parameter cov-*

erage we first identify parameter ranges for each functional unit and then compute, how extensively we stressed each of these parameters. Finally, in *functional coverage* we measure the amount of design features and operations that were exercised. Note, that first two metrics are functions of the implementation only, while the latter is based on the specification. Thus, in order to have fast and thorough simulation we need to generate our test vectors so that the total number of them would be as small as possible having the coverage measures as high as possible.

2.2 SAT AND SMT SOLVERS

Before the computer era, mathematicians had little or no interest to the questions related to the computability. In most cases it was enough to discover that a solution exists. On small, human manageable inputs there is not much difference how many steps your algorithm requires, 2^n or n^3. Everything has changed with the invention of the computers. Inspired by the increasing power of computing devices in the middle of XX$^{\text{th}}$ century, scientists started to dream about machines, solving tasks that would normally require "creative" approach, e.g., prove mathematical theorems [44]. This was led to the rapid progress in mathematical logic and theory of algorithms. Suddenly it appears that not all problems can be solved algorithmically, and for some of those that can be solved researchers were unable to propose anything better than some sort of brute force search. There was required a theory, that would answer the questions like whether there is an algorithm to handle a particular problem and how much resources it would need. That is how the notion of *computational complexity* and the *complexity theory*, which studies it, appear.

2.2.1 *Computational Complexity*

In this subsection we give just some basic definitions required to understand the latter content. Space limitations do not allow us to discuss different notions of algorithm, the kinds of Turing machines, etc. For further information in these topics please refer to [4].

Let us consider here an informal notion of algorithms, as something that can be run on a computer. Then, computational complexity of an algorithm is the number of basic operations it performs as a function of its input length. From this sentence we see, that actually, this function depends not only on input, but on the definition of these basic operations. And that is why this function is rarely used and instead we use the big-oh notation:

Definition 2.1. *Let $f, g : \mathbb{N} \to \mathbb{N}$ be two functions. Then we say that $f = O(g)$ if for some natural number c $f(n) \leq c \cdot g(n)$ for every $n \in \mathbb{N}$.*

This notation allows us get rid of low-level implementation details. Instead of thoroughly computing the number of operations and obtaining results like $f(n) = 100n^2 + 21n + 15n \log n + 5$ we may just say that $f = O(n^2)$. Assuming, that each operation takes one unit of time to complete we may talk about computation time.

Different problems can be grouped into several complexity classes depending on the number of steps it is required to solve them. Let us define some of these classes.

Definition 2.2. *We say that a problem belongs to class* **P** *if there exists $c > 0$, such that it can be solved by an algorithm, whose complexity is $O(n^c)$.*

Definition 2.3. *Similarly, a problem belongs to class* **EXP** *if there exists $c > 1$ such that it can be solved by an algorithm, whose complexity is $O(2^{n^c})$.*

A bit different from these classes is the next one:

Definition 2.4. *A problem belongs to class* **NP***, if there exists $c > 0$, such that for every solution candidate there exist an algorithm checking whether this candidate is a solution or not in $O(n^c)$ time.*

It is easy to see that $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$. The question whether the first relation is strict is still open an one of the most important questions of the contemporary mathematics.

Definition 2.5. *We say that a problem A is* **NP***-hard, if any problem $B \in \mathbf{NP}$ can be reduced to A in polynomial time. If A itself belongs to* **NP** *than we say that A is* **NP***-complete.*

The first known example of an **NP**-complete problem is *Boolean Satisfiability Problem* or *SAT*, which we study in the next subsection.

2.2.2 *Boolean Satisfiability Problem*

In SAT, we study instances of Boolean formulae $f : B^n \to B$, functions, whose parameters are vectors of zeros and ones and each value is also either zero or one. There are several ways to express these functions and one of them is so called *Conjunctive Normal Form (CNF)*. A function written in this form looks like the following:

$$f(x_1, \ldots, x_n) = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{m_i} l_{ij}$$

where each l_{ij} is either x_k or $\overline{x_k}$ for some $k \in \{1, \ldots, n\}$. Each set of disjunctions is called a *clause*. The problem can be formulated as follows:

Definition 2.6. *Given a Boolean formula in CNF form, determine whether this formula is satisfiable, i.e. there exists a variable assignment that evaluates it to 1.*

This is an example of a *decision problem*: a problem, whose answer is either yes or no. An algorithm, that solves such problem (i.e. always terminates with the correct answer) is called a *decision procedure*. Not every problem has such procedure, those, which does not have one, are called *undecidable problems*. Sometimes we can find an algorithm, such that if it returns 1, then the result is 1 and if it returns 0 or does not terminate, then we cannot say anything about the real answer. Those procedures are called *sound*. Another type are the *complete* ones: a procedure of this type must always terminate and when the answer is 1, it returns 1 (and may return 1 when the answer is 0). Thus, all decision procedures are sound and complete.

Many decision and optimization problems from various fields of mathematics and engineering can be translated to SAT instances, like Traveling Salesman Problem, Graph Coloring, Integer Programming, etc. The **NP**-completeness of SAT means that the algorithm that could efficiently decide all SAT instances is currently not known and probably there does not exist one. All known algorithms have exponential complexity in the worst case. However, **NP**-complete problems arise everywhere and require to be solved. Modern SAT solvers can efficiently solve large enough subset of instances to be usable in practice. We present here two main classes of such solvers.

First class of solvers employ a DPLL algorithm [31, 32], a method named after its inventors, Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland. It is based on *Shannon expansion* [86] of the Boolean Formula:

$$f(x_1, \ldots, x_n) = x_1 \wedge f(1, x_2, \ldots, x_n) \vee \overline{x_1} \wedge f(0, x_2, \ldots, x_n)$$

Two simple observations are applied as well:

- If there is a clause containing only one literal, then there is no choice than setting this literal to 1. This is called *unit propagation rule*.
- If all occurrences of a variable in literals contain only one polarity, than this is called *pure literal*. Such literals can always be assigned in a way that evaluates all clauses containing them to 1 and these clauses can be eliminated from the formula. This is called *pure literal elimination*.

The pseudocode of the DPLL procedure is given in Algorithm 2.1. By f_{x_i} we denote $f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots n)$, and $f_{\overline{x_i}}$ means the substitution of 0 instead

Algorithm 2.1 DPLL Procedure

```
1:  function DPLL(Boolean formula f(x_1, ..., x_n) in CNF form)
2:      if f is empty then
3:          return 1
4:      end if
5:      if f contains empty clause then
6:          return 0
7:      end if
8:      for all unit clauses c in f do
9:          UnitElimination(c, f)
10:     end for
11:     for all pure literals l in f do
12:         AssignPureLiteral(l, f)
13:     end for
14:     Choose i from 1 to n
15:     return x_i ∧ DPLL(f_{x_i}) ∨ ¬x_i ∧ DPLL(f_{¬x_i})
16: end function
```

of x_i. In step 14 we can apply some heuristics aiming to simplify recurrent formulas from the next step as much as possible. Proper choosing procedure can drastically increase the solver's speed.

Most modern DPLL-type solvers employ an additional technique called *Conflict-Driven Clause Learning (CDCL)* [10, 74, 92]. Assume we had selected a sequence of variables (step 14), which lead us to the state when we are required to return 0 (step 6). This state is called a *conflict*. Some other sequence could contain a subsequence, which results in the same conflict. In order to avoid this, once we have reached the conflicting state for the first time, we produce a new clause, which would prevent the same subsequence appear again, and add this clause to the formula.

In addition to this, the original DPLL algorithm employs a *chronological backtracking* scheme, when, if the choice at line 14 was unsuccessful, we undo it and continue the search from the previous position in the recursion stack. Alternatively, the information contained in the new learned clauses allow CDCL-type solvers jump to lower positions. This approach is called a *non-chronological backtracking* or *backjumping* [74].

Another class of algorithms employ stochastic local search techniques [85]. Here we start by picking a random assignment of variables. If it satisfies the formula, then we stop the procedure returning 1. If not, we flip variable values one by one under some heuristic observations to obtain satisfying assignment at the end. For example, WalkSAT algorithm examines the set of clauses that the

initial assignment evaluates to 0, pick one of them, flip a variable, that would make this clause satisfied and the total number of unsatisfied clauses would be reduced. These flips can achieve local minima of unsatisfied clauses. In this case the algorithm may restart with a new random assignment.

Generally, SAT instances obtained from "real life" problems, so called industrial benchmarks, has some internal structure, and DPLL/CDCL algorithms with good choice heuristics outperform the ones that exploit the local search technique, while the latter class of algorithms is usually better with randomly generated instances [61].

2.2.3 *Satisfiability Modulo Theories*

As we have mentioned in the previous subsection, SAT is the **NP**-complete complete problem, and any other problem from this class can be converted into it. However, during the translation into the SAT instance we often lose the domain-specific information of the initial task. For example, consider the following formula:

$$a + b < 5 \land b + a < 6,\ a, b \in \mathbb{N}$$

We can see that it can be simplified to $a + b < 5$, but SAT solvers do not know anything about commutativity of addition and need to rediscover it from the structure of the resulting logic expression.

Hence, it would be great to combine the power of modern SAT solvers with the expressiveness of the original domain, and this is the general idea of an SMT solver. Let us continue with some definitions:

Definition 2.7. *A first-order logic formula is the expression that contains the following elements:*

- *Variables.*
- *Standard logical symbols, like operators* $\lor$, $\neg$ *and* $\land$, *quantifiers* $\forall$ *and* $\exists$ *and parenthesis.*
- *Non-logical symbols: functional* ($+$, $-$, $\sin$, $\log$, *etc.), predicate* ($=$, $\leq$, $\approx$, *etc.) and constant (1, 5, 2.45e-10, etc) ones.*

Also, the set of syntactical rules for constructing such formulae must be defined. If these rules were not violated then the formula is called ***well-formed****.*

Constants, variables and functions are called *terms* and simple predicates without deeper propositional structure are *atoms*. In simple words, it is just an extension of Boolean expressions, where we put predicates instead of literals. However,

Table 2.1 – Examples of SMT expressions and theories

Expression	Theory
$x_1 \wedge (x_2 \vee \overline{x_3})$	Propositional logic
$x = y \wedge y = z \implies x = z$	Equality
$2x + 3 > 5 \vee 5y + 2z + 4x > 4 \implies 3y + 4z > x$	Linear Arithmetic
$(a \wedge 0\text{xF8}) \gg 2 > b$	Bit-vectors

this formula does not mean anything without *interpretation*. We must fix the domain set D, each constant symbol must refer to an element from D, each function symbol f is substituted by the actual function $f_I : D^n \to D$, and each predicate symbol P by predicate $P_I : D^n \to \{0, 1\}$.

SMT solvers operate instances of the first-order logic formulas from a set of fixed domains or *theories*. This is the main difference between them and general theorem provers that could find non-standard interpretation to satisfy given formula, and perform much slower thereafter. Some examples of theories, currently supported by most of the solvers are given in Table 2.1

The general principle of deciding such formulae is given below. Consider well-formed formula $\Phi(P_1, \ldots, P_n)$ of theory $\mathcal{T}$, where $P_1, \ldots, P_n$ are predicates.

1. Introduce new Boolean variables $x_1, \ldots, x_n$. Replace predicates by literals produced from these variables, obtaining the formula $\Phi(x_1, \ldots, x_n)$ (called a *propositional skeleton*).

2. Find a satisfying assignment $\alpha_1, \ldots, \alpha_n$ for the propositional skeleton with a SAT solver, or determine that it is unsatisfiable.

3. If $\Phi(x_1, \ldots, x_n)$ not satisfiable then return 0, else build an expression $\Phi_\alpha = P_{\alpha_1} \wedge \ldots \wedge P_{\alpha_n}$, where $P_{\alpha_i} = P_i$ if $\alpha_i = 1$ and $P_{\alpha_i} = \overline{P_i}$ otherwise.

4. Call a theory specific procedure for Φ_α. If it finds a satisfying assignment then return 1, else go to step 2.

Let us continue with some examples of theories and their decision procedures.

Example 2.1. *Quantifier-free linear arithmetic (QF_LRA):* in this theory predicates are linear inequalities over real numbers. This task is pretty similar to *linear programming (LP)*: an optimization problem, where we need minimize or maximize a linear function under a set of linear constraints (inequalities). The only difference from our case is the existence of such function, while we would be satisfied with any point from the region defined by those constraints. A variety of LP

Algorithm 2.2 Bit Flattening

```
1:  function BitFlatten(QF_BV formula f)
2:      φ ← PropositionalSkeleton(f)
3:      X ← BooleanVariables(φ)
4:      for all terms t of f do
5:          for i ← 1 to BitWidth(t) do
6:              X ← {x_{t_i}} ∪ X
7:          end for
8:      end for
9:      for all atoms a of f do
10:         φ ← φ ∧ BitConstraint(a, X)
11:     end for
12:     for all terms t in f do
13:         φ ← φ ∧ BitConstraint(t, X)
14:     end for
15:     return φ
16: end function
```

solving algorithms exist, all of them are easily convertible to our problem. Most notable ones are Simplex, Branch and Bound, Fourier-Motzkin variable elimination, Khachiyan's Ellipsoid, and Karmarkar's Interior Point methods [7]. Last two of them have a polynomial complexity.

We may have inequalities over integers instead of reals. This case is harder, Integer Linear Programming (QF_LIA) is **NP**-complete, like SAT.

Example 2.2. *Quantifier-free bit vectors (QF_BV):* instances of this theory appear naturally when we are trying to solve different task related to hardware design. Later we see some examples of these tasks. Unfortunately, in this case the general method of splitting tasks between the SAT solver and the theory-specific solver is hardly applicable. Instead, the technique called bit-flattening is used, which, in simple words, means that we convert everything into a SAT instance and solve it with the SAT solver. The conversion procedure is presented in Algorithm 2.2. In steps 10 and 13 it refers to function *BitConstraint*, which replace corresponding atom or term with its representation in new Boolean variables defined in step 6. For example, it could be a circuit implementation of the given function or predicate.

Other examples of theories include the theory of arrays, pointer logic, uninterpreted functions, etc. Some of them can be combined under certain conditions to solve more complex formulae. SMT solvers made tremendous progress in last

decade and now are widely used. In later sections we will get acquainted with some of their applications.

2.2.4 *Related Works*

In 1948 Shannon discovered a new way of manipulating Boolean functions, which was later called after him *Shannon expansion* [86]. Davis, Putnam, Logemann and Loveland in their works [31, 32] proposed CNF formulae for satisfiability testing and an algorithm to solve this problem, now known as DPLL, that used Shannon expansion as a basis. According to [42], the first paper containing systematic studies in computational complexity was [48] which laid out the definitions of time and space complexity and proved some hierarchy theorems. In 1971 Cook [26] and, independently, in 1973 Levin [68] proved a theorem that Boolean satisfiability problem is **NP**-complete. In 1972 Karp [55] published his list of 21 **NP**-complete problems, in which he showed that SAT can be reduced to 3SAT, where a CNF formula is allowed to have at most 3 literals per clause.

Since that time hundreds of different **NP**-complete problems were discovered [29, 38], but an important implication of Cook-Levin's theorem is that researchers, instead of trying to solve each his or her own problem can focus on just some generic tasks, do their best trying to solve them and obtain results they need by just translating the output to the languages of their problems. During the years SAT solvers made tremendous progress, being now able to solve instances of thousands of variables and millions of clauses [83], and became applicable to solve some complex industrial tasks.

However, as most of the tasks the industry require to determine satisfiability use richer languages than propositional logic, e.g. first-order theories, a tool that would combine the power of SAT solvers and expressiveness of first-order logic would be appreciated. That was the motivation to implement first SMT solvers. The foundations of SMT can be traced back to early works 1970s and 1980s, like [13, 75, 88, 89]. Modern research in that field began in 1990s with independent attempts to integrate some decision procedures into SAT solvers [3, 46, 77]. During the last decade a number of efficient SMT implementations appear [15, 33, 39] as well as verification tools that utilize them [8, 63].

2.3 FORMAL METHODS

We are already familiar with simulation-based verification. This method is simple and widely used in practice. It makes us sure, that our design is bug-free at certain isolated points, and this is the major disadvantage of the method. To over-

come it we require formal techniques that can prove the design correctness for larger sets of operational modes. In this section we describe some these techniques.

The behavior of digital system is often modeled as a *finite state machine (FSM).*

Definition 2.8. *The (Mealy-type) finite-state machine M is a tuple $(X, Y, S, S_0, \delta, \lambda)$, where X, Y and S are the sets of inputs, outputs and states, respectively, $S_0 \subset S$ denotes the set of initial states, $\delta : X \times S \to S$ and $\lambda : X \times S \to Y$ are the next-state and the output functions, respectively. If output depends only on state, then such machine is called Moore-type FSM.*

A state $s \in S$ is called *reachable* if there is a sequence of states $s_0 \in S_0$, $s_1, \ldots, s_n = s$ and inputs $x_0, \ldots x_{n-1}$ such that $s_i = \delta(s_{i-1}, x_{i-1})$ for $i = 1..n$.

2.3.1 *Equivalence Checking*

During the design cycle, different levels of hardware design are produced. At the moment, the most common approach is to implement the system at Register-Transfer Level (RTL) using one of the hardware description languages (HDL) and then convert it to the gate-level netlist using some of the EDA tools. Later we may append this netlist with an additional functionality, like Design for Testability structures, optimize the design, do some manual changes, etc. An approach to describe the hardware at higher levels, like Transaction-Level Modeling (TLM) is becoming more and more popular.

Once we have such variety of different versions of the design, we need to be sure, that all the versions do the same job, or, in other words, they are functionally equivalent. The process of certification that they are truly equivalent is called *equivalence checking*. Consider the most common case of it: RTL versus gate-level description. We convert both versions of the design into FSMs $M_1(X_1, Y_1, S_1, S_{0,1}, \delta_1, \lambda_1)$ and $M_2(X_2, Y_2, S_2, S_{0,2}, \delta_2, \lambda_2)$. Assume, that there is only one initial state for both machines $S_{0,1} = \{s_{0,1}\}$ and $S_{0,2} = \{s_{0,2}\}$, there is a one-to-one mapping between inputs and between outputs, and timing is identical

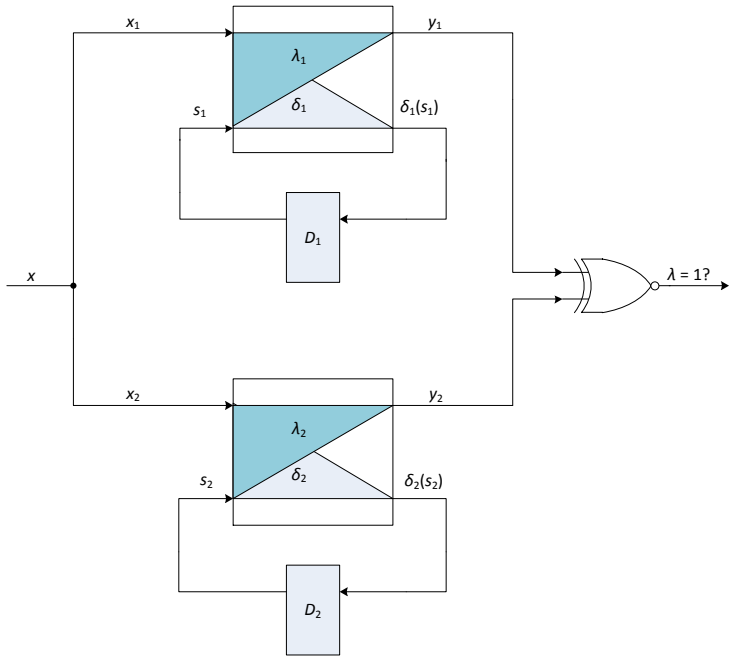

Figure 2.2 – Product machine for comparing two FSMs

for M_1 and M_2. Then we construct a *product machine* $M = (X, Y, S, S_0, \delta, \lambda) = M_1 \times M_2$ in the following manner:

$$
\begin{aligned}
X &= X_1 = X_2 \\
S &= S_1 \times S_2 \\
S_0 &= S_{0,1} \times S_{0,2} \\
Y &= \{0, 1\} \\
\delta(x, (s_1, s_2)) &= (\delta_1(x, s_1), \delta_2(x, s_2)) \\
\lambda(x, (s_1, s_2)) &= \begin{cases} 1 & \text{if } \lambda_1(x, s_1) = \lambda_2(x, s_2) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 2.2 shows a schematic view of the resulting machine.

FSMs M_1 and M_2 are said to be functionally equivalent iff for all reachable states s and input vectors x we have $\lambda(s, x) = 1$. To determine that we need to relate states of the initial machines between each other. This is done by determining a characteristic function $\rho : S \to \{0, 1\}$ such that

- $\rho(s_0) = 1 \; \forall s_0 \in S_0$

- if $\rho(s) = 1$ for some $s \in S$ then $\forall x \in X$
  - $\rho(\delta(x, s)) = 1$
  - $\lambda(x, s) = 1$

One can see from the last set of equations that the task of equivalence checking can be split into two parts. First, a candidate for the function ρ must be chosen, and second we should check the last two conditions for validity. The most common approach assumes a combinational equivalence checking paradigm: state elements have 1:1 correspondence and we must demonstrate that the outputs as well as the next-state functions of the correlated state elements are equivalent. The correspondence between M_1 and M_2 can be either obtained from the naming conventions or, in case of absence of such conventions, it can be heuristically guessed from their behaviors. Then we need to check the Boolean equivalence of a set of combinational circuits. There are two widely used methods for the latter task: Binary Decision Diagrams, which will be introduced in the next chapter and SAT, where two circuits, expressed by Boolean functions f and g are combined with an XOR gate[1], the function $f \oplus g$ is translated into CNF form and passed to SAT solver, which should verify its unsatisfiability. If we check the higher levels, then an SMT solver can be utilized instead.

It is important to mention, that in case of failing check we cannot distinguish whether the systems are inequivalent or just the wrong ρ has been guessed, and thus, described procedure is sound, but not complete. In practice, the incompleteness of the combinational equivalence checking is often addressed by rejecting all failing designs with a demand for design update with a valid register correspondence and equivalent input/output behavior.

