ADVANCED BDD OPTIMIZATION
Advanced BDD Optimization

by

RÜDIGER EBENDT
University of Bremen, Germany

GÖRSCHWIN FEY
University of Bremen, Germany

and

ROLF DRECHSLER
University of Bremen, Germany

Springer
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td></td>
<td>ix</td>
</tr>
<tr>
<td>1.</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>PRELIMINARIES</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>Notation</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Boolean Functions</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Decomposition of Boolean Functions</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Reduced Ordered Binary Decision Diagrams</td>
<td>14</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Syntactical Definition of BDDs</td>
<td>15</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Semantical Definition of BDDs</td>
<td>18</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Reduction Operations on SBDDs</td>
<td>20</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Variable Orderings of BDDs</td>
<td>24</td>
</tr>
<tr>
<td>2.4.5</td>
<td>BDD Levels and their Sizes</td>
<td>27</td>
</tr>
<tr>
<td>2.4.6</td>
<td>Implementation of BDD Packages</td>
<td>29</td>
</tr>
<tr>
<td>2.4.7</td>
<td>Basic Minimization Algorithm</td>
<td>32</td>
</tr>
<tr>
<td>2.4.8</td>
<td>Evaluation with BDDs</td>
<td>35</td>
</tr>
<tr>
<td>2.4.9</td>
<td>Paths in BDDs</td>
<td>37</td>
</tr>
<tr>
<td>3.</td>
<td>EXACT NODE MINIMIZATION</td>
<td>45</td>
</tr>
<tr>
<td>3.1</td>
<td>Branch and Bound Algorithm</td>
<td>47</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Classical Approach</td>
<td>47</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Lower Bound Technique</td>
<td>52</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Algorithm</td>
<td>60</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Experimental Results</td>
<td>63</td>
</tr>
<tr>
<td>3.2</td>
<td>$A^*$-based Optimization</td>
<td>67</td>
</tr>
<tr>
<td>3.2.1</td>
<td>State Space Search by $A^*$-Algorithm</td>
<td>67</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Best-First Search Algorithm</td>
<td>72</td>
</tr>
</tbody>
</table>
3.2.3 Monotonicity 80
3.2.4 Algorithm 86
3.2.5 Bottom-Up Approach 102
3.2.6 Experimental Results 109
3.3 Summary 118

4. HEURISTIC NODE MINIMIZATION 121
4.1 Efficient Dynamic Minimization 123
4.2 Improved Lower Bounds for Dynamic Reordering 126
4.3 Efficient Forms of Improved Lower Bounds 132
4.4 Combination of Improved Lower Bounds with Classical Bounds 136
4.5 Experimental Results 137
4.6 Summary 143

5. PATH MINIMIZATION 145
5.1 Minimization of Number of Paths 150
  5.1.1 Theory 150
  5.1.2 Efficient Minimization 155
  5.1.3 Experimental Results 159
5.2 Minimization of Expected Path Length 163
  5.2.1 Basic Approaches 163
  5.2.2 Fast Minimization Approach for Expected Path Length 164
  5.2.3 Node Weights 169
  5.2.4 Update Schema 175
  5.2.5 Experimental Results 180
5.3 Minimization of Average Path Length 183
  5.3.1 Applying the Unifying View 185
  5.3.2 Algorithm 190
  5.3.3 Experimental Results 191
5.4 Summary 192

6. RELATION BETWEEN SAT AND BDDS 197
6.1 Davis-Putnam Procedure 198
6.2 On the Relation between DP Procedure and BDDs 199
6.3 Dynamic Variable Ordering Strategy for DP Procedure 204
6.4 Experimental Results 206
6.5 Summary 209
Contents

7. FINAL REMARKS 211
References 213
Index 221
Preface

VLSI CAD has greatly benefited from the use of reduced ordered Binary Decision Diagrams (BDDs) and the clausal representation as a problem of Boolean Satisfiability (SAT), e.g. in logic synthesis, verification or design-for-testability. In recent practical applications, BDDs are optimized with respect to new objective functions for design space exploration. The latest trends show a growing number of proposals to fuse the concepts of BDD and SAT.

This book gives a modern presentation of the established as well as of recent concepts. Latest results in BDD optimization are given, covering different aspects of paths in BDDs and the use of efficient lower bounds during optimization. The presented algorithms include Branch and Bound and the generic $A^*$-algorithm as efficient techniques to explore large search spaces.

