Transformation Techniques for Decision Diagrams in Computer-Aided Design

•

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

•

vorgelegt beim Fachbereich IV
der Universität Trier

von

THORSTEN THEOBALD

August 1997

1. Berichterstatter: Prof. Dr. Ch. Meinel (Universität Trier)
2. Berichterstatter: Prof. Dr. B. Becker (Albert-Ludwigs-Universität Freiburg)

Datum der Disputation: 19. Dezember 1997

Contents

Introduction **1**

1 Basic Concepts **7**

1.1 Binary Decision Diagrams 7

1.1.1 Construction and Manipulation 9

1.1.2 Implementation Techniques 9

1.1.3 The Variable Order 10

1.1.4 Local Variable Swaps 12

1.1.5 The Sifting Algorithm 13

1.2 Variants of OBDDs 15

1.2.1 Ordered Functional Decision Diagrams 15

1.2.2 Zero-Suppressed Binary Decision Diagrams 15

1.3 Transformed Binary Decision Diagrams 16

1.4 Finite State Machines 18

1.5 Reachability Analysis 19

1.6 Relevant Software Systems 20

1.6.1 The CUDD Package 20

1.6.2 The SIS System 21

1.6.3 The VIS System 21

2 The Influence of the State Encoding **23**

2.1 Basic Framework 24

2.2 The Lower Bound 25

2.3 The Behavior of Important Encodings 27

2.3.1 The Standard Minimum-Length Encoding 27

2.3.2 The Gray Encoding 30

2.4 A Worst-Case Encoding 31

2.5 Related Topologies 37

2.6 Conclusion and Open Questions 37

3 Local Encoding Transformations **39**

3.1 Motivation: The Potential of Re-encoding 40

3.2 Local Re-encodings 42

3.3 The XOR-Transformation 43

3.3.1 Enumeration Results 44

3.3.2 Bounded Size Alteration 46

3.3.3 Stronger Bounds 46

3.3.4 General Two-Bit Re-encodings 48

3.3.5 Implementation Aspects 49

3.4 Who Profits From XOR ? 50

3.5 Experimental Results 52

3.6 Conclusion 53

3.7 Two Proofs 53

4 Linear Sifting **57**

4.1 Preliminaries 58

4.2 Linear Sifting 59

4.3 Experimental Results 63

4.4 Conclusion 64

5 Function Decomposition and Technology Mapping 67
5.1 Synthesis by Spectral Analysis 68
5.2 Synthesis by Linear Sifting 70
5.3 Experimental Results 71
5.3.1 Main Setup 71
5.3.2 Separate Mapping 72
5.3.3 Symbolic Synthesis 72
5.3.4 Summary of the Experiments 76
5.4 Linear Sifting and Adder Functions 76
5.4.1 Adder Behavior 78
5.4.2 The ISOP Algorithm 82
5.4.3 Open Questions 88

6 Related Approaches and Final Remarks 91
6.1 Related Approaches 91
6.2 Final Remarks 92

Bibliography 95

INTRODUCTION

In 1943, Thomas Watson, founder of IBM, remarked that he thought that there was a "world market for maybe five computers."

In 1949, the American magazine "Popular mechanics" reported optimistically that computers in the future might weigh no more than 1.5 tons.

In 1977, Ken Olsen, founder of Digital Equipment, scoffed that "there is no reason anyone wants a computer at home."

In 1997, there were tens of millions of computers, either at a place of work or at home. An advanced workstation may have a Gigabyte of main memory, its processor consists of several million transistors and can do a billion operations per second.

These historical facts and anecdotes may illustrate the rapid developments in the area of *VLSI* (*Very Large Scale Integration*) digital circuits and computer technology. Moreover, it may not be surprising that the design of circuits and systems with several million parts is a very difficult task and has raised problems which are far beyond the scope of manual design. Over the last three decades, *Computer-Aided Design* (*CAD*) tools have therefore been adopted for various tasks of the design process. Nowadays, these tools have not only become widely used, but they are indispensable.

However, applications in information processing, telecommunication or in industrial control systems permanently require the construction of even more powerful high-speed circuits. On the one hand, this imposes bigger and bigger challenges upon the CAD systems. On the other hand, all these systems underlie the inherent complexity in the manipulation of switching functions which has been extensively studied in theoretical computer science [Weg87, Mei89]. One of the main problems is to

get the immense number of combinations of mathematical objects, the so-called *combinatorial explosion*, under control.

A central problem in the design of CAD systems for VLSI circuits is the *representation* of the functional behavior of a circuit. For an illustration we will shortly consider the problem of *combinational circuit verification.* Hereby it is to check whether a combinational circuit C satisfies a given specification S. For the solution of this problem computer-internal representations for C and S have to be determined which can then be used to test the relevant properties. Of course, this approach only leads to a practical procedure, if both representations can be computed efficiently and practical algorithms are available to decide equivalence, satisfiability and similar properties by means of the representations. The mentioned representations are realized internally via *data structures.*

The underlying mathematical model of all these problems is the one of Boolean algebra. *Boolean functions* $f : \{0,1\}^n \rightarrow \{0,1\}$ are of particular importance in the design and analysis of digital circuits. For this reason they are also called *switching functions.* By introducing a suitable 0-1-encoding all finite problems can – at least in principle – be solved by means of switching functions. The great importance of switching functions is based on the possibility to obtain simplified, optimized and with optional properties provided circuits during the design process. Before applying optimization techniques, the switching functions must be represented efficiently.

Well-known classical representations of switching functions include *truth tables*, *disjunctive normal forms*, *Boolean formulas* or *multi-level representations* by means of net-lists. They are all based on the idea to describe a given switching function by means of a computation rule. With the permanently growing performance requirements the drawbacks of these representations have become much more serious: Descriptions in form of a truth table are never compact. For the more compact representations there are insurmountable problems in the algorithmic manipulation: Already the test if two disjunctive normal forms, two Boolean formulas or two net-lists represent the same function is co-NP-complete [GJ78], and hence we cannot expect efficient algorithms.

Within the last decade *ordered binary decision diagrams*, abbreviated *OBDDs*, have attracted much attention. Since their invention by Bryant in 1986 [Bry86] they have proven to be the most suitable data structure for Boolean functions in a large number of applications. OBDDs represent Boolean functions by means of a *decision process* in a rooted directed acyclic graph: Each internal node of the graph is labeled by one of the input variables and has two outgoing edges, labeled 1 and 0. Consequently, each assignment to the input variables defines a unique path through the graph culminating in an overall decision that can be either 1 or 0. An additional ordering restriction on the variable occurrences on every path and two reduction rules

serve to establish a canonical (i.e. unique) representation which can be manipulated efficiently.

However, as the running times of all algorithms on OBDDs are correlated to the size of the input graphs and of the resulting graphs, it is an important task to optimize the size of these graphs. This topic has become even more serious by the observation that for a substantial number of OBDD-based applications the choice of the optimization technique is the deciding factor whether a computation succeeds or not.

The main optimization parameter of OBDDs is the underlying variable order. Many research efforts have tried to characterize the complexity of the relevant variable ordering problems [THY93, MS94, BW96]. One of the main results is that the problem of finding the optimal variable order for a given OBDD is NP-hard [BW96]. Hence, researchers have tried to come up with efficient optimization algorithms for obtaining large size reductions without aiming at the global minimum. The currently most important algorithm for improving the variable order of an OBDD is called *Sifting* [Rud93]. Unfortunately, there are many important applications, in particular in the analysis of finite automata/finite state machines [CBM89, CM95], where all these optimization techniques reach their limits.

Contributions

In this work we propose and discuss new optimization approaches for decision diagrams. Our central topic is the use of

Transformation Techniques

to reduce the size of OBDDs. Within this framework we focus on two concepts that are related to each other.

Encoding transformations: We systematically analyze and exploit the fact that in the context of finite state machines the state encoding can be used as an additional optimization parameter.

Domain transformations: The concept of *domain transformations*, well-known in many areas of mathematics and physics (e.g. Fourier or Laplace transformation), has been proposed in [BMS95] to enlarge the optimization space of variable ordering. We characterize suitable classes of transformations and show how these constructions can be well integrated into existing algorithms for finding good variable orders.

One of the central achievements in the present work is the characterization of "suitable" transformations in the two optimization frameworks. It turns out that the class of *linear transformations* is of particular importance. Linear transformations are a classic element in many areas of mathematics and computer science. However, recent research results also discovered new applications of linear transformations in other fields of actual interest, for example in algebraic complexity theory [Sha93b] or cryptography (see [Sha93a, The95]). In the context of decision diagrams, the importance of linear transformations does not only origin from the well-known algebraic background, but as will be shown they are the most attractive candidate concerning efficient implementations.

The present work is structured as follows:

In Chapter 1 we review the basic concepts and state-of-the-art techniques of binary decision diagrams in VLSI design.

In Chapter 2 we discuss the additional optimization potential of state encoding when finite state machines are represented by OBDDs. For this purpose, we present a comprehensive case-study of counter-type finite state machines. We analyze the size of the transition relation when these machines are represented by OBDDs. As we want to investigate the influence of the state encoding we use a fixed appropriate variable order. The main result of the chapter is that even for the restricted model of counters the choice of the state encoding can make the difference between linear and exponential representations in the given framework. Furthermore, we present linear lower bounds for the OBDD-size of counters. By deriving the exact sizes for important encodings we show that these encodings nearly meet the lower bounds.

In Chapter 3 we present optimization techniques for exploiting the additional optimization parameter of state encoding. We analyze local encoding transformations which only involve a limited number of state bits. By proving several analogies to the variable reordering problem, we show that local encoding transformations are well-suited to achieve further reduction in the size of OBDDs. The central point of this chapter results from a classification of all re-encodings that involve only two bits. We show that

- this class of re-encodings contains (up to some symmetry) *exactly* one transformation which is not already captured by reordering techniques: a *linear (or exclusive-or) transformation,*
- linear transformations can be implemented efficiently.

For this reason we propose the use of linear transformations to find good re-encodings. Furthermore we show that OFDDs, a variant of OBDDs, exhibit a different behavior in the presence of exclusive-or transformations than OBDDs. Some

experimental results illustrate that the proposed method in fact yields a reduction of the OBDD-sizes.

Chapter 4 is devoted to the optimization of decision diagrams by means of domain transformations. In this framework, we show how to convert the general concept of linear transformations into a fully automatic optimization technique. We propose a new algorithm, called Linear Sifting, for this optimization that combines the efficiency of the Sifting algorithm and the power of linear transformations. The key idea of this algorithm is the following: An elementary linear transformation $x_i \mapsto x_i \oplus x_j$ (resp. its complement $x_i \equiv x_i \oplus x_j$) for two neighboring variables can be implemented in the same way as the swap of the two variables. Hence, it is possible to integrate elementary linear transformations into existing reordering algorithms like Sifting. We show that the new algorithm is applicable to large examples, and that in many cases it leads to substantially more compact diagrams when compared to simple variable reordering. In a comprehensive series of experiments the algorithm decreases the total number of nodes by 13% with a geometric mean of individual improvements of 22%. For some circuits, the new optimization algorithm reduces the size of the OBDDs up to 98%.

Chapter 5 deals with an application of the Linear Sifting algorithm. In order to simplify a synthesis task for particularly hard functions it is sometimes inevitable to decompose the function in a preprocessing step. We propose to use the Linear Sifting algorithm for automatically decomposing a target function within the synthesis process. This algorithm presents an alternative to existing algorithms. Using our method we are able to synthesize functions with standard tools which fail otherwise.

The second part of Chapter 5 presents theoretical results concerning limitations of the Linear Sifting algorithm and important effects in connection with symbolic synthesis algorithms. Our main result is the clarification of the following paradoxical observation: By using the Linear Sifting algorithm, the OBDD-size of adder functions can be reduced by 40%. Within a symbolic synthesis process, however, the transformed adder requires exponentially more space resources than the original one. In order to clarify this effect, we carefully analyze adder functions and their behavior in the presence of Linear Sifting and a symbolic synthesis environment.

Finally, in Section 6 we give a short description of related approaches concerning transformation techniques.

Parts of this work have already been published in a similar form: Chapter 2 in [MT97b], Chapter 3 in [MT96] Chapter 4 in [MST97], and some parts in the survey article [MT97a].

Acknowledgments

First of all, I wish to thank my advisor Prof. Christoph Meinel for all kind of support and cooperation during the last years.

Many thanks are also devoted to Prof. Fabio Somenzi, University of Colorado at Boulder, for a very fruitful collaboration, many hints and valuable advice.

Furthermore, I like to thank everybody who helped and supported me during my time at the University of Trier: in particular, all the members of Prof. Meinel's working group and the members of the DFG-graduate program "Mathematical Optimization" (Graduiertenkolleg "Mathematische Optimierung" der Deutschen Forschungsgemeinschaft).

Chapter 1

Basic Concepts

This chapter will cover the basic concepts relevant for our work. In particular, we survey some state-of-the-art techniques in the area of Boolean manipulation which form the starting point for our analyses and new optimization concepts.

Definition 1.1. *The following notations will be used throughout the text:*

- $\mathbb{B}$ *denotes the set* $\{0,1\}$.
- $\mathbb{B}_n$ *denotes the set of all switching functions* $\{0,1\}^n \to \{0,1\}$.

1.1 Binary Decision Diagrams

Ordered binary decision diagrams (**OBDDs**) [Bry86, Bry92] are rooted directed acyclic graphs representing switching functions. Each OBDD has two sink nodes which are labeled 1 and 0. Each internal (= non-sink) node is labeled by an input variable x_i and has two outgoing edges, labeled 1 and 0 (in the diagrams the 1-edge is indicated by a solid line and the 0-edge by a dotted line). A linear variable order π is placed on the input variables. The variable occurrences on each OBDD-path have to be consistent with this order. An OBDD computes a switching function $f \in \mathbb{B}_n$ in a natural manner: each assignment to the input variables x_i defines a unique path through the graph from the root to a sink. The label of this sink defines the value of the function on that input. Figure 1.1 shows two OBDDs for the function $f = bc + a\overline{b}\overline{c}$ w.r.t. the variable order $a < b < c$.

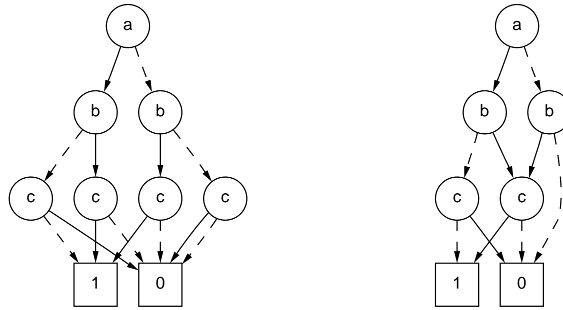


Figure 1.1: Two OBDDs for $f = bc + a\overline{b}\overline{c}$

An OBDD is called **reduced** if it does not contain any vertex v such that the 1-edge and the 0-edge of v lead to the same node, and it does not contain any distinct vertices v and v' such that the subgraphs rooted in v and v' are isomorphic. Equivalently, an OBDD is reduced if none of the following local reduction rules can be applied:

Deletion rule: If the 1- and the 0-edge of a node v lead to the same node u, then eliminate v and redirect all its incoming edges to u.

Merging rule: If the nodes u and v are labeled by the same variable, their 1-edges lead to the same node, and their 0-edges lead to the same node, then eliminate one of the two nodes u, v, and redirect all incoming edges to the other node.

Both reduction rules are illustrated in Figure 1.2.

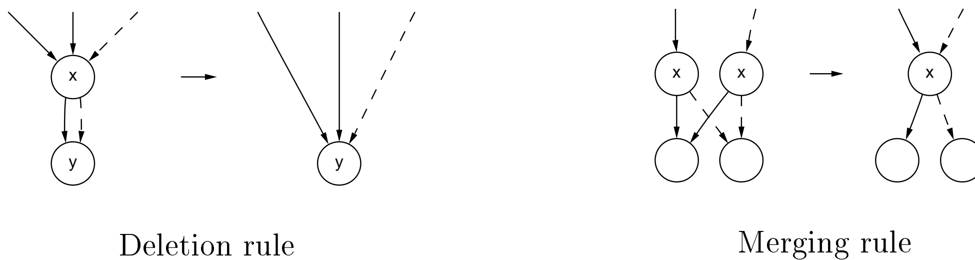


Figure 1.2: Reduction rules

Hence, the right OBDD in Figure 1.1 is reduced. For the algorithmic qualities of OBDDs the following canonicity property is of fundamental importance [Bry86].

Fact 1.2. *With respect to a fixed order, the reduced OBDD for each Boolean function $f \in \mathbb{B}_n$ is uniquely determined.*

The **size** of an OBDD is the number of its nodes. Several functions can be represented by a multi-rooted graph called **shared OBDD**, see Figure 1.3. In the following, all functions are represented by a shared OBDD.

1.1.1 Construction and Manipulation

OBDDs are not only a canonical representation, but they can also be manipulated quite efficiently. We will mention some important features. For more details, we refer to [Bry86, BRB90].

Let $*$ denote an arbitrary Boolean operation $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$, for example the conjunction or the disjunction. In order to compute $f * g$ from given OBDDs for the functions f, g, the so-called **Shannon expansion** w.r.t. the top variable of the order π can be used:

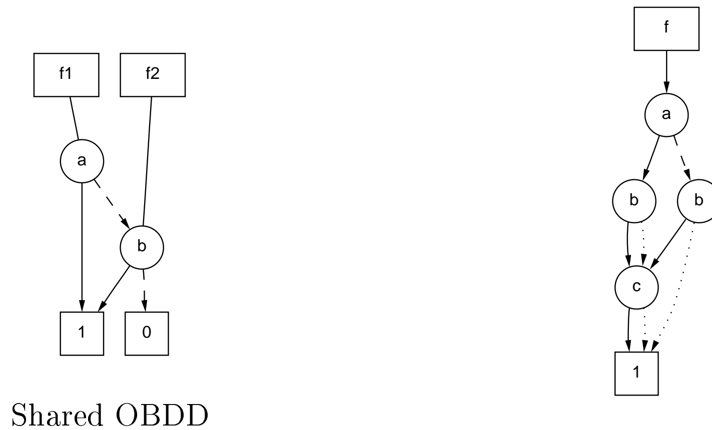
$$f * g \;=\; x\,(f|_{x=1} * g|_{x=1}) \;+\; \overline{x}\,(f|_{x=0} * g|_{x=0}),$$

where $f|_{x=1}$ denotes the subfunction which results from f by replacing the variable x by the value 1. By repeated application of this decomposition an OBDD for $f * g$ is computed. In order to perform this task efficiently, multiple calls with the same argument pairs are avoided – instead, the previously computed result will be looked up in a table. By using this technique, the originally exponential number of decompositions is now bounded by the product of the sizes of both OBDDs, and the basic algorithmic property of OBDDs follows:

Fact 1.3. *Let f_1 and f_2 be represented by two OBDDs P_1 and P_2 w.r.t. the same order. For each binary operation $*$ the reduced OBDD P for $f = f_1 * f_2$ can be computed in time $O(size(P_1) \cdot size(P_2))$.*

1.1.2 Implementation Techniques

In most current implementations, several techniques are used to increase the compactness of the representation and the efficiency of the algorithms [BRB90]. One device which will be relevant for our work is the use of **complemented edges**. Each edge obtains an additional attribute bit, the complement bit. If the bit is not set, the subfunction which the edge points to is interpreted in the original way. If instead the bit is set, then the connected subgraph is to be interpreted as the



Shared OBDD

$f = bc + a\overline{b}\overline{c}$ using complemented edges

Figure 1.3: Shared OBDD and complemented edges

complement of the ordinary subfunction. Hence two functions f and $\overline{f}$ can be represented by essentially the same graph, and it is possible to complement a function in constant time. Moreover, as the 0-sink can be represented by the complement of the 1-sink there is only one constant node necessary.