2.3.2 *Model Checking*

Checking the functional equivalence between two circuits is only one aspect of formal verification. Often the formal specification is not available, or it is incomplete, or the mapping between states cannot be determined. In these cases *model checking* is required. It is a more general problem than equivalence checking and its aim is to prove or disprove that the system implies some property, which is a part of a specification.

Example 2.3. Assume we have to design a traffic light controller. As usual, it should show three colors, green (G), yellow (Y) and red (R), one at a time, with the following sequence (..., G, Y, R, Y, G, ...). That are the properties of our system

1 the resulting circuit is called a *miter*

and we need to check them in order to make sure our controller is implemented correctly.

There are three common types of properties:

1. *Safety property.* These properties state that some undesired conditions must not happen.

2. *Liveness property.* This property type ensures that some essential conditions must be reached.

3. *Fairness property.* Here we ensure that some conditions happen repeatedly.

Returning to our example, the statement "R, G, and Y must be available" is the liveness property, "one color at a time is permitted" is the safety property and "the sequence (..., G, Y, R, Y, ...) should repeat" is the fairness property.

Often properties are expressed by Temporal Logic formulae where propositions are qualified in terms of time. Usual logic operators like $\vee$, $\wedge$, $\neg$ are complemented by temporal operators, like:

- Until: $\phi \; \mathcal{U} \; \psi$ – ψ holds now or shall hold in the future and until that moment ϕ has to hold. If ψ holds, ϕ does not have to.

- Release: $\phi \; \mathcal{R} \; \psi$ – ϕ releases ψ if ψ is true until the first position in which ϕ is true (or forever ϕ is never true).

- Next: $\mathcal{N} \; \phi$ – ϕ must hold at the next position.

- Future: $\mathcal{F} \; \phi$ – ϕ must hold eventually in the future.

There are plenty of ways how we can check properties. First, we can encode the property as an automaton, combine it with the system's automaton in a similar way as we did it for equivalence checking and then check whether some desired or undesired states are reachable, or, in case of a fairness property, whether there is some reachable cycle in the state transition graph.

The main disadvantage of this approach is that the number of states can be prohibitively large, disallowing us to construct the automaton explicitly. Instead, we can encode states and transitions between them with Boolean formulae. In the case of *Bounded Model Checking (BMC)* we then unfold this representation, making k consecutive copies, corresponding to consecutive time frames. Then we check with a SAT solver, whether the given property holds during these k steps. If the solver is able to generate us a counterexample then the property is invalid, otherwise it still can be invalid, just k is not reasonably large to prove it.

Thus, this method is complete but not sound. Again, SAT solver can be replaced by SMT in case of high-level designs.

Another method is called *symbolic simulation*. We start with assigning values to inputs, but leave some of them unassigned. Then, as in usual simulation we propagate these values to the outputs and obtain some function over the unassigned variables. Then we may compare it with the desired function. Thus, one symbolic simulation step can cover many usual simulation runs.

There is a lot of other model checking approaches, but describing all of them is beyond the scope of the current thesis. In case of interest one can consider reading [65] and [66].

2.3.3 *Related Works*

The birth of equivalence checking appeared when systems reached the complexity degree at which it became infeasible to model hardware at the logic level. In late 1970s IBM included RTL modeling into the design flow of its mainframe chips. At that time it was used mainly for faster simulation while the implementation still was done at the gate level. Different models required to be checked for equivalence and it was done [94]. Few years later, automatic logic synthesis tools were introduced, but RTL vs. gate-level check still remained important, due to possible bugs in logic synthesis software and as the synthesized netlist is a subject of frequent manual intervention. In the early 1990s, IBM introduced into their design flow a possibility to include custom-made transistor-level circuit blocks, whose correctness must also be ensured [62]. Similar methodologies were being adapted during the design of later versions of the Alpha microprocessor [11] and Intel's microprocessors [73]. In the mid-1990s, after logic synthesis tools had become widely available, the CAD tool vendors embedded the equivalence checking into their software.

In 1993 Brand introduced a miter structure [14]. Later, in the second half of 1990s some register matching problem solutions were proposed [20, 40]. The general case of sequential equivalence checking was studied in works like [27, 98].

Regarding the model checking paradigm, it appeared in early 1980s as well with works of E. M. Clarke and E. A. Emerson [23, 25, 41], and of J. Sifakis and J. P. Queille [78], where model checking of temporal logic formulae was studied. First three of mentioned authors got a Turing Award in 2007 for their contribution in the field. An automata-theoretic approach for property checking was proposed in [105]. In the late 1980s the symbolic model checking (SMC) method was proposed [22, 28] and in 1992 it was implemented in a tool called SMV [71]. In 1990

Bryant and Seger adapted the symbolic simulation technique for the task [19]. At the end of the last century, the notions of bounded [9] and unbounded [87] model checking using a SAT solver were introduced.

2.4 DESIGN ERROR CORRECTION

In the previous sections we have seen a number of methods that are able to detect errors. A common denominator of these methods is their output, given in a form of counterexample, providing conditions in which the design under verification is incorrect. Two more steps are needed, bug localization and fixing. At the moment it is mostly manual tedious work. However, some automated methods also exist. Two classes of such methods can be distinguished: *resynthesis* and *error matching*. A method of the former class, given an implementation and some sort of information about its faulty behaviour (e.g. failing test vectors, property etc.) tries to determine the changes the system would require to work correctly under those conditions, that caused faulty responses. The latter methods suggest fix candidates according to some heuristic rules and check them against some sort of a "golden model". Below we describe two representatives of these classes and after that briefly describe other ones.

2.4.1 *Resynthesis*

In 2005 Smith et al. presented a method of fixing multiple design errors in a gate-level netlist using SAT [93]. Consider we have a gate-level netlist $\mathcal{C}$ containing AND, OR, NAND, NOR, XOR, XNOR and NOT gates. In addition to this it can contain fault-free memory elements (D flip-flops) that can be reliably reset. We have simulated our design and some of the test vectors has failed. In case of combinational circuit let $V_{\mathcal{C}} = \{v_1, ...v_k\}$ be the set of failing test vectors. For sequential designs we might obtain a set $V_{\mathcal{S}} = \{V_1, ..., V_k\}$ of test sequences, where each sequence V_j consists of m_j consecutively simulated vectors $(v_{j1}, \ldots, v_{jm_j})$.

Let our circuit contain r primary inputs $X = (x_1, \ldots, x_r)$ and t primary outputs $Y = (y_1, \ldots, y_t) = f(X)$. In case of sequential circuits with initial state $Q_I = (q_1, \ldots, q_u)$ we have $Y = f(X, Q_I)$. Let $L = \{l_1, \ldots, l_n\}$ be the set of all circuit lines including stems and branches. We add a 2-to-1 multiplexer per each line l_i, and connect l_i to its 0-input and two new lines, w_i connected to its 1-input, and s_i carrying the control signal. Thus we have obtained two new sets of lines, $W = \{w_1, \ldots, w_n\}$ and $S = \{s_1, \ldots, s_n\}$. Consider we are running the test vector v_j. We denote the values of the sets X, Y, L and W at this vector by X^j, Y^j, L^j

and V^j, and for the single bits of these sets we put x_i^j, y_i^j, etc. Again, in the sequential case we use notations X^{jm} and x_i^{jm} for the input and similar ones for the other sets when we execute a test vector v_{jm}, where $m = 1, \ldots, m_j$. However, the values of control signals remain unchanged for all vectors, so $S = \{s_1, \ldots, s_n\}$ is used to indicate both variables and line names.

Our goal is to translate the problem of correcting system behavior for $V_{\mathcal{C}}$ or $V_{\mathcal{S}}$ into a CNF formula $\Phi_{\mathcal{C}}$ or $\Phi_{\mathcal{S}}$, respectively, an pass it to a SAT solver. Consider the combinational case. For each test vector v_j we must obtain the values $y_1^j, \ldots y_t^j$ at the primary outputs, stimulating inputs with $x_1^j, \ldots, x_r^j$. In order to do so, we must replace some of the intermediate signals l_i^j by correcting values w_i^j. We do not know, how much signals we need to correct, so we are trying every value from 1 to some arbitrarily chosen N.

Now we can produce a CNF that summarizes these observations,

$$\Phi_{\mathcal{C}} = E_p(S) \wedge \bigwedge_{j=1}^{k} \left(C^j(L^j, W^j, X^j, Y^j, S) \wedge x_1^j \wedge \ldots \wedge x_r^j \wedge y_1^j \wedge \ldots \wedge y_t^j \right)$$

Here, $C^j(L^j, W^j, X^j, Y^j, S)$ models the circuit behavior with all added multiplexers for the test vector v_j [2], and $E_p(S)$ is a cardinality constraint instructing the SAT solver to replace exactly p lines $l_{i_1}^j, \ldots l_{i_p}^j$ by values $w_{i_1}^j, \ldots, w_{i_p}^j$. E.g., $E_p(S)$ may be a circuit implementation of the predicate $s_1 + \ldots + s_n = p$, translated to CNF.

The formula $\Phi_{\mathcal{S}}$ for the sequential circuit $\mathcal{S}$ is the following

$$\Phi_{\mathcal{S}} = E_p(S) \wedge \bigwedge_{j=1}^{k} \bigwedge_{m=1}^{m_j} \left(C^{jm} \wedge \bigwedge_{i=1}^{r} x_i^{jm} \wedge \bigwedge_{i=1}^{u} q_i \wedge \bigwedge_{i=1}^{t} y_i^{jm} \right)$$

Here everything is similar to the sequential case except the expression $C^{jm} = C^{jm}(L^{jm}, W^{jm}, X^{jm}, Q_I, Y^{jm}, S)$. The difference is that now we are unrolling $\mathcal{S}$ in time, replacing it with m_j combinational circuits connected by its state signals and then we are able to convert the new combinational circuit to CNF. Regarding the signals S, now each signal s_i is branched to connect m_j corresponding multiplexers in order to fix each of the lines l_i^{jm} simultaneously.

Thus, we have obtained formulae $\Phi_{\mathcal{C}}$ or $\Phi_{\mathcal{S}}$. We pass it to the SAT solver and if it is satisfiable (i.e. cardinality constraint is properly chosen), we get the values s_i indicating where we should fix our circuit and w_i^j indicating by what we should replace the wrong signals.

2 in simple words, C^j is just a CNF representation of the circuit functionality with new hardware

2.4.2 *Error Matching*

The method presented in previous subsection has disadvantages that are not easy to overcome. First of all, it relies on SAT solvers which do not scale that well we would desire to, thus, modern designs with millions of gates are hard to handle. Moreover, nowadays nobody designs at the gate level, so bugs mostly appear at least at the RT level. Assume, a designer has put + instead of * and we have discovered it with a couple of test vectors. But how much test vectors do we need to turn addition back into multiplication? Maybe it is easier just to replace addition by other operators, check the same vectors, tell the designer that the desired output would be obtained if we replace "+" by "*", and let him decide, whether to accept this solution or not.

With these simple observations we have come to the idea of automated debug by *error matching*. The process of replacing design elements by other ones is called *mutation* and these new elements are called *mutants*. Moreover, the idea is easily applicable not only to hardware designs but to software programs as well. So, this approach looks pretty simple until you start thinking about its implementation, when you realize that it is too time consuming to replace everything by everything and we need to put some limits to the method.

Two simple observations help us to make the method more applicable in practice.

- It is not needed to check every design element, but the ones where the error most likely appear; so, we rank these elements by some heuristically computed suspiciousness and start to mutate the ones that got higher scores. E.g., to rank the candidates we might do the following. During the simulation we run several tests, some of them are passed, some of them not. Each test executes only some subset of design, reducing the search space to covered code lines. Assume that the code line l was executed in t_f failed and t_p passed test. Then its suspiciousness rank is computed by the following formula
$$r_l = \frac{t_f}{t_f + t_p}$$
- We should not try every mutant but only those that that more likely would fix the design. It means, that if we have a "+" operator put by mistake, then the designer probably wanted to perform some arithmetic computations and replacing it by bitwise AND would not help, so it is better to try other arithmetic operators. Of course, one can think of cases where it is worth to mutate arithmetic operators by logical ones or by other design elements, but such situations are not so often in practice. E.g., in [79] authors define

10 different groups of mutation operators and fix more than half of errors from the Siemens benchmark suite [91].

Note, that the last limitation make the method inapplicable, for example, when the error is hidden inside logical structure of the program, when some complex conditions are separated among different if-then-else operators, switch statements, etc.

A kind of error matching approach, that does not rely on simulation was presented in [60]. Erroneous behavior is identified via failing assertions, or, alternatively, a reference implementation might be given. A program is viewed as a set of components, which are executed depending on logical conditions. Components are assignments of the form $left\text{-}hand\text{-}side$ = $right\text{-}hand\text{-}side$, where $right\text{-}hand\text{-}side$ can contain errors. In order to cover bugs in logical conditions, new temporary Boolean variables are introduced, one per each condition. Then, new assignments of the form $temp\_var = condition$ are inserted before checking the condition, which itself is substituted by $temp\_var$ in the actual check.

Then the smallest set of components that behave abnormally, called a *diagnosis* is computed, using a method presented in [82]. In general case this procedure is undecidable, as the conditions, describing normal and abnormal behaviors of a component involve quantifiers. In order to overcome this limitation, concrete input values, for whose the specification is violated, are computed using an SMT solver, and the diagnosis is produced first for these inputs. This approach is unsound as it may return false positives – the sets that are not a diagnosis for the whole program. In this case just another diagnosis candidate need to be computed.

Once the diagnosis is found, the components it contains need to be repaired. This is done by substituting their right-hand sides by templates with unknown parameters and letting an SMT solver compute values of these parameters. E.g., a template can be of a kind $k_0 + k_1 v_1 + k_2 v_2$ where v_1 and v_2 are program variables and k_0, k_1 and k_2 are parameters. This template defines a fix as a linear combination over the program's variables. Generally, simple templates are tried first, and if they fail to repair the program, more complex ones are attempted. Such usage of templates allows us to classify this method as the error matching one.

2.4.3 *Related Works*

Automated design error correction methods for the logic level have been developed already decades ago. The error matching approaches [97, 106, 107] are of little practical use when systems fail due to error cases which are not covered by the

model. Resynthesis approaches [51, 70, 93] are more general and do not depend to the same extent on the types of errors. However, they are relying on given input stimuli and partial truth tables of the components to be corrected. Thus, the resynthesis "fixes" the bug only with respect to the input stimuli and potentially replaces it by a more "difficult" one. In general, the logic-level approaches are not capable of coping with larger digital systems due to their complexity.

Recent works have extended the logic-level methods to the Register-Transfer Level (RTL). In [21], Chang et al. propose a resynthesis approach at RTL. An error matching approach based on mutation operators has been developed by Debroy and Wong for correcting Java and C programs [34]. A common denominator of all these methods is their dependence on input stimuli and therefore, the resulting corrections hold for the given set of stimuli but not necessarily for all possible inputs.

During the last few years, the progress of SAT/SMT solvers inspired the appearance of fully formal error correction methods that do not rely on simulation. E.g., in an error-matching-type method from [60] authors obtain the information about erroneous behavior from failed assertions or a reference implementation; in a resynthesis-type method from [45] authors fix errors with respect to a reference implementation using the verification tool UCLID [63].

2.5 SUMMARY

The aim of this chapter was to give a reader an overview of related fields in verification and automated error correction, required for understanding the thesis domain and contribution. The evolution of the methods along with the basic principles and applications was presented. We started the chapter with the brief overview of simulation techniques as the most popular and wide-spread verification approach. Then we discussed SAT and SMT solvers, their architectures and the related theory. After that we presented some state-of-the-art formal verification techniques. The last section was about automated methods of error diagnosis and correction.

3

DECISION DIAGRAMS

In this chapter we introduce High-Level Decision Diagrams (HLDD) – the main mathematical model we use to represent digital systems at higher levels. But before we do it, we give a short overview of other types of decision diagrams. Thus, the first part of this chapter may be considered as a continuation of the previous background chapter.

We start with the binary case, presenting Binary Decision Diagrams (BDDs) as a way to represent propositional expressions. In Section 3.1 we give a general definition and in Subsection 3.1.1 and 3.1.2 we discuss some of its subtypes: Reduced Ordered Binary Decision Diagrams (ROBDD) and Structurally Synthesized Binary Decision Diagrams (SSBDD), respectively. Next, in Section 3.2 we switch to the higher level extensions of BDDs that is mostly used to operate word-level arithmetic and logic expressions by presenting a definition and some examples for some selected data structures. After that we continue with HLDDs. We define this structure, summarize some of its notable properties, and finish the section by presenting the methods of generating HLDDs from various types of hardware descriptions. Some of the results presented in this section were published [56, 99]. Section 3.4 provides an overview of notable works on discussed topics and, finally, Section 3.5 concludes the chapter.

3.1 BINARY DECISION DIAGRAMS

Decision Diagrams serve as a basis for various task related to digital systems design and test. We start our chapter with the discussion on their most famous representative, BDDs, which are now extensively used in various fields related to hardware design and test. For example, in the previous chapter we mentioned that BDDs (more precisely ROBDDs) can serve as a replacement of SAT solvers in equivalence checking task. Other examples of their usage can be found in Section 3.4.

A BDD which represents a Boolean function is a directed acyclic graph with a single root node, where all nonterminal nodes are labeled by Boolean vari-

ables (arguments of the function) and have always exactly two successor-nodes whereas all terminal nodes are labeled by constants 0 or 1. For all non-terminal nodes, a one-to-one correspondence exists between the values of the label variable of the node and the successors of the node. This correspondence is determined by the Boolean function inherent to the graph. Let us continue with the formal definitions of BDD.

Definition 3.1. *A BDD that represents a Boolean function $y = f(X)$, $X = \{x_1, x_2, \ldots, x_n\}$, is a directed acyclic graph $G_y = (V, E)$ with a single root node $v_0 \in V$, and two terminal nodes, v_0^T and v_1^T, where:*

- *Each non-terminal node is labeled by some variable $x \in X$. We denote the variable of node v by x_v.*
- *Terminal node v_0^T is labeled by constant 0, v_1^T is labeled by 1.*
- *Each non-terminal node has exactly two outgoing edges, labeled by 0 and 1, respectively. We denote an edge from vertex u to v by (u, v, c) where $c \in \{0, 1\}$.*

$\Gamma(v) \in V$ denotes the set of successor nodes of $v \in V$, $V = V^N \cup V^T$ consists of two types of nodes: non-terminal V^N and two terminal V^T nodes, or leafs. The designations v^0 and v^1 stand for the successors of vertex v for the values $x_v = 0$ and $x_v = 1$, respectively.

Definition 3.2. *For the value of $x_v = c$, $c \in \{0, 1\}$, we say the edge between nodes $v \in V$ and $v^c \in V$ is activated. Consider a situation where all the variables $x \in X$ are assigned by a Boolean vector $\alpha = (\alpha_1, \ldots, \alpha_n) \in B^n$ to some value. The edges activated by this vector form an activated path $l(v_0, v^T) \subset V$ from the root node v_0 to one of the terminal nodes $v^T \in V^T$.*

We say that a BDD $G_y = (V, E)$ represents a Boolean function $y = f(X)$, iff for all the possible vectors $\alpha = (\alpha_1, \ldots, \alpha_n) \in B^n$ a path $l(v_0, v_c^T) \in V$ is activated so that $y = f(\alpha) = c$, where $c \in \{0, 1\}$.

3.1.1 *Reduced Ordered Binary Decision Diagrams*

The most famous subclass of BDDs are *Reduced Ordered Binary Decision Diagrams (ROBDDs)*. Comparing to other decision diagram types, ROBDDs are so widely used that most of researchers talking about BDDs mean this particular subclass.

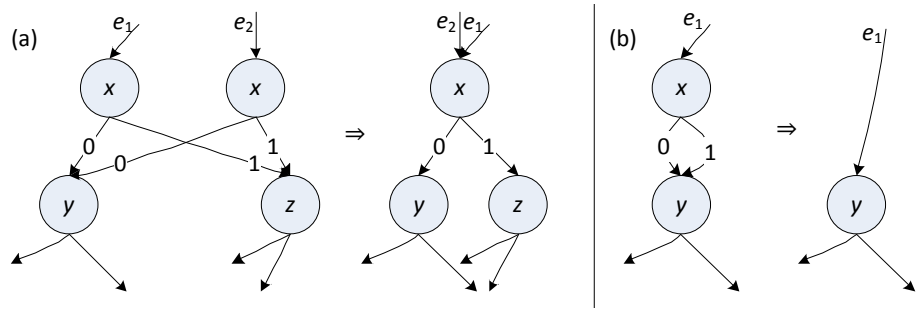

Figure 3.1 – Reduction operations: (a) Merge, (b) Eliminate

Definition 3.3. *A BDD representing Boolean function $f(X)$ is ordered, if for any two variables $x_i, x_j \in X$, $1 \leq i, j \leq |X|$ a node labeled by x_i precedes a node labeled x_j in some path $l(v_0, v^T)$ then in every other path containing vertices labeled by the same variables x_i precedes x_j.*

Assume we have 2 nodes v and w, such that $x_v = x_w$, $v^0 = w^0$ and $v^1 = w^1$. Then we can remove either v or w, redirecting its incoming edges to the remaining one. Such operation is called *merge* (Figure 3.1a). Assume now that we have 2 nodes, t and u such that $t^0 = t^1 = u$. Then we can remove t redirecting its incoming edges to u. This operation is called *eliminate* (Figure 3.1b).

Definition 3.4. *An OBDD is reduced if we cannot apply operations "merge" and "eliminate" any more.*

The key property of ROBDD, the reason why it is so appreciated is that this is a canonic representation of a Boolean function. That means that once we have fixed a variable order, we would obtain the same diagram for the same function regardless how it does look like initially.