The $A^*$-algorithm originates from Artificial Intelligence (AI), and the EDA community has been unaware of this concept for a long time. Recently, the $A^*$-algorithm has been introduced as a new paradigm to explore design spaces in VLSI CAD. Besides AI search techniques, the book also discusses the relation to another field of activity bordered to VLSI CAD and BDD optimization: the clausal representation as a SAT problem.

When regarding BDD optimization, mainly the minimization of diagram size was considered. The present book is the first to give a unified framework for the problem of BDD optimization and it presents the respective recent approaches. Moreover, the relation between BDD and SAT is studied in response to the questions that have emerged from the latest developments. This includes an analysis from a theoretical point of view as well as practical examples in formal equivalence checking.

This book closes the gap between theory and practice by transferring the latest theoretical insights into practical applications.
In this, a solid, thorough analysis of the theory is presented, which is completed by experimental studies. The basic concepts of new optimization goals and the relation between the two paradigms BDD and SAT have been known and understood for a short time, and they will have wide impact on further developments in the field.

Acknowledgement

We would like to acknowledge the cooperation with Wolfgang Günther, Sherif Reda, and Alex Orailoglu. We would also like to thank Stefan Höhne for his help with the implementation of the techniques described in Section 5.2. Furthermore, we acknowledge valuable discussions, criticisms, and help from friends and colleagues.

Finally, we want to thank Mark de Jongh and Cindy Zitter from Kluwer Academic Publishers for their help with the preparation of the final manuscript.

Bremen, February 2005

Rüdiger Ebendt

ebendt@informatik.uni-bremen.de

Görschwin Fey

fey@informatik.uni-bremen.de

Rolf Drechsler

drechsle@informatik.uni-bremen.de
Chapter 1

INTRODUCTION

During the last twenty years, the development of methods and tools for Computer Aided Design (CAD) of Very Large Scale Integrated (VLSI) circuits and systems has been a central issue in joint efforts of academia and industry. The interest in a further completion and improvement of Electronic Design Automation (EDA) will even grow in the future.

In the progress of this development it is important that computer-aided techniques keep up with the time. Facing the ever growing complexity of present-day circuits, these methods must still be able to provide a fault-free design.

The social relevance of EDA is obvious: fields of application vary from information and automation technology to electrical engineering, traffic engineering and medical technology.

Main problems of VLSI CAD are the automated synthesis of circuits and the optimization of circuits with respect to certain objective functions, e.g. chip area or operation frequency. Automated tools for these tasks have to accomplish the processing of very large amounts of data, often representable by so-called Boolean functions.

Already in 1938, Shannon proved that a two-valued Boolean algebra (whose members are most commonly denoted 0 and 1, or false and true) can describe the operation of two-valued electrical switching circuits. Today, Boolean algebra and Boolean functions are therefore used to model the digital logic of circuits where 0 and 1 represent the states of low and high voltage, respectively. To handle the large amounts of digital data arising during electronic design, efficient data structures and algorithms are necessary.
A well-known data structure for EDA is reduced ordered Binary Decision Diagram (BDD). BDD is a graph-based data structure for efficient representation and manipulation of Boolean functions and has been introduced in [Bry86]. It is known that BDDs allow a unique and very compact representation of Boolean functions. For these reasons, efficient algorithms for BDD manipulation do exist. Besides the clausal representation as a Boolean Satisfiability (SAT) problem, BDD is a state-of-the-art data structure for applications like equivalence checking, logic synthesis, formal verification of hardware, test and symbolic representation of state spaces in general, e.g. during model checking.

BDDs are well-known from hardware verification, e.g. from functional simulation, see [AM95, MMS+95, SDB97]. BDDs are also used by techniques for logic synthesis [BNNSV97, FMM+98, MSMSL99, MBM01, YC02] and for test. Here BDDs in particular support approaches to a synthesis for testability, see [Bec92, Bec98, DSF04].

All these applications benefit from optimization of BDDs with respect to different criteria, i.e. objective functions. In this, BDD optimizations happen at a deep and abstract logic level. In a design flow, this is an early stage before the step of technology mapping. After BDD optimization, it is often possible to directly transfer the achieved optimizations to the targeted digital circuits.

In the following, examples of BDD optimization and their applications are given.

- In BDD-based functional simulation, a simulator evaluates a given function by direct use of the representing BDD. A crucial point here is the time needed to evaluate a function. Hence, BDD optimizations have been proposed to minimize evaluation time [LWHL01, ISM03].