The problem one has to deal with in the presence of complemented edges is the loss of canonicity. In order to re-establish this property [Kar88] and [MB88] have stated rules for the allowed positions of complemented edges. The basis for these rules is the observation that some constellations of complement bit positions are functionally equivalent: In particular, if we consider for a node v the triple of an incoming and the two outgoing edges, there are eight possibilities for the triple of the three attribute bits. Out of the eight combinations four pairs are functionally equivalent. One possibility to re-establish canonicity is the rule that the 1-edge of every node cannot carry the complement attribute. Note that in the presence of complemented edges the pointer to the root node of the OBDD can also be complemented. Figure 1.3 shows an OBDD using complemented edges for the function $f = bc + a\overline{b}\overline{c}$ from Figure 1.1. In our diagrams the edges whose complement bit is set are represented by *dotted* lines.

1.1.3 The Variable Order

The size of an OBDD and therefore the complexity of its manipulation depends on the underlying variable order - this dependency can be quite strong. An extreme example is shown in Figure 1.4. The function

$$x_1 x_2 + x_3 x_4 + \ldots + x_{2n-1} x_{2n}$$

is represented by an OBDD of linear size w.r.t. to the variable order $x_1, x_2, \ldots, x_{2n-1}, x_{2n}$. In contrast, w.r.t. the variable order $x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n}$ the reduced OBDD grows exponentially in n.

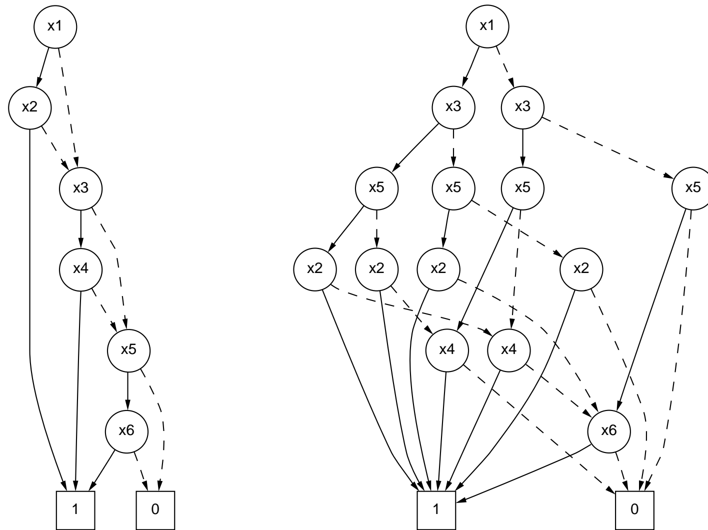


Figure 1.4: Influence of the variable order

The same effect occurs in the case of adder functions – here, the OBDD-size also varies from linear to exponential in the number of the input bits depending on the variable order. Other important functions like the binary multiplication of two n-bit integers have OBDDs of exponential size for all orderings [Bry91].

Due to the strong dependency of the OBDD-size on the chosen variable order it is a central problem in the manipulation of OBDDs to construct good orders. However, it is known that the problem to construct an optimal order for a given OBDD is NP-hard [BW96]. The currently best known exact algorithm is based on dynamic programming and has running time $O(n^2 \cdot 3^n)$ [FS90] resp. $O(n \cdot 3^n)$ for a slightly improved variant [BW96]. For serious applications this method is not useful. The practically relevant optimization strategies can be classified into two categories: *heuristics* and *dynamic reordering*.

Heuristics. Here, the aim is to deduce a priori information from the application which are useful for the determination of a good variable order. For the case of a **symbolic simulation**, i.e. the construction of an OBDD from a given net-list of a circuit, numerous methods have been developed in order to obtain a good order from the topological structure of the circuit [MWBS88]. The main drawback is that the effect of these heuristic methods is quite problem specific and that so far, no heuristic is known which is suitable for all cases.

Dynamic reordering. Another technique to minimize the OBDD-size is to improve the variable order during the manipulation dynamically. The currently best reordering strategy is called Sifting. As this algorithm is quite important for our work we will give a detailed description in Section 1.1.5.

1.1.4 Local Variable Swaps

All currently known dynamic reordering algorithms are exploiting the fact that two adjacent variables in the order can be efficiently swapped [FMK91]. It is indeed possible to perform such a swap by accessing only the nodes labeled by the two variables being exchanged.

Suppose that variables x_i and x_j are to be swapped, and that x_i immediately precedes x_j before the swap. Then the effect of the exchange on each node labeled x_i can be seen by applying Shannon's expansion w.r.t. both x_i and x_j. Assuming f is the function of a node labeled x_i, we have:

$$f = x_i x_j f_{11} + x_i \overline{x_j} f_{10} + \overline{x_i} x_j f_{01} + \overline{x_i}\ \overline{x_j} f_{00}.$$

Rearranging the terms such that x_i occurs before x_j in each term yields:

$$f = x_j x_i f_{11} + x_j \overline{x_i} f_{01} + \overline{x_j} x_i f_{10} + \overline{x_i}\ \overline{x_j} f_{00}.$$

In words, the effect of a swap is to interchange f_{10} and f_{01} in the OBDD. This effect is visualized in Figure 1.5. In a typical implementation the procedure of swapping the two variables in the order can be performed in a time that is only proportional to the number of nodes with label x_i or x_j.

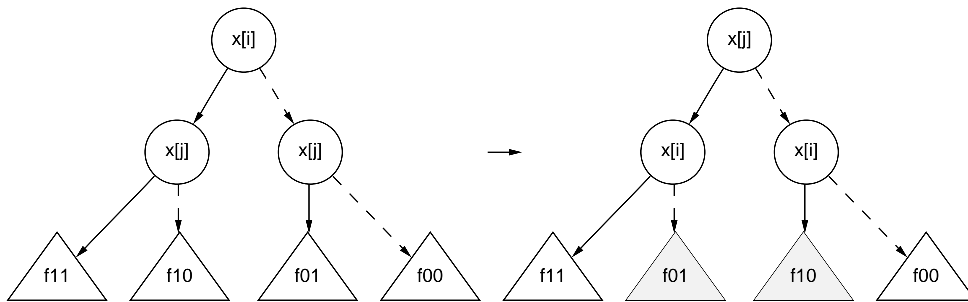


Figure 1.5: Swapping two variables x_i, x_j in the order

1.1.5 The Sifting Algorithm

The currently best reordering strategy has been invented by Richard Rudell [Rud93] and is called **Sifting**. It is a local search algorithm that iteratively improves the variable order by a series of swaps of adjacent variables.

Each variable is considered in turn and is moved up and down in the order so as to take all positions successively. The variable is then returned to the position where the minimum size of the OBDD was recorded. The process then continues with another variable. If during the algorithm the size increase of the OBDD exceeds a given factor $MaxGrowth$, the current searching subroutine is interrupted immediately.

A pseudo code for a basic variant of the Sifting algorithm is shown in Figure 1.6. BDD-size(P, π) denotes the size of the (shared) OBDD P w.r.t. the variable order π. For $1 \leq i \leq n$, $\pi[i]$ contains the index of the variable at position i in the order, and $\pi^{-1}[j]$ contains the position of variable x_j within the order. The algorithm uses a subroutine $swap_\pi(P, x_i, x_j)$ which exchanges x_i and x_j in the order and updates the OBDD P as well as the arrays $\pi[\ ]$ and $\pi^{-1}[\ ]$.

Efficient implementations of this basic variant also take care of the following extensions:

1. Originally, Rudell proposed to choose $MaxGrowth = 2$. In practice, however, a smaller value like $MaxGrowth = 1.2$ typically leads to a big improvement in time without loosing too much of the optimization quality.

2. As a heuristic preprocessing, the variables are sorted w.r.t. their number of occurrences in the OBDD. The idea is to investigate the variables with the highest optimization potential at first.

3. In the shown basic variant, each variable is moved at first to the top of the OBDD (descending in the order), and then to the bottom. As a typical heuristic, a variable is moved at first in the direction of the nearer end.

4. An important device to guarantee high speed in Sifting is the **interaction matrix**. This matrix tells the swapping procedure whether two variables appear together in the support of some function; in other words, whether there exists a root of the OBDD from which one can reach nodes labeled by both variables. If two variables do not interact, the swap can be performed in constant time. The interaction matrix is initialized before reordering starts, and it remains invariant throughout the execution of the Sifting algorithm.

5. During the searching subroutines lower bounds for the OBDD-size are computed. These lower bounds may tell us that during the movement of the vari-

```
Sifting(P_0, π_0) {
/* Input: an OBDD P_0 and a variable order π_0 */
/* Output: an OBDD P and a variable order π with
    BDD-size(P, π) ≤ BDD-size(P_0, π_0) */
    P = P_0; π = π_0;
    For all i ∈ {1, ..., n} {
        1.(a) /* Move variable x_i through the order */
            optsize = BDD-size(P, π);
            optpos = curpos = startpos = π^{-1}[i];
            For j = startpos − 1, ..., 1 (descending) {
                curpos = j;
                swap_π(P, x_{π[j]}, x_{π[j+1]});
                If BDD-size(P, π) < optsize {
                    optsize = BDD-size(P, π);
                    optpos = j;
                }
                Else If BDD-size(P, π) > MaxGrowth * optsize {
                    Exit(Step 1.(a));
                }
            }
        1.(b) For j = curpos + 1, ..., n {
                curpos = j;
                swap_π(P, x_{π[j−1]}, x_{π[j]});
                If BDD-size(P, π) < optsize {
                    optsize = BDD-size(P, π);
                    optpos = j;
                }
                Else If BDD-size(P, π) > MaxGrowth * optsize {
                    Exit(Step 1.(b));
                }
            }
        2. /* Put variable x_i to the optimal position found in step 1. */
            If curpos > optpos {
                For j = curpos − 1, ..., optpos (descending) {
                    swap_π(P, x_{π[j]}, x_{π[j+1]});
                }
            }
            Else {
                For j = curpos + 1, ..., optpos {
                    swap_π(P, x_{π[j−1]}, x_{π[j]});
                }
            }
    }
}
```

Figure 1.6: The Sifting Algorithm

able in the current direction, the minimum cannot be improved. In that case the process of moving the variable in this direction is interrupted immediately.

In many cases, the Sifting algorithm leads to very good variable orders. If the OBDD is not too large, its time consumption is acceptable in many applications, especially because the quality of the variable order can make the difference between success and failure in completing an application [Bry95].

By exploiting additional criteria like the symmetry relations between individual variables, the basic variant of Sifting can be further improved [PSP94, PS95].

1.2 Variants of OBDDs

1.2.1 Ordered Functional Decision Diagrams

If f denotes the function which is represented by a node in the OBDD with label x_i, and g, h denote the functions represented by the two sons, then Shannon's decomposition

$$f = x_i g + \overline{x_i} h$$

is satisfied. It is also possible to perform other decomposition types in the nodes, for example the so-called **Reed-Muller expansion**

$$f = g \oplus x_i h.$$

These **ordered functional decision diagrams** (**OFDDs**, [KSR92]) are particularly useful in connection with problems based on the exclusive-or operation, e.g. the minimization of AND-XOR-polynomials. Going a step further, it is shown in [DST+94] that the different decomposition types can be combined within the same graph while preserving good algorithmic properties (**ordered Kronecker functional decision diagrams**, OKFDDs).

1.2.2 Zero-Suppressed Binary Decision Diagrams

For many applications with a combinatorial background the corresponding Boolean functions are 1 at only very few positions. This observation is exploited by the so-called **zero-suppressed BDDs** (**ZBDDs**, **ZDDs**) by means of a modified elimination rule [Min93, Min96a]. In contrast to OBDDs, we will not eliminate the nodes with identical 0- and 1-successor, but those nodes whose 1-successor is the 0-sink, see Figure 1.7. By using this data structure many problems in the areas of two- and multi-level logic optimization have been solved efficiently [Cou94, Min96a].

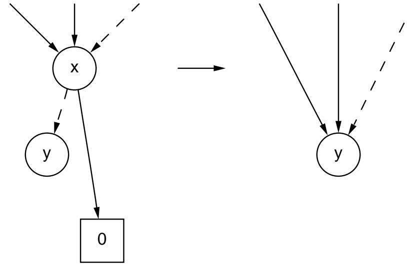


Figure 1.7: Elimination rule for ZDDs

1.3 Transformed Binary Decision Diagrams

As described in section 1.1.3 the main optimization parameter of OBDDs is the underlying variable order. In 1995, Bern, Meinel and Slobodovà [BMS95] (see also [MS97]) proposed a general framework for enlarging this optimization space. They showed that the concept of **domain transformations**, well-known in many areas of mathematics and physics (e.g. Fourier or Laplace transformation), can also be applied in the context of Boolean manipulation. They developed the concept of **transformed BDDs** (**TBDDs**) which allows to manipulate Boolean functions by using transformed versions of the functions.

More precisely, they construct **cube transformations** τ which are bijective mappings from $\mathbb{B}^n \to \mathbb{B}^n$. A cube transformation τ induces a mapping $\Phi_\tau : \mathbb{B}_n \to \mathbb{B}_n$ onto the Boolean algebra $\mathbb{B}_n$ with $\Phi_\tau(f)(a) = f(\tau(a))$ for every $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$.

Now, instead of representing and manipulating the original function $f \in \mathbb{B}_n$, the idea is to use the transformed function $\Phi_\tau(f) = f(\tau)$. This variable transformation preserves the efficient manipulation as shown by the following fact:

Fact 1.4. *If $\tau : \mathbb{B}^n \to \mathbb{B}^n$ is a cube transformation, and $f_1, f_2 \in \mathbb{B}_n$ are Boolean functions, then Φ_τ defines an automorphism on $\mathbb{B}_n$, i.e. the following holds:*

1. *$f_1 = g_1$ if and only if $\Phi_\tau(f_1) = \Phi_\tau(g_1)$.*
2. *Let $*$ be any binary operation on $\mathbb{B}_n$. If $f = f_1 * f_2$, then $\Phi_\tau(f) = \Phi_\tau(f_1) * \Phi_\tau(f_2)$.*

In other words, due to the second statement the polynomial complexity of the Boolean operations remain valid even if we work with the transformed functions. If,

for example, one wants to check two given functions for equivalence, statement 1 tells us, that in this situation it is not necessary at all to retransform the functions.

The realization of this general framework requires to find good techniques for finding suitable variable transformations τ which lead to small OBDD-sizes for the transformed versions of the relevant functions. The first idea which has been proposed by [BMS95] is based on **type-based** transformations:

Definition 1.5. *A* **complete type** σ *over the variables* $\{x_1, \ldots, x_n\}$ *is defined like a (not necessarily reduced) OBDD with three exceptions:*

1. *It has only one sink.*
2. *There does not need to exist an underlying variable ordering.*
3. *On each source-to-sink-path, each variable occurs exactly once.*

Figure 1.8 shows a complete type for the four variables x_1, x_2, x_3, x_4.

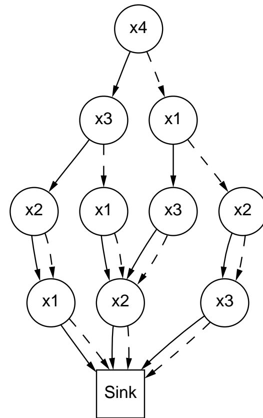


Figure 1.8: Complete type

With the help of complete types we may define cube transformations: In the following, let σ be a fixed complete type. Then each assignment $a = (a_1, \ldots, a_n) \in \{0,1\}^n$ defines a uniquely determined source-to-sink-path $p(a)$ within σ. $a^{[i]}$ denotes the index of the variable tested in the i-th position of $p(a)$. The cube transformation $\tau : \mathbb{B}^n \to \mathbb{B}^n$ for the given complete type σ is defined by

$$\tau(a_1, \ldots, a_n) = (a_{a^{[1]}}, \ldots, a_{a^{[n]}})$$

In [BMS95] heuristics have been proposed to construct suitable type-based transformations from a given circuit topology.

One of the main advantages of this transformation technique are that its realization works on top of a given data structure. Hence, in contrast to other OBDD-variants it is not necessary for the TBDD-realization to replace existing OBDD-packages, but instead the new optimization techniques can be integrated on top of an existing OBDD-package (see also [FKB95]).

1.4 Finite State Machines

Many problems in the area of VLSI design can be modeled by means of *finite state machines* which are the underlying model of sequential circuits. Figure 1.9 shows a small example.

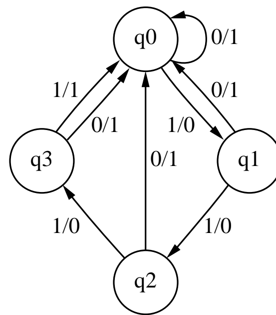


Figure 1.9: Simple finite state machine

Definition 1.6. *A* **finite state machine (FSM)** M *is defined by a tuple* $M = (Q, I, O, \delta, \lambda, Q_0)$*, where*

- Q *is the set of states,*
- I *is the input alphabet,*
- O *is the output alphabet,*
- $\delta : Q \times I \to Q$ *is the next-state function,*
- $\lambda : Q \times I \to O$ *is the output function, and*
- Q_0 *is the set of initial states.*

As usual in VLSI design, all components of the state machine are assumed to be binary encoded. Let p be the number of input bits, n be the number of state bits and m be the number of output bits. Then δ is a function $\mathbb{B}^n \times \mathbb{B}^p \to \mathbb{B}^n$, λ is a function $\mathbb{B}^n \times \mathbb{B}^p \to \mathbb{B}^m$, and Q_0 is a subset of $\mathbb{B}^n$.

The transition behavior of a binary encoded finite state machine can be represented via the transition functions $\delta_1, \ldots, \delta_n$ or via the transition relation:

Definition 1.7. *Let $M = (Q, I, O, \delta, \lambda, Q_0)$ be a binary encoded finite state machine with n state bits and p input bits. The (characteristic function of the)* **transition relation** $T : \mathbb{B}^{2n+p} \to \mathbb{B}$ *of M is defined by*

$$T(x_1, \ldots, x_n, y_1, \ldots, y_n, e_1, \ldots, e_p) = T(x, y, e) = \prod_{i=1}^{n} (y_i \equiv \delta_i(x, e)),$$

where $\equiv$ is the Boolean equivalence function. The variables $x_1, \ldots, x_n$ are called **current-state variables**, *and the variables $y_1, \ldots, y_n$ are called* **next-state variables**.

The importance of the transition relation originates from two aspects: On the theoretical side, it represents all possible transitions of the finite state machine within a single Boolean function. On the practical side, the transition relation is of basic importance for the algorithms on finite state machines that will be discussed in the next section.

1.5 Reachability Analysis

Many problems in the context of finite state machines, like equivalence checking, can be solved by means of symbolic OBDD-representations. The core of the OBDD-based method is to reduce the verification of global system properties to the verification of local properties which hold for all states that can be reached from the initial state. For this reason, **reachability analysis** is of particular importance in the process of formal verification: By using the OBDD data structure the set of reachable states is computed and compactly represented.

The standard reachability algorithm is based on a breadth-first traversal of the finite state machine [CBM89, CM95]. A basic variant can be described as follows: Let $\chi_j(x_1, \ldots, x_n) : \{0,1\}^n \to \{0,1\}$ be the characteristic function of the states which can be reached in at most j steps. The computation of the function χ_{j+1} starting from the function χ_j is described by the following Boolean equation which reflects the computation of the images of all states in χ_j under the next-state function δ:

$$\chi_{j+1}(y_1,\ldots,y_n) = \chi_j(y_1,\ldots,y_n) + \exists x_1,\ldots,x_n\ \exists e_1,\ldots,e_p \left(\prod_{i=1}^{n}(y_i \equiv \delta_i(x,e))\,\chi_j(x_1,\ldots,x_n)\right),$$

where $\exists x_i$ the Boolean existential quantifier

$$\exists x_i f = f|_{x_i=0} + f|_{x_i=1}.$$

This iteration is continued, until eventually a fixed point is reached which represents the set of reachable states. There are many refinements and variants of this form of image computation which all aim at keeping possible intermediate results during the computation small. The currently best methods for image computation are mainly based on the idea to partition the constant part of the above equation,

$$\prod_{i=1}^{n}(y_i \equiv \delta_i(x,e)),$$

the transition relation, in order to perform the quantifications as efficiently as possible [BCL+94, RAB+95].

1.6 Relevant Software Systems

The implementation of our optimization ideas which will be presented later on has been integrated into the environment of existing software systems. We will shortly mention the relevant systems.