To build an ROBDD from a propositional expression we need to do the following. As in the DPLL algorithm from the previous chapter, the basic idea is the Shannon expansion:

$$f(x_1, \ldots, x_n) = x_1 \wedge f_{x_1} \vee \overline{x_1} \wedge f_{\overline{x_1}}$$

However, now we need to keep the variable order, which means, e.g., if for f_{x_1} we choose x_2 as the next variable for expansion, we must choose the same variable expanding $f_{\overline{x_1}}$. So, for the subsequent variable x_i we create a new node, two outgoing edges and recursively apply this rule to $f_{\ldots x_i}$ and $f_{\overline{\ldots x_i}}$. If we apply this procedure until we use all the variables then we get a binary tree, which requires exponential size, instead of desired ROBDD. However, we can employ the following observation: let σ_1 and σ_2 be two sequences of literals, such that $f_{\sigma_1} = f_{\sigma_2}$

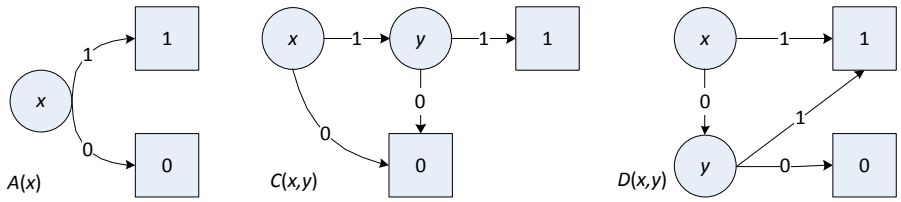

Figure 3.2 – Three basic SSBDDs

and we have already built a BDD for f_{σ_1}, then instead of calling recursion for f_{σ_2} direct its incoming edges to the diagram corresponding to f_{σ_1}.[1]

Unfortunately, this observation does not always help and sometimes we end up with the diagram of exponential size. There are functions, which produce BDDs of polynomial size. Classic example is symmetric Boolean functions, the ones whose values does not depend on the permutations of variables (thus the value of such function is the same for all assignments that contain equal number of zeroes and ones). There are, however, cases that always result in exponential BDDs. At least one of the BDDs corresponding to the outputs of n-bit logic multiplier has size $O(2^n)$. Most of the functions lie in between of these two extreme cases, which means that there is a variable ordering that would produce a compact diagram. The problem of finding the best ordering is **NP**-hard. Even a more relaxed version of this problem, for some $c > 1$ find an ordering that produce a diagram at most c times larger than the optimal one is **NP**-hard.

3.1.2 *Structurally Synthesized Binary Decision Diagrams*

Structurally Synthesized Binary Decision Diagrams (SSBDDs) form another special subclass of BDDs. Historically, this data structure is a predecessor of HLDDs, which were invented as a generalization of SSBDDs for higher abstraction levels. SSBDDs are used for test generation, fault simulation, design error diagnosis, etc.

Three basic diagrams, $A(x)$, $C(x,y)$ and $D(x,y)$, corresponding to Boolean functions $f_1(x) = x$, $f_2(x,y) = x \wedge y$ and $f_3(x,y) = x \vee y$, respectively, are depicted in Figure 3.2. In SSBDDs it is allowed to label a node not only by a variable but by its negation as well, i.e. any single literal can be a node label. Generally, SSBDD is a graph generated from an arbitrary Boolean formula f and graphs A, C, D by applying the following algorithm, called a *superposition procedure*.

1 This technique is called dynamic programming.

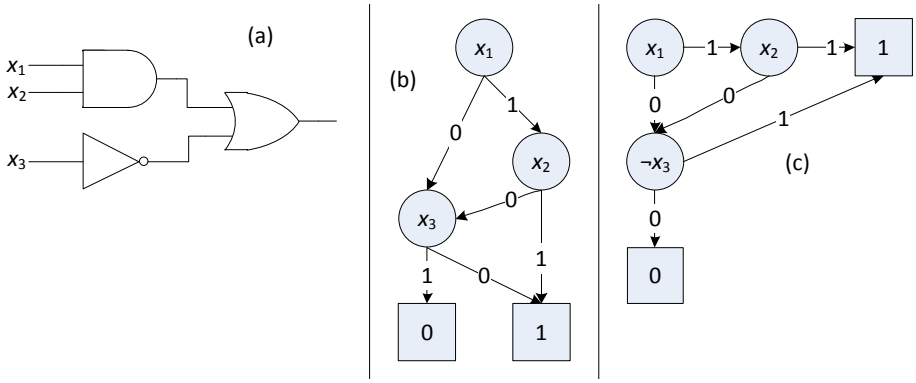

Figure 3.3 – Three representations of $x_1 \wedge x_2 \vee \overline{x_3}$: (a) Gate-level circuit, (b) ROBDD, (c) SSBDD

1. Convert f to *Negation Normal Form (NNF)* – a representation of Boolean formula where only $\vee, \wedge, \neg$ and parenthesis are allowed and negation of subformulas larger than single variable is prohibited.

2. If $f = l$ where l is single literal, return $A(l)$, if $f = l_1 \wedge l_2$ or $f = l_1 \vee l_2$, where l_1 and l_2 are literals, return $C(l_1, l_2)$ or $D(l_1, l_2)$ respectively.

3. If $f = g \wedge h$, where g and h are propositional expressions, then apply current procedure recursively to g and h and let these two recursive calls return us graphs G and H, respectively. Then in the graph $C(g, h)$ perform a *graph superposition* – replace the node g by the graph G in the following way:
   - Remove terminals in G.
   - Redirect all incoming edges of 1-terminal in G to $g^1 = h$.
   - Redirect all incoming edges of 0-terminal in G to $g^0 = 0$.
   - Remove g.

   After that do the same for the node h and the graph H.

4. If $f = g \vee h$ where g and h are propositional expressions, then, again, call the current procedure recursively for g and h and perform the graph superposition for $D(g, h)$ and diagrams G and H.

An example of different representations of a Boolean function is given in Figure 3.3.

Note, that not every path in SSBDD can be activated by a particular variables' assignment. Two nodes labeled by the same variable can appear in the path,

which means that once we moved towards 1-edge for the first node, we cannot choose 0-edge for the second one. A path that can be activated by an assignment of the variables is called *feasible*, otherwise it is *infeasible*.

3.2 WORD-LEVEL EXTENSIONS

In this section we give a short overview of different types of word-level diagrams. As we have seen in Subsection 3.1.1, ROBDDs fail to represent outputs of an n-bit multiplier in a compact way. On the other hand, modern systems contain a lot of different arithmetic circuits, which need to be verified. To address this issue, several data structures, inspired by BDDs, but treating the functions in different ways, were proposed. An n-bit arithmetic circuit is usually modeled as a function $f : B^n \to \mathbb{Z}$. All of the presented data structures are canonical with respect to particular variable ordering.

*Multiplicative Binary Moment Diagrams (*BMD)* [18]. This is a directed acyclic graph with a single root node, where each non-terminal node is labeled by a Boolean variable, edges are labelled by pairs $< c_1, c_2 >$, where $c_1 \in B$ and $c_2 \in \mathbb{Z}$, and terminal nodes by integer constants. There can be any number of terminals. *BMD is based on Boole-Shannon expansion: $f = (1 - x)f_{\overline{x}} + xf_x$, rearranged in the following way: $f = f_{\overline{x}} + x(f_x - f_{\overline{x}})$. Note, that if we expand f in this way to the end, we obtain a polynomial, where maximum degree of a single variable is 1. To build a *BMD from polynomial $P : B^n \to \mathbb{Z}$ we can apply the following recursive procedure.

1. Create node labeled by x, the first variable in the order.
2. Create two outgoing edges, labeled by $< 0, c_0 >$ and $< 1, c_1 >$, where c_0 and c_1 are the greatest common divisors of the coefficients in $P_{\overline{x}}$ and P_x respectively.
3. If we have already built a *BMD for $P_{\overline{x}}$ then link the edge $< 0, c_0 >$ with this diagram (analogue of the merge operation for ROBDDs). Otherwise recursively create the *BMD for $P_{\overline{x}}$.
4. Repeat the last step for $< 1, c_1 >$ and P_x.

Edge-Valued Binary Decision Diagrams (EVBDD) [64]. The basis of this structure is the following decomposition $f = c + x(f_x + c_x) + (1 - x)f_{\overline{x}}$, where c and c_x are integer constants. So, an EVBDD is a tuple (c, f) where c is a constant value and f is a directed acyclic graph consisting of two types of nodes:

- There is a single terminal node with value 0 (denoted by 0).

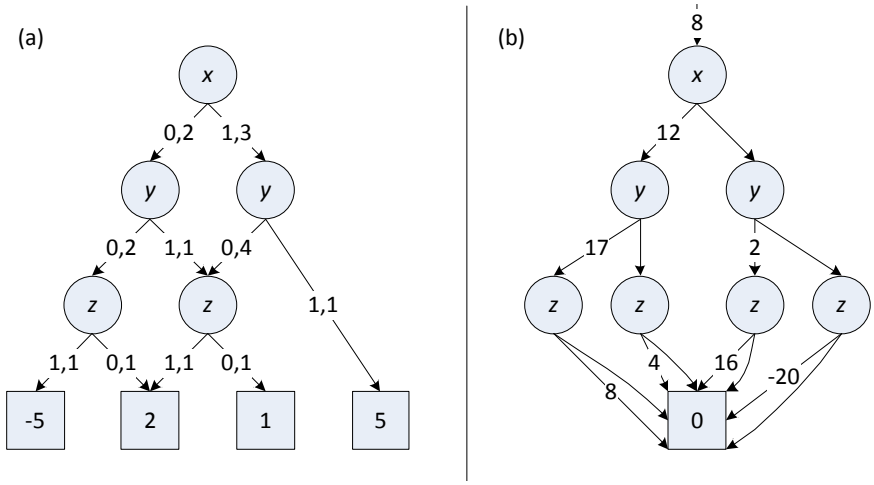

Figure 3.4 – Examples of (a) *BMD and (b) EVBDD. Both represent the function $8 - 20z + 2y + 4yz + 12x + 24xz + 15xy$.

- A non-terminal node v is a 4-tuple (x, l, r, c_x) where x is a binary variable, l is a left child corresponding to f_x, r is a right child, corresponding to $f_{\overline{x}}$, and c_x is an integer constant.

Constant c is usually represented in a graph as *dangling* edge to the root node. An EVBDD is reduced if there is no non-terminal node v with $l = r$ and $c_x = 0$, and there are no two non-terminal nodes u and v such that $u = v$.

Multiple-valued Decision Diagrams (MDD) [72]. This is a direct extension of BDDs to the case of n-valued logic, so each node has n outgoing edges and there are n terminals. Similar observations regarding variable ordering and reductions of redundancies can lead us to the notion of Reduced Ordered MDDs, which is a canonical representation of a function $f : \{0, 1, \ldots, n-1\}^k \to \{0, 1, \ldots, n-1\}$, just like an ROBDD is a canonical representation of a Boolean function.

3.3 HIGH-LEVEL DECISION DIAGRAMS

BDDs and Word-level DDs are good in representing different components of a system, but often we need to describe the system as a whole at higher levels, like behavioral or RTL. For this purpose High-Level Decision Diagrams were introduced [101, 102]. Until now this data structure was successfully applied in many areas related to hardware design and test (see Section 3.4), but formal verification was not among them. In current work we obviate this disadvantage by applying it for equivalence checking and error correction.

Consider a digital system (Z, F) as a network of subsystems or components where Z is the set of variables (Boolean, Boolean vectors or integers), which represent connections between components, inputs and outputs of the network. Let $Z = X \cup Y$, where X is the set of function arguments and Y is the set of function values where $Q = X \cap Y$ is the set of state variables. $D(z)$ denotes the finite set of all possible values for $z \in Z$ and $D(Z')$ is the set of all possible vectors for all $Z' \subseteq Z$. Obviously, if $Z' = \{z'_1, \ldots, z'_n\}$ then $D(Z') = D(z'_1) \times \ldots \times D(z'_n)$. Let F be a set of discrete functions: $y_k = f_k(X_k)$ where $y_k \in Y$, $f_k \in F$, and $X_k \subseteq X$ (k iterates over all elements in F).

Definition 3.5. *The high-level decision diagram representing the function $f_k : D(X_k) \to D(y_k)$ is a directed acyclic multigraph $G = (V, E)$ with one root node and a set of terminal nodes where:*

- *Each non-terminal node is labeled by some input or control variable $x \in X$.* [2] *We shall denote the variable of node v by x_v.*
- *Each terminal node w is labeled by some function $g_w : D(X_w) \to D(y_k)$ (possibly a constant or a single variable) where $X_w \subseteq X_k$.*
- *Each edge e from node v to u is labeled by some constant $c \in D(x_v)$. We denote such edge by (v, u, c).*
- *Each two edges $e_1 = (v, u_1, c_1)$ and $e_2 = (v, u_2, c_2)$ going from the same source node are labeled by different constants $c_1 \neq c_2$.*
- *If the node v is labeled by x_v then the number of edges going from this node is $|D(x_v)|$.*

In simple words, HLDD is a data structure similar to BDD, but with many edges originating from a particular node and a number of functions at the end, instead of constants 0 and 1. We shall denote the set of terminal nodes by V^T, the set of non-terminal nodes by V^N and the set of all successors of the node v by $\Gamma(v)$. For non-terminal nodes $v \in V^N$ an onto function exists between the values $c \in D(x_v)$ of labels x_v and the successors $v^c \in \Gamma(v)$ of v. By v^c we denote the successor of v for the value $x_v = c$. The edge (v, v^c, c), which connects nodes v and v^c, is called *activated* iff there exists an assignment $x_v = c$. Activated edges, which connect v_i and v_j, make up an *activated path* $l(v_i, v_j) \subseteq V$. An activated path $l(v_0, v^T)$ from the root node v_0 to a terminal node v^T is called *full activated path* and v^T itself is *activated terminal node*. Without loss of generality we assume further that each variable has at least two values, i.e. $\forall z \in Z,\ |D(z)| > 1$.

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

Remark 3.1. *Each BDD is an HLDD as well, with two terminal vertices labeled by constant functions* 0 *and* 1*, and* $D(x) = \{0, 1\}$ *for every variable* x*.*

The nodes of HLDD may represent different high-level functional components of a system. In RTL description we usually partition the system into control and data parts. Non-terminal nodes in HLDDs correspond to control paths and they are labeled by control variables or logical conditions, whereas terminal nodes correspond to data paths, and they are labeled by the data or functions on data.

Before we can apply HLDDs for our purposes we need to obtain them from other types of hardware descriptions. We present here HLDD generation algorithms from the following hardware description types: HDL source code, network of components [99] and microprocessor instruction sets [104].

3.3.1 *Synthesis of HLDDs from Procedural Descriptions*

Consider a procedure representing a behavior level description of a digital system. The procedure can be represented by a flow graph which is a directed graph, and a path can be defined as in graph theory. A path can be represented by a sequence of assignment statements and conditional expressions (i.e. assertions). A path is executable (or feasible) if there are input data such that the program is executed along that path. Otherwise, it is un-executable (or infeasible). For deciding the feasibility of program paths, symbolic execution of programs and constraint solving techniques are used [109]. In this paper, a procedure similar to symbolic execution is used for HLDD synthesis.

In the following we assume that all the assignment statements correspond to updating of the corresponding data-path registers during a current cycle. Based on that we introduce a "simplified" cycle-based symbolic execution procedure.

First, the states are inserted into the flow graph of the procedure under analysis in a similar way as the algorithmic state machines are marked by states during synthesis of FSM [30]. For example, in the graph in Figure 3.5 there are six states $s_0, s_1, \ldots, s_5$, inserted in such a way that during the state transfer each data variable can be changed only once. A global state variable q is introduced to map the states to values of q. For example, $g = i$ for the state s_i.

As a result of cycle based symbolic execution the path trees are created only for the restricted regions between neighboring states, avoiding in this way the explosion of paths. As a result of tracing all the transfers in the flow graph a table is constructed where each row corresponds to a path between neighboring states of the procedure. An example of the result of cycle based symbolic execution of the procedure in Figure 3.5 is presented in Table 3.1.

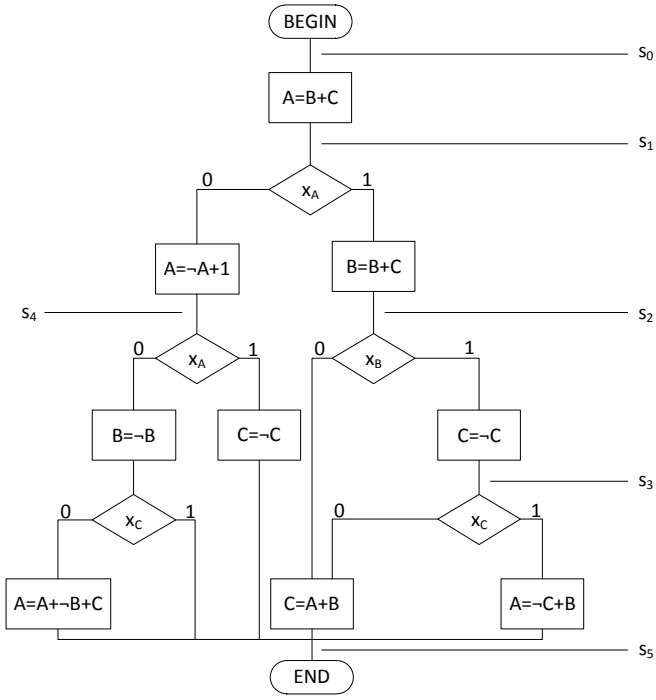

Figure 3.5 – A procedure under symbolic execution

Table 3.1 – Results of symbolic execution

Constraints				Assignment statements
q	x_A	x_B	x_C	
0				$A = B + C;\ q = 1$
1	0			$A = \neg A + 1;\ q = 4$
1	1			$B = B + C;\ q = 2$
2		0		$C = A + B;\ q = 5$
2		1		$C = \neg C;\ q = 3$
3			0	$C = A + B;\ q = 5$
3			1	$A = \neg C + B;\ q = 5$
4	0			$B = \neg B$
4	0		0	$A = A + B + C;\ q = 5$
4	0		1	$q = 5$
4	1			$C = \neg C;\ q = 5$

Table 3.2 – Results of symbolic execution for variable A

q	x_A	x_B	x_C	A
0				$B + C$
1	0			$\neg A + 1$
3			1	$\neg C + B$
4	0		0	$A + B + C$

For all the left-hand side variables A, B, C and q in Table 3.1 we create HLDDs which describe the cycle-based behavior of these variables during execution of the procedure in Figure 3.5.

Consider a table created as the result of symbolic execution as a set of tuples $N = N_i, N_i = (Ci, Si)$ where C_i is a logical condition (a set of constraints), and S_i is a set of assignment statements. Each assignment $s \in S_i$ in a form $z_{i,s} = E_{i,s}$, where $z_{i,s}$ is a variable and $E_{i,s}$ is an algebraic expression, will be fulfilled iff the set of constraints C_i is satisfied. By collecting all the assignments s from N for a left-hand variable z we can represent the behavior of the variable z as

$$z = \vee C_i E_{i,s} \tag{3.1}$$

where C_i may have a logic value 0 or 1. It is easy to realize that the set of all constraints C_i in N satisfies the requirements for orthogonality and completeness. In Table 3.2 the behavior of the variable A is explicitly highlighted. Based on the Table 3.2 the formula (3.1) for the data variable A can be derived as

$$\begin{aligned} A \;=\; & (q = 0)(B + C) \vee (q = 1)(x_A = 0)(\neg A + 1) \vee \\ & (q = 3)(x_C = 1)(\neg C + B) \vee (q = 4)(x_A = 0)(x_C = 0)(A + B + C) \end{aligned}$$

From the formula (3.1), a HLDD for the variable y can be derived in a similar way as BDDs are derived by using Shannon factorization. The only difference is that instead of Boolean factorization we may use multi-valued factorization, depending on the possible number of values of the constraint variable.

The HLDDs created by factorization of the formulas (3.1) for all the four variables A, B, C and q are depicted in Figure 3.6. Here, z' denotes the previous value of the variable $z \in Z$.

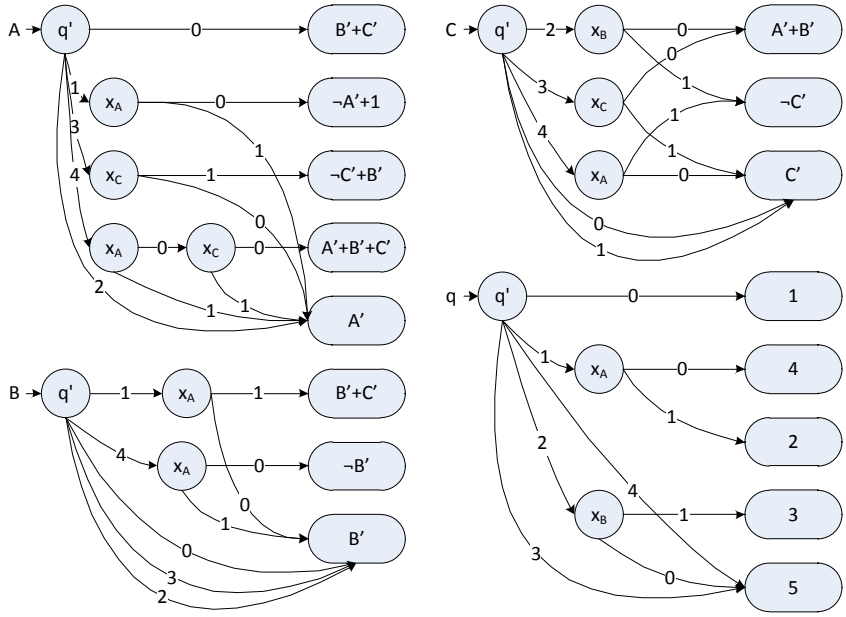

Figure 3.6 – HLDDs for the procedure in Figure 3.5.

3.3.2 *Synthesis of HLDDs by Iterative Superposition*

The description of a system is usually given as a network of components described in a hardware description language. HDLs can be classified into procedural and non-procedural languages. In the procedural case for each procedure (e.g. process in VHDL) the HLDD model can be created as in the previous section. In non-procedural cases we can create the formulas (3.1) as the basis for HLDDs in a more straightforward way without symbolic execution, since the pairs of constraints and assignments are then given directly. In many cases where the system is described as a network of components or subsystems represented already as a set of HLDDs, the whole model can be further compressed by iterative superposition in a similar way as SSBDDs are created.