- One way to synthesize a circuit for a given function is to directly map the representing BDD into a multiplexor circuit. It is known that optimizations of the BDD (e.g. BDD size, expected or average path length in BDDs) directly transfer to the derived circuit (e.g. area and delay minimization). With that, the targeted applications are in the field of logic synthesis.

This book presents new work in the field of BDD optimization. Efficient search algorithms are presented to determine a good or optimal variable ordering of BDDs. The classical methods are based on different search paradigms, e.g. hill climbing and Branch and Bound (B&B). A recent suggestion is to use the generic A*-algorithm. This is a fundamental search paradigm in Artificial Intelligence (AI). The use of this concept in EDA, however, is a recent innovation.
The classical criterion for the optimality of a BDD is its size, i.e. the number of nodes in the diagram. Classically, this criterion is addressed both with exact and heuristic approaches. After that, alternative criteria for BDD optimality are considered. Criteria different from BDD size have been studied in the classical works of [AM95, MMS$^+$95] and in the more recent works of [SB00, LWHL01, FD02a, NMSB03, FSD04, EGD04b]. Applications for these alternative optimizations in logic synthesis, functional simulation and test are given.

The motivation for classical size-driven BDD optimization was the reduction of memory requirement and run time of the algorithms operating on BDDs. There has always been a strong demand for BDD size optimizations since BDDs are sensitive to variable ordering.

That is, dependent on the fixed order, in which input variables are tested along the paths from the root node to the terminal node, the size of the diagram may vary from linear to exponential [Bry86].

Hence, many approaches to determine at least a “good” ordering have been proposed in the past.

These methods can be roughly classified as:

- topology-based heuristics
- dynamic reordering
- simulation-based methods
- exact methods

The first class of methods are topology-based heuristics where structural information about a given circuit is used to determine a good ordering of inputs, e.g. see [FOH93]. Yet, these approaches are fast, but are often designed for classes of circuits with certain structural properties only and they do not guarantee an optimal result. In fact, experiments have shown that they can yield results up to 100 times the size of an optimal solution.

Second, dynamic reordering based on Rudell’s sifting algorithm [Rud93] has become a widespread and very successful method. Here, a good ordering is determined dynamically by the system in situations where the application gets low on memory. This approach computes an ordering with what essentially is a so-called hill-climbing strategy, known from AI, e.g. [Ric88]. Since this normally happens fully automated and does not involve any inputs or actions from the user of such a software system, this method has become very popular. Today, many applications based on BDDs benefit from this widespread method. But still run time is an important issue and there is demand for faster solutions.
Further, simulation-based methods known from AI like *Evolutionary Algorithms* (EA) and *Simulated Annealing* (SA) have been proposed to obtain better results than the heuristic approaches before [BLW95, DGB96, DG97, GD00]. While these methods can obtain better results than mere “rules of thumb”, the run time is usually much higher.

Finally, *exact* methods have been studied which are the only approaches that guarantee to determine an *optimal* variable ordering [FS90, ISY91, JKS93, DDG00, EGD03a, Ebe03, EGD04a, EGD05]. Except for the last two publications, these methods are all based on the B&B paradigm.

These classes of methods are given in increasing order of run time complexity. That is, exact BDD minimization is the hardest problem. In fact it has been shown that it is NP-complete to decide whether the number of nodes of a given BDD can be improved by variable reordering [BW96]. Yet there are applications in logic synthesis where a minimal number of nodes is needed since a reduction in the number of BDD nodes directly transfers to a smaller chip area: sub-optimal solutions can yield multiples of the minimum size and are not acceptable here. These applications follow multiplexor based design styles like *Pass Transistor Logic* (PTL). They allow to consider layout aspects during the synthesis step and by this guarantee high design quality. Recently, the interest in these approaches has been renewed, e.g. see [MBSV95, BNNSV97, FMM+98, MSMSL99, MBM01, DG02, YC02].

Pass Transistor Logic uses only two transistors to realize a multiplexor (a wired OR of two MOS transistors, see Figure 1.1). However, both input polarities are needed to drive the pass transistor multiplexor. The advantages of PTL circuits as an alternative to static CMOS design are higher energy efficiency (low power), less circuit area and higher circuit speed. In 1995, Kuroda and Sakurai successfully used hand-designed PTL circuits to design digital systems [KS95]. A drawback of this early works however was the lack of automated synthesis tools. Later, several approaches for automatic PTL synthesis based on BDDs have been proposed [YSRS96, BNNSV97, CLAB98, SB00]. They all
benefit from the close correspondence of BDDs and PTL circuits which can be used for a straightforward mapping (see Figure 1.2).