1.6.1 The CUDD Package

The **CUDD** package (**Colorado University Decision Diagrams**) is a software library written in the programming language C. It provides large number of functions to manipulate OBDDs and ZDDs. Furthermore it can handle so-called algebraic decision diagrams (ADDs, [BFG+93])) which serve to represent functions from $\{0,1\}^n$ to an arbitrary set. The package has been developed by Fabio Somenzi and his working group at the University of Colorado at Boulder. Its initial version became available in April 1996. Meanwhile, the last publicly available version is numbered 2.1.2.

Recent experimental studies have shown that the package provides one of the most efficient implementations of BDD-based algorithms [Sen96, MGS97].

CUDD's primary distinguishing features are a large number of algorithms and heuristics for dynamic variable reordering. In particular, support is provided for Sifting, group Sifting (Sifting on user-defined variable groups), window permutations (all variable-order permutations among a group of adjacent variables in the BDD are tested [FMK91, ISY91]), identification and linking of symmetric variables [PS95], simulated annealing [BLW95], and genetic algorithm-based reordering [DBG95].

1.6.2 The SIS System

SIS (**Sequential Interactive Synthesis**) is an interactive tool for synthesis and optimization of combinational and sequential circuits [SSL+92, SSM+92]. It has been developed at the University of California at Berkeley, and is currently available in version 1.3.

Given a state transition table, a signal transition graph, or a logic-level description of a sequential circuit, it produces an optimized net-list in the target technology while preserving the input/output behavior. As a special case it can produce an optimized net-list in the target technology when starting from the logic-level description of a combinational circuit.

SIS has been a pioneering system in the area of modern design tools, and contains a large number of state-of-the-art algorithms. It has also been widely used within commercial environments, and has influenced many industrial systems [Mic94].

The SIS program employs OBDDs as internal data structure.

1.6.3 The VIS System

VIS (**Verification Interacting with Synthesis**) is a software system that integrates the synthesis, simulation and formal verification of finite state machines. It has been developed jointly at the University of California at Berkeley and at the University of Colorado at Boulder [BHS+96]. Currently, the newest version is release 1.2. In VIS, the underlying data structure for all algorithms are OBDDs and ZDDs. It is possible to choose between different OBDD packages. Currently, the following packages are supported:

- The CUDD package.
- The package of David Long developed at Carnegie Mellon University [Lon92].
- The so-called breadth-first package which has been developed at the University of California at Berkeley [SRBS96].

VIS can interact with SIS to optimize the existing logic by reading and writing the BLIF format (Berkeley Logic Interchange Format, [SSL+92]) which describes circuits by means of net-list descriptions.

Chapter 2

The Influence of the State Encoding

As described in the previous chapter, the main optimization parameter of OBDDs is the underlying variable order. Many research efforts have tried to characterize the complexity of the relevant variable ordering problems and to come up with efficient optimization algorithms for obtaining large size reductions without aiming at the global minimum. Unfortunately, there are many important applications, in particular in the analysis of finite state machines, where this optimization technique reaches its limits.

In connection with finite state machines the OBDD-size does not only depend on the variable order but also on the state encoding. For a fixed state encoding there are many finite state machines whose OBDD-representations are large w.r.t. all variable orders [ATB94]. The general idea which will be investigated in this chapter and the following one is to use the state encoding as an additional optimization potential.

This general optimization idea immediately raises questions from different aspects:

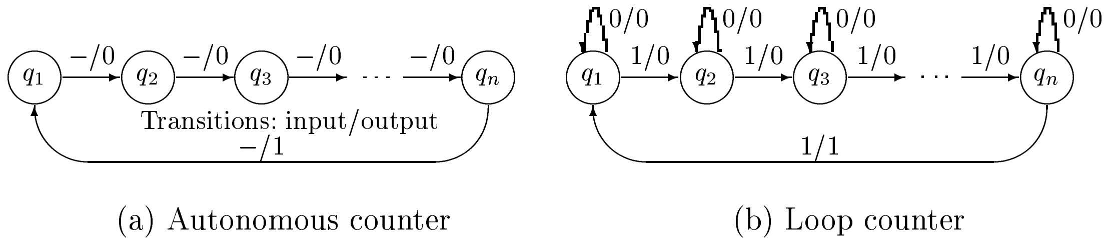
Principle aspects: *To which extent* can the choice of the state encoding influence the OBDD-size at all ?

Practical aspects: *Which techniques* can be used to exploit this optimization potential ?

In this chapter we will give results concerning the first question. Later on, in the next chapter, we will turn towards the second question.

For the characterization of the optimization potential, we consider classes of counters which have a simple structure but which appear in numerous practical examples. For these classes, we analyze the relationship between the state encoding and the OBDD-size from a combinatorial point of view and give some precise answers concerning this relationship.

In particular, we consider the autonomous and the loop counter shown in Figure 2.1 [GDN92].



(a) Autonomous counter (b) Loop counter

Figure 2.1: Counter types

In Figure 2.1, the input symbol '−' at an edge means that this edge is used for both inputs 1 and 0. In addition to the two depicted counters we analyze an acyclic counter which can be constructed out of the autonomous counter by deleting the "backward" edge from the last state to the first state. The results, although derived for a quite specific class of finite state machines, serve as reference examples for the task of finding re-encodings. The main contributions of this chapter are:

1. When fixing the variable order in a reasonable way, we derive the exact OBDD-sizes for the counters w.r.t. some important encodings.

2. We present lower bounds for the OBDD-sizes of counter encodings which are very close to the derived OBDD-sizes for the standard encoding. These bounds underline the suitability of the standard encoding in this context.

3. We construct worst-case encodings which lead to exponential-size OBDDs and hence demonstrate the sensitivity in choosing the appropriate state encoding.

2.1 Basic Framework

In order to prove precise statements about the relation between the state encoding and the OBDD-size we will use the transition relation which has been introduced in Definition 1.7.

The application which stands behind the transition relation is to compute the set of reachable states. Therefore, we can consider equivalently the transition relation of the underlying non-deterministic finite state machine in which the inputs have been eliminated. In terms of Boolean manipulation this modification corresponds to an existential quantification over the inputs. Note that the transition relation can be used to represent a finite state machine even if the next-state function does not exist, for example if the state machine is non-deterministic.

As already mentioned in Section 1.5 the most efficient implementations of this general concept work with a *partitioned* transition relation. However, for the investigation of the question in how far the choice of the state encoding can influence the OBDD-size, we chose to consider the *unpartitioned* transition relation for the following reasons:

- The unpartitioned transition relation represents the given finite state machine in a single function.
- The analysis does not depend on the specific choice of partitioning heuristics.
- Although for large state spaces, the transition relation is typically significantly larger than other representations (like a partitioned variant), the general techniques which we present remain valid for many types of representations.

For a finite state machine M, we derive the reduced OBDD-size for the characteristic function of its transition relation. This size is shortly called **OBDD-size of** M. The OBDD-size crucially depends on the chosen variable ordering. There are two variable orderings which often appear in connection with finite state machines: the **separated** ordering $x_1, \ldots, x_n, y_1, \ldots, y_n$ and the **interleaved** ordering $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$ [ATB94, Kri94].

For practical applications, the interleaved variable ordering is often superior to the separated ordering. If one considers for example a deterministic autonomous (i.e. input-independent) machine with a bijective next-state function, then the OBDD w.r.t. the separated ordering has exponential size. The reason is that after reading the variables $x_1, \ldots, x_n$ all induced subfunctions are different. The restriction to fix the variable ordering is reasonable in our context, as we want to analyze the effect of different state encodings.

2.2 The Lower Bound

We will investigate lower bounds for the OBDD-size of an autonomous counter with 2^n states where we keep the interleaved variable order fixed and vary over all $(2^n)!$

possible n-bit state encodings. For the first lower bound we make use of the fact that the next-state function of the autonomous counter is bijective (as the inputs are not relevant for an autonomous machine, we can consider the next-state function as a function from Q to Q). Note, that every finite state machine with a bijective next-state function is autonomous (i.e. input-independent).

Theorem 2.1. *Let M_{2^n} be an autonomous finite state machine with 2^n states, n encoding bits and a bijective next-state function. The OBDD-size of M_{2^n} w.r.t. the interleaved variable order is at least $3n + 2$.*

Proof. We show that for each $1 \leq i \leq n$, there are at least 3 nodes with label x_i or y_i. As the next-state function is bijective, each of the $2n$ variables appears on every path from the root to the 1-sink. There exists a node A labeled by x_i whose 1-edge leads to a sub-OBDD which does not represent the constant 0 (otherwise the next-state function cannot be a total function). In this sub-OBDD there exists a path from the root to the 1-sink, and therefore the root must be labeled by y_i (see Figure 2.2 (a)). Analogously, there exists a node B labeled by x_i whose 0-edge leads to a sub-OBDD with root label y_i.

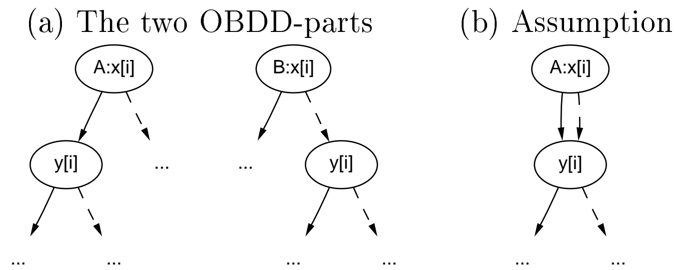


Figure 2.2: Lower bound

If we assume that A and B are identical and that the two roots of the sub-OBDDs are identical, then node A can be reduced, see Figure 2.2 (b). The resulting OBDD has a path from the root to the 1-sink on which the variable x_i does not appear, a contradiction. Altogether we have at least $3n$ internal nodes and 2 sink nodes. □

The next theorem improves the lower bound by explicitly using the properties of a counter.

Theorem 2.2. *The OBDD-size of an autonomous counter with 2^n states, n encoding bits and interleaved variable order is at least $4n + 1$.*

Proof. First, we show that every variable $x_2, \ldots, x_n$ must appear at least twice in the OBDD. Assume for a contradiction that x_i appears only once. As the next-state function is bijective each 1-path in the OBDD goes through the vertex labeled by x_i. Every assignment $(x_1, y_1, \ldots, x_{i-1}, y_{i-1}) \in \{0,1\}^{2i-2}$ that leads to the x_i-node can be combined with every assignment $(x_i, y_i, \ldots, x_n, y_n) \in \{0,1\}^{2n-2i+2}$ that leads from the x_i-node to the 1-sink in order to construct a transition of the counter. Intuitively, these two groups of variables act independently. For each $(x_i, \ldots, x_n)$ there exists a vector $(y_i, \ldots, y_n)$ such that the assignment $x_i, y_i, \ldots, x_n, y_n$ leads from the x_i-node to the 1-sink. After exactly 2^{n-i} transitions of the form $x_i \ldots x_n \to y_i \ldots y_n$ the cycle w.r.t. the last $n-i+1$ variables is finished. Analogously, the first $i-1$ variables form a cycle of length 2^{i-1}. The resulting cycle is of length 2^n if and only if the least common multiple of 2^{i-1} and 2^{n-i+1} is 2^n which is impossible for $2 \leq i \leq n$. Analogously, we can show that each variable y_i must appear at least twice in the OBDD. Altogether there are at least $4n-1$ internal nodes. □

2.3 The Behavior of Important Encodings

2.3.1 The Standard Minimum-Length Encoding

First, we consider the standard encoding where 2^n states are represented by n bits, and the encoding of a state q_i is the binary representation of i (see Figure 2.3). Let M_{2^n} be the autonomous counter with 2^n states under this encoding.

state	encoding
q_0	$00\ldots000$
q_1	$00\ldots001$
q_2	$00\ldots010$
$\vdots$	$\vdots$
q_{2^n-1}	$11\ldots111$

Figure 2.3: The standard encoding

The **MSB-first** (most significant bit first) variable order is the interleaved variable order which reads the bits in decreasing significance. In contrast, the **LSB-first** variable order reads the bits in increasing significance.

Lemma 2.3. *For $n \geq 2$, the reduced OBDD for M_{2^n} w.r.t. the MSB-first variable order has $5n-1$ nodes.*

Proof. The idea is to use the OBDD for $M_{2^{n-1}}$ in order to construct the OBDD for M_{2^n}. Formally, this leads to a proof by induction. We show: The reduced OBDD is of the form like in Figure 2.4 (a) (in the sense that the shown nodes exist and are not pairwise isomorphic) with sub-OBDDs A and B and has exactly $5n-1$ nodes. The case $n=2$ can easily be checked.

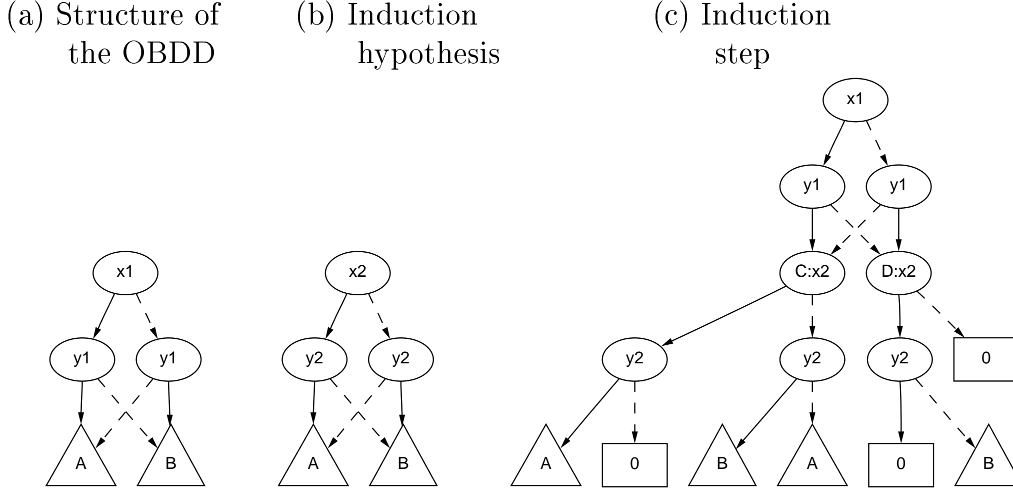


Figure 2.4: MSB-first

Induction step: The OBDD for the $n-1$ bits $x_2, \ldots, x_n$ has the form like in Figure 2.4 (b). Let $|x|$ be the binary number that is represented by a bit-string $x \in \{0,1\}^+$. With this notation we have
A: leads to the 1-sink if and only if $|y_3 \ldots y_n| = |x_3 \ldots x_n| + 1$, $x_3 \ldots x_n \neq 11 \ldots 1$.
B: leads to the 1-sink if and only if $x_3 = \ldots = x_n = 1$, $y_3 = \ldots = y_n = 0$.

We construct the reduced OBDD for M_{2^n} like in Figure 2.4 (c). The subfunctions rooted in C and D have the following meanings:
C: leads to the 1-sink if and only if $|y_2 \ldots y_n| = |x_2 \ldots x_n| + 1$, $x_2 \ldots x_n \neq 11 \ldots 1$.
D: leads to the 1-sink if and only if $x_2 = \ldots = x_n = 1$, $y_2 = \ldots = y_n = 0$.

It can easily be checked that all the subfunctions rooted in the new invented nodes are pairwise different. Therefore size(M_{2^n}) = size$(M_{2^{n-1}}) - 3 + 8$ = size$(M_{2^{n-1}}) + 5$. □

Lemma 2.4. *For $n \geq 2$, the reduced OBDD for M_{2^n} with respect to the LSB-first variable order has $5n-1$ nodes.*

Proof. The case $n = 2$ can easily be checked.
Induction step: The OBDD for the $n-1$ variables $x_2, \ldots, x_n$ has the form like in Figure 2.5 (a). We have
A: leads to the 1-sink if and only if $|y_n \ldots y_3| = |x_n \ldots x_3| + 1$ or $x_3 = \ldots = x_n = 1$, $y_3 = \ldots = y_n = 0$.
B: leads to the 1-sink if and only if $|y_n \ldots y_3| = |x_n \ldots x_3|$.

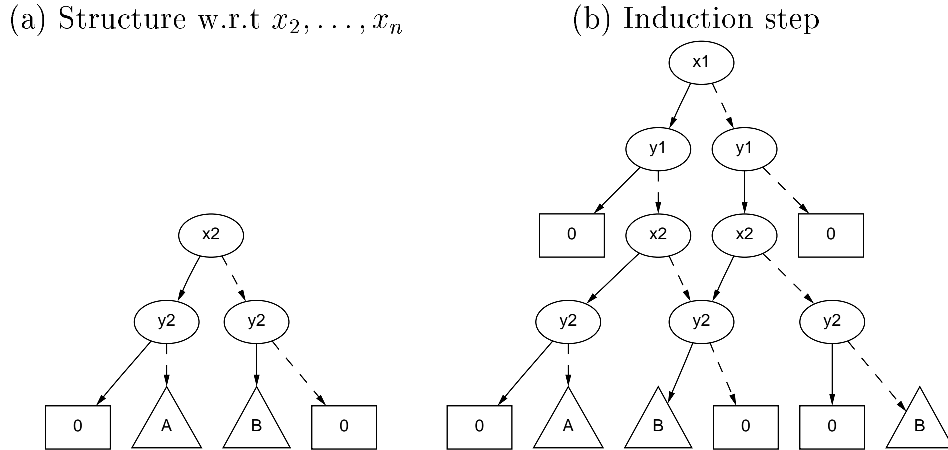


Figure 2.5: LSB-first

The construction of the reduced OBDD for M_{2^n} like in Figure 2.5 (b) yields

$$\text{size}(M_{2^n}) = \text{size}(M_{2^{n-1}}) - 3 + 8 = \text{size}(M_{2^{n-1}}) + 5.$$

□

Note that the OBDD-sizes for the MSB- and the LSB-variable order are equal, although the OBDDs are not isomorphic. The main reason for the equality in size is the fact that in both OBDDs there is only one bit of information that has to be passed from the level of y_{i-1} to the level of x_i for $2 \leq i \leq n$.

It is quite remarkable that these OBDD-sizes for the standard encoding nearly meet the lower bound of $4n+1$. We conjecture that the standard encoding is even optimal. In order to prove a better lower bound, one might have to establish much more sophisticated communication complexity arguments which connect local OBDD-properties with the global single-cycle-property of a counter.

2.3.2 The Gray Encoding

Another important minimum-length encoding is the one where the encoding of state i is the Gray code representation of i. The Gray code has the property that all successive code words differ in only one bit. The n-bit Gray code can be constructed by reflecting the $(n-1)$-bit Gray code. To all the new code words, a leading 1 is added. Figure 2.6 shows a 4-bit Gray code with least significant bit x_1.

		Decimal number															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	x_4	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Gray	x_3	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
code	x_2	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
	x_1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0

Figure 2.6: 4-bit Gray code

As in the case of the standard encoding, the MSB-first variable order reads the bits in *decreasing* significance: x_1 and y_1 correspond to the most significant bit.

Lemma 2.5. *For* $n \geq 2$, *the OBDD-size of the autonomous counter with respect to the Gray code and MSB-first variable order is* $10n - 9$.

Proof. The case $n = 2$ can easily be checked.
Induction step: Let $M^G_{2^n}$ be the autonomous counter with 2^n states under the Gray encoding. The OBDD for $M^G_{2^{n-1}}$ has the form like in Figure 2.7.