Let us have a set of HLDDs $G = G_k$ where each is representing a digital function $z_k = f_k(Z_k)$. If a terminal node v_0 in a graph G_k is labeled by a data variable x_{v_0} which is represented by another graph $G(x_{v_0})$ then the procedure is trivial: the node v_0 in G_k can be simply substituted by the graph $G(x_{v_0})$.

Example 3.1. Consider a data path in Figure 3.7 and the descriptions of the operations of the components in Table 3.3. Variables R_1 and R_2 represent registers, *IN* represents the input bus, integer variables y_1, y_2, y_3, y_4 represent control signals, M_1, M_2, M_3 are multiplexers, and the functions $R_1 + R_2$ and $R_1 \cdot R_2$

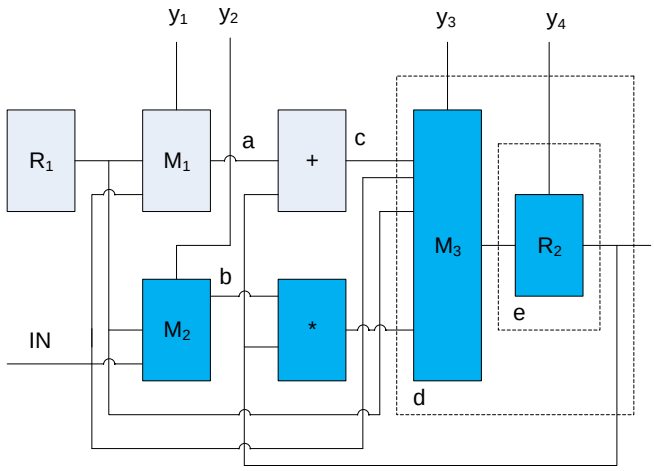

Figure 3.7 – RTL data path of a digital system

Table 3.3 – Functions of the blocks of system in Figure 3.7.

y_1	M_1/a	y_2	M_2/b	y_3	M_3/e	y_4	R_2
0	R_1	0	R_1	0	$M_1 + R_2$	0	0
1	IN	1	IN	1	IN	1	R_2
				2	R_1	2	M_3
				3	$M_2 \cdot R_2$		

represent adder and multiplier, respectively. The superposition procedure of two graphs for the register input logic R_2 and the multiplexer M_3 (internal variable e) is illustrated in Figure 3.8, and the whole single fully compressed HLDD for the data-path in Figure 3.7 is depicted in Figure 3.9.

Each node in the HLDD represents a subcircuit of the system (e.g. the nodes y_1, y_2, y_3, y_4 represent multiplexers and decoders). The bold path in Figure 3.9 shows the active mode of the system for input control vector $(y_1, y_2, y_3, y_4) = (1, 0, 3, 2)$, which means that during this clock cycle the system calculates the multiplication $R_2 = R_1 \cdot R_2$. The structural relationships between the HLDD and the original system are highlighted by dotted lines in Figure 3.9.

Superposition of a non-terminal node v labeled by x_v in G_k by the graph $G(x_v)$ is only then possible when all the values $c \in D(x_v)$ have one-to-one mapping to the terminal nodes of $G(x_v)$ labeled by constant values from the same set $D(x_v)$. The procedure is as follows:

1. The node v will be removed from G_k.

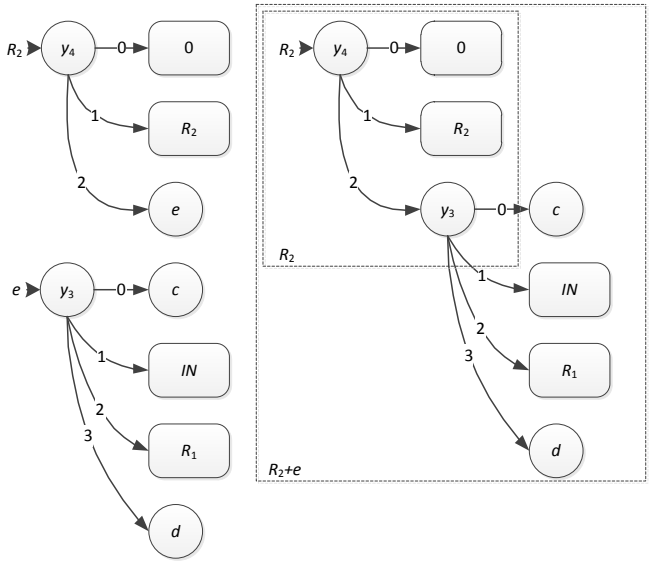

Figure 3.8 – Superposition of HLDDs.

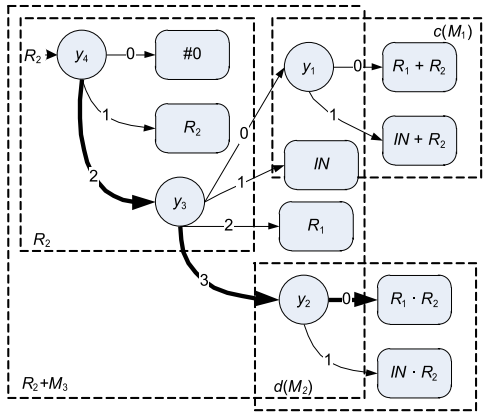

Figure 3.9 – Full HLDD model for the data path in Figure 3.7

2. All the edges in $G(x_v)$ that were connected to terminal nodes $v_c^T \in G(x_v)$ will be cut and then connected, correspondingly, to the successors v^c of the node v in G_k.

3. All the incoming edges of v in G_k will be now incoming edges for the root node v_0 in $G(x_v)$.

Note, that this procedure corresponds exactly (!) to the superposition procedure developed for SSBDDs with the only difference in the ranges of values for c (binary vs. integer).

Table 3.4 – Microprocessor instructions

#	Instruction	Behaviour	#	Instruction	Behaviour
I_1:	MVI A,D	A = IN	I_6:	MOV A,M	A = IN
I_2:	MOV R,A	R = A	I_7:	ADD R	A = A + R
I_3:	MOV M,R	OUT = R	I_8:	ORA R	A = A $\vee$ R
I_4:	MOV M,A	OUT = A	I_9:	ANA R	A = A $\wedge$ R
I_5:	MOV R,M	R = IN	I_{10}:	CMA A	A = $\neg$A

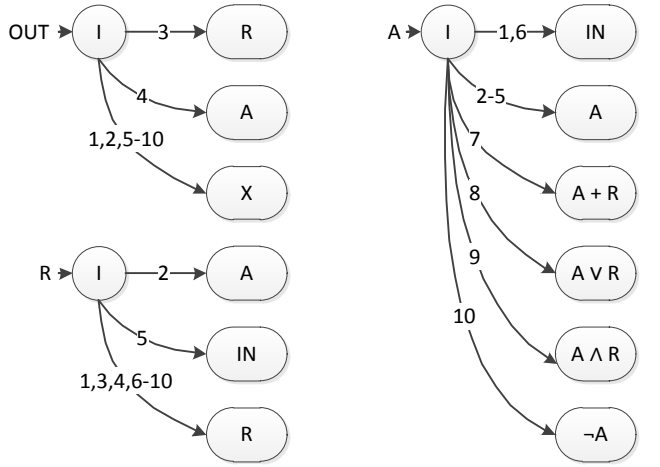

Figure 3.10 – HLDDs for the processor from Table 3.4

3.3.3 *Synthesis of HLDDs from Microprocessor Instructions*

Microprocessor instruction sets can be given as a set of 4-tuples (i, f, X, y) where i is instruction number, f is operation name, X is a list of operands and y is the result destination (register or memory cell).

Example 3.2. A simple instruction set of a hypothetical microprocessor is shown in Table 3.4. Here A and R are registers, M is a memory cell address, D is a constant.

Translating such instruction sets to HLDDs is fairly easy. Instruction number serves as a state variable and we create the set of HLDDs for each of the registers and for the output bus depending on the instruction behavior. Resulting DDs for the processor from Table 3.4 is shown in Figure 3.10

3.4 RELATED WORKS

In 1959 C.Y.Lee introduced a method for representing digital circuits by Binary Decision Programs [67]. In 1976 the same model called alternative graphs [100] was introduced for test generation purposes. Independently the same model was introduced into the field of test generation by Akers [2] under the name of Binary Decision Diagrams (BDD).

In 1986 Bryant in his seminal paper [81] proposed ROBDDs and showed how most Boolean algebra operations can be efficiently implemented on this structure. At the moment this paper is one of the most cited papers in computer science. Today the theory of BDDs is developing quickly with the main purpose of efficient manipulation of logic expressions [2, 35, 81]. Its application domain include digital circuit design and verification, sensitivity and probabilistic analysis of digital circuits, finite-state system analysis, etc. [17].

However, disadvantages of the proposed structure were revealed soon. In 1991 Bryant proved that at least one of the diagrams produced from the circuit implementation of integer multiplication is exponential in size with respect to the number of input bits, while the circuit itself has polynomial size [16]. Billig and Wegener in 1996 proved the **NP**-completeness of finding good variable orderings for building a BDD [12].

A number of alternative representations targeting mostly arithmetic functions have been proposed. In 1992 Lai and Sastry proposed EVBDDs [64], which was the first word-level data structure for representing functions $f : B^n \to \mathbb{Z}$. However, it still was not able to model multiplication efficiently. This problem was solved by Bryant and Chain in their paper [18] introducing *BMDs. Several extensions of this structure have been proposed since then, e.g. Hybrid Decision Diagrams (HDD) [24] and Kronecker multiplicative binary moment diagrams (K*BMD) [36]. A good survey on different word-level diagrams and their capabilities to model basic arithmetic functions can be found in [50].

In 1980s hardware design process moved from gate-level to RTL. In order to solve various tasks related to this process a structure, capable to represent systems at the same level they were designed, was desired. In 1988 High-Level Decision Diagrams (HLDD) were proposed by Ubar as such structure [101, 102]. Since then HLDDs were applied to high-level test generation [99], simulation [54], fault diagnosis [80], dependability evaluation [103], etc.

3.5 SUMMARY

The first part of this chapter continued to acquaint the reader with the background theory, focused this time on different decision diagrams. We discussed general BDDs, ROBDDs and SSBDDs. After that an overview of some representatives of word-level diagrams were given: *BMDs, EVBDDs and MDDs.

Then we proceeded with the section describing HLDDs – the basis of our further research. We gave a formal definition of this class of diagrams, which is the first contribution of the thesis. In the following few subsections we presented algorithms of compiling HDL sources, networks of components and microprocessor instruction sets into the sets of HLDDs.

A brief survey of related works concluded the chapter.

4

CHARACTERISTIC POLYNOMIALS

High-Level Decision Diagrams have been used in many fields related to VLSI design and test, but, unfortunately, formal verification is not among them, due to uncanonicity of the HLDD representation. This chapter addresses this issue developing a polynomial-based path analysis technique that is able to restore the lost canonicity for the control part of the graph. Main results of this chapter was published in [56], except the Section 4.4 which appeared in [58].

In Section 4.1 we formulate the problem and observe some initial approaches to solve it. In Section 4.2 we define characteristic polynomials: the main tool we use in this and the next chapters to solve equivalence checking and error correction problems. The next section provides an algorithm of obtaining polynomials from an HLDD, Section 4.5 is a short survey of similar works and Section 4.6 summarizes the chapter contents.

4.1 INITIAL OBSERVATIONS

HLDDs generated from different sources generally preserve the structure of the original description, which allows making them compact. However, variables in a path usually appear in the same order their corresponding modules or conditions appear in the source, so, they can be before or after each other in the same diagram. We could define similar transformations as for ROBDDs, but we must pay at least the same price – possible exponential blow-up. Is there some way to extract canonical properties of the underlying discrete function of the HLDD without changing the variable order?

Example 4.1. Consider the diagram from Example 3.1, shown in Figure 3.9. Suppose in some other system implementing the same functionality we have obtained different diagrams, like the ones depicted in Figures 4.1 - 4.3. Figure 4.1 shows the result of swapping around variables y_1 and y_3. In Figure 4.2 we introduce a new variable x. The subgraph after the edge $x \xrightarrow{0} y_3$ is the same as shown in the original diagram from Figure 3.9; the subgraph after $x \xrightarrow{1} y_1$ is the same

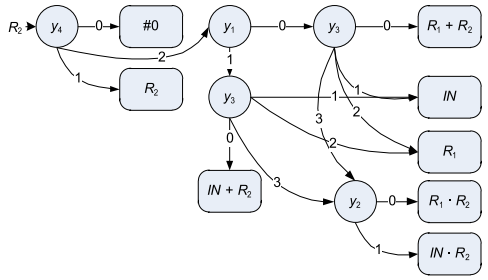

Figure 4.1 – Variables in different order

as in the previous diagram with reversed order of y_1 and y_3. Easy to see that in all these cases the actual function is the same.

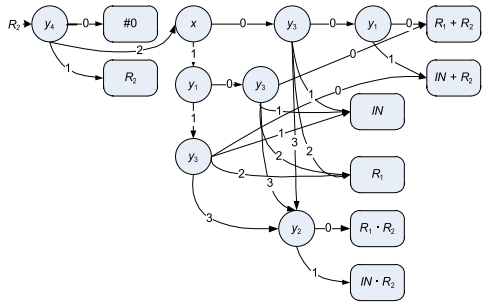

Figure 4.2 – Redundant node x

The case shown in Figure 4.3 is a bit more complicated, because here we see a function with another argument set. This means that the argument sets should be somehow normalized in such case. Later in Section 4.4 we show how to deal with this, but first let us try to solve a simpler case, when variables are the same

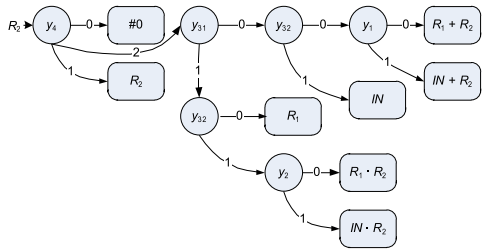

Figure 4.3 – Variable's decomposition

Consider again the digital system (Z, F), where $Z = X \cup Y$ represented by the set of HLDDs $\mathcal{G} = \{G_k : k = 1..|F|\}$. A function $f \in F$ is split by its diagram $G \in \mathcal{G}$ into two parts: a graph and terminal-node functions. To compute

$f(\alpha)$ for some variable assignment α we first activate a path to particular terminal node and then compute the final value using the terminal node function. Usually these terminal node functions are simple word-level arithmetic or logic expressions, so, building its canonic representation is generally not a problem. We might use structures like *BMD or EVBDD presented in the previous chapter. We can even choose different diagrams for different functions, depending on the diagram sizes for certain function types. Alternatively, we may consider comparing these functions with an SMT solver.

Thus, we need to describe the paths from source to terminals in a way it would be the same for all diagrams with the same functionality, like the ones we have seen in Figures 4.1 and 4.2.

So, let $f \in F$, $G = (V, E) \in \mathcal{G}$ is the HLDD representation of f and let D_i designate a subset of $D(X)$, such that assignments from it will activate the terminal node $v_i^T \in V^T \subset V$, so $D(X)$ is being partitioned to non-intersecting sets $D_1, ..., D_t$, where $t = |V^T|$. More formally,

$$\bigcup_{i=1}^{t} D_i = D(X) \bigwedge \forall i, j (i \neq j \Rightarrow D_i \cap D_j = \emptyset)$$

For each such subset D_i we define a characteristic function $\chi_i : D(X) \rightarrow \{0,1\}$, i.e. $\alpha \in D(X) \wedge \chi_i(\alpha_1, \ldots, \alpha_n) = 1 \Leftrightarrow (\alpha_1, \ldots, \alpha_n) \in D_i$. Our goal is to get an algebraic expression for f. Let $\alpha \in D(X)$ be some assignment to input and control variables and $\chi_i : D(X) \rightarrow \{0,1\}$ be the characteristic function of the set D_i, i.e. $\chi_i(\alpha_1, \ldots, \alpha_n) = 1 \Leftrightarrow (\alpha_1, \ldots, \alpha_n) \in D_i$ HLDD may be seen as graph representation of the following algorithm:

begin
if $\alpha \in D_1$ **then** $f(X) = g_1(X_1)$ **endif**
...
if $\alpha \in D_t$ **then** $f(X) = g_t(X_t)$ **endif**
end

Here, $X_i \subseteq X$, $i = 1..t$. As a shorthand for the algorithm we will use a formula

$$f(X) = \mathbf{case}_{i=1}^{t} \chi_i(X) \rightarrow g_i(X_i) \tag{4.1}$$

In the next section we deduce a more algebraic representation of this expression.

4.2 FROM CHARACTERISTIC FUNCTIONS TO CHARACTERISTIC POLYNOMIALS

Suppose we have two diagrams G_1 and G_2 with the same sets of variables and terminal nodes, but different control structures and wish to prove that they are equivalent. As we saw in the previous section, it is sufficient to show that the function (4.1) for both diagrams is the same. Our assumption implies that everything in (4.1) is the same except characteristic functions $\chi_i(X)$. Without loss of generality we may assume that $D(x_i) = \{1, \ldots, |D(x_i)|\}$. If not we define a bijective mapping $h : D(z_i) \to \{1, \ldots, |D(x_i)|\}$ and use the values of this function instead of their originals. Then the following theorem takes place:

Theorem 4.1. *The characteristic function of the set D_i, $\chi_i(X)$, can be represented by a unique polynomial $P_i : \mathbb{Q}^n \to \mathbb{Q}$ of degree at most $\sum_{j=1}^{n} \left(\left|D(x_j)\right| - 1\right)$ where we have for each vector $\alpha = (\alpha_1, ..., \alpha_n) \in D(X)$*

$$\chi_i(\alpha_1, ..., \alpha_n) = P_i(\alpha_1, ..., \alpha_n)$$

.

Proof. We have $D(X)$ different vectors in $\mathbb{Q}^n$. The sought-for polynomial should be equal to 1 for vectors from D_i and to 0 for vectors from $D(X) \backslash D_i$. In numerical analysis the Lagrange interpolation polynomial is well-known [5]: consider we have measured the values for some function $f : [a, b] \to \mathbb{R}$ at points $x_0, ..., x_n$ and obtained results are $y_0, ..., y_n$, then we can interpolate them to the whole segment $[a, b]$ by a polynomial of degree at most n:

$$f(x) \approx P(x) = \sum_{i=0}^{n} y_i \frac{(x - x_0)...(x - x_{i-1})(x - x_{i+1})...(x - x_n)}{(x_i - x_0)...(x_i - x_{i-1})(x_i - x_{i+1})...(x_i - x_n)}$$

If f is a polynomial of such degree then we get the exact result, otherwise there will be some error. Note that for $x = x_0, ..., x_n$ we will always get the exact result: $P(x_i) = y_i = f(x_i)$ and it is a polynomial of lowest degree that gives such result. This is the property we are interested in the current paper. Although in numerical analysis textbooks only the case of one-variable function is usually studied, these results can be easily transferred to the multiple-variable case. So, our sought-for polynomial P_i is the Lagrange polynomial that evaluates to 1 for each vector from D_i and to 0 for each vector from $D(X) \backslash D_i$:

$$P_i(x_1, ..., x_n) = \sum_{(\alpha_1, ..., \alpha_n) \in D_i} \prod_{j=1}^{n} \prod_{\substack{k=1 \\ k \neq \alpha_j}}^{\left|D(x_j)\right|} \frac{x_j - k}{\alpha_j - k} \tag{4.2}$$

The degree of this polynomial is at most $\sum_{i=1}^{n} (|D(x_i)| - 1)$. Let us prove that this is the only polynomial of such degree:

- *The basis.* Let $n = 1$. Assume we have 2 polynomials, $P(x_1)$ and $Q(x_1)$, $\deg P = \deg Q = |D(x_1)| - 1$ and $\forall (i \in 1..|D(x_1)|)\ P(i) = Q(i)$. Then the polynomial $P - Q$ has $|D(x_1)|$ roots: $1, 2, \ldots, |D(x_1)|$, which could be only in case $P \equiv Q$.
- *The induction step.* The proof is similar to the basis case one: assume we have 2 polynomials, $P(x_1, ..., x_n)$ and $Q(x_1, ..., x_n)$, $\deg P = \deg Q = \sum_{i=1}^{n} (|D(x_i)| - 1)$. After assigning values to x_1 we get $|D(x_1)|$ pairs of $(n-1)$-variable polynomials. They are pairwise equal by induction hypothesis. Thus, the polynomial function $P - Q : \mathbb{Q} \to \mathbb{Q}[x_2, ..., x_n]$ of degree at most $|D(x_1)| - 1$ has $|D(x_1)|$ roots in $\mathbb{Q}[x_2, ..., x_n]$. Thus, $P \equiv Q$.

□

Corollary 4.1. *Two diagrams G_1 and G_2 are equivalent iff they have equal sets of control variables, terminal nodes and the polynomial representations* (4.2) *of characteristic functions for any two corresponding terminal nodes are the same.*

We shall call the right side of formula (4.2) the *characteristic polynomial* of the node v_i^T. The formula 4.1 now can be rewritten in the following way

$$f(X) = \sum_{i=1}^{t} P_i(X) g_i(X_i) = \sum_{i=1}^{t} g_i(X_i) \left(\sum_{(\alpha_1,...,\alpha_n) \in D_i} \prod_{j=1}^{n} \prod_{\substack{k=1 \\ k \neq \alpha_j}}^{|D(x_j)|} \frac{x_j - k}{\alpha_j - k} \right) \tag{4.3}$$

The last equation would be the complete representation of f if we could explicitly enumerate vectors $\alpha \in D_i$, but the only input we have is G, so we need to be able to obtain characteristic polynomials from it. The next section shows how to do this.

4.3 FROM HLDDS TO CHARACTERISTIC POLYNOMIALS

HLDDs are directed and acyclic multigraphs, thus their vertices can be easily arranged in topological order, which means that if $v < w$ for some $v, w \in V$, then there is no paths from w to v. Then we might iteratively grow up our polynomials starting from $P \equiv 1$ in the source node. Keeping in mind these observations one could come up with something like Algorithm 4.1. Would it return us desired characteristic polynomials? Not really. There is a special case we should handle before we can apply it.