A disadvantage of PTL designs is the high delay of transistor chains which is quadratic in the number of transistors used. Therefore, modern synthesis tools provide automatic buffer insertion and minimization [ZA01]. A second disadvantage, the possibility of sneak paths, i.e. connections of VDD and ground, already has been overcome with the use of BDD-based PTL synthesis since circuits derived from BDDs do not contain sneak paths.

This book starts the study of BDD optimization by presenting classical and the latest approaches to minimization of the number of nodes in BDDs. After basic notations and definitions are given in Chapter 2, algorithms are presented that exactly minimize BDD size in Chapter 3. The approaches vary from classical B&B methods to recent, more efficient approaches, which utilize extended B&B techniques. The latest development in the field is the shift to a new algorithmic paradigm, the generic \( A^* \)-algorithm [HNR68, Pea84, Ric88]. While this is one of the central concepts in AI, the EDA community has been more or less unaware of this paradigm for a long time. Recently, this approach has been suggested for exact size-driven BDD minimization [EGD04a, EGD05].

Besides presentation of the approaches, important algorithmic concepts and the theory behind them are introduced, such as heuristic state
space search, ordered best-first search and the $A^*$-algorithm as well as the use of lower bounds during search.

All presented methods are evaluated by experimental results with benchmarks. In the case of recent approaches to exact BDD minimization, reductions in run time of more than 80% have been observed when compared to the best classical method. When applying the new method to arithmetic functions, the gain is even larger, achieving speed-ups of up to one order of magnitude.

Next, classical and recent approaches to dynamic reordering are given in Chapter 4, reflecting the latest developments. The latest promising method is based on Rudell’s sifting algorithm and makes use of lower bounds to restrict the search space.

After these contributions to classical size-driven BDD optimization, in Chapter 5 the book focuses on alternative criteria for the optimality of BDDs. As a growing number of methods require the optimization of BDDs with respect to different objective functions, these new criteria become more and more important in comparison to the “classical” notion of BDD optimality, i.e. small or minimal BDD size.

The new criteria are related to paths in BDDs and they are motivated by applications in SAT solving, functional simulation, logic synthesis and synthesis for testability.

First, the minimization of the number of paths in BDDs is considered. The book gives a theoretical study as well as a minimization algorithm. The study is completed by experimental results.

Next, recent approaches to optimize BDDs with respect to the expected path length in BDDs are given and compared to previous approaches. The latest method is based upon Rudell’s sifting algorithm. In contrast to the first approach of [LWHL01], the recent method is based on fully local operations, saving the high cost of touching large parts of the graph in every step. Experimental results with the new method are given, showing speed-ups of up to two orders of magnitude while preserving high quality of the results.

A unifying view is applied to compare the idea and the algorithmic hardness of the respective path minimization approaches. Then a new algorithm is derived, applying the unified view. The algorithm implements the minimization with respect to a new criterion for BDD optimality, the average path length in BDDs. This criterion, among others, has recently been suggested as a starting point for the synthesis for path delay fault testable circuits [DSF04]. The presentation of modern BDD optimization techniques is finished by giving experimental results for this last optimization approach.
State-of-the-art verification tools often use SAT solvers besides BDDs. Both SAT and BDD are well-established concepts. When looking at the experienced run times when using both paradigms for particular problem classes, the two concepts can behave orthogonally. The lastest trends in current and future research move towards the fusion of SAT and BDD.

In the book, the relation between SAT and BDD is studied in Chapter 6, considering formal equivalence checking as an example. This gives theoretical insight for a better understanding of the new trend and reflects the latest developments in the field.

Finally, concluding remarks are given in Chapter 7.
In this chapter, some definitions and basic notations of Boolean algebra and reduced ordered Binary Decision Diagrams (BDDs) are given, as far as they are used in the following chapters. First, Boolean functions are defined and an important method for their decomposition is explained. Then a formal definition of BDDs is given.