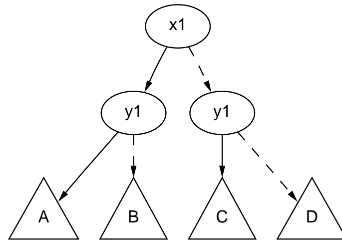


Figure 2.7: Gray code OBDD-structure

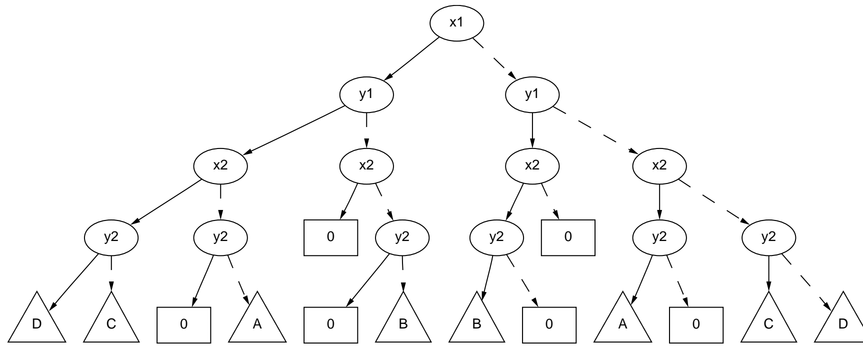


Figure 2.8: Gray code induction step

We construct the reduced OBDD for $M^G_{2^n}$ as shown in Figure 2.8. The frontier of the constructed OBDD illustrates the reflecting property of the Gray code. It follows

$$\text{size}(M^G_{2^n}) = \text{size}(M^G_{2^{n-1}}) - 3 + 13 = \text{size}(M^G_{2^{n-1}}) + 10.$$

□

The OBDD-representation for the Gray encoding is also linear but the corresponding factor is bigger than the factor for the standard encoding. The essential reason for this difference is the reflecting property of the Gray code which does not allow an immediate use of the OBDD for $M^G_{2^{n-1}}$ when constructing the OBDD for $M^G_{2^n}$.

It can be checked that the LSB-first variable order yields exactly the same number of nodes as the MSB-first order. The construction for LSB-first works as follows: Consider the $n-1$ most significant bits of each Gray code word. For all i, the truncated code words corresponding to the decimal numbers $2i$ and $2i+1$ are identical. If one considers only the code words which correspond to the even (or equivalently odd) decimal numbers, one obtains a Gray code of lower dimension. The recursive OBDD-construction adds in each step 10 nodes like in the case of the MSB-first order.

2.4 A Worst-Case Encoding

Of course, it is easy to construct autonomous finite state machines with bijective next-state function together with an encoding which have exponential OBDD-size w.r.t. the interleaved variable order. Things become more difficult if one wants to construct the encoding of an autonomous *counter* that leads to exponential OBDD-size. In the following we construct an encoding in which the first nodes in the OBDD

labeled by $x_1, y_1, \ldots, x_{n/2}, y_{n/2}$ and their outgoing edges lead to a complete tree. The number of nodes with label $x_{n/2+1}$ will therefore be 2^n and the OBDD will have more than 2^{n+1} nodes.

There are 2^n different assignments to the **leading variables** $x_1, y_1, \ldots, x_{n/2}, y_{n/2}$ and 2^n transitions in the finite state machine. To ensure that the leading variables in the OBDD generate a complete tree, we construct the next-state function in the following way:

1. For each assignment to the leading variables, there exists exactly one assignment to the **tail variables** $x_{n/2+1}, y_{n/2+1}, \ldots, x_n, y_n$ which leads to the 1-sink in the OBDD.

2. Each of the 2^n assignments to the tail variables appears exactly once in the construction.

For the construction of the worst-case counter, we build up two tables, the **transition table** and the **tail table**.

Transition table: This table consists of 2^n rows. Each row describes one transition $x_1 \ldots x_n \rightarrow y_1 \ldots y_n$ of the counter. Initially, some of the bits are already set to 1 or 0: All the 2^n assignments for the leading variables $x_1, y_1, \ldots, x_{n/2}, y_{n/2}$ are inserted in the table in the following way: In row i of the table, $0 \leq i < 2^n$, the integer i is represented in binary by the bits $x_1 \ldots x_{n/2} y_1 \ldots y_{n/2}$. The bits of the tail variables are not affected by this initial construction – these entries are marked by stars and will be filled during the construction. The transition table after this initial step is shown in Figure 2.9.

There is an additional entry called *visited* in each row which helps to keep track which rows of the table have already been filled during the construction algorithm. Initially, every entry in the *visited* column is set to 'FALSE'.

$x_1 \ldots x_{n/2}$	$x_{n/2+1} \ldots x_n$	$y_1 \ldots y_{n/2}$	$y_{n/2+1} \ldots y_n$	*visited*
00 ... 00	$\star$	00 ... 00	$\star$	FALSE
00 ... 00	$\star$	00 ... 01	$\star$	FALSE
⋮	⋮	⋮	⋮	⋮
00 ... 00	$\star$	11 ... 11	$\star$	FALSE
00 ... 01	$\star$	00 ... 00	$\star$	FALSE
⋮	⋮	⋮	⋮	⋮
11 ... 11	$\star$	11 ... 11	$\star$	FALSE

Figure 2.9: Initial structure of the transition table

Tail table: This table consists of 2^n rows which contain the 2^n different assignments for the tail variables $x_{n/2+1}, \dots, x_n, y_{n/2+1}, \dots, y_n$. In each row there is an additional entry called *used* which helps to ensure that each assignment is only used once during the construction. Initially, every entry in the *used* column is set to 'FALSE'. The tail table is shown in Figure 2.10.

$x_{n/2+1} \dots x_n$	$y_{n/2+1} \dots y_n$	*used*
$00 \dots 00$	$00 \dots 00$	FALSE
$00 \dots 00$	$00 \dots 01$	FALSE
$\vdots$	$\vdots$	$\vdots$
$11 \dots 11$	$11 \dots 11$	FALSE

Figure 2.10: Initial structure of the tail table

The entries of the transition table which are marked with a star are filled during the construction. The task is to put each assignment for the tail variables into one row of the transition table in such a way that the induced transitions $x_1 \dots x_n \rightarrow y_1 \dots y_n$ form a cycle of length 2^n. Note that this construction guarantees that the two properties are satisfied. We use the algorithm of Figure 2.11.

Claim. The finite state machine that is constructed by the algorithm is a counter, i.e. the cycle has length 2^n.

The claim follows from three statements:

1. In step 3.(a) there exists an assignment in the tail table with the desired properties.

2. If the while-condition in step 3 is not satisfied (i.e. the while-loop terminates), then $x_1 \dots x_n = 00 \dots 0$ in the present row of the transition table.

3. When the while-loop terminates, all 2^n assignments for the tail variables have been marked as used.

Before proving the statements we show why the claim follows from them: From statement 1 it follows that step 3.(b) is well-defined in each processing of the loop body. Due to the finiteness of the tail table and step 3.(c), the algorithm terminates. Statements 2 and 3 guarantee that the construction builds a cycle of length 2^n.

1. For each row of the transition table: Set $x_{n/2+1} \dots x_n := y_1 \dots y_{n/2}$.
2. Set the present row to the top row of the transition table.
3. While *visited* in the present row is set to 'FALSE'
   (a) Set the *visited* entry in the present row of the transition table to 'TRUE'.
   (b) Choose the maximal (w.r.t. the represented binary number) assignment for the tail variables from the tail table which matches the assignment for $x_{n/2+1} \dots x_n$ in the present row and whose *used* entry is set to 'FALSE'.
   (c) Set the *used* entry for the chosen assignment in the tail table to 'TRUE'.
   (d) Set $y_{n/2+1} \dots y_n$ in the transition table to the chosen assignment for $y_{n/2+1} \dots y_n$.
   (e) Let R be the row in the transition table in which the assignment for $x_1 \dots x_n$ is identical with the assignment for $y_1 \dots y_n$ in the present row.
   (f) Set the present row to row R.

Figure 2.11: Construction of the worst-case counter

Proofs of the statements:
1. The *visited* column of the transition table guarantees that each row of the transition table is at most once the present row – otherwise, the while-loop immediately terminates. For a *fixed* assignment X to $x_{n/2+1} \dots x_n$ the number of rows in the transition table in which the assignment X appears is $2^{n/2}$. Therefore there are at most $2^{n/2}$ situations during the run of the algorithm in which an assignment is needed in step 3.(b) that extends X. On the other hand there are also $2^{n/2}$ assignments for the tail variables in the tail table which extend X. Therefore in each processing of step 3.(b), there is at least one previously unused suitable assignment left.

2. Consider the assignment for $y_1 \dots y_n$ in the present row after step 3.(d). Due to step 1 of the algorithm this assignment is equal to $x_{n/2+1} \dots x_n y_{n/2+1} \dots y_n$ in the present row. Due to step 3.(d) and (e) this assignment determines the assignment for $x_1 \dots x_n$ in the new present row. It follows: Whenever a row with a given bit sequence for $x_1 \dots x_n$ becomes the present row in step 3.(f), this bit sequence has just been marked as used in the tail table in step 3.(c). This makes it impossible

that a row becomes the present row in step 3.(f) more than once.

After the first processing of the loop body, the top row is marked as visited, but the bit sequence $00\ldots0$ has not been marked as used in the tail table. Therefore the only possibility to enter a row whose *visited* entry is already set to 'TRUE' is to enter the top row.

3. Due to statement 2, the last chosen assignment for the tail variables is $00\ldots0$. As in step 3.(b) the maximal assignment is chosen, all $2^{n/2}$ assignments of the form $00\ldots0\ldots \in 0^{n/2}\{0,1\}^{n/2}$ must have already been used now. Therefore, due to the bit shifting in step 1 and step 3.(e), all assignments $\ldots00\ldots0 \in \{0,1\}^{n/2}0^{n/2}$ in the tail table must have already been used now. Using again that in 3.(b) the maximal assignment is chosen, it follows that all 2^n assignments for the tail variables have been used when the while-loop terminates. □

Example 2.6. We will illustrate the processing of the worst-case construction for the case $n = 4$. The transition table and the tail table after step 1 of the algorithm are shown in Figure 2.12.

Transition table				
x_1x_2 x_3x_4		y_1y_2 y_3y_4		*visited*
00	00	00	⋆	FALSE
00	01	01	⋆	FALSE
00	10	10	⋆	FALSE
00	11	11	⋆	FALSE
01	00	00	⋆	FALSE
01	01	01	⋆	FALSE
01	10	10	⋆	FALSE
01	11	11	⋆	FALSE
10	00	00	⋆	FALSE
10	01	01	⋆	FALSE
10	10	10	⋆	FALSE
10	11	11	⋆	FALSE
11	00	00	⋆	FALSE
11	01	01	⋆	FALSE
11	10	10	⋆	FALSE
11	11	11	⋆	FALSE

Tail table		
x_3x_4 y_3y_4		*used*
00	00	FALSE
00	01	FALSE
00	10	FALSE
00	11	FALSE
01	00	FALSE
01	01	FALSE
01	10	FALSE
01	11	FALSE
10	00	FALSE
10	01	FALSE
10	10	FALSE
10	11	FALSE
11	00	FALSE
11	01	FALSE
11	10	FALSE
11	11	FALSE

Figure 2.12: Worst-case construction for $n = 4$ after step 2

Figure 2.13 shows the two tables at the end of the construction. Here, the *visited* and the *used* columns also contain information at which moment of the algorithm the flags are set to 'TRUE': An entry i in the *visited* resp. *used* column indicates that this flag is set to 'TRUE' in the i-th iteration of the while-loop.

Transition table		
$x_1x_2\ x_3x_4$	$y_1y_2\ y_3y_4$	*visited*
00 00	00 11	1
00 01	01 01	14
00 10	10 10	9
00 11	11 11	2
01 00	00 00	16
01 01	01 00	15
01 10	10 00	12
01 11	11 00	7
10 00	00 01	13
10 01	01 10	11
10 10	10 01	10
10 11	11 01	5
11 00	00 10	8
11 01	01 11	6
11 10	10 11	4
11 11	11 10	3

Tail table	
$x_3x_4\ y_3y_4$	*used*
00 00	16
00 01	13
00 10	8
00 11	1
01 00	15
01 01	14
01 10	11
01 11	6
10 00	12
10 01	10
10 10	9
10 11	4
11 00	7
11 01	5
11 10	3
11 11	2

Figure 2.13: Worst-case construction for $n = 4$

Initially, the top row is the present row. Consequently, the *visited* entry of the top row contains a 1 in Figure 2.13. In step 3.(b) of the algorithm the entry $x_3x_4y_3y_4 = 0011$ is chosen which is therefore labeled by a 1 in the tail table. Due to step 3.(d) and 3.(e) this chosen assignment makes the row with entry $x_1x_2x_3x_4 = 0011$ become the present row in step 3.(f) and hence during the next iteration of the while-loop. Consequently, this row is labeled by a 2 in the *visited column*.

The next chosen entry from the tail table is 1111, and hence the bottom row is the present row at the beginning of the third iteration of the while-loop. In this iteration, the entry 1110 from the tail table is chosen. At the end of the algorithm, the transition table contains all the transitions of a counter with the stated properties.

Remark. The first paragraph of this section implies that the constructed OBDD has at least 2^n subfunctions of the form $f_{x_1=a_1,\ldots,x_{n/2}=a_{n/2},y_1=b_1,\ldots,y_{n/2}=b_{n/2}}$ for $a_1,\ldots,a_{n/2}$,

$b_1, \ldots, b_{n/2} \in \{0, 1\}$. Hence, the worst-case construction also holds for a bigger class of variable orders than only the fixed interleaved order $x_1, y_1, \ldots, x_n, y_n$: Namely, it also holds for all variable orders in which all variables from the upper half come from the set $\{x_1, \ldots, x_{n/2}, y_1, \ldots, y_{n/2}\}$ and all variables from the lower half come from the set $\{x_{n/2+1}, \ldots, x_n, y_{n/2+1}, \ldots, y_n\}$.

2.5 Related Topologies

The exact OBDD-sizes for the standard and the Gray encoding are summarized in Figure 2.14. All results are valid for $n \geq 3$.

	autonomous counter	loop counter	acyclic counter
Standard MSB-first	5n-1	5n-1	5n-1
Standard LSB-first	5n-1	5n-2	5n-1
Gray MSB-first	10n-9	10n-9	10n-11
Gray LSB-first	10n-9	15n-21	10n-11

Figure 2.14: Related topologies

With similar techniques most of the results that have been proven for the autonomous counter can also be established for the loop counter and the acyclic counter: The lower bound of $4n + 1$ can be transferred to the acyclic counter. The construction can be slightly modified to prove a $4n$ lower bound for the loop counter which becomes non-deterministic after the elimination of the inputs. The worst-case construction for the autonomous counter can also be used to construct a worst-case encoding for the acyclic counter.

2.6 Conclusion and Open Questions

We have given some precise results on the relation between state encodings and the size of OBDD-representations. These results show a strong dependence and therefore recommend to investigate effective re-encoding techniques to exploit this optimization potential. This will be done in the next chapter.

The general open problems concerning the material of the current chapter are to analyze non-linear topologies of finite state machines, more general variable orders and the behavior during reachability analysis in the context of state encodings. For these tasks, the presented results and analysis techniques form the basic ingredients.

Chapter 3

Local Encoding Transformations

In this chapter we will be concerned with techniques for exploiting the extended optimization framework of OBDDs that has been investigated in the previous chapter.

The underlying general problem of all these efforts is the following: Given the OBDDs for the next-state and output functions of a finite state machine – if one is interested in the input/output behavior of the machine, in how far can the internal state encoding be exploited to minimize OBDD-sizes ? Our approach targets at applying *local* encoding transformations, i.e. transformations which involve only a limited number of encoding bits. These transformations can be interpreted as a re-encoding of the symbolic states. The aim is to minimize OBDD-sizes by iterated application of local transformations. The advantage of this approach is that the costs of applying these transformations are still manageable.

The chapter is structured as follows: We begin with considering the principle optimization potential of re-encoding techniques from an asymptotical point of view. Then, in Section 3.2, we analyze the advantages of local encoding transformations. In Section 3.3 we propose the application of the *XOR-transformation* and show why this transformation is most promising among the set of all encoding transformations. At the end of the section we describe an implementation of this transformation and give some experimental results which illustrate the positive impact of the presented ideas.

3.1 Motivation: The Potential of Re-encoding

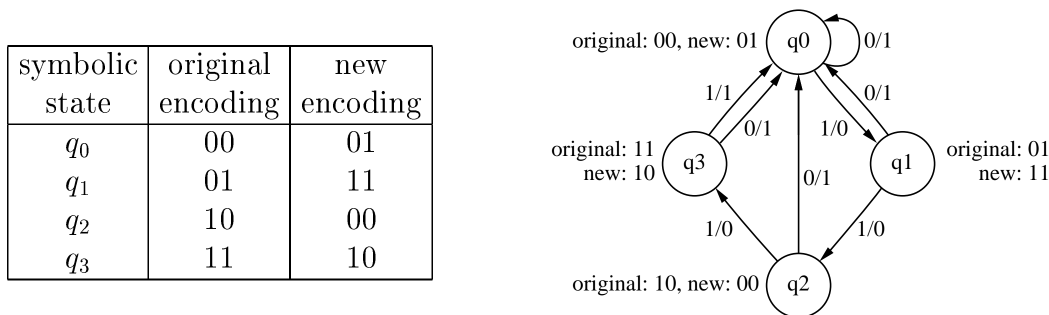
As a starting point for the presented techniques, let us again consider the autonomous counter from Figure 2.1, this time from an asymptotical point of view.

The following theorem shows that almost all encodings for the autonomous counter lead to exponential-size OBDDs, even for their optimal variable order. The size refers to the shared OBDD-size of the transition functions $\delta_1, \ldots, \delta_n$.

Theorem 3.1. *Let $e(n)$ denote the number of n-bit encodings for the autonomous counter with 2^n states which lead to a (shared) OBDD of size at most $2^n/n$ w.r.t. their optimal variable order. Let $a(n) = (2^n)!$ denote the number of all possible counter encodings. Then the ratio $e(n)/a(n)$ converges to zero as n tends to infinity.*

The proof of the theorem can be found at the end of the chapter, in Section 3.7. It is based on ideas of [LL92] and classical counting results of Shannon [Sha49]. An analogous result can be established for the characteristic function of the transition relation and its OBDD-size.

Definition 3.2. *An* **encoding transformation**, *shortly called a* **re-encoding**, *is a bijective function $\rho : \mathbb{B}^n \to \mathbb{B}^n$ that transforms the given state encoding to a new encoding. (For an example see Figure 3.1.) If a state s is encoded by a bit-string $c \in \mathbb{B}^n$, then its new encoding is $\rho(c)$.*



symbolic state	original encoding	new encoding
q_0	00	01
q_1	01	11
q_2	10	00
q_3	11	10

Figure 3.1: Encoding transformation $\rho(c_1, c_2) = (c_2, \overline{c_1})$

This modification of the internal state encoding does not modify the input/output behavior of the state machine. The machine with the new encoding is denoted by

$M' = (Q', I, O, \delta', \lambda', Q'_0)$. Its encoded next-state function, output function and set of initial states are computed as follows:

$$\begin{aligned} \delta'(s,e) &= \rho(\delta(\rho^{-1}(s),e)), \\ \lambda'(s,e) &= \lambda(\rho^{-1}(s),e), \\ Q'_0 &= \rho(Q_0), \end{aligned} \tag{3.1}$$

where $\delta, \delta' : \mathbb{B}^n \times \mathbb{B}^p \to \mathbb{B}^n$, $\lambda, \lambda' : \mathbb{B}^n \times \mathbb{B}^p \to \mathbb{B}^m$, and $Q_0, Q'_0 \subset \mathbb{B}^n$.

The transition relation of the re-encoded machine M' can be obtained from the transition relation of M as follows:

Lemma 3.3. *Let $T(x,y,e)$ be the characteristic function of the transition relation of M. Then the characteristic function $T'(x,y,e)$ of the transition relation of M' is*

$$\prod_{i=1}^{n} \left(\rho_i^{-1}(y) \equiv \delta_i(\rho^{-1}(x)) \right).$$

Therefore $T'(x,y,e)$ can be obtained from $T(x,y,e)$ by the substitutions $y_i \mapsto \rho_i^{-1}(y)$ and $x_i \mapsto \rho_i^{-1}(x)$, $1 \le i \le n$.