Algorithm 4.1 Obtaining characteristic polynomials from an HLDD

```
1:  function HLDD2CP(HLDD G = (V, E))
2:      order all nodes in G topologically
3:      T ← array of ordered nodes
4:      for i ← 1..(|V| − 1) do
5:          P_{T[i]} ← 0
6:      end for
7:      P_{T[0]} = 1
8:      for k ← 0..(|V| − 1) do
9:          v ← T[k]
10:         for all w ∈ Γ(v) do
11:             i ← c_{(v,w)}
12:             P_w ← P_w + P_v ∏_{j=1, j≠i}^{|D(v)|} (x_v − j)/(i − j)
13:         end for
14:     end for
15:     return {P_{v^T} | v^T ∈ V^T}
16: end function
```

In Algorithm 4.1 we grow up our polynomial for all the paths from the source node to terminals. Once we are in vertex v we have full polynomial P_v and use it for the descendants of v. But what if node's variable, x_v, has appeared before in some of the predecessors of v? Then Algorithm 4.1 will take into account infeasible paths which we should avoid. Such paths could appear if we generate an HLDD using the method described in Subsection3.3.2, the method from Subsection 3.3.1 is guaranteed to produce diagrams free of such paths. However, unlike the case of SSBDDs, this is a quite rare problem at higher levels, even if we use the superposition procedure to obtain an HLDD. A variable appearing twice in a path generally means a designer checks the corresponding condition twice. For example, the author has never seen such paths in the diagrams generated from real designs.

However, nobody prohibits designers to check conditions as many times as they need and we should be able to handle such situations. If we encounter the same variable two times in a path we can duplicate all variables between the occurrences of this variable and make an equivalent diagram without such redundancies. Let some path contain nodes labeled by variables $x_0, x_1, ..., x_k, x_0$. We should produce a new diagram, where this chain would be replaced by two new chains, $x_0, x_1, ..., x_k$ and $x_1, ..., x_k, x_0$. The following theorem will help us:

Theorem 4.2. *Suppose an HLDD containing a chain of non-terminal nodes labeled by variables $x_0, x_1, ..., x_k, x_0$ was transformed in the following way:*

- *Remove nodes labeled $x_0, x_1, ..., x_k, x_0$*
- *Add $2(k+1)$ nodes labeled $x_1, ..., x_k, x_0$ (the first chain) and $x_0, x_1, ..., x_k$ (the second chain).*
- *All connections from and to the nodes of the first chain will remain the same.*
- *All connections from and to x_0 occurrence in the second chain will remain the same.*
- *Connections from other nodes to $x_1, ..., x_k$ in the second chain will be removed*

Then the result is equivalent to original diagram.

Proof. We shall prove that the second diagram contain all paths from the first one that could be activated and vice versa. $\Rightarrow$:

- Obviously, all paths not containing mentioned nodes remain unchanged.
- All paths containing the first occurrence of the x_0 can be activated in the second chain.
- All paths containing the second occurrence of the x_0 or only intermediate nodes $x_1, ..., x_k$ can be found in the first chain.

$\Leftarrow$: The similar check for all possible paths in our two chains shows that they can be activated in the first diagram. □

Example 4.2. *Figure 4.4 illustrates the transformation described above. Some vertex and all edge labels are not shown because they are not important in current context. We have a path $x \to y \to z \to x$ that is being split in two chains $x \to y \to z$ and $y \to z \to x$:*

Now we can prove the following theorem:

Theorem 4.3. *The Algorithm 4.1 produces the set of characteristic polynomials for the diagram G if G does not contain infeasible paths.*

Proof. Let L be a set of all paths from the root node to some terminal node v^T. Each path $l \in L$ activated by the assignment $(x_{v_1} = \alpha_1, ..., x_{v_k} = \alpha_m)$ will be represented in the result polynomial by the following summand:

$$\prod_{j=1}^{m} \prod_{\substack{k=1 \\ k \neq i_j}}^{\left|D(x_{v_j})\right|} \frac{x_{v_j} - k}{\alpha_j - k} \tag{4.4}$$

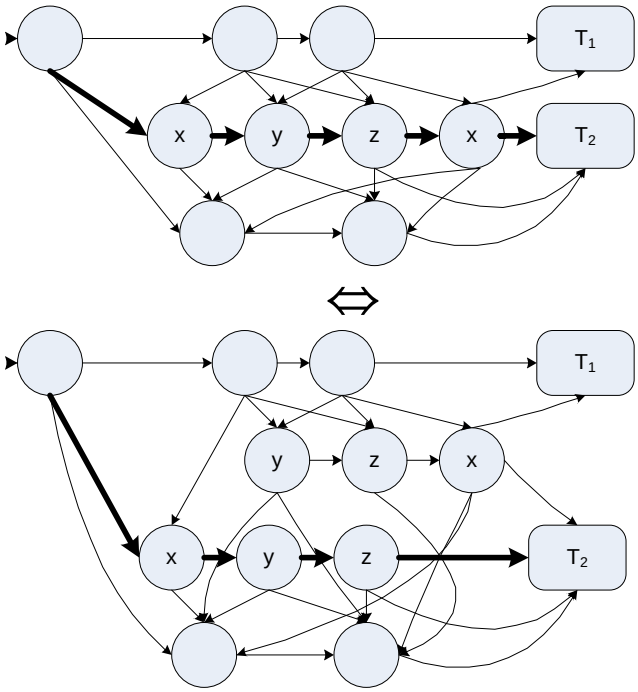

Figure 4.4 – Removing duplicate variables

(This can be easily proven by induction). The resulting polynomial will be the sum of these summands over all paths from L. As G does not contain infeasible paths, all variables in l are different. So, $m \leq n$. The only difference between summands in (4.2) and (4.4) is the bound of the first product sign: generally, the path l should not contain all variables; some of them may be missing. This means that one path actually represents $|D(\tilde{x}_1) \times ... \times D(\tilde{x}_{n-m})|$ assignments $\{(x_{v_1} = \alpha_1, ..., x_{v_m} = \alpha_m, \tilde{x}_1 = \alpha_{m+1}, \ldots, \tilde{x}_{n-m} = \alpha_n\}$, where $\{\tilde{x}_1, ..., \tilde{x}_{n-m}\} = X \backslash \{x_{v_1}, ..., x_{v_m}\}$ But we have

$$\sum_{(\alpha_{m+1},...,\alpha_n) \in D(\tilde{x}_1) \times ... \times D(\tilde{x}_{n-m})} \left(\prod_{j=1}^{n-m} \left(\prod_{\substack{k=1 \\ k \neq \alpha_{m+j}}}^{|D(\tilde{x}_j)|} \frac{\tilde{x}_j - k}{\alpha_{m+j} - k} \right) \right) = 1 \qquad (4.5)$$

This is the Lagrange polynomial that evaluates to 1 for all possible values from $D(\tilde{x}_1) \times ... \times D(\tilde{x}_{n-m})$. The simplest polynomial with such property is the constant 1, so they should coincide with each other. Multiplying this polynomial with our summand gives us the sum of $D(\tilde{x}_1) \times ... \times D(\tilde{x}_{n-m})$ summands each representing certain assignment for the whole set of variables. Finally, adding

them together results in the formula (4.2) (none of the summands will appear twice, because all paths have the same source node and thus assignments of corresponding paths will differ for at least one variable; otherwise they would never branch off). □

Example 4.3. Let us now find the characteristic polynomials for the HLDD that evaluates the next state for variable C from Figure 3.6. First of all we change the labels of edges labeled by 0 to 5 for the one going from node q and to 2 for others. We have 4 paths to the first terminal node: $(q = 5)$, $(q = 1)$, $(q = 3, x_C = 1)$, $(q = 4, x_A = 2)$. Thus,

$$\begin{aligned} P_1(q, x_A, x_B, x_C) \quad = \quad & \frac{(q-1)(q-2)(q-3)(q-4)}{(5-1)(5-2)(5-3)(5-4)} + \\ & \frac{(q-5)(q-2)(q-3)(q-4)}{(1-5)(1-2)(1-3)(1-4)} + \\ & \frac{(q-5)(q-1)(q-2)(q-4)(x_C-2)}{(3-5)(3-1)(3-2)(3-4)(1-2)} + \\ & \frac{(q-5)(q-1)(q-2)(q-3)(x_A-1)}{(4-5)(4-1)(4-2)(4-3)(2-1)} = \\ & -\frac{1}{6}q^4x_A - \frac{1}{4}q^4x_C + \frac{3}{4}q^4 + \frac{11}{6}q^3x_A + 3q^3x_C - \frac{53}{6}q^3 \\ & -\frac{41}{6}q^2x_A - \frac{49}{4}q^2x_C + \frac{143}{4}q^2 + \frac{61}{6}qx_A \\ & +\frac{39}{2}qx_C - \frac{173}{3}q - 5x_A - 10x_C + 31 \end{aligned}$$

For the second node we have two paths $(q = 2, x_B = 2)$ and $(q = 3, x_C = 2)$, so the second polynomial will be

$$\begin{aligned} P_2(q, x_A, x_B, x_C) \quad = \quad & \frac{(q-5)(q-1)(q-3)(q-4)(x_B-1)}{(2-5)(2-1)(2-3)(2-4)(2-1)} + \\ & \frac{(q-5)(q-1)(q-2)(q-4)(x_C-1)}{(3-5)(3-1)(3-2)(3-4)(2-1)} = \\ & -\frac{1}{6}q^4x_B + \frac{1}{4}q^4x_C - \frac{1}{12}q^4 + \frac{13}{6}q^3x_B - 3q^3x_C + \frac{5}{6}q^3 \\ & -\frac{59}{6}q^2x_B + \frac{49}{4}q^2x_C - \frac{29}{12}q^2 + \frac{107}{6}qx_B - \frac{39}{2}qx_C \\ & +\frac{5}{3}q - 10x_B + 10x_C \end{aligned}$$

Finally, paths $(q = 2, x_B = 2)$ and $(q = 4, x_A = 1)$ give us the third polynomial:

$$\begin{aligned} P_3(q, x_A, x_B, x_C) &= \frac{(q-5)(q-1)(q-3)(q-4)(x_B-2)}{(2-5)(2-1)(2-3)(2-4)(1-2)} + \\ &\frac{(q-5)(q-1)(q-2)(q-3)(x_A-2)}{(4-5)(4-1)(4-2)(4-3)(1-2)} = \\ &\frac{1}{6}q^4x_A + \frac{1}{6}q^4x_B - \frac{2}{3}q^4 - \frac{11}{6}q^3x_A - \frac{13}{6}q^3x_B + 8q^3 \\ &+\frac{41}{6}q^2x_A + \frac{59}{6}q^2x_B - \frac{100}{3}q^2 - \frac{61}{6}qx_A - \frac{107}{6}qx_B \\ &+56q + 5x_A + 10x_B - 30 \end{aligned}$$

Thus, we can compare the control parts of HLDDs by computing their characteristic polynomial sets. The only remaining issue belongs to the case shown in Figure 4.3. If variable sets are different, then functions are different by definition, so we cannot apply our method unless we make those sets the same. The next section presents an algorithm created for this purpose.

4.4 NORMALIZATION OF SET OF VARIABLES

Assume we have two digital systems doing the same operations but their procedural descriptions contain different variables with different dimensions. From the HLDD point of view this means that if we assign the same bits to our control variables then we will reach the same terminal node. How to compare two HLDDs, if the sets of control variables are not same but there exists one-to-one mapping between bits of these variables? The answer is to make those sets the same: we just need to split variables forth and use the superposition procedure producing the diagrams we can compare. Let $X = (x_1, ..., x_n)$, $Y = (y_1, ...y_m)$ be two sets of variables, where $\sum_{i=1}^{n} \dim x_i = \sum_{j=1}^{m} \dim y_j$ ($\dim t$ means the length of variable t in bits). We need to obtain a set Z of variables where $\forall t \in (X \cup Y) \exists z_i, ..., z_j (t = z_i \cup ... \cup z_j)$ using Algorithm 4.2.

After running this algorithm we receive the sought for set $Z = (z_1, ...z_k)$. Each variable t from X and Y will be represented by a tree-like HLDD (Figure 4.3) representing some function $f(z_i, ..., z_j)$, where $D(t) \subseteq D(f(z_i, ..., z_j))$. Using the superposition procedure and characteristic polynomials we can now check if our diagrams are equivalent or not.

Algorithm 4.2 Normalisation algorithm

```
1:  Z ← ∅
2:  while X ≠ ∅ do
3:      x_i ← GetFirstVariable(X)
4:      y_j ← GetFirstVariable(Y)
5:      if dim x_i > dim y_j then
6:          t ← y_j
7:          u ← x_i \ t
8:          Z ← Z ∪ {t}
9:          X ← X \ {x_i}
10:         Y ← Y \ {y_j}
11:         X ← X ∪ {u}
12:     else if dim x_i < dim y_j then
13:         t ← x_i
14:         u ← y_j \ t
15:         Z ← Z ∪ {t}
16:         X ← X \ {x_i}
17:         Y ← Y \ {y_j}
18:         Y ← Y ∪ {u}
19:     else
20:         Z ← Z ∪ {x_i}
21:         X ← X \ {x_i}
22:         Y ← Y \ {y_j}
23:     end if
24: end while
```

4.5 RELATED WORKS

There are works studying similar polynomials for different diagram types. In [1] and [53] authors studied the characteristic polynomials of BDDs. In [37] the method was applied to MDDs. These methods are not much known, as in the binary case ROBDDs are mostly used, which are canonical and do not require equivalence tests rather than direct comparison.

Regarding the case of multivalued logic, Reduced Ordered MDDs are canonical as well and do not require further equivalence checking either. General MDD structure, however, is not that flexible as HLDD. Generating an MDD from a general circuit design could be a challenge, except for the designs exploiting Multiple-Valued Logic Field Programmable Gate Arrays (MVL FPGA) technology, where a mapping between the MDD and the circuit can be easily obtained [72].

4.6 SUMMARY

HLDDs cannot be applied directly, like BDDs, in formal verification, as it is not a canonical data structure. We need algorithms comparing different HLDDs expressing the same functionality and this can be established using characteristic polynomials with a combination of canonic word-level diagrams applied to terminal node functions.

A characteristic polynomial is a formal way to express, what subset of diagram paths lead to a particular terminal node. Unlike BDD and MDD models, certain obstacles exist preventing the direct use of these polynomials. Some designs may result in diagrams, permitting multiple occurrences of a variable in a path. An algorithm of transforming these diagrams into the equivalent ones without the presence of multiple variables is given.

Another obstacle is the possibility of splitting n-bit variable into a number of variables of lesser bitwidth, resulting in functions that cannot be equivalent mathematically as they depend on different variable sets. In order to overcome this obstacle we must normalize the variable sets and the algorithm of such normalization is given.

Having these problems resolved we can compute characteristic polynomials and compare them for the corresponding terminal node pairs checking the equivalence of the graph structures of HLDDs. An algorithm of producing the set of polynomials from an HLDD is provided along with the proof of its correctness.

All mentioned algorithms provide a way to transform an uncanonical HLDD into a fully canonical structure, making it possible to apply HLDDs for equivalence checking at the higher levels, and form the main contribution of the current chapter.

5

PROBABILISTIC EQUIVALENCE CHECKING

In this chapter we will see how to apply characteristic polynomials for equivalence checking. The main obstacle preventing computing them directly is their size. To overcome this we developed a probabilistic technique based on characteristic polynomials, which allow us to compare diagrams with a negligible chance to get a collision. The first section of the chapter addresses the problem of mapping states. Next, in Section 5.2 we present our probabilistic technique. After that, in Section 5.3 we give an estimation of the probability of collision. Section 5.4 provides some experimental results and Section 5.5 concludes the chapter.

5.1 MAPPING STATE VARIABLES

In Subsection 2.3.1 we described the task of combinational equivalence checking between RTL and low-level descriptions. It consists of two subtasks: first we propose a function ρ that relate state elements from one level to ones from another level, assuming that such 1:1 relationship exists; then we check outputs and next state functions using SAT/SMT solvers or BDDs.

Our task is slightly different. We compare two different high-level descriptions, e.g. behavioral vs. RTL or software model vs. RTL, etc. In this case we also need the first step: map variables and states (some HLDD variables correspond to system registers which are part of the state space when we check RTL vs. gate-level description), but we cannot assume 1:1 relationship anymore.

When we implement some computations, we may use temporary variables to store intermediate results, e.g. to make code more clear. Such temporary variables affect the state variable and make systems harder to compare. Thus, in order to apply methods of guessing candidates for ρ, developed for state-of-the-art equivalence checking approach [66], we must first get rid of such temporary variables. Here we present a technique to remove them and make systems implementing the same algorithms at different levels directly comparable. Consider the following example.

Example 5.1. A hypothetical digital system, according to its specification presented in Listing 1, consists of a variable x and four inputs: in_x, a, b and c. Depending on conditions c_a, c_b and c_c the value at x should be increased by a, b or c while the value of x is less than some numeric constant $some\_num$.

Listing 1 – System Specification

```
x := in_x;
while x < some_num loop
  if c_b and not c_a and not c_c then
    x := x + b;
  else if c_c and not c_a and not c_b then
    x := x + c;
  else if c_a and not c_b and not c_c then
    x := x + a;
  end if;
end loop;
```

Listing 2 shows the RTL implementation of the system. In the design process from specification to RTL, additional details appear in the description, e.g. new variables: reg_t and $state$. We see, that from the algorithmic point of view, reg_t is unnecessary here. It is needed just to store the intermediate result of addition, and then it transfers the value back to reg_x. The order of checking predicate values is also changed. We also injected a little copy-paste error there to make equivalence checking fail later in our case-study: condition c_c should hold when we are adding in c, not the negation of c_c.

First of all we generate HLDDs using methods described in Section 3.3. Figure 5.1 depicts the flowchart for Listing 1 with inserted states. After the symbolic execution procedure we will obtain the set of two diagrams shown in Figure 5.2a. The system implementation results in three graphs from Figure 5.2b.

Register reg_x takes value reg_t at state s_3, and reg_t takes its value at the previous state, s_2. Let us denote the fact that some variable a takes value $f(A)$ at state s, where A is some set of variables, by $a \overset{s}{=} f(A)$. So, we have $reg\_x \overset{s_3}{=} reg\_t \overset{s_2}{=} f(reg\_x, in\_a, \ldots)$. As s_3 is the next state for s_2 and there are no actions at s_3 any more, we can remove variable reg_t and action s_3, having $reg\_x \overset{s_2}{=} f(reg\_x, in\_a, \ldots)$. Then we remap states to make them compatible with specification states (i.e. rename s_4 to s_3) and obtain two new diagrams that we already can compare with the specification ones. The new state diagram appears to be exactly the same as in the specification, so in Figure 5.3 we show only the one for register reg_x.

Listing 2 – System Implementation

```
case state is
  when s0 =>
    reg_x <= in_x;
    state := s1;
  when s1 =>
    if reg_x < some_num then
      state := s2;
    else
      state := s4;
    end if;
  when s2 =>
    state := s3;
    if c_a and not c_b and not c_c then
      reg_t <= reg_x + in_a;
    else if c_b and not c_a and not c_c then
      reg_t <= reg_x + in_b;
    else if not c_c and not c_a and not c_b then --bug, should be if
        c_c
      reg_t <= reg_x + in_c;
    else
      reg_t <= reg_x;
    end if;
  when s3 =>
    reg_x <= reg_t;
    state := s2;
  when s4 =>
    state := s0;
end case;
```

Let us now generalize this approach. Consider we have two sets of variables, Z_{spec} and Z_{impl}. A subset of Z_{spec} can be mapped to some subset of Z_{impl}, as we a talking about the same algorithms, and auxiliary variables remain in both sets. Then, for each such variable z_a and for each appearance of it in the right side of an assignment expression, where at the left side there is a mapped variable z_m try to do the following:

1. If $z_m \overset{s_m}{=} f(z_a, ...)$ and $z_a \overset{s_a}{=} g(...)$, where the next state for s_a is s_m or if there are some intermediate states between s_a and s_m, there are no modifications of the variable z_m depends on, then replace the first expression with $z_m \overset{s_m}{=} f(g(...), ...)$.

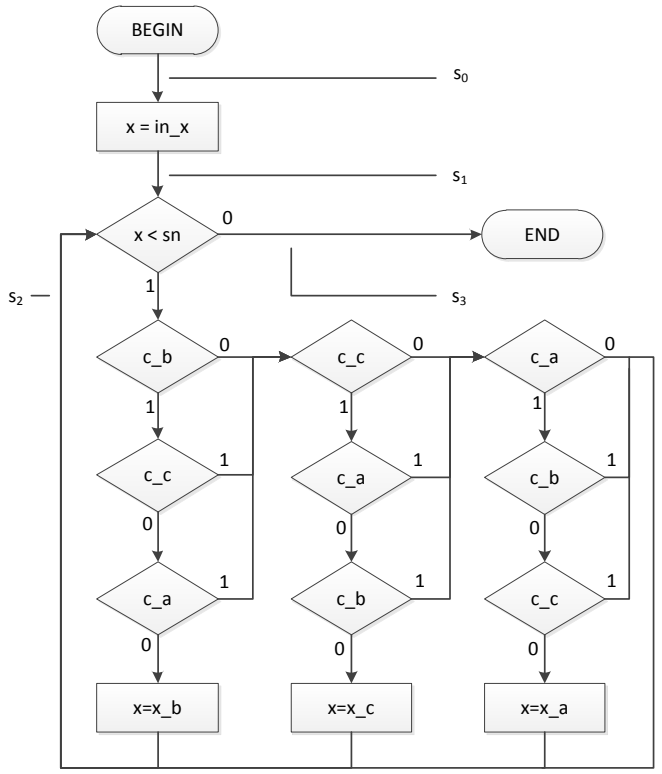

Figure 5.1 – Flowchart for Listing 1

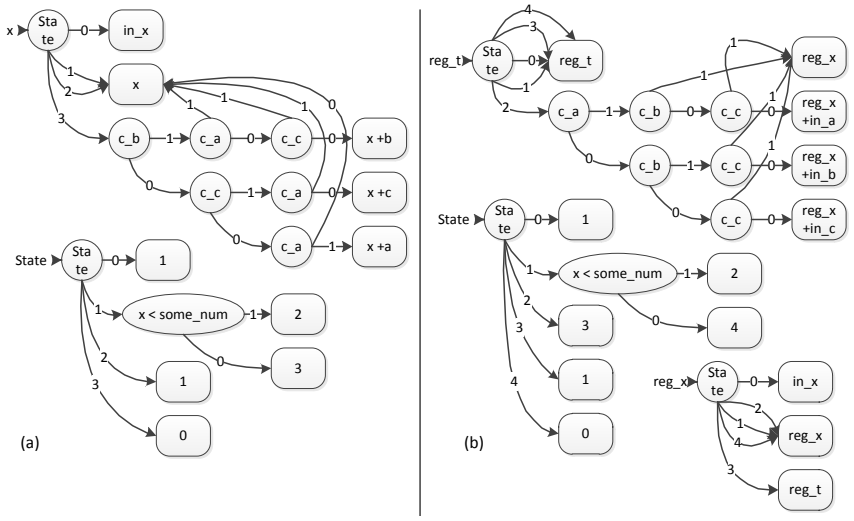

Figure 5.2 – HLDD sets derived from system (a) specification and (b) implementation

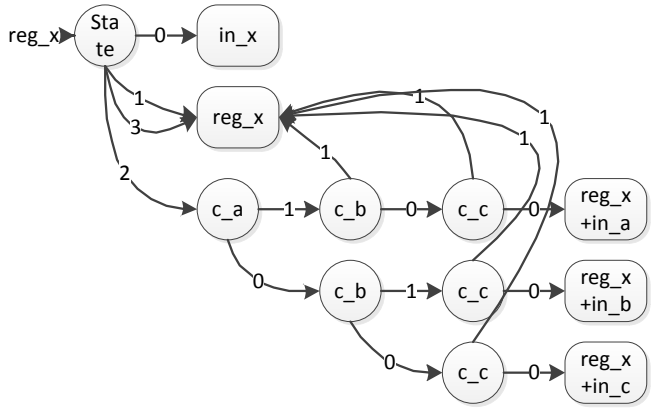

Figure 5.3 – A modified HLDD for *reg_x*.