2.1 Notation

This section explains general notations used throughout this book. Often sets and power sets are considered. The notation for the power set of a given set $M$ is

$$2^M = \{ S | S \subseteq M \}.$$ 

$\mathbb{N}$ denotes the set of natural numbers not including zero, i.e.

$$\mathbb{N} = \{ 1, 2, \ldots \}.$$ 

Variables which are assumed to have values in $\mathbb{N}$ are most of the time denoted by the letters $i, j, k, m$ and $n$. Then, if the range of these variables is given by the context, sometimes a specification like “$n \in \mathbb{N}$” is omitted for simplicity.

When giving a result which expresses a both-way implication “if and only if” between a left and a right side of the statement, often the notation “iff” will be used as abbreviation, e.g.

$$a = \sqrt{b} \text{ iff } a^2 = b.$$ 

Functions usually are denoted using the identifiers $f, g,$ and $h$. A function $f$ is given as a mapping from a domain $X$ to domain $Y$. Domain
and co-domain are stated in advance, e.g. \( f: X \to Y \). The mapping then is defined with an expression of the form
\[
f(x) = \text{an expression using } x
\]
or
\[
(x_1, x_2, \ldots, x_k) \mapsto \text{an expression using } x_1, x_2, \ldots, x_k
\]
(e.g., in the case of a function \( f: \mathbb{N}^2 \to \mathbb{N} \), the function may be given in the form \( (x_1, x_2) \mapsto x_1 \cdot x_2 \)).

However, in the case of captions of figures etc. a shorter notation may be used to save space: the function above might be given in the short form
\[
f = x_1 \cdot x_2.
\]
This is done only if domain and co-domain are clear or given by the context (e.g. if \( f \) above is already known to be a Boolean function, it is clear from the short notation that \( f: \{0,1\}^2 \to \{0,1\} \)).

### 2.2 Boolean Functions

Let \( \mathbf{B} := \{0, 1\} \) and \( n \in \mathbb{N} \). Boolean variables, typically denoted by Latin letters, e.g. \( x, y, z \) are bound to values in \( \mathbf{B} \). Variables are referred to by subscripts which are from the set \( \{1,2,\ldots,n\} \), e.g.
\[
x_1, x_2, \ldots, x_n.
\]
To denote the set \( \{x_1, x_2, \ldots, x_n\} \) of “standard” variables we use the notation \( X_n \). Later, in Chapter 4, also the notation \( X^j_i = \{x_i, x_{i+1}, \ldots, x_{j-1}, x_j\} \) will be used to refer to several subsets of \( X_n \). The following is an introduction of notations defining Boolean functions.

**Definition 2.1** Let \( m, n \in \mathbb{N} \). A mapping
\[
f: \mathbf{B}^n \to \mathbf{B}^m
\]
is called a Boolean function. In the case of \( m = 1 \) we say \( f \) is a single-output function, otherwise \( f \) is called a multi-output function.

These terms are used because Boolean functions are used to describe the digital logic of a circuit. A circuit transforms inputs, i.e. a vector of incoming Boolean signals to a vector of outputs, thereby following a certain logic. This logic can be described by a Boolean function.

Other properties of a circuit (e.g. critical path delay or area requirement) need a more sophisticated representation (e.g. as BDD which is a special form of a graph). Let \( f: \mathbf{B}^n \to \mathbf{B}^m \) be a Boolean function. To
put emphasis on the arity $n$ of $f$, we may choose to write $f^{(n)}$ instead of $f$. This notation will be used for functions in general, e.g. later on we use it to denote variable orderings $\pi^{(n)}$ (see Section 2.4). Sometimes we may even write $f^{(n,m)}$ to reflect the whole signature of a (Boolean) function $f$.

A multi-output function $f: B^n \to B^m$ can be interpreted as a family of $m$ single-output functions $(f_i^{(n)})_{1 \leq i \leq m}$. The $f_i$'s are called component functions.

To achieve a standard, in this book the set of variables of a Boolean function $f^{(n)}$ will always be assumed to be $X_n$. If not stated otherwise, Boolean functions are assumed to be total (completely specified), i.e. there exists a defined function value for every vector of input variables. The Boolean functions constantly mapping every variable to 1 (to 0) are denoted one (zero), i.e.

- one : $B^n \to B^m$; $(x_1, x_2, \ldots, x_n) \mapsto 1$,
- zero : $B^n \to B^m$; $(x_1, x_2, \ldots, x_n) \mapsto 0$.

A Boolean variable $x_i$ itself can be interpreted as a first example of a Boolean function

$$x_i: B^n \to B; \ (a_1, a_2, \ldots, a_i, \ldots, a_n) \mapsto a_i.$$ 

This function is called the projection function for the $i$-th component.