Proof. The lemma is a consequence of the following equivalences:

$$\begin{aligned} T'(x,y,e) = 1 &\iff \forall i \; y_i = \rho_i(\delta(\rho^{-1}(x))) &&\iff y = \rho(\delta(\rho^{-1}(x))) \\ &\iff \rho^{-1}(y) = \delta(\rho^{-1}(x)) &&\iff \prod_{i=1}^{n} \left(\rho_i^{-1}(y) \equiv \delta_i(\rho^{-1}(x)) \right) = 1 \end{aligned}$$

□

Example 3.4. The large potential of re-encoding techniques can now be demonstrated at the example of the autonomous counter: By Lemma 2.3 there exists an encoding such that the transition relation of the autonomous counter with 2^n states and n encoding bits has at most $5n-1$ nodes even if the variable order is fixed to $x_1, y_1, \ldots, x_n, y_n$. Hence, for each given encoding of an autonomous counter, there exists a re-encoding which leads to OBDDs of linear size. As according to Theorem 3.1 most encodings lead to OBDDs of exponential size, the gain between the original OBDD and the OBDD after a suitable re-encoding is exponential in most cases. The aim now is to *find* the suitable re-encoding that leads to small OBDD-sizes.

3.2 Local Re-encodings

In the previous section we have shown that re-encodings may have a large impact on the OBDD-size. It is possible that the OBDD becomes much smaller, but in the case of a badly chosen re-encoding the OBDD could even become much larger. This situation is comparable to the problem of finding a good variable order for an OBDD. When changing the variable order of an OBDD, the graph may become much smaller in the best case or much larger in the worst case. This sensitivity is the main reason why it is hard to *find* a good re-encoding or a good variable order.

For the effective *construction* of good variable orders it has turned out that the most efficient strategies are based on local exchanges of variables. The presently best strategies for finding good variable orders dynamically are based on the Sifting algorithm of Rudell that has been described in Section 1.1.5. The main principle of this algorithm is based on a subroutine which finds the optimum position for one variable, if all other variables remain fixed. This subroutine is repeated for each variable. There are two main reasons why this strategy works efficiently:

Bounded size alteration: If one variable x_i is moved to another position in the OBDD, the size of the OBDD cannot change arbitrarily much, in particular it cannot explode. [BLW95] have shown the following theorem:

Theorem 3.5. *Let P be an OBDD. If a variable x_i is moved to a later position in the variable order, then the size of the resulting OBDD P' satisfies*

$$size(P)/2 \leq size(P') \leq size(P)^2.$$

If a variable x_i is moved to an earlier position in the variable order, then the size of the resulting OBDD P' even satisfies the relation

$$size(P)^{1/2} \leq size(P') \leq 2\ size(P).$$ □

Practical studies have shown that in most cases the resulting sizes are even far below the worst-case estimations. Hence, each application of the above mentioned subroutine keeps the size of the OBDD manageable. However, this bounded size alteration for the subroutine does not mean that the optimization potential is limited. The iteration of this subroutine allows to minimize OBDDs very effectively.

Continuity: The procedure for moving a variable x_i to a different position in the order works continuously: During this process only the variables between the original and the new position of x_i are involved, and all nodes labeled by

the remaining variables remain untouched. In particular, the time complexity of this operation is very small if x_i is moved to an adjacent position, and it increases with the number of variables between the original and the new position of x_i in the variable order.

In the case of re-encoding the situation is analogous. It seems to be very hard to find the right *global* re-encoding, whereas it is very promising to combine and iterate operations with restricted local effect. Our approach to construct *local* re-encodings $\rho : \mathbb{B}^n \to \mathbb{B}^n$ is to keep most of the bits fixed (i.e. $\rho_i(x) = x_i$) and vary only on a small number of bits. In particular, if we vary only 2 bits, we will speak of **two-bit re-encodings**. In this case it follows from the worst-case bounds for the synthesis and the substitution of OBDDs that the OBDDs remain polynomial.

Example 3.6. The **exchange variables** re-encoding $\rho_{i,j} : \mathbb{B}^n \to \mathbb{B}^n$, $1 \leq i, j \leq n$, is defined as follows (The following definition shows the case $i < j$, the case $i > j$ is defined analogously):

$$(q_1, \ldots, q_n) \mapsto (q_1, \ldots, q_{i-1}, q_j, q_{i+1}, \ldots, q_{j-1}, q_i, q_{j+1}, \ldots, q_n)$$

Obviously, the exchange variables re-encoding has the same effect on the next-state functions as exchanging the state variables x_i and x_j in the variable order. From all the $(2^n)!$ possible encoding transformations $n!$ can be generated by the iterated application of this transformation type. The inverse mapping ρ^{-1} from Equation 3.1 does not affect the size of the resulting OBDDs as this mapping only causes the renaming of the two functions δ_i and δ_j.

Note, that the transformation which exchanges the encodings of two fixed states may not be seen as a local operation, although the transformation seems to be very simple.

3.3 The XOR-Transformation

We will now propose XOR (exclusive or)-transformations. This transformation is a local re-encoding which operates on two bits.

Definition 3.7. An **XOR-transformation** $\rho_{i,j}$, $1 \leq i, j \leq n$, is defined by

$$(q_1, \ldots, q_n) \mapsto (q_1, \ldots, q_{i-1}, q_i \oplus q_j, q_{i+1}, \ldots, q_n)$$

Short notation: $q_i \mapsto q_i \oplus q_j$. For an example see Figure 3.2.

original encoding		new encoding	
q_1	q_2	q_1^{new}	q_2^{new}
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

Figure 3.2: The XOR-transformation $q_1 \mapsto q_1 \oplus q_2$

Indeed, XOR-transformations provide a solid basis for the design of effective re-encodings due to the following facts:

1. The number of possible re-encodings generated by the iterated application of XOR-transformations is much larger than the number of possible variable orders. Thus, XOR-transformations considerably enlarge the optimization space. On the other hand, the number of these re-encodings is much smaller than the number of all re-encodings which makes it possible to keep the search space manageable.

2. The size influence of this transformation is bounded in a reasonable way like in the case of local changes in the variable order.

3. A precise analysis even shows that an XOR-transformation contains the same asymmetry as the movement of one variable in the variable order. Namely, the bounds for the effect of a transformation $x_i \mapsto x_i \oplus x_j$ depends on the position of x_i and x_j in the variable order.

4. The XOR-transformation is in fact the only new possible re-encoding on two variables.

5. The XOR-transformation can be implemented efficiently like an exchange of two variables in the order.

In the following subsections we will prove these statements.

3.3.1 Enumeration Results

The following combinatorial statements characterize the size of the optimization space provided by the use of XOR-transformations.

Lemma 3.8. *(1) Let $t(n)$ be the number of possible encoding transformations that can be generated by the iterated application of XOR-transformations. It holds*

$$t(n) = \prod_{i=0}^{n-1} (2^n - 2^i).$$

(2) The quotient of $t(n)$ and the number of all possible encoding transformations converges to zero as n tends to infinity.
(3) Let $v(n) := (2n)!$ denote the number of possible variable orders for the transition relation of an autonomous finite state machine with n state bits. The fraction $v(n)/t(n)$ converges to zero as n tends to infinity.

Statement 3 says that in the case of autonomous state machines, there are much more encoding transformations generated by XOR-transformations than variable orders for the transition relation. This relation also holds when the number of input bits is fixed and the number of state bits becomes large.

Proof. (1) Obviously, each XOR-transformation is a regular linear variable transformation over the field $\mathbb{Z}_2$. Moreover, the XOR-transformations provide a generating system for all regular linear variable transformations. Therefore the state encodings which can be obtained by iterated XOR-transformations are in 1-1-correspondence with the regular $n \times n$-matrices over $\mathbb{Z}_2$.

The number of these matrices can be computed as follows: The first row vector b_1 can be chosen arbitrarily from $\mathbb{Z}_2^n \setminus 0$. The i-th row vector b_i, $2 \leq i \leq n$, can be chosen arbitrarily from

$$\mathbb{Z}_2^n \setminus \Big\{ \sum_{j=1}^{i-1} \lambda_j b_j \ : \ \lambda_1, \ldots, \lambda_{i-1} \in \mathbb{Z}_2 \Big\}.$$

These are $2^n - 2^{i-1}$ possibilities for the vector b_i. This proves the claimed number.

(2) This statement follows from the relation

$$\frac{t(n)}{(2^n)!} \leq \frac{2^{n^2}}{(2^n)!} \leq \frac{(n^2)!}{(2^n)!} \to 0 \ \text{ as } \ n \to \infty.$$

(3) It holds

$$\frac{(2n)!}{t(n)} = \frac{2n(2n-1)}{2^n - 2^0} \cdot \frac{(2n-2)(2n-3)}{2^n - 2^1} \cdots \frac{2 \cdot 1}{2^n - 2^{n-1}} \to 0 \ \text{ as } \ n \to \infty. \qquad \square$$

In particular, the number of possible encoding transformations which can be generated by the iterated application of XOR-transformations is smaller than 2^{n^2} which is exactly the number of all $n \times n$-matrices over $\mathbb{Z}_2$.

It follows from the previous proof that all exchanges of two state variables can be simulated by the iterated application of XOR-transformations.

3.3.2 Bounded Size Alteration

Let ρ be the XOR-transformation $q_i \mapsto q_i \oplus q_j$. Then the inverse transformation is defined by

$$\rho^{-1} : q_i \mapsto q_i \oplus q_j,$$

i.e. we have $\rho = \rho^{-1}$. The effect of $\rho(\delta(\rho^{-1}(\cdot)))$ in Equation 3.1 can be split into two parts:

1. Substitute the current-state variable x_i by $x_i \oplus x_j$.

2. Replace the function δ_i by $\delta_i \oplus \delta_j$.

It does not matter which of these two steps is executed first.

Lemma 3.9. *Let $P_1, \ldots, P_n$ be the OBDDs for $\delta_1, \ldots, \delta_n$ and $P'_1, \ldots, P'_n$ be the OBDDs after the application of the XOR-transformation $q_i \mapsto q_i \oplus q_j$. The following holds:*

$$\begin{array}{rcccll} \Omega(size(P_i)^{1/2}/size(P_j)^{1/2}) & \leq & size(P'_i) & \leq & \mathcal{O}(size(P_i)^2 \cdot size(P_j)^2), & \\ \Omega(size(P_k)^{1/2}) & \leq & size(P'_k) & \leq & \mathcal{O}(size(P_k)^2), & k \neq i. \end{array}$$

The upper bound immediately follows from the facts that the substitution of an OBDD P_2 into one variable of an OBDD P_1 leads to an OBDD of size at most $\mathcal{O}(\text{size}(P_1)^2 \cdot \text{size}(P_2))$, and that the operation $P_1 \oplus P_2$ leads to an OBDD of size at most $\mathcal{O}(\text{size}(P_1) \cdot \text{size}(P_2))$. For the lower bounds it suffices to observe $\rho = \rho^{-1}$.

In case of the transition relation both the current-state variables and the next-state variables have to be substituted. This leads to the result

$$\Omega(\text{size}(P)^{1/4}) \leq \text{size}(P') \leq \mathcal{O}(\text{size}(P)^4),$$

where P and P' are the original and the re-encoded OBDD for the transition relation, respectively.

3.3.3 Stronger Bounds

For a more refined analysis of the XOR-transformation we use the following theorem from [SW93]. In particular, we will refine the analysis for the substitution of a variable x_i by $x_i \oplus x_j$ in an OBDD.

Theorem 3.10. *The reduced OBDD representing f with variable order $x_1, \ldots, x_n$ contains as many x_i-nodes as there are different functions f^S, $S \subset \{1, \ldots, i-1\}$, depending essentially on x_i (i.e. $f^S_{x_i=0} \neq f^S_{x_i=1}$). Here $f^S = f_{x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$ where $a_j = 0$ if $j \notin S$, and $a_j = 1$ otherwise.* □

Let s_k be the number of nodes labeled by x_k in the OBDD P and s^*_k be the number of nodes labeled by x_k in the OBDD P' which is the result of the transformation.

Theorem 3.11. *The size of an OBDD w.r.t. the variable order $x_1, \ldots, x_n$ after the application of the substitution $x_i \mapsto x_i \oplus x_j$ is bounded from above by*

$$2(s_1 + \ldots + s_{i-1}) + 1 + s_{i+1} + \ldots + s_n + 2 \quad \textit{in case } j < i$$

and by

$$s_1 + \ldots + s_i + (s_{i+1} + \ldots + s_{j-1} + s_{j+1} + \ldots + s_n + 2)^2 \quad \textit{in case } i < j.$$

The proof of this theorem can be found at the end of the chapter, in section 3.7. It applies ideas from [BLW95], in which local changes in the variable order are analyzed.

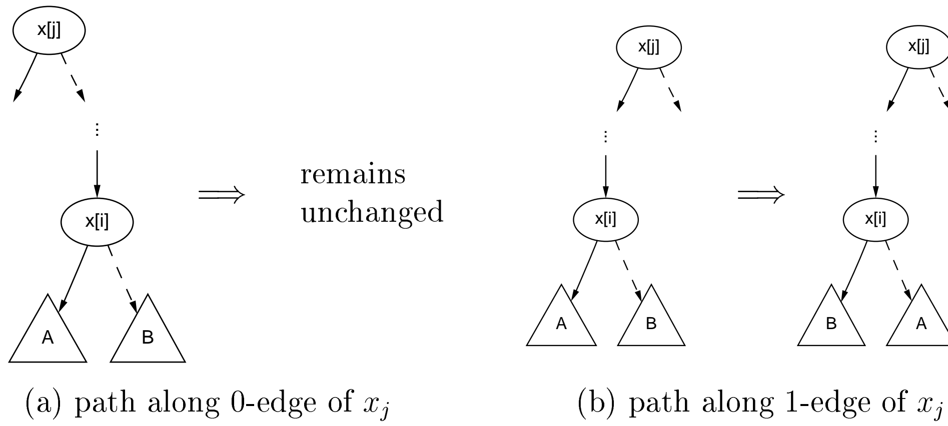
Corollary 3.12. *Let P be an OBDD and P' the resulting OBDD after the substitution $x_i \mapsto x_i \oplus x_j$. Then*

$$\begin{array}{lllll} size(P)/2 & \leq & size(P') & \leq & 2\ size(P) & \text{if } x_j < x_i, \\ size(P)^{1/2} & \leq & size(P') & \leq & size(P)^2 & \text{if } x_i < x_j. \end{array}$$

□

The analogy between the behavior of the XOR-transformation and the local changes in the variable order recommends to use XOR-transformations for the optimization of OBDD-sizes.

The XOR-transformation $x_i \mapsto x_i \oplus x_j$ for $j < i$ and $j+1 \leq k \leq i$ can be visualized as shown in Figure 3.3. Let A and B be the two sub-OBDDs whose roots are the children of an x_i-node. Consider a path from x_j to x_i. If this path contains the 0-edge of x_j, the subgraph rooted in x_i remains unchanged. If instead the path contains the 1-edge of x_j, the 0- and the 1-successor of the x_i-node are exchanged. This modification can prevent subgraph-isomorphisms in the new sub-OBDDs which are rooted in an x_k-node, $j+1 \leq k \leq i$.



(a) path along 0-edge of x_j (b) path along 1-edge of x_j

Figure 3.3: Mutation $x_i \mapsto x_i \oplus x_j$ for $x_j < x_i$

3.3.4 General Two-Bit Re-encodings

The effect of each two-bit re-encoding can be split into the two-parts "Substitute the two variables x_i and x_j by some functions" and "replace the two functions δ_i and δ_j by some functions". The variable substitution has an impact on all functions which depend essentially on x_i or x_j, whereas the function replacement only affects the functions δ_i and δ_j.

Figure 3.4 shows that all re-encodings which are induced by the $4! = 24$ bijective functions $f : \mathbb{B}^2 \to \mathbb{B}^2$ can be obtained by a combination of maximal one XOR-transformation, an exchange variable transformation and the identity. Hence, beside the exchange variable transformation merely XOR-transformations are needed to produce all two-bit re-encodings. We write a two-bit re-encoding which is induced by f as

$$\rho_{i,j}^f : (x_1, \ldots, x_n) \mapsto (x_1, \ldots, x_{i-1}, f_1(x_i, x_j), x_{i+1}, \ldots, x_{j-1}, f_2(x_i, x_j), x_{j+1}, \ldots, x_n)$$

The substitution $x_i \mapsto \overline{x_i}$ does not affect the size of the OBDD. As $\overline{x_i \oplus x_j} = \overline{x_i} \oplus x_j = x_i \oplus \overline{x_j}$, each of the above 24 transformations has the same effect w.r.t. the OBDD-size as a combination of the exchange variables transformation, the XOR-transformation and the identity operation. Moreover, for each of the 24 transformations, a combination of at most *two* of the "basis" transformations suffices.

We remark that in connection with decomposition types of decision diagrams similar completeness results have been established in [BD95].

	f(00)	f(01)	f(10)	f(11)	function
1	00	01	10	11	$(x_1, x_2) = \mathrm{id}$
2	00	01	11	10	$(x_1, x_1 \oplus x_2)$
3	00	10	01	11	(x_2, x_1)
4	00	10	11	01	$(x_1 \oplus x_2, x_1)$
5	00	11	01	10	$(x_2, x_1 \oplus x_2)$
6	00	11	10	01	$(x_1 \oplus x_2, x_2)$
7	01	00	10	11	$(x_1, \overline{x_1 \oplus x_2})$
8	01	00	11	10	$(x_1, \overline{x_2})$
9	01	10	00	11	$(x_2, \overline{x_1 \oplus x_2})$
10	01	10	11	00	$(x_1 \oplus x_2, \overline{x_2})$
11	01	11	00	10	$(x_2, \overline{x_1})$
12	01	11	10	00	$(x_1 \oplus x_2, \overline{x_1})$
13	10	00	01	11	$(\overline{x_1 \oplus x_2}, x_1)$
14	10	00	11	01	$(\overline{x_2}, x_1)$
15	10	01	00	11	$(\overline{x_1 \oplus x_2}, x_2)$
16	10	01	11	00	$(\overline{x_2}, x_1 \oplus x_2)$
17	10	11	00	01	$(\overline{x_1}, x_2)$
18	10	11	01	00	$(\overline{x_1}, x_1 \oplus x_2)$
19	11	00	01	10	$(\overline{x_1 \oplus x_2}, \overline{x_2})$
20	11	00	10	01	$(\overline{x_2}, \overline{x_1 \oplus x_2})$
21	11	01	00	10	$(\overline{x_1 \oplus x_2}, x_1)$
22	11	01	10	00	$(\overline{x_2}, \overline{x_1})$
23	11	10	00	01	$(\overline{x_1}, \overline{x_1 \oplus x_2})$
24	11	10	01	00	$(\overline{x_1}, \overline{x_2})$

Figure 3.4: General two-bit re-encodings

3.3.5 Implementation Aspects

In this section we will describe how to implement the XOR-substitution $x_i \mapsto x_i \oplus x_j$ efficiently. Our starting point is the consideration of local changes in the variable order. In order to modify the variable order of OBDDs we iterate exchanging variables in adjacent levels. Since an exchange of adjacent variables is a local operation consisting only of the relinking of nodes in these two levels, this can be done efficiently as shown in Figure 3.5. In order to move a variable x_i behind an arbitrary variable x_j in the order, the exchanges of adjacent variables are iterated.

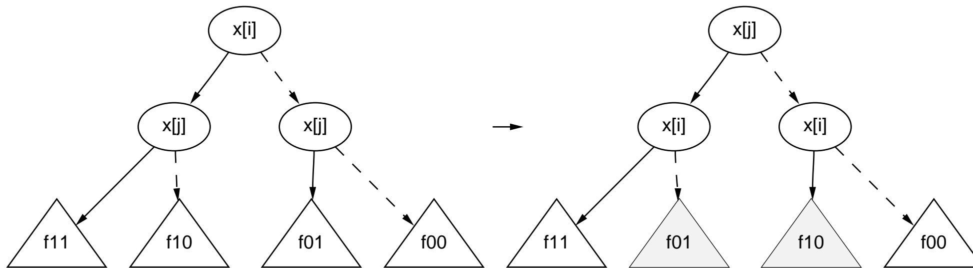


Figure 3.5: Swapping two variables x_i, x_j in the order

In case of the XOR-operation and adjacent variables x_i and x_j, we can proceed analogous to the level exchange. Figure 3.6 shows the case where x_j is the direct

successor of x_i in the order. The case where x_i is the direct successor of x_j in the order works analogously. If x_i and x_j are not adjacent, it would of course be helpful if we could simulate the substitution $x_i \mapsto x_i \oplus x_j$ by a sequence like $x_i \mapsto x_i \oplus x_{i+1}$, $x_i \mapsto x_i \oplus x_{i+2}$, ..., $x_i \mapsto x_i \oplus x_j$. However, this straightforward idea does not work, as this would require operations in the intermediate steps which influence more than two adjacent levels.