2. If $z_m \overset{s_m}{=} f(z_a, \ldots)$ and $z_a \overset{s_a}{=} g(\ldots)$, and the transition $s_a \to s_m$ is enabled only when some predicate P is true, then replace the first expression with the statement **if** P **then** $z_m \overset{s_m}{=} f(g(\ldots), \ldots)$.

3. If the statement $z_a \overset{s_a}{=} g(\ldots)$ is not used any more then remove it. If state s_a contains no more actions then remove it.

So, if all auxiliary variables are successfully removed and variable sets and states mapped, then we are ready to proceed to the second part of the equivalence checking process – comparing the diagrams.

5.2 POLYNOMIAL VALUES AT RANDOM POINTS

Consider two digital system $S_1 = (Z_1, F_1)$ and $S_2 = (Z_2, F_2)$. All necessary mappings end normalizations are done, so $Z_1 = Z_2 = Z$ and we have translated both systems to into the sets of HLDDs $\mathcal{G}_1$ and $\mathcal{G}_2$. We are comparing two graphs, $G_1 \in \mathcal{G}_1$ and $G_2 \in \mathcal{G}_2$, which represent functions $f^{(1)}(X) = \sum_{i=1}^{t} P_i^{(1)}(X) g_i^{(1)}(X_i)$ and $f^{(2)}(X) = \sum_{i=1}^{t} P_i^{(2)}(X) g_i^{(2)}(X_i)$.

As it was shown in the previous chapter, we check non-terminal and terminal parts of HLDDs separately, the former with characteristic polynomials and the latter with existing methods, like canonic word-level diagrams (alternatively, we can create QF_BV theory instances in form $g_i^{(1)} \equiv g_i^{(2)}$ for each pair of corresponding terminal node functions $g_i^{(1)}$ and $g_i^{(2)}$ and check these instances with arbitrary SMT solver). Regarding the non-terminal part, the explicit computation of characteristic polynomials for a large modern digital system would take huge amount of time and memory.

Algorithm 5.1 Compute characteristic polynomial value at random point

```
1:  function RandomPoint(HLDD G = (V, E), r ∈ ℝ^n)
2:      order all nodes in G topologically
3:      T ← array of ordered nodes
4:      for i ← 1..(|V| − 1) do
5:          y_{T[i]} ← 0
6:      end for
7:      y_{T[0]} = 1
8:      for k ← 0..(|V| − 1) do
9:          v ← T[k]
10:         for all w ∈ Γ(v) do
11:             i ← c_{(v,w)}
12:             y_w ← y_w + y_v ∏_{j=1, j≠i}^{|D(v)|} (r_v − j)/(i − j)
13:         end for
14:     end for
15:     return {y_{v^T} | v^T ∈ V^T}
16: end function
```

Consider a diagram G containing a path of length n in a form $v_0 \to v_1 \to \ldots \to v_{n-1}$ from the source node v_0 to one of the terminals $v_{n-1} \in V^T$. This path would add to the polynomial $P_{v_{n-1}}$ the following summand:

$$\prod_{j=0}^{n-1} \prod_{\substack{k=1 \\ k \neq \alpha_{v_j}}}^{|D(x_{v_j})|} \frac{x_{v_j} - k}{\alpha_{v_j} - k}$$

Here, α_{v_j} is the assignment to x_{v_j}. It is n-variable polynomial where degree of each variable x_{v_j} is $|D(x_{v_j})| - 1$. Thus, in the worst case, when none of its monomials equals to 0, we need $|D(x_{v_0})| \cdot \ldots \cdot |D(x_{v_{n-1}})|$ memory cells to store its coefficients. Easy to see that in the worst case this number is exponential to the size of original diagram.

Fortunately, we can extract a lot of useful information about the polynomial without computing an analytical form of it. The idea is to treat the polynomial as a usual function $P : \mathbb{R}^n \to \mathbb{R}$ and to evaluate its value at some random point $r = (r_1, ..., r_n) \in \mathbb{R}^n$. Algorithm 5.1 presents the procedure, that, given a diagram G and random vector r returns characteristic polynomial values at r. Note, that it is almost the same algorithm as Algorithm 4.1 except the line 12 where instead of variable x_v we use its random assignment value r_v.

Although the algorithm is mathematically correct, in the real life computations we cannot use real numbers. We might generate random vectors of integers or floating point numbers. Using floating point numbers implies performing calculations with an error $\epsilon > 0$. In case of choosing integer vectors we face two problems: first of all, at line 12 we do a lot of multiplications which would cause overflows. Secondly, as the computation includes divisions, to implement support of rational numbers. Fortunately, the following lemma takes place.

Lemma 5.1. *Let $t, d, i \in \mathbb{N}$ where $i < d$. Then*

$$\prod_{\substack{j=1 \\ j \neq i}}^{d} \frac{t-j}{i-j} \in \mathbb{Z} \tag{5.1}$$

Proof. If $t \leq d$ then the result of the left-side expression in (5.1) is either 1 or 0. Assume $t > d$. Then we have

$$\begin{aligned}
&\frac{(t-1)\cdot \ldots \cdot (t-(i-1))(t-(i+1))\cdot \ldots \cdot (t-d)}{(i-1)\cdot \ldots \cdot (i-(i-1))(i-(i+1))\cdot \ldots \cdot (i-d)} && = \\
&\frac{(t-1)\cdot (t-2)\cdot \ldots \cdot (t-(i-1))}{1\cdot \ldots \cdot (i-1)} \times \frac{(t-(i+1))\cdot \ldots \cdot (t-d)}{(-1)\cdot \ldots \cdot (i-d)} && = \\
&(-1)^{d-i}\frac{(t-1)\cdot ((t-1)-1)\cdot \ldots \cdot ((t-1)-(i-1)+1)}{1\cdot \ldots \cdot (i-1)} && \times \\
&\frac{(t-(i+1))\cdot \ldots \cdot (t-d)}{1\cdot \ldots \cdot (d-i)} && = \\
&(-1)^{d-i}\binom{t-1}{i-1}\frac{(t-(i+1))\cdot \ldots \cdot (t-d)}{1\cdot \ldots \cdot (d-i)} && = \\
&(-1)^{d-i}\binom{t-1}{i-1}\frac{(t-i-1)\cdot \ldots \cdot (t-i-1-(d-i)+1)}{1\cdot \ldots \cdot (d-i)} && = \\
&(-1)^{d-i}\binom{t-1}{i-1}\binom{t-i-1}{d-i}
\end{aligned}$$

As t is at least $d+1$ and i is at most d then $t-1 \geq d > i-1$ and $t-i-1 \geq d+1-i-1 = d-i$, so the binomial coefficients are correctly defined. Thus,

$$\prod_{\substack{j=1 \\ j \neq i}}^{d} \frac{t-j}{i-j} = (-1)^{d-i}\binom{t-1}{i-1}\binom{t-i-1}{d-i} \in \mathbb{Z} \tag{5.2}$$

□

This result allows us to propose the next method.

1. Generate n random integer numbers, $r_1, \ldots, r_n \in \mathbb{Z}$, where $\forall i\ r_i < p$ for some large number p. As vectors from $D(X)$ only select one of the paths, in order to avoid this we also need to select numbers that exceed the corresponding variable's domain cardinality: $\forall i\ r_i > |D(x_i)|$. For simplifying further computations we strengthen a bit the last condition, by choosing numbers that exceed the maximum cardinality of the domains. Thus, the final condition looks like $\forall i\ \max_{j=1}^{n} |D(x_j)| < r_i < p$.

2. Evaluate binomial coefficients from Lemma 5.1 modulo p.

3. Apply Algorithm 5.1 using just obtained binomial coefficients at step 12 and doing all computations modulo p as well.

As it is more convenient to use prime number moduli, from now we suppose that $p \in \mathbb{P}$. For example, if we are doing our computations using traditional 32-bit integers, we can choose $p = 4294967291$, which is the closest prime number from below to $2^{32} = 4294967296$.

Usage of binomial coefficients allows us to utilize some of their well-known properties to speed-up our computations. It is easy to see that

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \tag{5.3}$$

$$\binom{n}{k} = \frac{n-k+1}{k}\binom{n}{k-1} \tag{5.4}$$

Consider the lines 10-13 of Algorithm 5.1. We iterate over all descendants of v, which means that c takes values from 1 to $|D(x_v)|$. According to Lemma 5.1 we need to compute the following expression with the arbitrary sign at line 12:

$$\binom{r_v - 1}{c-1}\binom{r_v - c - 1}{|D(x_v)| - 1} \mod p \tag{5.5}$$

The first multipliers in (5.5) form the sequence $1 = \binom{r_v-1}{0}, \binom{r_v-1}{1}, \ldots, \binom{r_v-1}{|D(x_v)|-1}$, so the property (5.4) is applicable to pairs of its consequent elements. The sequence of second multipliers, $\binom{r_v-2}{|D(x_v)|}, \binom{r_v-3}{|D(x_v)|-1}, \ldots, \binom{r_v-|D(x_v)|-1}{0} = 1$ allows us to utilize the property (5.3) for consequent pairs downwards.

Algorithm 5.2 presents the optimized version of Algorithm 5.1. The function from line 10, $ComputeBinomialCoefficients(v)$, is used to calculate and store in memory binomial coefficients for the descendants of v in just described way.

Easy to see that we can compute expressions (5.5) for the whole set $\Gamma(v)$ by using $2|D(x_v)|$ divisions and $3|D(x_v)|$ multiplications in $\mathbb{Z}_p$. As $|D(x_v)|$ equals the number of outgoing edges of v and we iterate over all vertices at line 8 the

Algorithm 5.2 Algorithm 5.1 with optimizations

```
1:  function RandomPointOpt(HLDD G = (V, E), r ∈ Z_p^n)
2:      order all nodes in G topologically
3:      T ← array of ordered nodes
4:      for i ← 1..(|V| − 1) do
5:          y_{T[i]} ← 0
6:      end for
7:      y_{T[0]} = 1
8:      for k ← 0..(|V| − 1) do
9:          v ← T[k]
10:         ComputeBinomialCoefficients(v)
11:         for all w ∈ Γ(v) do
12:             d ← ( y_v (−1)^{|D(x_v)|−c} C(r_v − 1, c − 1) C(r_v − c − 1, |D(x_v)| − c) )  mod p
13:             y_w ← (y_w + d)  mod p
14:         end for
15:     end for
16:     return {y_{v^T} | v^T ∈ V^T}
17: end function
```

total number of divisions and multiplications is $O(|E|)$. Thus we have proven the following theorem:

Theorem 5.1. *The complexity of Algorithm 5.2 is* $O(|E|)$. □

5.3 COLLISION PROBABILITY

If we are dealing with randomized algorithms we should estimate the probability Pr of getting a collision: let P and Q be different characteristic polynomials; what is the probability of $(P - Q)(r_1, \ldots, r_n) \equiv 0 \pmod{p}$ for some uniformly distributed random numbers $r_1, \ldots, r_n \in \{\max_{j=1}^{n} |D(x_j)| + 1, \ldots, p - 1\}$?

Let $d_{max} = \max_{j=1}^{n} |D(x_j)|$. We can use the basic formula of probability theory:

$$Pr = \frac{\texttt{number of acceptable try-outs}}{\texttt{number of all try-outs}}$$

The denominator's value is obviously $(p - d_{max} - 1)^n$. Let a designate the value of the numerator. Our goal is to find its upper bound.

Lemma 5.2. *Let* $\mathbb{F}$ *is a field,* $P(x_1, \ldots, x_n) \in \mathbb{F}[x_1, \ldots, x_n]$, $d_1, \ldots, d_n$ *are maximum degrees of* $x_1, \ldots, x_n$ *correspondingly and* M *is a finite subset of* $\mathbb{F}$*, where*

$m = |M|$. Then the number of possible solutions of the equation $P(t_1, \ldots, t_n) = 0$, where $t_i \in M$ for all $i = 1, \ldots, n$, is at most

$$\frac{(m^n - 1)d}{n(m-1)}$$

where $d = \sum_{j=1}^{n} d_j$ – maximum possible degree of P.

Proof. Let a is the sought for number of solutions, variables are ordered in a way that $d_1 \leq \ldots \leq d_n$ and let us assign some values to $t_2, ..., t_n$. We have two possible results:

1. P becomes a one-variable polynomial with degree at most d_1 and thus has at most d_1 roots.
2. P collapses to a trivial equation $0 = 0$. That means that if we treat P as the one-variable polynomial in $(\mathbb{F}[x_2, ..., x_n])[x_1]$, all its coefficients after the substitution must be equal to 0.

It is easy to see that these two results are independent. Thus, $a \leq d_1 m^{n-1} + a'$, where a' denotes the number of vectors which lead to the second case. There we should solve the similar problem, this time for vectors of length $n-1$ and the system of equations $P_j(t_2, \ldots, t_n) = 0$, where $j = 0, \ldots, d_1$. It is easy to see that the maximum number of solutions will be achieved if the system has only one non-trivial equation. This gives us the following estimation:

$$a \leq d_1 m^{n-1} + a' \leq d_1 m^{n-1} + d_2 m^{n-2} + a'' \leq \sum_{i=1}^{n} d_i m^{n-i}$$

$$\leq \frac{1}{n} \left(\sum_{i=1}^{n} d_i \right) \cdot \left(\sum_{i=0}^{n-1} m^i \right) = \frac{m^n - 1}{m-1} \cdot \frac{d}{n}$$

The last inequality here is Chebyshev's sum inequality. □

According to (4.2) our polynomial $P - Q$ belongs to $\mathbb{Q}[x_1, \ldots, x_n]$, but we intend to use $\mathbb{Z}_p$ for our computations. In order to apply Lemma 5.2 we need a polynomial from $\mathbb{Z}_p[x_1, \ldots, x_n]$. However, it is easy to check that if divide numerators by denominators in rational coefficients of P and Q according to $\mathbb{Z}_p$ rules and then evaluate the value for some assignment, the result would be the same as if we do it in a proposed way using binomial coefficients. Thus, Lemma 5.2 helps us to estimate the probability of a collision:

$$Pr = \frac{a}{(p - d_{max} - 1)^n} \leq \frac{((p - d_{max} - 1)^n - 1)d}{n(p - d_{max} - 1)^n (p - d_{max} - 2)} < \frac{d}{n(p - d_{max} - 2)}$$

Assuming that $\frac{d}{n} \ll p$ (we always can choose such p) we get $Pr \approx 0$. E.g., consider two large diagrams with 1000000 8-bit variables. The probability of getting a collision in this case for $p = 4294967291$ is only

$$Pr = \frac{2^8 \cdot 1000000}{1000000 \cdot (4294967291 - 2^8 - 2)} = 0.0000000596$$

Remark 5.1. In order to estimate the probability of collision in similar cases one may use Schwartz-Zippel theorem [84, 110] which states, that for given n-variable polynomial $P \not\equiv 0$ the probability that $P(t_1, \ldots, t_n) = 0$, where $(t_1, \ldots, t_n) \in S$ for some finite set S does not exceed the value $\frac{\deg P}{|S|}$. E.g. this theorem was cited in [37]. Although our result is slightly less accurate in the worst case, when $d_1 = d_2 = \ldots = d_n = d_{max} = \deg(P - Q)$ and $d = nd_{max}$ (it would give us an estimate $Pr < \frac{\deg(P-Q)}{p-d_{max}-2}$, while Schwartz-Zippel bound is $Pr \leq \frac{\deg(P-Q)}{p-d_{max}-1}$), in the general case our estimation is much better, achieving almost n times lesser value when the degree of polynomial difference is maximum possible (equals to d)[1].

5.4 EXPERIMENTS

In this section we present the experiments to assess the proposed method. The first set of experiments is based on the motivational examples shown in the previous sections. In Section 4.1 we made several transformations to obtain Fig. 4.2 from Fig. 3.9. Assume we made a mistake during that transformation and the edge from y_3 to $R_1 + R_2$ is redirected to $R_1 \cdot R_2$.

Then this error should affect characteristic polynomials for both terminal nodes. First of all we generate a random vector (x, y_1, y_2, y_3, y_4) = (766319080, 2130684362, 4026180015, 3459714997, 3086748849). Next, for every variabe $z \in \{x, y_1, y_2, y_3, y_4\}$ we define a mapping $h_z : D(z) \to \{1, \ldots, |D(z)|\}$ in the following way: $h_z(k) = k$ if $k \neq 0$ and $h_z(0) = |D(z)|$. Then, we compute characteristic polynomials for two selected nodes from Figure 3.9, after that we apply the same procedure to Figure 4.2 to check whether we obtain the same values and, finally, we try to detect the error for the diagram with redirected edge. At the end of the computation we get $(P_{R_1+R_2}, P_{R_1 \cdot R_2})$ = (2400174328, 3598364564) for both correct diagrams, while the corresponding values for the diagram with error are equal to (541580543, 1161991058).

1 A polynomial generally achieves maximum possible degree when the diagram contains a path ending in the corresponding terminal node, for which a variable that would not label any of its nodes does not exist.

Table 5.1 – Equivalence checking experimental results.

Benchmark	Diagrams	Nodes	Terminal Nodes	Edges	Simulated Errors	Polynomials	Time	Avg. Time
b00	4	35	16	42	206	886	0.004882	2.4×10^{-5}
b02	2	16	9	24	124	822	0.003201	2.6×10^{-5}
b03	15	137	54	187	730	3102	0.021511	2.9×10^{-5}
b04	12	58	30	59	158	630	0.002925	1.9×10^{-5}
b06	5	44	18	77	301	1347	0.00529	1.8×10^{-5}
b09	5	44	18	62	228	864	0.003747	1.6×10^{-5}
HC11	98	979	326	8856	166554	1891929	104.456	6.2×10^{-4}
CRC	35	234	86	282	770	2704	0.020822	2.7×10^{-5}

Consider our other example from the beginning of current chapter. One can remember that we injected an error there, the unnecessary inversion of the last occurrence of c_c. We compare the bottommost diagram from Figure 5.2a with the HLDD from Figure 5.3. First, we generate a random vector r = ($state$, c_a, c_b, c_c) = (3410646289, 3410646289, 3410646289, 1572882280). The characteristic polynomial values of the terminal nodes of implementation diagram from Figure 5.3 are presented here: (in_x, reg_x, $reg\_x + in\_a$, $reg\_x + in\_b$, $reg\_x + in\_c$) = (4090461347, 4216261509, 2982678470, 1627467403, 4263000411). The specification diagram for variable x gives us, however, the following output: (in_x, x, $x + b$, $x + c$, $x + a$) = (4090461347, 4216261509, 1627467403, 4263000411, 2982678470). We see that values at x/reg_x and $x + c/reg\_x + in\_c$ differ from each other.

Table 5.1 displays results for a more comprehensive experiment. We generated several diagrams for circuits from the ITC99 benchmark set [30], supplemented with a commercial processor core HC11 [47] from Green Mountain Inc. and a VERTIGO benchmark CRC (Circular Redundancy Check) [43]. First five columns provide some information about the complexity of the benchmark circuits: name of circuit, number of graphs (each graph represents a subcircuit) , number of internal and terminal nodes in the model, and number of edges in all graphs. Column 6 represents the number of errors we simulated for each circuit, succeeded by the column showing the number of polynomials used for representing the circuits and evaluated during the simulation. The simulated errors are similar to the one we started this section with: we tried to redirect each edge to every possible node and checked the polynomial values (here the word "possible" means that the diagram should remain acyclic). As it turned out, all these errors were detected using only one random vector. The run times (in seconds) on each set of diagrams are provided in column 8. Finally, the last column shows the average time per error. Experiments were carried out on a computer with Intel Core 2 Duo P7550 2.26 GHz processor and 2 GB RAM.

Since the number of simulated errors is well correlated with the structural complexity of the model, we can evaluate the scalability of the proposed method by the correlation between the number of simulated errors and the average time of the verification run (i.e. the average time of simulating an error). Since the number of simulated errors for the most complex circuit HC11 is in range of 280 - 1300 times higher than for other circuits, and the average error simulation time for HC11 is only in the range of 30-40 times higher than for other circuits, we can conclude that the proposed polynomial HLDD-based approach of equivalence checking is well scalable in the case when the partitioning of the verification model is well balanced.