**Definition 2.2** The complement of a Boolean variable $x_i$ is given by the mapping

$$\overline{x_i}: B^n \to B; \ (a_1, a_2, \ldots, a_i, \ldots, a_n) \mapsto \overline{a_i}$$

where $\overline{a_i} = 1$ iff $a_i = 0$.

An interesting class of Boolean functions are (partially) symmetric functions. Later, in Chapter 3, algorithms for BDD minimization will be presented which exploit (partial) symmetry to reduce run time.

**Definition 2.3** Let $f: B^n \to B^m$ be a multi-output function. Two variables $x_i$ and $x_j$ are called symmetric, iff

$$f(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, x_j, x_{i+1}, \ldots, x_{j-1}, x_i, x_{j+1}, \ldots, x_n).$$

Symmetry is an equivalence relation which partitions the set $X_n$ into disjoint classes $S_1, \ldots, S_k$ called the symmetry sets. A function $f$ is called partially symmetric, iff it has at least one symmetry set $S$ with $|S| > 1$. If a function $f$ has only one symmetry set $S = X_n$, then it is called totally symmetric.
2.3 Decomposition of Boolean Functions

**Definition 2.4** Let $f: \mathbb{B}^n \to \mathbb{B}^m$ be a Boolean function. The cofactor of $f$ for $x_i = c$ ($c \in \mathbb{B}$) is the function $f_{x_i=\overline{c}}: \mathbb{B}^n \to \mathbb{B}^m$. For all variables in $X_n$ it is defined as

$$f_{x_i=\overline{c}}(x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) = f(x_1, x_2, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_n).$$

A cofactor of $f$ is the function derived from $f$ by fixing a variable of $f$ to a value in $\mathbb{B}$. Formally, a cofactor of $f^{(n)}$ has the same arity $n$. In contrast to all variables different from $x_i$, the variable $x_i$ is not free in the cofactor $f_{x_i=\overline{c}}$. Hence the cofactor does not depend on this variable (see Definition 2.5).

Despite the generality of the last definition covering multi-output functions, sometimes only the cofactors of single-output functions $f: \mathbb{B}^n \to \mathbb{B}$ are of interest. When a multi-output function $f^{(n,m)} = (f_i^{(n)})_{1 \leq i \leq m}$ is given, we often consider the cofactors of the component functions $f_i$ only. These cofactors then are single-output functions of arity $n$. A cofactor of a multi-output function $f$ can be interpreted as a family of cofactors of the component functions of $f$.

A cofactor $f_{x_i=\overline{c}}$ is sometimes called a direct cofactor to emphasize that there is only one variable bound to a value in $\mathbb{B}$. This opposes to a cofactor in more than one variable. E.g., for $k \leq n$, $x_{i_1}, \ldots, x_{i_k} \in X_n$ and $c_1, \ldots, c_k \in \mathbb{B}$, the function $f_{x_{i_1}=c_1, x_{i_2}=c_2, \ldots, x_{i_k}=c_k}$ is a cofactor in multiple variables. This cofactor is equivalent to several direct cofactors, e.g. to

$$(f_{x_{i_1}=c_1, x_{i_2}=c_2, \ldots, x_{i_k}=c_k})_{x_{i_j}=c_{i_j}}.$$  

In general it is equivalent to

$$(f_{x_{i_1}=c_1, x_{i_2}=c_2, \ldots, x_{i_j}=c_{i_j}, x_{i_{j+1}}=c_{i_{j+1}}, \ldots, x_{i_k}=c_k})_{x_{i_j}=c_j}$$

for any $1 \leq j \leq k$. A cofactor in multiple variables is uniquely determined regardless of the order in which we fix these variables. Hence, these cofactors can also be thought of being obtained by simultaneously fixing all the involved variables. To obtain increased readability, sometimes a "|" sign is used to separate the function symbol from the list of variable bindings, e.g. we write $f_j|_{x_{i_1}=c_1}$ for a cofactor in a component function $f_j$.

**Definition 2.5** Let $f: \mathbb{B}^n \to \mathbb{B}^m$ be a Boolean function and let $x_i \in X_n$. Then function $f$ is said to essentially depend on $x_i$ iff

$$f_{x_i=0} \neq f_{x_i=1}.$$
The set of variables which $f$ essentially depends on, is called the support of $f$ and is denoted $\text{support}_{n,m}(f)$. Usually $n, m$ are clear from the context and hence we simply write $\text{support}(f)$.