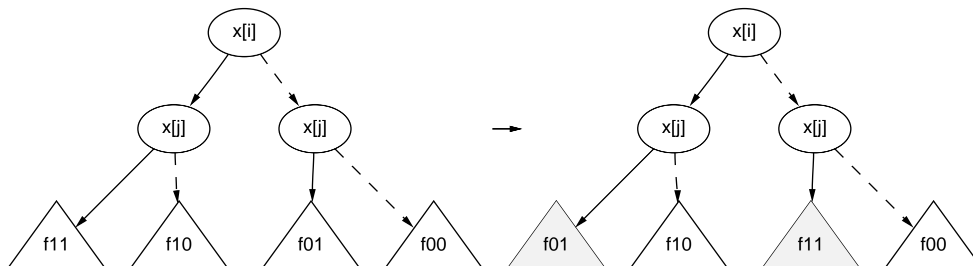


Figure 3.6: Performing $x_i \mapsto x_i \oplus x_j$ for two neighboring variables x_i, x_j

A method that works and is only slightly more expensive than the exchange of two non-adjacent variables is the following: First, shift the variable x_i to a new position in the order which is adjacent to x_j. Then perform the XOR-operation, and then shift the variable x_i back to its old position. This technique retains the locality of the operation, as only nodes with a label x_k are influenced whose position in the order is between x_i and x_j.

3.4 Who Profits From XOR ?

In principle, the applicability of the XOR-transformation is not restricted to the use of OBDDs as underlying data structure. It can also be applied to other data structures for Boolean functions. However, the strong relationship between the XOR-transformation and local changes in the variable order like in the case of OBDDs does not always transfer to other representations. We will demonstrate this effect on OFDDs.

Ordered functional decision diagrams (OFDDs) are a modification of OBDDs which have been described in Section 1.2.1. Each node v with label x_i in an OBDD represents a Shannon decomposition

$$f = x_i \cdot g + \overline{x_i} \cdot h,$$

whereas each node v with label x_i in an OFDD represents a Reed-Muller decomposition

$$f = g \oplus x_i \cdot h.$$

In both decompositions the functions g and h are independent of x_i and are the functions which are represented by the subgraphs rooted in the two successor nodes of v.

It has been shown in [BLW95] that local changes in the variable order have the same effect on OFDDs like on OBDDs. In particular, the exchange of two variables x_i and x_j in the order only affects the nodes of an OBDD resp. OFDD which are labeled by a variable whose position in the order is between x_i and x_j. These observations justify the notions of *local* changes. From the proof of Theorem 3.11 it follows that the XOR-transformation for OBDDs has also this pleasant local property. However, in spite of the fact that the Reed-Muller decomposition seems to operate well with XOR-transformations, the substitution $x_i \mapsto x_i \oplus x_j$ for OFDDs does not have the local property like in the case of OBDDs.

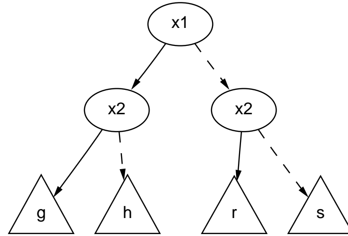


Figure 3.7: OFDD for $f = g \oplus x_2 \cdot h \oplus x_1 \cdot r \oplus x_1 \cdot x_2 \cdot s$

To prove this statement, consider the OFDD in Figure 3.7 which represents the function

$$f = g \oplus x_2 \cdot h \oplus x_1 \cdot r \oplus x_1 \cdot x_2 \cdot s$$

for some functions g, h, r, s independent of x_1, x_2. The function $f'(x_1, \ldots, x_n) = f(x_1 \oplus x_2, x_2, \ldots, x_n)$ is

$$\begin{aligned} f' &= g \oplus x_2 \cdot h \oplus (x_1 \oplus x_2) \cdot r \oplus (x_1 \oplus x_2) \cdot x_2 \cdot s \\ &= g \oplus x_2 \cdot (h \oplus r \oplus s) \oplus x_1 \cdot r \oplus x_1 \cdot x_2 \cdot s. \end{aligned}$$

Hence, there must exist a node in the OFDD for f' representing the function $h \oplus r \oplus s$. As h, r and s are arbitrary functions, the substitution operation $x_i \mapsto x_i \oplus x_j$ does not have the local property. However, as the $\oplus$-operation is a polynomial operation on OFDDs, the result of the substitution remains polynomial.

A tight relationship between our re-encoding techniques and specific OBDD-variants is the following: In a more general setting, the concept of domain transformations has been proposed for the manipulation of switching functions. This concept has been described in Section 1.3.

It turns out that every re-encoding function ρ defines a transformation within the TBDD-concept. However, the OBDDs for the next-state functions of a re-encoded machine with re-encoding function ρ are not isomorphic to the ρ-TBDDs, as in the OBDDs of the re-encoded machine, the transformation ρ^{-1} is also involved (see Equation (3.1) in section 3.1).

3.5 Experimental Results

In this section we present some experimental results on the extended optimization techniques for OBDDs. We built up some routines on top of the OBDD-package of D. Long [Lon92] and used the ISCAS89 benchmark circuits s1423, s5378, s9234 which have a large number of state bits and have also formed the set of examples in [RS95]. Each optimization run consists of three phases: First, we applied Rudell's Sifting algorithm [Rud93] for finding a good variable order. Then some minimization based on XOR-transformations is performed. Finally, Sifting is applied once more to re-establish a suitable variable order. The table shows the obtained shared OBDD-sizes of the next-state functions in comparison to the sizes that were obtained *without* the minimization by XOR-transformations.

Circuit	# state bits	without XOR	with XOR
s1423	74	3054	2853
s5378	121	1801	1517
s9234	228	4621	4359

The minimization based on XOR-transformation works as follows: In a preprocessing step we compute promising pairs (i, j) for an XOR-transformation. The heuristic criteria for considering a pair (i, j) as promising are:

1. the next-state functions δ_i and δ_j have a nearly equal support, or
2. the variables x_i and x_j appear in nearly the same functions.

Then, as long as improvements are possible, the best XOR-transformation among these pairs are applied. In order to avoid the expensive computations $\delta_i \oplus \delta_j$, we only

perform this step if the variable substitution $x_i \mapsto x_i \oplus x_j$ yields a good intermediate result.

It must be admitted that in the above experiments, the running times are significantly higher than the running times for pure Sifting. This is due to a non-optimal implementation of the XOR-transformation within the OBDD-package of D. Long and due to the large number of performed XOR-transformations. However, big improvements in the implementation techniques are the focus of the next chapter.

3.6 Conclusion

We have proposed and analyzed new re-encoding techniques for minimizing OBDDs. In particular, we have proposed the XOR-transformation and shown that this transformation is in fact the only new transformation on two state variables. This transformation can in certain cases significantly enrich the set of basic operations for the optimization of OBDDs. Some experimental results have shown the validity of our approach.

3.7 Two Proofs

Proof of Theorem 3.1

Proof. Let $K = \lfloor 2^n/n \rfloor$. Suppose there are c_i variables with label x_i in the OBDD, and let $c_{n+1} = K - \sum_{i=1}^{n} c_i$. c_{n+1} serves to cover the case $\sum_{i=1}^{n} c_i < K$. Since each c_i, $1 \leq i \leq n+1$, is a nonnegative integer, the equation $\sum_{i=1}^{n+1} c_i = K$ has exactly $\binom{n+K}{K}$ solutions. Suppose that all the vertices are ordered by increasing positions in the variable order, then each of the two edges of a vertex can only connect to its successors, including the nonterminal vertices 0 and 1. Hence there are at most $(K+1)^2\, K^2\, (K-1)^2 \cdot \ldots \cdot 2^2 = ((K+1)!)^2$ ways to place the edges. Since there are $n!$ different variable orders, there are at most

$$n! \binom{n+K}{K} ((K+1)!)^2$$

shared OBDDs with at most K vertices for their optimal variable order. Due to the cyclic symmetry of the autonomous counter it follows

$$\begin{aligned} e(n) &\leq 2^n\, n!\, (n+K)!\, ((K+1)!)^2 \;\leq\; 2^n\, (2n+3K+2)! \\ &\leq 2^n (2n + 3 \cdot 2^n/n + 2)! \;\leq\; 2^n (4 \cdot 2^n/n)! \quad \text{for sufficiently large } n, \end{aligned}$$

and therefore

$$\frac{e(n)}{a(n)} \to 0 \quad \text{as} \quad n \to \infty.$$

□

Proof of Theorem 3.11

Proof. Let $f(x_1, \ldots, x_n)$ be a switching function,

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

and P, P' the OBDDs of f resp. f'.

CASE 1: $j < i$. As there is a 1-1-correspondence between (x_i, x_j) and $(x_i \oplus x_j, x_j)$, Theorem 3.10 implies directly that $s_k^* = s_k$ for $k \leq j-1$ or $k \geq i+1$. In particular, a subfunction $f_{x_1=a_1,\ldots,x_{k-1}=a_{k-1}}$ depends essentially on x_k if and only if $f'_{x_1=a_1,\ldots,x_{k-1}=a_{k-1}}$ depends essentially on x_k for $k \leq j-1$.

Next we show $s_j^* \leq s_1 + \ldots + s_{j-1} + 1$. It can happen that a subfunction $f_{x_1=a_1,\ldots,x_{j-1}=a_{j-1}}$ does not depend essentially on x_j but the subfunction $f'_{x_1=a_1,\ldots,x_{j-1}=a_{j-1}}$ does. Due to this effect s_j^* can become large and there is no upper bound for s_j^* depending only on s_j. Each x_j-node in P' has to be reached from at least one node labeled by x_k, $k \leq j-1$, or the source is the only node labeled by x_j. There are at most

$$\begin{aligned} 2s_1^* + \ldots + 2s_{j-1}^* - s_1^* - s_2^* - \ldots - s_{j-1}^* + 1 &\leq s_1^* + s_2^* + \ldots + s_{j-1}^* + 1 \\ &= s_1 + s_2 + \ldots + s_{j-1} + 1 \end{aligned}$$

edges starting in a node labeled by x_k, $k \leq j-1$, and leading to a node labeled by x_j. Hence, $s_j^* \leq s_1 + \ldots + s_{j-1} + 1$.

Now we prove $s_k^* \leq 2s_k$, $j+1 \leq k \leq i$. Again, we use that $f_{x_1=a_1,\ldots,x_{k-1}=a_{k-1}}$ depends essentially on x_k if and only if $f'_{x_1=a_1,\ldots,x_{k-1}=a_{k-1}}$ does. Further, each subfunction $f'_{x_1=a_1,\ldots,x_{k-1}=a_{k-1}}(x_k, \ldots, x_n)$ is identical to

$$\begin{aligned} & f_{x_1=a_1,\ldots,x_{k-1}=a_{k-1}}(x_k, \ldots, x_i, \ldots, x_n) \quad \text{(in case } a_j = 0) \\ \text{or to} \quad & f_{x_1=a_1,\ldots,x_{k-1}=a_{k-1}}(x_k, \ldots, \overline{x_i}, \ldots, x_n) \quad \text{(in case } a_j = 1). \end{aligned}$$

These two subfunctions do not have to be necessarily different. If follows $s_k^* \leq 2s_k$.

CASE 2: $i < j$. For $k \leq i$ or $k \geq j+1$ we have $s_k^* = s_k$ due to Theorem 3.10. In particular, this relation holds for the index i, as $f_{x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$ depends essentially on x_i if and only if $f'_{x_1=a_1,\ldots,x_{i-1}=a_{i-1}}$ does.

For $i+1 \leq k \leq j-1$ the number s_k^* is equal to the number of functions f'^S, $S \subset \{1,\ldots,k-1\}$, depending essentially on x_k. By Shannon's expansion it holds

$$f'^S = \overline{x_j} f'^S_{x_j=0} + x_j f'^S_{x_j=1}.$$

The function f'^S depends essentially on x_k if and only if at least one of the two subfunctions $f'^S_{x_j=0}$ and $f'^S_{x_j=1}$ depends essentially on x_k. By computing upper bounds on the number of different subfunctions $f'^S_{x_j=0}$ and $f'^S_{x_j=1}$, we can derive an upper bound for s_k^*. The number of different subfunctions $f'^S_{x_j=a}$, $a \in \{0,1\}$, is equal to the number of different subfunctions $f^S_{x_j=a'}$, $a' \in \{0,1\}$, (as both x_i and x_j are specified and due to the 1-1-correspondence between (x_i, x_j) and $(x_i \oplus x_j, x_j)$), and this number is at most the number of different subfunctions f^S. All the subfunctions f^S are represented in the OBDD P. Combining the three possible cases which of the two subfunctions $f'^S_{x_j=0}$ resp. $f'^S_{x_j=1}$ depend on x_k we obtain

$$s_k^* \leq s_k^2 + 2s_k(s_{k+1} + \ldots + s_n + 2).$$

It remains to find an upper bound for s_j^*. Let $S \subset \{1,\ldots,j-1\}$. Then f'^S depends essentially on x_j if and only if $f'^S_{x_j=0}$ and $f'^S_{x_j=1}$ are different. The number of subfunctions $f'^S_{x_j=a}$, $a \in \{0,1\}$, is identical to the number of subfunctions $f^S_{x_j=a'}$, $a' \in \{0,1\}$, as x_i and x_j are specified. The subfunctions $f^S_{x_j=a}$ are represented in the OBDD P. There are at most $s_{j+1}+\ldots+s_n+2$ choices for $f^S_{x_j=0}$ and then at most $s_{j+1}+\ldots+s_n+1$ choices for $f^S_{x_j=1}$. The size of an OBDD after the XOR-transformation $x_i \mapsto x_i \oplus x_j$ with $i<j$ is therefore

$$\begin{aligned}
\text{size}(P') \;\leq\; & \sum_{k=1}^{i} s_k + \sum_{k=i+1}^{j-1} \Big(s_k^2 + 2s_k(s_{k+1} + \ldots + s_n + 2)\Big) \\
& + (s_{j+1} + \ldots + s_n + 2)(s_{j+1} + \ldots + s_n + 1) + \sum_{i=j+1}^{n} s_k + 2 \\
\leq\; & s_1 + \ldots + s_i + (s_{i+1} + \ldots + s_{j-1} + s_{j+1} + \ldots + s_n + 2)^2 .
\end{aligned}$$

□

Chapter 4

Linear Sifting

In this section we will propose new techniques for the optimization of decision diagrams within the context of domain transformations from Section 1.3.

In practice it is necessary to identify a cube transformation that maps the functions of interest onto functions with simple representations, and whose representation is also compact. Since the exhaustive search of all possible automorphisms is prohibitive even for small problems, efficient heuristic algorithms are sought for the automatic identification of appropriate transformations.

In the previous chapter we have studied linear transformations, concluding that they form a class very well suited to the implementation of efficient algorithms. In this chapter we present one such algorithm in the framework of domain transformations and report on its experimental evaluation. Our algorithm combines linear transformations with variable reordering techniques. In contrast to the static concept of transformations based on complete types (see Section 1.3) linear transformations can be applied dynamically.

Linear transformations replace one variable, x_i, with a linear combination of variables. A linear combination is obtained by taking the exclusive or of the arguments or its complement. We restrict ourselves to transformations of the form

$$x_i \mapsto x_i \equiv x_j, \tag{4.1}$$

where $\equiv$ indicates the equivalence function, that is, the complement of the exclusive or.

Before we examine the reasons for this choice and enter into the details of the algorithm, we briefly review two applications that demonstrate the usefulness of this kind of transformation.

In [HS96] a technique based on spectral analysis is presented, whose purpose is to decompose a circuit into the cascade of two subcircuits: The first is a linear block, composed of exclusive or gates fed by the primary inputs of the overall circuit. Such a decomposition can simplify the synthesis task—sometimes dramatically. The realization of this application will be presented in Chapter 5.

In [CCC+92] it is shown that the reachability analysis of the product of two very similar sequential machines can be made more efficient if a transformation essentially identical to (4.1) is applied to the state variables of one of the two machines. A program that can apply linear transformations, can therefore automatically apply the technique of [CCC+92] and, in addition, take advantage of similarities between subsets of the two machines or even between different parts of one machine. In other words, the technique we present in this chapter can basically be used to automatically re-encode finite state systems to make reachability analysis more efficient.

The rest of this chapter is organized as follows: In Section 4.1 we recall some definitions relevant for the current chapter. In Section 4.2 we discuss the Linear Sifting algorithm. In Section 4.3 we report on our experiments. Section 4.4 concludes the chapter.

4.1 Preliminaries

Complemented edges. The implementation of the optimization algorithm will be described for decision diagrams with complemented edges as introduced in Section 1.1.2.

The Sifting algorithm. As shown in Section 1.1.3 the size of BDDs depends, sometimes critically, on the variable order π. Various algorithms have been proposed in recent years to determine good variable orders. The one that is most important to our treatment is Rudell's *Sifting* algorithm that has been described in Section 1.1.5. The effectiveness of Sifting stems from its ability to move a variable to any position in the order in a short time. Its time efficiency relies on the ability to quickly swap adjacent variables. It is indeed possible to perform such a swap by accessing only the nodes labeled by the two variables being exchanged.

Linear transformations. An automorphism τ on $\mathbb{B}^n$ (a bijective mapping from $\mathbb{B}^n$ onto itself) induces an automorphism on $\mathbb{B}_n$, ϕ, with $\phi(f)(a) = f(\tau(a))$. This function automorphism forms the basis of the transformation concept from Section 1.3. The function automorphism is completely characterized by the images of the

projection functions, $x_1, \ldots, x_n$. We define linear transformations as those automorphisms in which the images of the projection functions are obtained by composing a sequence of elementary transformations of the form $x_i \mapsto x_i \equiv x_j$.

It is possible – and convenient in our context – to regard variable reordering as an automorphism. Quite naturally, the exchange of variables x_i and x_j in the order corresponds to the application of the transformation $x_i \mapsto x_j, x_j \mapsto x_i$.

The number of linear transformations is shown in Lemma 3.8 to be

$$\prod_{i=0}^{n-1} (2^n - 2^i).$$

For large n, the number of linear transformations is much smaller than the number of all possible automorphisms ($2^n!$), but much larger than the number of transformations corresponding to variable permutations ($n!$).

On the other hand, the more limited is a set of transformations, the easier it is to represent it. Indeed, variable permutations require $O(n)$ space; linear transformations take $O(n^2)$ space; while general automorphisms require exponential space in the worst case. (In practice, the space requirements of linear transformations are quite far from the worst-case quadratic bound.)

Linear transformations therefore considerably enlarge the search space for the BDD optimization problem, when compared to variable reordering; still they enjoy efficient manipulation. Moreover, in the context of the mentioned applications, linear transformations are even more highly desired because they form a key step in the solution of the problem.

4.2 Linear Sifting

The Sifting algorithm (see Section 1.1.5) proceeds by a sequence of pairwise exchanges of variables. Suppose that variables x_i and x_j are to be swapped, and that x_i immediately precedes x_j before the swap. Then the effect of the exchange on each node labeled x_i can be easily seen by applying Shannon's expansion theorem w.r.t. both x_i and x_j. Assuming f is the function of a node labeled x_i, we have:

$$f = x_i x_j f_{11} + x_i \overline{x_j} f_{10} + \overline{x_i} x_j f_{01} + \overline{x_i}\ \overline{x_j} f_{00}. \tag{4.2}$$

Exchanging x_i and x_j and rearranging terms yields in terms of the automorphism notation:

$$\phi(f) = x_i x_j f_{11} + x_i \overline{x_j} f_{01} + \overline{x_i} x_j f_{10} + \overline{x_i}\ \overline{x_j} f_{00}. \tag{4.3}$$

In words, the effect of a swap is to interchange f_{10} and f_{01}. If we repeat the same process for the application of the elementary linear transformation from Equation 4.1, we obtain:

$$\phi(f) = x_i x_j f_{11} + x_i \overline{x_j} f_{00} + \overline{x_i} x_j f_{01} + \overline{x_i}\ \overline{x_j} f_{10}. \qquad (4.4)$$

A comparison of Equations 4.3 and 4.4 shows that the only difference is that f_{00} is involved in the interchange with f_{10}, instead of f_{01}, see Figure 4.1. The fundamental techniques applied to the efficient swapping of two variables can be used also for their linear combination.