5.5 SUMMARY

This chapter addresses the equivalence checking problem at higher levels. We started it with the short discussion on the problem of mapping states of the comparing systems and proposed an algorithm that removes auxiliary variables, which is the first contribution of the current chapter.

The next contribution is the algorithm of computing polynomial values at given random vector on given diagram. Usage of Lemma 5.1 allows to speed up the basic approach and obtain the algorithm that computes the values in $O(|E|)$ time.

Proposing the probabilistic algorithm we were required to give an estimation of the collision probability and two such estimates were provided; one based on the Schwartz-Zippel theorem and another one developed by the author, where the latter gives better estimation for our case and forms the third contribution of the chapter.

The experimental results performed on the subset of ITC'99 benchmarks supplemented with some industrial designs are reported in the last section. These benchmarks prove the feasibility of proposed methodology detecting mismatched diagrams in milliseconds.

6

AUTOMATED ERROR CORRECTION

In the previous chapter the probabilistic method of detecting design errors was presented. However, the root cause of error must be diagnosed and the bugs corrected. Here we present a method that is able to correct a subset of design errors detected by probabilistic equivalence checking algorithm on HLDDs. The main results of this chapter were published in [57, 59].

Section 6.1 describes an underlying model of design errors, Section 6.2 presents the main algorithm, Section 6.3 provides the experimental results to assess this method and Section 6.4 concludes the chapter.

6.1 ERROR MODELING

Before we start fixing errors, we must first understand what a design error is. Consider the case of design given as HDL source code. Rich syntax of modern languages means a variety of ways the bug can appear in the design and a variety of ways it can be fixed. Enumerating all such ways is an immense challenge. On the other hand, using the set of HLDDs is a simple and strict way to describe digital systems that removes all syntactic sugar of HDL, but remains pretty close to it – one can easily track the correspondences between diagram nodes and design statements. Thus it is better to model errors in HLDDs, fix them and bring fixes back into the design.

Each HLDD represents the full behavior of a variable of the system and each path in the HLDD describes the behavior of the variable during a specific mode of operation of the system. Such splitting down of the behavior of a digital system into a set of well manageable atomic behaviors of variables resembles program slicing [108]. Activating paths on HLDDs is equivalent to dynamic program slicing [6] and providing efficient modeling for cause-effect relationships when handling system errors. Each path in a HLDD represents the possible locations of error causes during the related mode of operation of the system. The causes can be related either to the nodes on the path, or to the edges between the nodes.

Once an error has been detected and located in an HLDD, a subgraph of HLDD, which contains the possible causes of the error will be fixed.

Consider a buggy design, translated into HLDDs. Its errors should somehow appear in the diagrams. It appears that, unlike the case of HDL source, there is not so much choice. Something can be wrong with intermediate nodes. Something can be wrong with their interconnections. Finally, something can be wrong with terminal node functions. Thus, we can define three classes of HLDD errors:

- Node-related errors.
- Edge-related errors.
- Errors in terminal node functions.

These errors are distinguished by the procedure of fixing them. A node-related error class contains two subclasses: *missing nodes* and *unnecessary nodes*. They can be fixed by either adding or removing a node (with its edges) to/from the diagram, respectively. Edge-related errors could be of three types:

- There is an additional edge.
- An edge is missing.
- Edge directs to the wrong node.

As we do not consider verification of terminal node functions here, modeling their bugs is also behind the scope of current thesis.

Let us check how our error model correlates with others. There are not so much high-level models available. One of them is microprocessor error model presented in [96]. Consider some microprocessor, like one described in Example 3.2. Its faults can be divided into the following classes:

F1: no source is selected;

F2: a wrong source (operation) is selected;

F3: more than one source (operations) is selected;

F4: no destination is selected;

F5: instead of, or in addition to the selected correct destination, one or more other destinations are selected.

F6: one or more micro-orders not activated;

F7: micro-orders are erroneously activated;

F8: a different set of microinstructions is activated instead of, or in addition to the given microinstructions;

F9: one or more cells are stuck at 0 or 1;

F10: one or more cells fail to make a $0 \to 1$ or $1 \to 0$ transition;

F11: two or more pairs of cells are coupled;

F12: one or more lines can be stuck at 0 or 1;

F13: one or more lines may form a wired-OR or wired-AND function due to shorts or spurious coupling.

F14: data processing functional fault model, still left open.

In [104] this model was reduced to 3 HLDD fault classes:

D1: the output edge for $x_v = c$ of a node v is broken (models F1, F4, F6);

D2: the output edge for $x_v = c$ of a node v is always activated (F3, F5, F7, F8);

D3: instead of the edge for $x_v = c$ of a node v, another edge for $x_v = d$, or a set of edges $\{ d \}$ is activated (F2, F5, F8, F9-F14).

Class D1 means missing node in our model, D2 can be viewed as either all edges direct to v^c or all edges except (v, v^c, c) are missing, D3 means that edge labeled by c is pointing to v^d instead of v^c, i.e. edge directs to a wrong node. In case of multiple new destinations d a new node that models combined behavior of all descendants v^d should be introduced, then we may say that the edge is pointing to this new node.

As we see, the model from [96] can be reduced to a subset of our model. Current thesis is focused on fixing edge-related errors. An extension of this method to node-related errors is currently under development.

6.2 ERROR CORRECTION

As we saw in previous section, edge-related errors could be of three types, additional unnecessary edge, missing edge and edge with wrong destination node. First two types of errors are detectable by HLDD definition. There is a one-to-one correspondence between edges going from a node and values of the node variable. So, if the sets $D(x)$ and $D'(x)$ do not match for some variable x in graphs G and G' then the error is detected. The last type is detectable by using polynomials, as described in previous chapter.

Fixing the first type of bugs is quite trivial. As we have all the information about the error before calculating polynomials, we just remove unnecessary edges and compute the polynomial values to check that the diagrams are equivalent now. The second type of errors is also detectable before the computations and this case can be easily turned into the third one. One way to do this is to add a new dummy terminal vertex, direct the new edges there and then redirect them to the correct nodes. So, the rest of the paper describes the last case.

Let $\mathbf{r} = (r_1, \ldots, r_n) \in \mathbb{Z}_p^n$. Suppose we are computing characteristic polynomials for the diagram G using this vector. During the computation, when we are moving from a parent node v to one of its child nodes through an edge (v, w, c), we are multiplying the value P_v by the number

$$p_{(v,w,c)} = (-1)^{|D(v)|-c} \binom{r_v - 1}{c - 1} \binom{r_v - c - 1}{|D(v)| - c} \mod p$$

For every node $v \in V^N$ by G^v we denote a subgraph of G with root node v and all its descendant nodes. Also, for every pair of nodes v, w, such as w is the descendant of v, we denote by G_w^v a subgraph with the root node v, the only terminal node w and all nodes that lie in different paths from v to w. It is known that all such relationships between pairs (v, w) produce a transitive closure of the graph G, which we, as usual, denote by $G^+ = (V, E^+)$. Every graph G_w^v has its own characteristic polynomial. Let the values of these polynomials at point $\mathbf{r}$ be the edge labels of the graph G^+. We denote these labels by $P_{(v,w)}$ for every $(v, w) \in E^+$. These values can be easily computed in polynomial time. Note, that $\forall (v \in V)\ P_{(v,v)} = 1$.

Theorem 6.1. *Let $G = (V, E)$ be an HLDD, $v \in V^N$, $w, w' \in V$ and v_0 be the root node. If we redirect an edge (v, w, c) to the new destination w', then, for every terminal node v^T, the value $P_{(v_0,v^T)}$ changes by the following expression:*

$$\Delta P_{(v_0,v^T)} = P_{(v_0,v)} \cdot p_{(v,w,c)} \cdot \left(P_{(w',v^T)} - P_{(w,v^T)} \right). \tag{6.1}$$

Proof. A characteristic polynomial for some pair of vertices is a way to express paths from one vertex to another. Redirecting an edge means removing the edge (v, w, c) and adding the new edge (v, w', c), so all paths through (v, w, c) disappear and the new paths through (v, w', c) appear instead. Each path $v_0 \xrightarrow{c_0} v_1 \xrightarrow{c_1} \ldots \xrightarrow{c_{n-1}} v_n \xrightarrow{c_n} v^T$ adds $p_{(v_0,v_1,c_0)} p_{(v_1,v_2,c_1)} \ldots p_{(v_n,v^T,c_n)}$ to the polynomial value.

So, we have to subtract those products, that contain $p_{(v,w,c)}$ and add the products for the new paths:

$$\begin{aligned}
&\Delta P_{(v_0,v^T)} && = \\
&\quad \sum_{\substack{\{l(v_0,v^T)| \\ (v,w',c)\in l(v_0,v^T)\}}} \left(\prod_{e\in l(v_0,v)} p_e\right) p_{(v,w,c)} \prod_{e\in l(w',v^T)} p_e && - \\
&\quad \sum_{\substack{\{l(v_0,v^T)| \\ (v,w,c)\in l(v_0,v^T)\}}} \left(\prod_{e\in l(v_0,v)} p_e\right) p_{(v,w,c)} \prod_{e\in l(w,v^T)} p_e && = \\
&p_{((v,w,c))} \left(\sum_{\{l(v_0,v)\}} \prod_{e\in l(v_0,v)} p_e\right) && \times \\
&\left(\sum_{\{l(w',v^T)\}} \prod_{e\in l(w',v^T)} p_e - \sum_{\{l(w,v^T)\}} \prod_{e\in l(w,v^T)} p_e\right) && = \\
&P_{(v_0,v)} \cdot p_{(v,w,c)} \cdot \left(P_{(w',v^T)} - P_{(w,v^T)}\right). &&
\end{aligned}$$

□

Corollary 6.1. *Assume we have redirected an edge (v,w,c) to some new node w' and calculated the new value $P^{new}_{(v,v^T)}$. Then, for some node $u \neq v,w$, $\Delta P_{(u,v^T)} = P_{(u,v)}\left(P^{new}_{(v,v^T)} - P^{old}_{(v,v^T)}\right)$, where $P^{old}_{(v,v^T)}$ denotes the value of corresponding characteristic polynomial before modifying the diagram.*

Proof. As $P_{(v,v^T)} = \sum_{v^c\in\Gamma(v)} p_{(v,v^c,c)} P_{(v^c,v^T)}$, then using Theorem 6.1 we obtain

$$\begin{aligned}
&\Delta P_{(u,v^T)} = P_{(u,v)} p_{(v,w,c)} \left(P_{(w',v^T)} - P_{(w,v^T)}\right) && = \\
&P_{(u,v)} \sum_{\substack{v^d\in\Gamma(v) \\ (v,v^d,d)\neq(v,w',c)}} p_{(v,v^d,d)} P_{(v^d,v^T)} && - \\
&P_{(u,v)} \sum_{\substack{v^d\in\Gamma(v) \\ (v,v^d,d)\neq(v,w,c)}} p_{(v,v^d,d)} P_{(v^d,v^T)} && + \\
&P_{(u,v)} \left(p_{(v,w',c)} P_{(w',v^T)} - p_{(v,w,c)} P_{(w,v^T)}\right) && = \\
&P_{(u,v)} \left(P^{new}_{(v,v^T)} - P^{old}_{(v,v^T)}\right) &&
\end{aligned}$$

□

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two diagrams with different sets of characteristic polynomials, giving us different values at vector $\mathbf{r}$. Let S_1 and S_2 be these sets of values. The goal of the error correction procedure is to make these two sets equal. For that purpose we redirect up to k edges from one node to another, and recalculate the values in S_2, where the choice of k is empirical. The algorithm contains 4 steps:

1. Choose a subset E_2' of E_2 where $|E_2'| \leq k$.
2. Choose a subset $W_2' \subset V_2$, $|W_2'| = |E_2'|$ and redirect all $e = (v, w, c) \in E_2'$ to their new destinations $w' \in W_2'$, making sure that the new edges would not produce any cycles.
3. Recalculate the values from S_2, obtaining a new set S_2'.
4. If $S_2' = S_1$ generate a new random vector $\mathbf{r}'$, recalculate S_2' and S_1 using the new vector. If the equality holds then finish, otherwise go to step 1.

For the first two steps a variety of algorithms from exhaustive search to advanced heuristics can be applied. In our implementation we use the following method. First of all, giving a new destination to an edge affects certain subset of terminal nodes (maybe the full set V^T), so we choose only those sets E_2' and W_2' that affect values of failed nodes leaving correct values intact. It is easy to see, which value is affected by redirection and which not if we have computed relationships of the transitive closure. After that we applied the next observation: assume, that for some subsets E_2' and W_2' of length $l < k$ some, but not all failed values were fixed. Probably, this means that we need to add more elements to those sets, so that these elements would fix the rest of the values. Thus, we suspend the search of solution among other l-elemental subsets and switch to supersets of E_2' and W_2'. As you will see in the next section this method is quite fast.

For the 3rd step we apply Algorithm 6.1, which uses the results obtained in Theorem 6.1 and Corollary 6.1. It is easy to see, that the redirection of one particular edge does not affect any polynomial value for vertices that lie after its source node v. On the other hand, polynomial values for nodes, that are located before v are about to change. That is why in the for-cycle at line 8 we iterate through the set of errors starting from the ones that are located closer to terminals according to topological order. At line 12 we store the current polynomial value for further use and at the next line we apply Theorem 6.1 for graph $G^v_{v^T}$, keeping in mind that $P_{(v,v)} = 1$. This operation changes polynomial values for all vertices that appear before v in topological order. But we do not need to update all of them, only the values we are going to use further should be changed. Those values are:

Algorithm 6.1 Recalculate characteristic polynomial values

1: **function** RECALCULATEPOLYNOMIALS($G = (V, E)$, $E' \subset E$, $W' \subset W$)
Require: $|E'| = |W'|$
2: order all nodes in G topologically
3: order all edges in E' topologically by source nodes
4: order all nodes in W' according to E' ordering
5: $T[1 \dots |E'|] \leftarrow$ array of ordered edges from E'
6: $U[1 \dots |W'|] \leftarrow$ array of ordered nodes from W'
7: $v_0 \leftarrow rootnode$
8: **for** $i = |E'|$ **downto** 1 **do**
9: $e = (v, w, c) \leftarrow T[i]$
10: $w' \leftarrow U[i]$
11: **for all** $v_t \in V^T$ **do**
12: $P^{old}_{(v,v_t)} = P_{(v,v_t)}$
13: $P_{(v,v_t)} \leftarrow P_{(v,v_t)} + p_e(P_{(w',v_t)} - P_{(w,v_t)})$
14: **end for**
15: $updated = \emptyset$
16: **for all** $(v_1, w_1, c_1) \in \{T[1], \dots, T[i-1]\}$ **do**
17: $w'_1 \leftarrow$ corresponding vertex from W'
18: $UpdateValues(w'_1, v, updated)$
19: $UpdateValues(w_1, v, updated)$
20: $UpdateValues(v_1, v, updated)$
21: **end for**
22: $UpdateValues(v_0, v, updated)$
23: **end for**
24: **return** $\{P_{(v_0,v_t)} \mid v_t \in V^T\}$
25: **end function**

26: **procedure** UPDATEVALUES(v, w, $updated$)
27: **if** $v \notin updated$ **then**
28: **for all** $v_t \in V^T$ **do**
29: $P_{(v,v_t)} \leftarrow P_{(v,v_t)} + P_{(v,w)}(P_{(w,v_t)} - P^{old}_{(w,v_t)})$
30: **end for**
31: $updated = updated \cup \{v\}$
32: **end if**
33: **end procedure**

- Source, old and new destination nodes for each error that still remains unprocessed. We do this at lines 16-21, using the function $UpdateValues$.

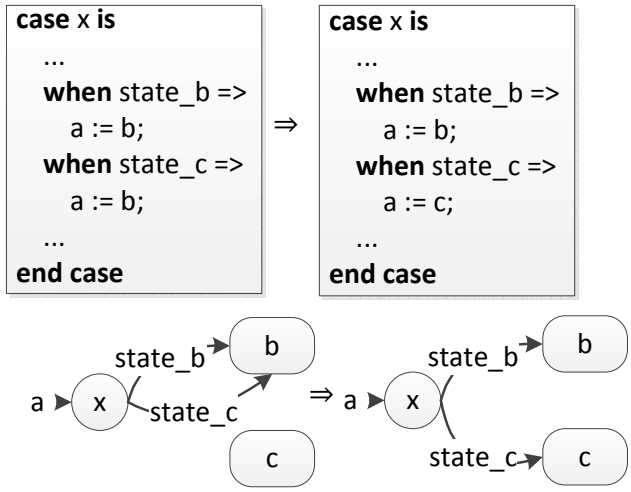

Figure 6.1 – Fix by replacing the right side of assignment

Note, that some destination node w_i may appear after the node v in topological order, in this case $P_{(w_i,v)} = 0$ and $P_{(w_i,v^T)}$ remain intact.

- The root node. However, in this case if there is some edge with source v_0 in E', then it is already updated at the previous step. Otherwise we update it at line 22.

In function $UpdateValues$ we check first whether the polynomial values for vertex v was not updated before, and if not then apply Corollary 6.1.

So, after i-th iteration of the external cycle we obtain the polynomial values of graph $G_{v^T}^{v_0}$ with $|E'| - i + 1$ redirected edges, and thus, after $|E'|$ steps we get desired output.

The complexity of Algorithm 6.1 can be found in rather straightforward way. The function $UpdateValues$ has one cycle of length $|V^T|$, in which we perform elementary addition and multiplication operations, thus its complexity is $O(|V^T|)$. The main algorithm has the outer cycle of length $|E'| \leq k$ and two inner cycles of lengths $|V^T|$ and $|E'|$. The second inner cycle contain the third one, hidden inside the function $UpdateValues$. Thus, overall complexity is $O(k^2|V^T|)$. This speeds up the polynomial recalculation process, comparing to the straightforward way of applying Algorithm 5.2 each time we redirect an edge, as the latter approach result a procedure of complexity $O(k * |E|) = O(k * |V|^2)$.

Generally, each intermediate node has a corresponding condition inside some **if**, **case**, **while**, etc, statement. Redirecting its outgoing edge(s) means making changes either in this condition or, if we move the edge from one terminal node

Table 6.1 – Error Correction experimental results.

Benchmark	Diagrams	Nodes	Terminal Nodes	Edges	Bugs	Time	Avg. Time
b00	4	35	16	42	4	0.032	0.008
b01	3	26	13	44	5	0.022	0.0044
b02	2	16	9	24	4	0.015	0.0038
b03	15	137	54	187	14	0.468	0.033
b04	12	58	30	59	11	0.041	0.0037
b06	5	44	18	77	8	0.163	0.02
b09	5	44	18	62	4	0.027	0.0068
b10	15	127	57	266	13	0.17	0.013
b13	24	178	94	252	14	0.065	0.0046
HC11	98	979	326	8856	14	5.905	0.42
CRC	35	234	86	282	14	3.657	0.28
UART	421	1747	872	9706	14	0.325	0.023

to another, in the assignment statement of the variable, that behavior is expressed by the diagram. Figure 6.1 provides an example of the latter case. Thus, our method is able to fix **inconsistencies in the control part of design and incorrect assignment statements, including state transitions**.

6.3 EXPERIMENTS

In this section we present the experiments which had the goal to investigate the feasibility of the method and to estimate the average time cost needed for repairing design errors in the HLDD based design description.

Let us first check, how our method would fix the bug from Example 5.1, where we forgot to remove inversion in condition c_c from the third **if** statement. This condition corresponds to the bottommost node labelled by c_c. We denote it by c_c_3. In Section 5.4 we got the following results of equivalence checking procedure: for vector r = ($state$, c_a, c_b, c_c) = (3410646289, 3410646289, 3410646289, 1572882280) the characteristic polynomial values of the terminal nodes of implementation diagram (Figure 5.3) were (in_x, reg_x, $reg\_x + in\_a$, $reg\_x + in\_b$, $reg\_x + in\_c$) = (4090461347, 4216261509, 2982678470, 1627467403, 4263000411), while the specification ones were (in_x, x, $x + b$, $x + c$, $x + a$) = (4090461347, 4216261509, 1627467403, 4263000411, 2982678470). After searching among possible fix candidates we find that swapping around edges 1 and 0 from c_c_3 would produce the desired output: (in_x, reg_x, $reg\_x + in\_a$, $reg\_x + in\_b$, $reg\_x + in\_c$) = (4090461347, 4216261509, 1627467403, 4263000411, 2982678470).

Let us now switch to the real designs. Again, as in the previous chapter, we generated HLDDs for circuits from the ITC99 benchmark set [30], supplemented with a commercial processor core HC11 [47] from Green Mountain Inc., a VERTIGO benchmark CRC (Circular Redundancy Check) [43] and a top-level design of communication controller UART16750 [76] from the OpenCores community. Then we randomly selected some diagrams from the design, some edges in these diagrams, with no more then 3 edges per single graph, and redirected these edges to new, again, randomly selected destinations. After that we applied described algorithms to fix the bugs. The only information about the specification was polynomial values and terminal node mapping from correct graphs to the faulty ones (faulty graphs may have different topological order) which was enough to restore the correct circuit.

Table 6.1 summarizes results of our experiments. First five columns provide some information about the complexity of benchmark circuits: name of circuit, number of graphs (each graph represents a subcircuit) , number of internal and terminal nodes in the model, and number of edges in all graphs. Column 6 represents the number of errors we injected for each circuit, keeping the maximum number of bugs per graph to 3. The run times (in seconds) on each set of diagrams are provided in column 7. Finally, the last column shows the average time required to fix one bug. Experiments were carried out on a computer with Intel Core 2 Duo P7350 2.0 GHz processor and 3 GB RAM.

6.4 SUMMARY

Contemporary HDL languages have rich syntax and trying to enumerate all error types a designer can make writing code is a complex task. Luckily, HDL designs can be compiled into the set of HLDDs the variety of HDL bugs can be translated into a few atomic HLDD error classes. This HLDD-based error model is the first contribution of the current chapter.

A subclass of our model, edge-related errors can be automatically fixed. The straightforward way to do it is to redirect all edges to all possible new destinations and recalculate polynomials. Fortunately, we can apply certain filters allowing us to throw away the redirections that certainly would not fix the diagram and organize the search in order to find the whole set of fixes faster. This is the second contribution of this chapter.