Formally, $\text{support}_{n,m}$ is a mapping $\{ f \mid f: B^n \rightarrow B^m \} \rightarrow 2^{X_n}$. If not stated otherwise, a given Boolean function $f: B^n \rightarrow B^m$ is always defined over the variable set $X_n$ and always support($f$) = $X_n$ is assumed. The next definition characterizes cofactors of a Boolean function which are derived by fixing at least one variable of the support of the function.

**Definition 2.6** Let $f: B^n \rightarrow B^m$ be a Boolean (single or multi-output) function over $X_n$. A cofactor $f_{x_i_1=a_1,x_i_2=a_2,...,x_i_k=a_k}$ with $x_i_1, x_i_2, \ldots, x_i_k \in X_n, (a_1, a_2, \ldots, a_k) \in B^k$ is called true iff

$$\{x_i_1, x_i_2, \ldots, x_i_k\} \cap \text{support}(f) \neq \emptyset.$$ 

Note that true cofactors of a function cannot be equivalent to the function itself.

Often it is more useful to consider a set of cofactors of a Boolean function rather than considering just one particular cofactor. This is reflected by the next definition.

**Definition 2.7** Let $f: B^n \rightarrow B^m, f = (f_i^{(n)})_{1 \leq i \leq m}$ be a Boolean multi-output function essentially depending on all its input variables and let $I \subseteq X_n$. The set of non-constant cofactors of $f$ with respect to the variables in $I$ is denoted $\text{cof}_{n,m}(f, I)$. Formally, a function $\text{cof}_{n,m}$ is given as

$$\text{cof}_{n,m}: \{ f \mid f: B^n \rightarrow B^m \} \times 2^{X_n} \rightarrow 2\{f | f: B^n \rightarrow B^m\};$$

$$(f, \{x_i_1, \ldots, x_i_k\}) \mapsto \{f_i|_{x_i_1=a_1,...,x_i_k-a_k} \text{ non-constant} \mid 1 \leq i \leq m, (a_1, \ldots, a_k) \in B^k\}$$

for $1 \leq k \leq n$.

The set $\text{cof}_{n,m}(f, I)$ is the set of all distinct (non-constant and single-output) cofactors of $f$ ($f$ is interpreted as a family of $m$ $n$-ary single-output functions) with respect to all variables in $I$. Note that this is not a multiset, hence functionally equivalent cofactors are eliminated and thus do not contribute to $|\text{cof}_{n,m}(f, I)|$.

If $n$ and $m$ are clear from the context, we simply write $\text{cof}(f, I)$.
Next, we restrict this set to contain only cofactors that are true cofactors of one of the single-output functions $f_i$.

**Definition 2.8** Let $f : B^n \rightarrow B^m$; $f = (f_i^{(n)})_{1 \leq i \leq m}$ be a Boolean multi-output function essentially depending on all its input variables and let $I \subseteq X_n$. The set of non-constant true cofactors of $f$ with respect to the variables in $I$ is denoted $\text{tcof}_{n,m}(f, I)$. Formally, a function $\text{tcof}_{n,m}$ is given as

$$\text{tcof}_{n,m} : \{f \mid f : B^n \rightarrow B^m\} \times 2^{X_n} \rightarrow 2\{f | f : B^n \rightarrow B^m\};$$

$$(f, \{x_{i_1}, \ldots, x_{i_k}\}) \mapsto \{ f_{|x_{i_1}-a_1, \ldots, x_{i_k}-a_k} \text{ non-constant and true cofactor of } f_i | 1 \leq i \leq m, (a_1, \ldots, a_k) \in B^k \}$$

for $1 \leq k \leq n$.

Let $f_i \neq f_j$ be two distinct single-output functions in the family $(f_i^{(n)})_{1 \leq i \leq m}$. Note that a true cofactor in $f_i$ can be functionally equivalent to a cofactor of $f_j$ that is not true. In other words, the cofactors in the set $\text{tcof}_{n,m}(f, I)$ are not required to be true in every single-output function, it is only required that at least one such single-output function exists.

Again if $n$ and $m$ are given from the context, we simply write $\text{tcof}(f, I)$.

The following well-known theorem [Sha38] allows to decompose Boolean functions into “simpler” sub-functions.