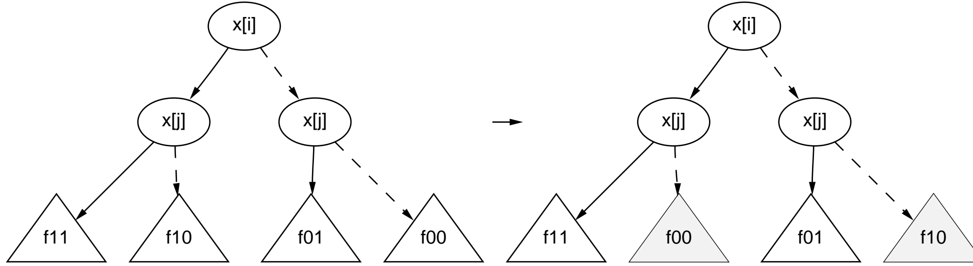


Figure 4.1: $x_i \mapsto x_i \equiv x_j$ via local interchange

Moreover, examination of Equations 4.2-4.4 shows that the application of both swapping and linear combination produces, if the initial state is included, all three ways of partitioning $\{f_{11}, f_{10}, f_{01}, f_{00}\}$ into two subsets of equal size. Therefore, by combining swapping and linear transformations it is possible to reduce the size of the BDD in more cases than by swapping only. This observation forms the basis for our Linear Sifting algorithm, which proceeds as follows.

Each variable is considered in turn, and, as in Sifting, it is moved up and down in the order. Let x_i be the chosen variable, and let x_j be the variable immediately following it in the order. One basic step of Linear Sifting consists of the following three phases:

1. Variables x_i and x_j are swapped; let the size of the BDD after the swap be s_1.

2. The linear transformation $x_j \mapsto x_i \equiv x_j$ is applied; let the resulting size of the BDD be s_2.

3. If $s_1 \leq s_2$ then the linear transformation is undone. This is obtained by simply applying the transformation again, since it is its own inverse.

The net effect of the three-phase procedure is that x_i is moved one position onward in the order, and possibly linearly combined with x_j. Conversely, if x_j is the variable being moved, it is first swapped and then x_i is combined with it. ($x_j \mapsto x_i \equiv x_j$.)

The usual formulation of linear transformations is in terms of exclusive or, rather than its complement. We use the equivalence function ($\equiv$) because our BDDs use complemented edges; hence, we cannot swap f_{11} with one of $\{f_{10}, f_{01}, f_{00}\}$, all of which could be complemented. Since the 1-edges cannot be complemented, we would have to change the complementation of edges into the node labeled (initially) x_i. This would make the operation non-local, which is highly undesirable.

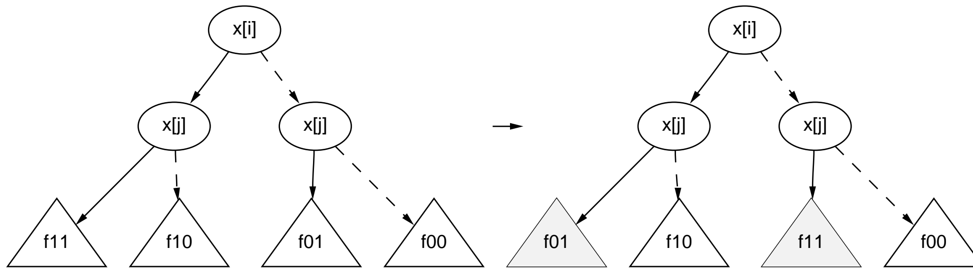


Figure 4.2: $x_i \mapsto x_i \oplus x_j$ via local interchange

Figure 4.2 recalls the realization of the exclusive or operation, and Figure 4.3 illustrates an example in which the use of complemented edges causes problems with the locality of the operation. We start from a sub-OBDD whose root is labeled by x_i. The 0-edge of this node is complemented and points to a sub-OBDD g whose root is not labeled by x_j. The exclusive or operation now causes the 1-edge of the left x_j-node to become complemented which is indicated by a bold line. To re-establish canonicity, the complement bit has to be moved up *at least* to a position above the node labeled by x_i. However, in the worst case, the adjustment of the complement bits can lead up to the top of the whole OBDD.

Nothing is lost by using the equivalence function instead of the exclusive or. Figure 4.1 shows the realization of the equivalence operation. It has been shown in the previous chapter that the combination of variable exchange and the transformation of Equation 4.1 subsumes all invertible transformations that affect only two variables.

An algorithm that reorders the variables of a BDD normally keeps track of the permutation produced. Likewise, our algorithm for Linear Sifting has to maintain the images of the projection functions (the transformed $x_1, \ldots, x_n$). The most expedient way to do that is to maintain the projection functions in the BDD package. Application of the transformations to the BDDs of the projection functions yield

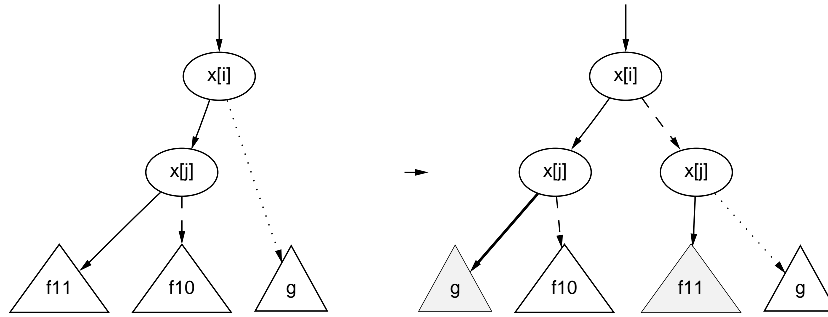


Figure 4.3: The non-locality of exclusive or in the presence of complemented edges

the BDDs of their transforms, which can be used to build functions directly in the transformed space. This minimizes the impact of Linear Sifting on the applications that use it. As an example, two combinational circuits may be tested for equivalence by building BDDs for both. By using the transformed BDDs of the primary inputs one guarantees that the two circuits are equivalent if and only if they have the same BDD. Functions like *Apply* [Bry86] or *ITE* [BRB90] are oblivious of Linear Sifting.

An important device to guarantee high speed in Sifting is the *interaction matrix* already mentioned in Section 1.1.5. In standard Sifting, this matrix tells the swapping procedure whether two variables appear together in the support of some function; in other words, whether there exists a root of the BDD from which one can reach nodes labeled by both variables. If two variables do not interact, the swap can be performed in constant time. The interaction matrix is initialized before reordering starts, and, when swapping is the only allowed transformation, it remains invariant throughout the execution of the Sifting algorithm.

The interaction matrix is effective also for Linear Sifting. On the one hand, it allows cheap swaps for non-interacting variables. On the other hand, it is used to avoid linear combinations of variables that do not interact. In contrast to swapping operations, however, linearly combining variables may cause variables that did not interact to become interacting. Consider the case of two functions of three variables:

$$\begin{aligned} f &= x \oplus y \\ g &= y \equiv z. \end{aligned}$$

Variables x and z do not interact, because neither f nor g has both of them in its support. However, the transformation $y \mapsto x \equiv y$ yields

$$\begin{aligned} \phi(f) &= y' \\ \phi(g) &= x \oplus y \oplus z, \end{aligned}$$

where all variables interact. Therefore, when two variables are combined, the Linear Sifting algorithm updates the interaction matrix accordingly.

Linear Sifting is slower than normal Sifting in most cases, because the cost of a linear transformation is comparable to the cost of a variable swap. If we assume that the graphs manipulated by the two algorithms are approximately of the same size, then Linear Sifting will be approximately three times slower than standard Sifting. This analysis is approximate and does not take into account, for instance, the effects of the linear transformations on the interaction matrix. However, it is confirmed by our experiments in most cases.

We close this section by recalling that several variants of decision diagrams exist. We have discussed Linear Sifting for BDDs, but the algorithm applies with null or minor modifications to all the diagrams that use Shannon's expansion (as opposed to the Reed-Muller expansion from Section 1.2.1 or moment [BC95] expansions). In the case of zero-suppressed BDDs, the transformation $x_i \mapsto x_i \equiv x_j$ is not very effective in changing the size of the diagrams: A node is suppressed if its 1-edge points to the 0 sink node. This condition is not affected by swapping f_{10} and f_{00}. The graph size may still change as a result of increased or decreased sharing of nodes, but it is advisable to use $x_i \mapsto x_i \oplus x_j$ instead. The exclusive or transformation is also preferable for Edge-Valued BDDs [LS92, LPV94]: In the typical case when the weight of the 0-edge is constrained to be 0, using $x_i \mapsto x_i \equiv x_j$ would lead to a non-local transformation.

4.3 Experimental Results

We have implemented Linear Sifting in the CUDD package [Som96] and applied it to the BDDs of a large collection of circuits. The collection includes well-known benchmark circuits [BF85, BBK89, Yan91], as well as several large industrial examples. Figures 4.4 and 4.5 (Figure 4.5 continues Figure 4.4) summarize our experiments, in which we compare the best results we have obtained by applying variable reordering (specifically, different variants of Sifting [PS95]) to the best results obtained by combining normal Sifting and Linear Sifting.

Two main strategies were used for Linear Sifting: One was to build the BDDs with a known good order and then applying Linear Sifting. The second strategy consisted of applying Linear Sifting also while building the BDDs, whenever their sizes doubled. The former strategy gave better results in more cases, but the latter produced the most striking results.

The reduction in size goes from 0% in 19 cases out of 77, to 98% for C499 and C1355. The cases of 0% improvement are mostly for small circuits. We left these small circuits in our test suite in spite of their limited significance, because we

insisted on using the entire validation test suite of CUDD, so that no bias in favor or against Linear Sifting were introduced by the selection process.

The average improvement, measured by the geometric mean of the individual improvements, is 22%. The total number of nodes is decreased by 13% by the application of Linear Sifting. The CPU times should be considered as merely indicative and are shown for the sole purpose of supporting the claim that Linear Sifting is time efficient in practice. However, since different circuits give their best results when subjected to different methods, it is impossible to compare individual CPU times. More significant is the fact that for most cases the number of linear transformations applied is comparable to the number of swaps performed by the algorithm. When Sifting and Linear Sifting produce graphs of almost equal sizes, Sifting is about twice as fast. Improvements to the implementation of Linear Sifting should close this gap somewhat.

Though not all circuits equally benefit from the application of Linear Sifting (e.g., multipliers), in some cases like C499 or C1908 the reduction is such as to make a dramatic difference in the subsequent processing of the circuit.

4.4 Conclusion

The optimization of decision diagrams has applications in both verification and synthesis. Whenever the cost of creating and optimizing the diagram is a minor fraction of the total elaboration cost; or whenever the optimality of the diagram may have a direct impact on the quality of the solution, the extra expense of time required by a more sophisticated algorithm may well be justified. We have presented in this chapter Linear Sifting, which extends Rudell's variable reordering algorithm by combining it with linear transformations. We have shown that Linear Sifting reduces the combined size of the BDD and the transformation in most cases, with a geometric mean of 22% over a large set of experiments. Several applications can benefit from a more compact representation that preserves the ease of use and canonical properties of plain decision diagrams. One application in logic decomposition for synthesis will be investigated in the next chapter.

The main open question is: In how far can the additional optimization potential of Linear Sifting be transferred to the reachability analysis of finite state machines ? We have outlined one interesting connection between our work and the verification of similar finite state machines of [CCC+92]. In order to profit from Linear Sifting within the traversal of a single finite state machine one has to realize an *efficient* adaption of the existing techniques and heuristics for image computation.

Name	In	Ordering	Linear Sifting		
		Size	Size	Ratio	Time
9symml	9	25	25	1.00	0.1
alu4	14	350	350	1.00	0.1
t481	16	21	21	1.00	0.3
vda	17	485	472	0.97	0.2
cordic	23	42	29	0.69	0.1
s499	23	351	319	0.91	0.4
s344	24	104	83	0.80	0.1
ttt2	24	109	109	1.00	0.1
s1196	32	599	599	1.00	0.2
C1908	33	5708	1822	0.32	4.2
adder16	33	82	49	0.60	0.1
s635	34	128	128	1.00	0.1
C432	36	1209	1209	1.00	0.3
mm9b	38	1527	1476	0.97	1.0
too_large	38	319	319	1.00	0.5
mm9a	39	1112	972	0.87	0.3
C1355	41	25866	484	0.02	5.1
C499	41	25866	520	0.02	3.0
k2	45	1246	1235	0.99	1.2
s967	45	389	380	0.98	0.2
C3540	50	23828	23823	1.00	31.0
s1269	55	1810	1792	0.99	0.6
C880	60	4083	4080	1.00	3.8
comp32	64	97	97	1.00	0.1
adder32	65	162	97	0.60	0.4
adsb32	65	345	191	0.55	0.4
s938	66	161	161	1.00	0.3
sbc	68	917	917	1.00	0.5
alu32	73	478	388	0.81	0.4
alu32r	73	478	388	0.81	0.4
dalu	75	689	594	0.86	3.0
s991	84	707	707	1.00	0.6
s1512	86	572	476	0.83	1.4
xi30	90	91	91	1.00	0.3
s1423	91	1796	1477	0.82	3.1
dpath32	93	485	396	0.82	0.5
ellen_c	93	155432	154756	1.00	1203.4
ellen_t	109	442	442	1.00	0.3
M1	118	252153	153006	0.61	774.2

Figure 4.4: Experimental results (first part). The column labeled `In` gives the number of inputs of the circuit. The column labeled `Ordering size` gives the best results we have achieved by variable ordering only. Totals are shown at the bottom of Figure 4.5.

Name	In	Ordering	Linear Sifting		
		Size	Size	Ratio	Time
mm30a	123	11065	9065	0.82	18.6
i3	132	133	133	1.00	0.2
i8	133	1276	1276	1.00	1.0
apex6	135	498	498	1.00	0.4
s3271	142	832	671	0.81	1.7
frg2	143	963	918	0.95	3.9
ind1sn	148	1796	1730	0.96	8.9
s4863	153	64088	61701	0.96	404.4
ind2sn	172	2989	2818	0.94	13.9
ind3sn	174	3201	3071	0.96	19.6
ellen_s	176	135007	134990	1.00	607.3
C5315	178	2104	1808	0.86	9.5
s5378	199	1932	1664	0.86	9.1
C7552	207	3730	1921	0.52	71.3
ellen_l	216	176022	170419	0.97	1010.1
s3384	226	710	597	0.84	0.8
C2670	233	2008	998	0.50	37.2
s9234.1	247	3045	2731	0.90	16.8
comp128	256	385	385	1.00	2.1
des	256	2945	2085	0.71	8.6
add128	257	642	385	0.60	10.4
adsb128	257	1401	767	0.55	9.9
i10	257	20660	19295	0.93	172.9
alu128	265	1918	1540	0.80	6.5
s6669	322	18043	17904	0.99	23.1
cps1364	328	20522	20510	1.00	109.7
OW	333	8842	5470	0.62	42.0
OW2	333	6684	4021	0.60	30.7
clma	415	738	715	0.97	3.4
clmb	415	690	618	0.90	5.2
dsip	452	3033	2594	0.86	14.0
bigkey	486	1595	1406	0.88	2.2
s15850.1	611	10111	8990	0.89	112.5
s13207.1	700	2941	2859	0.97	54.2
trainf	925	127987	124470	0.97	1103.6
s38584.1	1464	13303	12934	0.97	255.9
s38417	1664	266772	266404	1.00	2156.5
s35932	1763	5206	5168	0.99	162.5
Totals	16979	1436081	1250009	0.87	8562.9

Figure 4.5: Experimental results (continued). See Figure 4.4 for the legend.

Chapter 5

Function Decomposition and Technology Mapping

Decomposition techniques date back to the very early days of computer-aided circuit design [Koh78]. By revealing the structural properties of a switching function it is possible to establish representations by means of several simpler functions. Utilizing these functional properties enables us to decompose large and complex circuits into a system of smaller subcircuits which may be readily available and economically maintained.

Recent developments concerning symbolic representations of switching functions by means of decision diagrams have renewed the interest in decomposition techniques [HS96, NJFS96]: By extracting functional properties and decomposing the switching functions the well-known power of symbolic manipulation techniques can be even further extended.

In particular, the recent work [HS96] dealt with the problem that most multi-level synthesis tools like SIS run into severe problems for complex functions. By decomposing the circuit into the cascade of two suitable subcircuits the synthesis task can be simplified. However, efficient algorithms to find such a decomposition are available only for very specific types of decomposition. An attractive candidate is to divide the function f into a linear block σ and a non-linear block f' (see Figure 5.1) such that $f(x) = f'(\sigma(x))$. The main synthesis task now is to find a linear block σ that minimizes the complexity of f. In [HS96] this problem has been tackled by methods of spectral analysis. However, although there is a well-established theory on these techniques [HMM85] the practical implementation of these ideas leads to

very complex algorithms and the improvements are limited. Hence an important question is not only to find better algorithms but also to find algorithms which can be integrated much more easily into existing design systems.

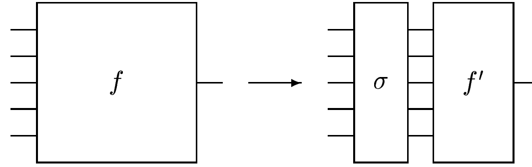


Figure 5.1: Linear decomposition.

In this chapter we propose to use the Linear Sifting algorithm from Chapter 4 for the computation of the linear block σ. This algorithm is an extension of the well-known Sifting algorithm [Rud93], which is currently the state-of-the-art method for finding good variable orders of binary decision diagrams. In the Linear Sifting algorithm, not only swaps of two neighboring variables are applied as elementary operations, but also linear transformations among neighboring variables. This algorithm modifies a given function, but preserves the canonicity in representation and the efficiency of manipulation. We show in what respect Linear Sifting can be used to perform the above mentioned synthesis task and we prove the validity of our approach by experimental results.

The chapter is structured as follows: In Section 5.1 we review the approach of [HS96] based on spectral analysis. In Section 5.2 we present our new approach based on Linear Sifting and contrast it to the existing approach. Experimental results are given in Section 5.3. Finally, in Section 5.4 we precisely analyze some unexpected but important effects that occur in the interplay of adder functions, linear transformations and symbolic synthesis algorithms.

5.1 Synthesis by Spectral Analysis

In this section we revisit the algorithm of [HS96]. This description will form the basis for the development and judgement of our algorithm.

Let us recall that a Boolean function $\sigma : \mathbb{B}^n \to \mathbb{B}^n$ is called linear if there exists an $n \times n$-matrix m_σ with Boolean entries such that $\sigma(x) = m_\sigma \cdot x$ where the matrix product $m_\sigma \cdot x$ is evaluated over the Galois field $GF(2)$. Of course, σ and m_σ can be used interchangeably. Additionally we allow to complement some bits of $\sigma(x)$.

The problem: Given a Boolean function $f \in \mathbb{B}_n$, find a linear transformation σ and a remaining function $f' \in \mathbb{B}_n$ such that the decomposition $f(x) = f'(\sigma(x))$ simplifies the synthesis task and leads to smaller circuits.

The choice of suitable linear transformations is achieved by assigning each Boolean function a complexity measure that is heuristically related to the complexity of the final circuit implementation. The cost of the transformation σ is neglected in this complexity measure on the grounds that the transformation itself is typically much less complex than the remaining function after the transformation. (In our experience this is true in most cases.)

In order to estimate the complexity of the Boolean function f one can use the number of vector pairs (x_1, x_2) which have a Hamming distance $d_H(x_1, x_2)$ of 1 and which have identical function values, i.e.

$$C^n(f) = \#\{(x_1, x_2) : d_H(x_1, x_2) = 1 \text{ and } f(x_1) = f(x_2)\}.$$

This value lies in the range 0 to $n2^n$. As a heuristic criterion, large values of $C^n(f)$ indicate a simple AND/OR-implementation with the extreme case $C^n(1) = n2^n$. On the other hand, small values of $C^n(f)$ indicate an expensive AND/OR-implementation with the extreme case $C^n(x_1 \oplus x_2 \oplus \cdots \oplus x_n) = 0$. Due to the observation that many multi-level synthesis systems perform poorly without a good sum-of-product representation of the target function, the authors of [HS96] claim that the complexity measure C can also be used for multi-level synthesis.