Cashing intermediate results and using results of Theorem 6.1 and Corollary 6.1 allow us to recalculate polynomial values on the fly. The algorithm of fast recalculating polynomial values is the third contribution of this chapter.

It is important to note that proposed method is fully formal and does not rely on simulation results as most of the other error correction methods do.

This chapter also reports the experimental results performed on the subset of ITC'99 benchmarks supplemented with some industrial designs. These results prove the feasibility of proposed methodology making multiple fixes in just few seconds for most complex designs.

7

CONCLUSIONS AND FUTURE WORK

The aim of the thesis is to propose a novel methodology for equivalence checking and automated error correction at higher abstraction levels. The proposed methodology appends the theory of high-level decision diagrams, served as a basis for representing digital systems in this work, with canonical characteristic polynomials. Their mathematical properties allow developing new algorithms for manipulating discrete functions represented by HLDDs and apply them for solving these tasks.

This chapter summarizes the thesis, bringing together its contributions and discusses the promising directions of further research.

7.1 CONCLUSIONS

The thesis presents a new approach of equivalence checking and error correction and uses High-Level Decision Diagrams as the key data structure for modeling digital systems, originally presented in some traditional form, like HDL source code. An HLDD translator compiles the design into a set HLDDs, using, depending on original description, the symbolic execution or iterative superposition procedure or a combination of them. Then, given two HLDD sets, representing one system at different levels we are able, after some preparation, compute characteristic polynomial values at a random point and perform the equivalence checking. In case of a failing check we can automatically fix some subset of errors.

The feasibility of proposed methodology is proven by the set of experiments done for each of the two tasks addressed by the thesis. ITC99 benchmarks and larger industrial designs have been used to assess presented methods.

7.1.1 *Contributions*

As the thesis addresses related, but different problems, we can divide the contributions of it into three groups.

- ***General contributions for the theory of HLDDs***.
  - A formal definition of HLDD as a data structure for modeling digital system at higher levels is given.
  - Characteristic polynomials are applied to canonically represent the non-terminal part of HLDDs.
  - An algorithm for normalizing different variable sets is given.
  - An algorithm of removing miscomparing states produced by auxiliary variables.
- ***Contributions to equivalence checking methodology***.
  - An algorithm of computing characteristic polynomial values at random vectors on the given diagram, with speed-up achieved due to the use of binomial coefficients
  - A new measure of collision probability for this algorithm, more accurate in the general case than Schwartz-Zippel theorem usually applied for similar tasks.
- ***Contributions to the problem of automated error correction***.
  - A novel HLDD-based error model.
  - A new algorithm for fixing edge-related errors on HLDDs by manipulating characteristic polynomial values.

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

7.1.2 *Advantages*

The proposed method of analyzing discrete functions by manipulating characteristic polynomial values over HLDDs achieves very promising results in the field of formal verification. Its fast probabilistic algorithms allow comparing HLDDs in milliseconds and fixing errors in seconds. The close correspondence between HLDDs and original representations permits easily bring fixes back into the design, saving valuable time for design and verification engineers. Presented techniques are fully formal and do not rely on simulation in any extent. All proposed

approaches work completely at higher levels, in compliance with modern tendencies in the area of electronic design automation.

The most important drawback of HLDDs, their uncanonicity, is now successfully resolved enabling their use in areas, where canonical representations are required, like the case of formal verification addressed by this thesis. This result gives a way for very interesting and promising ideas of further investigation of HLDD properties and their application in various other fields. Some of these ideas are listed in the next section.

7.2 FUTURE WORK

This section outlines the issues, which require further research in order to improve and advance the proposed techniques and discovers new possible fields of application for the theory of HLDDs.

The proposed error correction algorithm is targeting edge-related errors on HLDDs. The next step is to extend it for covering node-related ones. Fixing a node-related error generally means that we append the design implementation with a small piece of missing functionality. At first glance, the task seems harder than fixing edge-related errors, but not so much, as clauses similar to Theorem 6.1 can be proven for node related-errors as well, and the problem is just to develop efficient filtering procedure that would restrict the search space.

Another promising direction is to cover partially specified systems, where a full HLDD model for specification cannot be constructed. In this case two ideas can be investigated: incorporate partial HLDDs into the set of implementation diagrams and compare the result with the unaltered set; develop the theory of characteristic polynomials further making possible not only to test the equivalence relation but inclusion as well.

In this thesis we studied the problem of equivalence checking. Another field of formal verification is model checking. It is possible to express PSL properties with the extended HLDD model called THLDD [54]. It would be interesting to try to adapt proposed techniques to this field.

In the theory of HLDDs one of the most promising ideas is to apply this structure to model software programs, as the development of system software and hardware design are quickly merging at the moment. One of the direct results would be paralleling sequential programs. In addition to this, software analogues of methods developed for various fields of hardware design and test can be investigated.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] V. D. Agrawal and D. Lee. Characteristic polynomial method for verification and test of combinational circuits. In *Proceedings of the 9th International Conference on VLSI Design: VLSI in Mobile Communication*, VLSID '96, pages 341–, Washington, DC, USA, 1996. IEEE Computer Society.

[2] S.B. Akers. Functional testing with binary decision diagrams. *J. of Design Automation and Fault-Tolerant Computing*, 2:311–331, 1978.

[3] Alessandro Armando and Enrico Giunchiglia. Embedding complex decision procedures inside an interactive theorem prover. *Annals of Mathematics and Artificial Intelligence*, 8:475–502, 1993.

[4] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[5] K.A. Atkinson. *An Introduction to Numerical Analysis*. John Wiley and Sons, 2 edition, 1988.

[6] J. Laski B. Korel. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[7] M.S. Bazaraa, J.J. Jarvis, and H.D. Sherali. *Linear Programming and Network Flows*. John Wiley & Sons, 2009.

[8] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST: Applications to software engineering. *INT. J. SOFTW. TOOLS TECHNOL. TRANSFER*, 2007.

[9] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[10] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[11] G. P. Bischoff. Formal implementation verification of the bus interface unit for the alpha 21264 microprocessor. In *Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, ICCD '97, pages 16–, Washington, DC, USA, 1997. IEEE Computer Society.

[12] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *Computers, IEEE Transactions on*, 45(9):993 –1002, sep 1996.

[13] R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. In J. E. Hayes, D. Michie, and J. Richards, editors, *Machine intelligence 11*, pages 83–124. Oxford University Press, Inc., New York, NY, USA, 1988.

[14] Daniel Brand. Verification of large synthesized designs. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, ICCAD '93, pages 534–537, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[15] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] Randal E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. Comput.*, 40(2):205–213, February 1991.

[17] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992.

[18] Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits with binary moment diagrams. In *IN PROCEEDINGS OF THE 32ND ACM/IEEE DESIGN AUTOMATION CONFERENCE*, pages 535–541, 1995.

[19] Randal E. Bryant and Carl johan Seger. Formal verification of digital circuits using symbolic ternary system models. pages 121–146. American Mathematical Society, 1990.

[20] Jerry R. Burch and Vigyan Singhal. Robust latch mapping for combinational equivalence checking, 1998.

[21] K.-H. Chang, I. Wagner, V. Bertacco, and I. L. Markov. Automatic error diagnosis and correction for RTL designs. *High Level Design Validation and Test, (HLDVT) IEEE Int'l Workshop*, pages 65–72, 2007.

[22] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In Rajeev Alur and Thomas Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422. Springer Berlin / Heidelberg, 1996.

[23] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[24] E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, ICCAD '95, pages 159–163, Washington, DC, USA, 1995. IEEE Computer Society.

[25] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[26] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[27] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, November 1989.

[28] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 365–373, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[29] Pierluigi Crescenzi and Viggo Kann. A compendium of NP optimization problems `http://www.nada.kth.se/~viggo/problemlist/compendium.html`.

[30] Scott Davidson. ITC'99 benchmark circuits - preliminary results. In *ITC*, page 1125. IEEE Computer Society, 1999.

[31] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[32] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[33] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. *Software Testing, Verifcation and Validation (ICST), International Conference on*, 2010.

[35] R. Drechsler and B. Becker. *Binary Decision Diagrams: Theory and Implementation.* Kluwer Academic Publishers, 1998.

[36] Rolf Drechsler, Bernd Becker, and Stefan Ruppertz. The k*bmd: A verification data structure. *IEEE Des. Test*, 14(2):51–59, April 1997.

[37] Elena Dubrova and Harald Sack. Probabilistic verification of multiple-valued functions. In *ISMVL*, pages 460–466, 2000.

[38] P.E. Dunne. An annotated list of selected NP-complete problems `http://www.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html`.

[39] Bruno Dutertre and Leonardo De Moura. The yices SMT solver. Technical report, Computer Science Laboratory, SRI International, 2006.

[40] C. A. J. Van Eijk and J. A. G. Jess. Detection of equivalent state variables in finite state machine verification. In *In Proc of the International Workshop on Logic Synthesis*, pages 3–35, 1995.

[41] E. Emerson and Edmund Clarke. Characterizing correctness properties of parallel programs using fixpoints. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer Berlin / Heidelberg, 1980.

[42] Lance Fortnow and Steve Homer. A short history of computational complexity. In *The History of Mathematical Logic*. North-Holland, 2002.

[43] FP6 VERTIGO Project. CRC benchmark `http://www.vertigo-project.eu`.

[44] John Franco and John Martin. *A History of Satisfiability*, chapter 1, pages 3–74. Volume 185 of Biere et al. [10], February 2009.

[45] Amir Masoud Gharehbaghi and Masahiro Fujita. Formal verification guided automatic design error diagnosis and correction of complex processors. *High-Level Design, Validation, and Test Workshop, IEEE International*, 0:121–127, 2011.

[46] Fausto Giunchiglia and Roberto Sebastiani. Building decision procedures for modal logics from propositional decision procedures — the case study of modal k. In M. McRobbie and J. Slaney, editors, *Automated Deduction — Cade-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 583–597. Springer Berlin / Heidelberg, 1996.

[47] Green Mountain Computing Systems, Inc. HC11 CPU core, `http://www.gmvhdl.com/hc11core.html`.

[48] Juris Hartmanis and Richard Stearns. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306, 1965.

[49] Matthew Hicks, Murph Finnicum, Samuel T. King, Milo M. K. Martin, and Jonathan M. Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. *Security and Privacy, IEEE Symposium on*, 0:159–172, 2010.

[50] Stefan Höreth and Rolf Drechsler. Formal verification of word-level specifications. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '99, New York, NY, USA, 1999. ACM.

[51] Shi-Yu Huang, Kwang-Ting Cheng, Kuang-Chien Chen, and Juin-Yeu Joseph Lu. Fault-simulation based design error diagnosis for sequential circuits. In *Proceedings of the 35th annual Design Automation Conference*, DAC '98, pages 632–637, New York, NY, USA, 1998. ACM.

[52] Intel Corporation. FDIV replacement program, `http://www.intel.com/support/processors/pentium/sb/CS-012748.htm`.

[53] Jawahar Jain, Jim Bitner, Donald S. Fussell, and Jacob A. Abraham. Probabilistic design verification. In *ICCAD*, pages 468–471, 1991.

[54] Maksim Jenihhin. *Simulation-Based Hardware Verification with High-Level Decision Diagrams*. PhD thesis, Tallinn University of Technology, Tallinn, Estonia, 2008.

[55] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[56] Anton Karputkin, Raimund Ubar, Jaan Raik, and Mati Tombak. Canonical representations of high-level decision diagrams. *Estonin Journal of Engineering*, 16:39–55, 2010.

[57] Anton Karputkin, Raimund Ubar, Mati Tombak, and Jaan Raik. Interactive presentation abstract: Automated correction of design errors by edge redirection on high-level decision diagrams. In *Proc. of 16th IEEE International High Level Design Validation and Test Workshop (HLDVT)'11*, page 83, 2011.

[58] Anton Karputkin, Raimund Ubar, Mati Tombak, and Jaan Raik. Probabilistic equivalence checking based on high-level decision diagrams. In *Proc. of 14th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)'11*, pages 423–428, 2011.

[59] Anton Karputkin, Raimund Ubar, Mati Tombak, and Jaan Raik. Automated correction of design errors by edge redirection on high-level decision diagrams. In *Proc. of 13th International Symposium on Quality Electronic Design (ISQED)'12*, pages 686–693, 2012.

[60] R. Konighofer and R. Bloem. Automated error localization and correction for imperative programs. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 91 –100, 30 2011-nov. 2 2011.

[61] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[62] Andreas Kuehlmann, Arvind Srinivasan, and David P. Lapotin. Verity – a formal verification program for custom cmos circuits. *IBM JOURNAL OF RESEARCH AND DEVELOPMENT*, 39:149–165, 1994.

[63] Shuvendu K. Lahiri, Sanjit A. Seshia, Randal E. Bryant, and Al E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID, 2002.

[64] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the 29th ACM/IEEE Design*

Automation Conference, DAC '92, pages 608–613, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[65] William K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2008.

[66] Luciano Lavagno, Grant Martin, and Louis Scheffer. *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set*. CRC Press, Inc., Boca Raton, FL, USA, 2006.

[67] C. Y. Lee. Representation of switching circuits by binary decision programs. *The Bell System Technical Journal*, pages 985–999, 1959.

[68] Leonid A. Levin. Universal search problems. *Problemy Peredaci Informacii*, 9:115–116, 1973.

[69] M. H. MacDougall. Computer system simulation: An introduction. *ACM Comput. Surv.*, 2(3):191–209, September 1970.

[70] J.C. Madre, O. Coudert, and J.P. Billon. Automating the diagnosis and the rectification of design errors with PRIAM. In *Proc. of IEEE International Conference on Computer-Aided Design (ICCAD)'89*, pages 30 – 33, 1989.

[71] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[72] D. Michael Miller. Multiple-valued logic design tools. In *ISMVL*, pages 2–11, 1993.

[73] John Moondanos, Carl Seger, Ziyad Hanna, and Daher Kaiss. Clever: Divide and conquer combinational logic equivalence verification with false negative elimination. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 131–143. Springer Berlin / Heidelberg, 2001.

[74] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *ANNUAL ACM IEEE DESIGN AUTOMATION CONFERENCE*, pages 530–535. ACM, 2001.

[75] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[76] OpenCores. UART16750 communication controller: `http://opencores.org/project,uart16750`.

[77] Amir Pnueli, Yoav Rodeh, Ofer Shtrichman, and Michael Siegel. Deciding equality formulas by small domains instantiations. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 687–688. Springer Berlin / Heidelberg, 1999.

[78] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin / Heidelberg, 1982.

[79] Jaan Raik, Urmas Repinski, Hanno Hantson, Maksim Jenihhin, Giuseppe Di Guglielmo, Graziano Pravadelli, and Franco Fummi. Combining dynamic slicing and mutation operators for ESL correction. In *IEEE 17th European Test Symposium*, 2012.

[80] Jaan Raik, Urmas Repinski, Raimund Ubar, Maksim Jenihhin, and Anton Chepurov. High-level design error diagnosis using backtrace on decision diagrams. In *Proc. of 28th Norchip Conference*. IEEE, 2010.

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

[82] R Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, April 1987.

[83] SAT Competition 2011. Results of SAT11 competition `http://www.cril.univ-artois.fr/SAT11/`.

[84] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, October 1980.

[85] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *DIMACS SERIES IN DISCRETE MATHEMATICS AND THEORETICAL COMPUTER SCIENCE*, pages 521–532, 1995.

[86] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28:59–98, 1948.

[87] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 108–125, London, UK, UK, 2000. Springer-Verlag.

[88] Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, July 1978.

[89] Robert E. Shostak. Deciding combinations of theories. In *Proceedings of the 6th Conference on Automated Deduction*, pages 209–222, London, UK, UK, 1982. Springer-Verlag.

[90] MG Siegler. Google has a secret fleet of automated toyota priuses; 140,000 miles logged so far, `http://techcrunch.com/2010/10/09/google-automated-cars/`.

[91] Siemens AG. Siemens benchmark suite, `http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/`.

[92] João P. Marques Silva and Karem A. Sakallah. Grasp – a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[93] Alexander Smith, Andreas Veneris, Moayad Fahim Ali, and Anastasios Viglas. Fault diagnosis and logic debugging using boolean satisfiability. *IEEE TRANS. ON CAD*, 24:1606–1621, 2005.

[94] Gordon L. Smith, Ralph J. Bahnsen, and Harry Halliwell. Boolean comparison of hardware and flowcharts. *IBM J. Res. Dev.*, 26(1):106–116, January 1982.

[95] Arthur G. Stephenson, Lia S. LaPiana, Daniel R. Mulville, Peter J. Rutledge, Frank H. Bauer, David Folta, Greg A. Dukeman, and Robert Sackheim et al. Mars climate orbiter mishap investigation board phase i report. *Press Release*, 1999.

[96] S. M. Thatte and J. A. Abraham. Test generation for microprocessors. *IEEE Trans. Comput.*, 29(6):429–441, June 1980.

[97] Masahiro Tomita, Tamotsu Yamamoto, Fuminori Sumikawa, and Kotaro Hirano. Rectification of multiple logic design errors in multiple output circuits. In *Proceedings of the 31st annual Design Automation Conference*, DAC '94, pages 212–217, New York, NY, USA, 1994. ACM.

[98] H J Touati, H Savoj, B Lin, R K Brayton, and A Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's, 1990.

[99] R. Ubar, J. Raik, A. Karputkin, and M. Tombak. Synthesis of high-level decision diagrams for functional test pattern generation. In *Mixed Design of Integrated Circuits & Systems, 2009. MIXDES'09. MIXDES-16th International Conference*, pages 519–524. IEEE, 2009.

[100] Raimund Ubar. Test generation for digital circuits using alternative graphs. *Proc. of Tallinn Technical University*, 409:75–81, 1976.

[101] Raimund Ubar. Alternative graphs and technical diagnosis of digital devices. *Electronic technique*, 8(5(132)):33–57, 1988.

[102] Raimund Ubar. Test synthesis with alternative graphs. *IEEE Design and Test of Computers*, 13(1):48–57, 1996.

[103] Raimund Ubar, Gert Jervan, Jaan Raik, Maksim Jenihhin, and Peeter Ellervee. Dependability evaluation in fault-tolerant systems with high-level decision diagrams. In *Proc. of the Computer Science meets Automation*, pages 147 – 152, 2007.

[104] Raimund Ubar, Jaan Raik, Artur Jutman, Maksim Jenihhin, Martin Instenberg Marina Brik, and Heinz-Dietrich Wuttke. Diagnostic modeling of microprocessors with high-level decision diagrams. In *Proc. of 11th Intl. Biennial Baltic Electronic Conference*, pages 147–150, 2008.

[105] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[106] Andreas Georgios Veneris. *Multiple design error diagnosis and correction in digital VLSI circuits*. PhD thesis, Champaign, IL, USA, 1998. UML Order No. GAX99-12408.

[107] A. Wahba and D. Borrione. Automatic diagnosis may replace simulation for correcting simple design errors. In *Proceedings of the conference on European design automation*, EURO-DAC '96/EURO-VHDL '96, pages 476–481, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

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

[109] Jian Zhang. Symbolic execution of program paths involving pointer and structure variables. In *Proceedings of the Quality Software, Fourth International Conference*, QSIC '04, pages 87–92, Washington, DC, USA, 2004. IEEE Computer Society.

[110] R. Zippel. An explicit separation of relativised random polynomial time and relativised deterministic polynomial time. *Inf. Process. Lett.*, 33(4):207–212, December 1989.

CURRICULUM VITAE

CURRICULUM VITAE

PERSONAL DATA

Name: Anton Karputkin
Date of Birth: October 10, 1983
Place of Birth: Tallinn, Estonia
Citizenship: Estonian

CONTACT DATA

Address: 15 Raja St., 12618 Tallinn, Estonia
Phone: +372 55 687 493
E-mail: akarputkin@gmail.com

EDUCATION

2008 – ...	PhD studies in Information and Communication Technology, Tallinn University of Technology (TUT)
2005 – 2008	M.Sc. in Computer Science, University of Tartu
2002 – 2005	B.Sc. (cum laude) in Computer Science, University of Tartu
1999 – 2002	Upper Secondary Education from Tallinn Tõnismäe Real School
1990 – 1999	Secondary Education from Lasnamäe Russian Upper Secondary School

CAREER

2010 – 2011	Software Developer, Sanoma Baltics AS
2008 – 2009	Software Developer, Exact Holdings OÜ
2006 – 2008	Software Developer, Roosa Gepard OÜ

AWARDS

2011	Scholarship of Estonian Information Technology Foundation (EITSA)
2006	1st Prize (28th Place), 13th International Mathematics Competition, Odessa, Ukraine

ELULOOKIRJELDUS

ISIKUANDMED

Nimi: Anton Karputkin
Sünniaeg: 10. Oktoober, 1983
Sünnikoht: Tallinn, Eesti
Kodakondsus: Eesti

KONTAKTANDMED

Aadress: Raja 15, 12618 Tallinn, Eesti
Tel.: +372 55 687 493
E-post: akarputkin@gmail.com

HARIDUSKÄIK

2008 – ...	doktoriõpe, info- ja kommunikatsioonitehnoloogia õppekava, Tallinna Tehnikaülikool (TTÜ)
2005 – 2008	Magistrikraad informaatikas, Tartu Ülikool
2002 – 2005	Bakalaureusekraad (cum laude) informaatikas, Tartu Ülikool
1999 – 2002	Keskharidus, Tallinna Tõnismäe Realkool
1990 – 1999	Põhiharidus, Lasnamäe Vene Gümnaasium

TEENISTUSKÄIK

2010 – 2011	Tarkvaraarendaja, Sanoma Baltics AS
2008 – 2009	Tarkvaraarendaja, Exact Holdings OÜ
2006 – 2008	Tarkvaraarendaja, Roosa Gepard OÜ

SAAVUTUSED JA PREEMIAD

2011	Eesti Infotehnoloogia Sihtasutuse (EITSA) stipenidium
2006	1. auhind (28. koht), 13. Rahvusvaheline Matemaatika Võistlus, Odessa, Ukraina

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.