**Theorem 2.9** Let $f : B^n \rightarrow B^m$ be a Boolean function (over $X_n$). For all $x_i \in X_n$ we have:

$$f = x_i \cdot f_{x_i=1} + \overline{x_i} \cdot f_{x_i=0}. \quad (2.1)$$

It is straightforward to see that the sub-functions obtained by subsequent application of Theorem 2.9, called the Shannon decomposition, to a function $f$ are uniquely determined. Note that this ensures the well-definedness of cofactor set definitions.

### 2.4 Reduced Ordered Binary Decision Diagrams

Many applications in VLSI CAD make use of reduced ordered Binary Decision Diagrams (BDDs) as introduced by [Bry86]:
A BDD is a graph-based data structure. Redundant nodes in the graph, i.e. nodes not needed to represent $f$, can be eliminated. BDDs allow a unique (i.e. canonical) representation of Boolean functions. At the same time they allow for a good trade-off between efficiency of manipulation and compactness. Compared to other techniques to represent Boolean functions, e.g. truth tables or Karnaugh maps, BDDs often require much less memory and faster algorithms for their manipulation to exist.

In the following, a formal definition of BDDs is given. We start with purely syntactical definitions by means of Directed Acyclic Graphs (DAGs). First, single-rooted Ordered Binary Decision Diagrams (OBDDs) are defined. This definition is extended to multi-rooted graphs, yielding Shared OBDDs (SBDDs). Next, the semantics of SBDDs is defined, clarifying how Boolean functions are represented by SBDDs.

After that, reduction operations on SBDDs are introduced which preserve the semantics of an SBDD. This leads to the final definition of reduced SBDDs that will be called BDDs for short in this book.

Finally, some definitions and notations are given which allow to discuss various graph-oriented properties of BDDs, among them paths in BDDs and their (expected) length.

Examples are given to illustrate the formal definitions where appropriate.

### 2.4.1 Syntactical Definition of BDDs

**Definition 2.10** An Ordered Binary Decision Diagram (OBDD) is a pair $(\pi, G)$ where $\pi$ denotes the variable ordering of the OBDD and $G$ is a finite DAG $G = (V, E)$ ($V$ denotes the set of vertices and $E$ denotes the set of edges of the DAG) with exactly one root node (denoted root) and the following properties:

- A node in $V$ is either a non-terminal node or one of the two terminal nodes in $\{1, 0\}$.
- Each non-terminal node $v$ is labeled with a variable in $X_n$, denoted $\text{var}(v)$, and has exactly two child nodes in $V$ which are denoted $\text{then}(v)$ and $\text{else}(v)$.
- On each path from the root node to a terminal node the variables are encountered at most once and in the same order.

More precisely, the variable ordering $\pi$ of an OBDD is a bijection

$$\pi: \{1, 2, \ldots, n\} \rightarrow X_n$$
where $\pi(i)$ denotes the $i$-th variable in the ordering. The above condition “in the same order” states that for any non-terminal node $v$ we have

$$
\pi^{-1}(\text{var}(v)) < \pi^{-1}(\text{var(then}(v)))
$$

iff then$(v)$ is also a non-terminal node and

$$
\pi^{-1}(\text{var}(v)) < \pi^{-1}(\text{var(else}(v)))
$$

iff else$(v)$ is also a non-terminal node.

Even though this might look a bit “over-formal”, the notation is required in the following to prove the correctness of the algorithms.

**Example 2.11** In Figure 2.1 two different types of binary decision diagrams are depicted. Solid lines are used for the edges from $v$ to then$(v)$ whereas dashed lines indicate an edge between $v$ and else$(v)$. In both diagrams the variables in $\{x_1, x_2, x_3\}$ are encountered at most once on every path. Whereas the right diagram is not ordered since the variables are differently ordered along distinct paths, the left one respects the ordering $\pi(1) = x_1, \pi(2) = x_2, \pi(3) = x_3$. Both diagrams represent the function $f: \mathbb{B}^3 \rightarrow \mathbb{B}; (x_1, x_2, x_3) \mapsto x_1 \cdot x_2 + \bar{x}_1 \cdot x_3$, as will be explained in a later section.

Where appropriate, we will speak of an OBDD over $X_n$ to put emphasis on the set of variables used as node labels. However, if not stated otherwise, an OBDD is always assumed to be an OBDD over $X_n$.

Note that OBDDs are connected graphs, as all nodes must be connected via at least one path to the (only) root node: to see this, assume the graph consists of more than one connected component. But then,