It has been shown that the complexity measure $C(f)$ can be expressed by the Walsh spectrum of f [HMM85]. The *Walsh spectrum* of f is the 2^n element vector

$$S^n(f) = (s_0, \ldots, s_{2^n-1}) = W^n Y^n(f),$$

where $Y^n(f)$ is the 2^n element truth table vector of f (with 0, 1 replaced by +1, −1) and W^n is the $2^n \times 2^n$ *Hadamard matrix* defined by

$$W^n = \begin{bmatrix} W^{n-1} & W^{n-1} \\ W^{n-1} & -W^{n-1} \end{bmatrix}, \qquad W^0 = 1.$$

If $||u||$ is defined as the number of 1's in the binary representation of u, then the complexity measure C can be written as a weighted sum of squares of the Walsh spectral coefficients

$$C^n(f) = n2^n - \frac{1}{2^n} \sum_{n=0}^{2^n-1} ||u|| \; s_u^2 \; .$$

The main idea in [HS96] to find a suitable linear filter is to construct linear transformations that minimize the complexity C. In order to find suitable candidates

for this minimization process the Walsh coefficient representation of the complexity measure is used. To avoid the exponential costs of computing the Walsh transformation symbolic BDD-techniques are used in connection with extensions of the Walsh transformation.

5.2 Synthesis by Linear Sifting

Our approach to achieve the described decomposition task is based on the Linear Sifting algorithm from Chapter 4. This algorithm applies linear transformations on a given Boolean function in a fully automatic manner. It is very time-efficient, and it sometimes leads to significant reductions in the sizes of the OBDDs compared to the "conventional" Sifting algorithm.

Our heuristic for extracting a linear filter is to use the (shared) BDD-sizes of the functions f', σ in the decomposition $f(x) = f'(\sigma(x))$ as a complexity measure:

$$C^n(f', \sigma) = \text{BDD-size}(f', \sigma).$$

Our algorithm for extracting a linear filter is to apply the Linear Sifting algorithm on the given target function as explained above.

Hence, our complexity measure coincides with the optimization criterion of the Linear Sifting algorithm. Particularly, we can exploit the fact that our implementation represents the linear transformation σ within the shared BDD itself. The measure has been inspired by the observation that in some cases Linear Sifting yields significant reductions in the BDD-size compared to the conventional Sifting algorithm. Although there is by far no strict correlation between BDD-size and the difficulty in the mapping step, we expect functions with relatively small BDD-size to be mapped easily. The advantage of our measure is that it is the most attractive one for the preprocessing step (minimizing the BDD-size is equivalent to minimizing the memory consumption) and that it somehow seems to be particularly adequate to analyze Boolean functions whose implementation will be obtained through modern BDD-based logic synthesis tools.

Although we did not particularly aim at fully symbolic synthesis algorithms like the one by Minato [Min96b] we also investigated these methods, see Section 5.3.3 and Section 5.4. These algorithms start by transforming the OBDD of a function into a ZDD (zero-suppressed BDD) of its prime-irredundant two-level form before proceeding.

5.3 Experimental Results

5.3.1 Main Setup

For the experimental evaluation we tried to use a very similar setup as the one in [HS96].

Our algorithm has been implemented using the CUDD BDD-package and SIS 1.3. After building the BDDs in a suitable manner (see below) they are dumped to a BLIF file. In case of using Linear Sifting the transformation is also encoded into this file (in form of a subcircuit). Notice that, for sequential benchmarks, only the combinational portion of the logic has been considered, that is, state inputs and state outputs have been treated as primary inputs and primary outputs, respectively.

The functions are then synthesized under SIS 1.3 using `script.rugged`. Mapping was done using the library `lib2.genlib` with the command `map -m 0` which optimizes the circuit for area. The memory limit was 128 MB, the time limit 20000 s on a Sun Sparc 10.

Our results for a set of 62 benchmark circuits are shown in Table 5.2 and 5.3. The column `In` represents the number of inputs resp. outputs of the function. The column `Area` shows the area of the mapped circuit, where all numbers are divided by 464, the greatest common divisor of all gate sizes. The delay of the mapped circuit and the time that SIS needs for all the above mentioned tasks are listed in the next two columns. The following two columns show the size of the BDD after the preprocessing step (= the symbolic simulation using Sifting resp. Linear Sifting) and the running times of this preprocessing.

In our experiments we were especially interested in the question if it is possible by using an appropriate preprocessing to synthesize those functions which failed when using only conventional variable reordering. Hence, our experiments were conducted as follows: We used the publicly available very good variable orders from the CUDD distribution [Som96] and used the mapping from the corresponding BDDs as our reference. These variable orders were obtained by a large number of different variable reordering variants. Then we used three combinations of Sifting and Linear Sifting in order to see whether a circuit can be improved or a failed attempt can now be completed. The three combinations are:

1. Start from the well-known good order, then apply a final Linear Sifting.

2. Dynamically apply Linear Sifting.

3. Dynamically apply Linear Sifting until Convergence.

By these methods the number of fails could be reduced from 9 to 6. As 2 of these 6 functions completed when starting from a good order, we now only have 4 functions that did not complete for any variant. Concerning the size of the circuits for which both approaches succeed, the decomposition generally seems to produce some overhead. However, in some cases like s1423, C7552 or C2670 significant gains could be achieved by using Linear Sifting. As it is proven by the CPU times, the preprocessing step is very time-efficient.

As a further reference we want to mention the results of [HS96]. Unfortunately, although they report on a large number of intermediate results of their implementation, they give their final mapping results only for two circuits whose BDDs have more than 31 nodes and compare it to a mapping from a hardware description language, see Figure 5.4.

5.3.2 Separate Mapping

In the above described experimental setup, we were mainly interested in the effect of Linear Sifting when acting as a preprocessing for introducing some structure into the function representation.

In particular for the cases where Linear Sifting leads to far superior results we were interested in the question in how far it is advisable to encode the transformation and the remaining function into one BLIF file or to do the mappings separately. Table 5.5 shows what happens to the circuits C499, C1355, C1908 when mapping the transformation and the remaining function separately. It reflects the general observation that separate mapping typically does not improve the mapping results in our environment. However, we did not use a specialized algorithm for mapping the transformation (like in [HS96]). This might improve our results.

5.3.3 Symbolic Synthesis

Minato [Min96a, Min96b] has proposed the use of symbolic BDD- and ZDD-techniques for optimizing multi-level logic circuits. They are mainly based on generating a ZDD-representation for a prime-irredundant two-level form of the given BDD and then using efficient divisor extraction algorithms on ZDDs for optimizing the circuit. As these techniques seem to be especially suited when starting a mapping procedure from a given BDD (instead e.g., of a multi-level description), we explored their use for our problem. In our experimental setup, we first used the symbolic algorithms for synthesizing the transformed function and then used SIS for doing the final mapping step, see Figure 5.6. For the implementation of the symbolic synthesis algorithms we used the corresponding procedures of the newest VIS releases.

		Starting from very good order					Best results - 3 Lin. Sifting BDDs				
	In	Area	Delay	Time	BDD	Pre	Area	Delay	Time	BDD	Pre
9symml	9	116	11.8	8	25	0.7	115	9.9	5	25	0.1
alu4	14	1210	36.8	156	350	0.2	1251	35.6	251	350	0.4
t481	16	90	12.5	3	21	0.9	90	12.5	4	21	1.0
vda	17	1748	25.1	265	478	0.3	1800	29.7	269	478	0.5
cordic	23	137	14.4	6	42	0.1	145	18.0	5	31	0.2
s499	23	561	19.8	49	330	0.1	1031	29.5	105	336	0.6
s344	24	300	18.2	12	104	0.1	366	18.3	14	83	0.3
ttt2	24	341	19.1	14	107	0.1	349	21.6	12.5	107	0.2
s1196	32	1509	29.3	362	598	0.3	1462	33.5	842	599	0.8
C1908	33	TIME OUT			5526	1.7	5148	74.9	3714	1209	23.1
adder16	33	366	73.2	9	82	0.1	481	51.2	18	49	0.6
s635	34	569	40.4	24	128	0.2	569	40.4	25	128	0.6
C432	36	2308	43.5	1995	1064	0.3	2279	45.5	3168	1064	1.1
mm9b	38	4472	46.9	6800	1527	0.7	4394	48.6	12941	1502	1.8
toolarge	38	754	26.7	121	319	1.8	737	29.1	98	319	2.4
mm9a	39	3851	52.5	4259	1111	0.5	TIME OUT			980	2.0
C1355	41	TIME OUT			25866	5.1	3110	54.6	757	484	18.1
C499	41	TIME OUT			25866	4.1	3419	58.0	968	520	10.7
k2	45	3667	35.2	1209	1246	0.6	3681	53.9	1438	1236	4.8
s967	45	1088	22.3	163	366	0.2	1012	24.3	169	383	0.5
C3540	50	TIME OUT			23828	7.7	TIME OUT			23823	120.2
s1269	55	3872	42.3	16456	1695	0.4	3719	46.9	3573	1480	2.2
C880	60	13047	77.6	1761	4053	0.8	13150	78.7	1585	4052	5.2
comp32	64	568	80.1	42	97	0.1	527	76.2	38	97	0.7
adder32	65	734	146.2	30	162	0.2	1019	126.7	58	97	2.2
adsb32	65	1323	145.6	136	345	0.2	1323	110.6	82	191	1.5
s938	66	717	40.1	48	161	0.2	726	44.1	46	161	1.4
sbc	68	1869	21.7	222	917	0.5	1861	19.6	193	917	2.0
alu32	73	2032	95.4	307	478	0.4	2078	82.1	281	388	1.8
alu32r	73	2032	95.4	305	478	0.4	2100	95.1	279	388	3.0
dalu	75	1665	32.5	488	689	0.8	1634	42.1	382	594	12.1

Figure 5.2: Experimental results I.

		Starting from very good order					Best results - 3 Lin. Sifting BDDs				
	In	Area	Delay	Time	BDD	Pre	Area	Delay	Time	BDD	Pre
s991	84	1329	79.0	82	328	0.3	1485	116.6	82	328	1.8
s1512	86	1451	23.6	104	566	0.4	1358	29.5	102	526	1.6
xi30	90	675	157.4	22	91	0.3	675	157.4	20	91	1.3
s1423	91	3025	39.7	898	1796	0.6	2533	35.8	577	1477	11.4
dpath32	93	1776	133.6	272	485	0.4	1511	100.9	254	396	3.2
mm30a	123	MEMORY OUT			11065	4.5	27038	226.1	8470	8979	47.8
i3	132	254	9.9	11	133	0.2	290	11.0	13	133	1.0
i8	133	2788	45.4	541	1276	1.3	2860	52.9	719	1276	4.1
apex6	135	1406	25.2	84	498	0.3	1432	34.3	158	498	1.6
s3271	142	2564	29.0	160	831	0.8	2666	29.4	145	671	2.9
frg2	143	1823	36.1	313	963	0.6	1837	35.1	321	921	3.5
s4863	153	MEMORY OUT			64088	26.3	MEMORY OUT			64079	419.0
C5315	178	6374	53.0	542	1719	1.3	5746	64.1	3448	1537	8.5
s5378	199	3727	27.6	468	1932	1.3	MEMORY OUT			1688	56.0
C7552	207	9189	143.3	638	2212	2.0	6182	113.5	315	1150	157.8
s3384	226	2605	83.3	141	693	1.2	2737	42.9	160	597	4.5
C2670	233	6475	71.1	433	1774	0.9	5003	51.8	308	1401	9.3
s9234.1	247	9007	45.1	733	3045	2.2	9045	48.1	688	2731	56.2
comp128	256	2296	318.9	3392	385	0.3	2103	311.2	2852	385	8.9
des	256	7148	120.6	1950	2945	1.7	8106	224.9	4012	2087	162.5
add128	257	2942	584.5	1220	642	0.9	4091	497.3	189	385	46.2
adsb128	257	5435	579.5	10750	1401	1.4	5550	428.8	366	767	24.5
i10	257	TIME OUT			20660	6.2	TIME OUT			19343	650.4
alu128	265	TIME OUT			1918	4.2	8550	323.4	623	1540	60.1
s6669	322	TIME OUT			18033	25.3	TIME OUT			17912	157.3
clma	415	1481	33.3	117	738	9.0	1501	29.7	128	718	13.0
clmb	415	1388	28.4	134	690	9.0	1244	23.7	103	618	24.2
dsip	452	5310	119.7	1192	3033	3.1	10595	155.8	3848	2703	31.1
bigkey	486	5529	126.5	1173	1595	1.4	7817	144.2	1480	1406	22.7
s15850.1	611	25154	111.8	4766	9712	15.8	31514	137.6	7727	10934	236.7
s13207.1	700	7487	55.0	1322	2863	12.8	8622	43.2	1187	2854	42.0

Figure 5.3: Experimental results II.

		Starting from a HDL			Using transformation		
	In	Area	Delay	BDD	Area	Delay	BDD
sec	8	674	23.02	109	357	13.95	111
des	256	6213	126.36	7388	7824	117.53	4887

Figure 5.4: Reference results from [HS96].

		Mapping remaining part			Mapping transformation				
	In	Area	Delay	Time	Area	Delay	Time	BDD	Pre
C1355	41	1667	26.0	82	1817	68.6	75.9	484	18.1
C499	41	1663	25.2	93	2296	83.2	83.7	520	10.7
C1908	33	4400	42.6	1906	732	29.9	15.9	1209	32.1

Figure 5.5: Results from separate mapping.

The results for some typical cases where Linear Sifting reduced the BDD-size significantly are shown in Figure 5.7. The comparison with Figure 5.5 shows that the overhead from the two-phase mapping (VIS and SIS) seems to be too big in our context.

A very astonishing result which is also of independent interest in the connection between BDDs and ZDDs is obtained in the case of adders. When using Minato's algorithm to transform a BDD for an n-bit adder to the ZDD of its prime-irredundant two-level form, the ZDD is of linear size. When applying Linear Sifting on the BDD of the adder, the BDD-size is reduced by 40 % of the original size. If, however, this linearly sifted BDD is transformed to the ZDD of its irredundant sum-of-product, the size of the resulting ZDD seems to grow exponentially. A thorough description of this effect will be given in Section 5.4.

Figure 5.6: Symbolic synthesis.

		Mapping remaining function		
	In	Area	Delay	Time
C1355	41	1945	22.5	314
C499	41	1801	28.9	191
C1908	33	TIME OUT		

Figure 5.7: Results from symbolic synthesis.

5.3.4 Summary of the Experiments

We have presented and experimentally evaluated a new approach for synthesizing and mapping a Boolean function after performing a suitable decomposition. Our method is an alternative to the one proposed in [HS96]. The experimental results show that this approach may produce good results in cases where the function cannot be synthesized without a decomposition. Our approach is easy to implement in all environments which use dynamic reordering for BDDs, the resulting algorithm is simpler than the one in [HS96]. It also very well supports the concept of symbolic BDD-representations and may immediately profit from further improvements of the Linear Sifting algorithm.

In general, we think that the full power of decomposition techniques for mapping from a BDD has not been exploited so far. Additional investigations of the auxiliary routines (like Linear Sifting), their combination and the optimization criteria should help to close this gap. Specifically, the results concerning adder functions suggest that the minimization of the ZDD for the cover of the given function rather than the minimization of the BDD for the function itself may provide a more reliable cost function. (The number of nodes in that ZDD corresponds to the number of literals in a factored form for the function.)

5.4 Linear Sifting and Adder Functions

In this section we give some detailed analyses of adder functions in the presence of symbolic synthesis algorithms and Linear Sifting. The goal it not only to analyze the blowup for adders reported in Section 5.3.3 but also to provide a new reference result for unexpected yet important effects caused by combining seemingly unrelated algorithms on BDDs and ZDDs.

Let $f(a_{n-1}, b_{n-1}, \ldots, a_0, b_0, c_{in}) : \mathbb{B}_{2n+1} \rightarrow \mathbb{B}_{n+1}$ be the n-bit adder function that takes as input two n-bit strings $a_{n-1} \ldots a_0$, $b_{n-1} \ldots b_0$ and an incoming carry c_{in},

and computes their binary sum $s_{n-1} \dots s_0$ and the outgoing carry bit c_n, see Figure 5.8.

$$\begin{array}{ccccccc} & a_{n-1} & a_{n-2} & \dots & a_2 & a_1 & a_0 \\ & b_{n-1} & b_{n-2} & \dots & b_2 & b_1 & b_0 \\ & & & & & & c_{in} \\ \hline c_n & s_{n-1} & s_{n-2} & \dots & s_2 & s_1 & s_0 \end{array}$$

Figure 5.8: n-bit adder

We prove the following behavioral properties of adders:

1. By linear transformations, the OBDD-size of adders (w.r.t. the optimal order) can be reduced by at least 40%. An upper bound for the reduction is 60%. In practice, the Linear Sifting algorithm indeed finds a transformation which leads to a 40% reduction in size.

2. When establishing a prime-irredundant two-level form in ZDD-representation for adders via the ISOP algorithm, the size of the ZDD for the original adder remains linear. On the other hand, the size of this ZDD for the transformed adder seems to grow exponentially.

We proceed as follows:

1. The variable order $b_{n-1}, a_{n-1}, \dots, b_0, a_0, c_{in}$ leads to an OBDD-size of $5n+2$ for the n-bit adder.

2. *For all* variable orders, n-bit adders have an OBDD-size of at least $5n - C$ for a fixed constant C.

3. For every n-bit adder there exists a linear transformation τ_n for which the transformed OBDD has size $3n+1$.

4. (Trivial:) Every linearly transformed n-bit adder has size at least $2n+2$.

5. The application of the ISOP algorithm on the adders of the first statement leads to ZDDs of size $11n+2$.

6. The size of the ZDDs which are the results of the ISOP algorithm when applied to the transformed adders seem to grow exponentially.

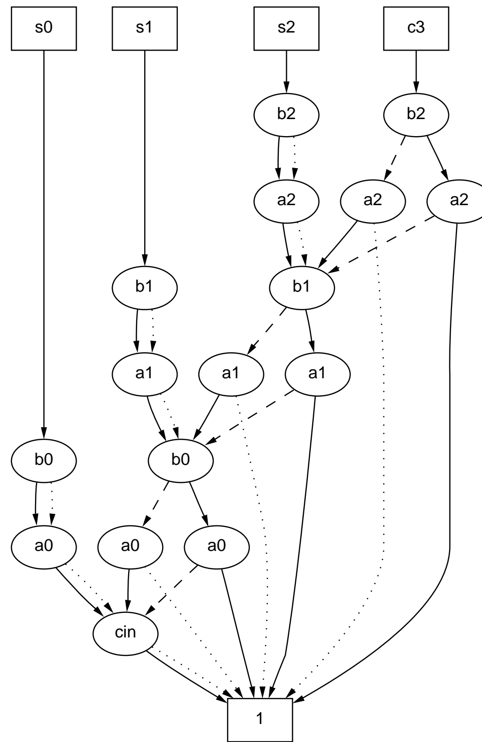


Figure 5.9: 3-bit adder

5.4.1 Adder Behavior

It is well-known that good BDD-orders for the n-bit adder require to keep a_i and b_i together in the order, $0 \leq i \leq n-1$. In Lemma 5.1 we will prove that the variable order $b_{n-1}, a_{n-1}, \ldots, b_0, a_0, c_{in}$ leads to the size $5n+2$ in the presence of complemented edges. After that, in Theorem 5.2, we will prove that this size is asymptotically optimal, as *every* variable order leads to a size of at least $5n - C$.

Lemma 5.1. *For $n \geq 1$, the reduced OBDD using complemented edges for the n-bit adder w.r.t. the variable order $b_{n-1}, a_{n-1}, \ldots, b_0, a_0, c_{in}$ has exactly $5n+2$ nodes.*

Proof. The lemma can be proven by induction on the length of the adder. The idea for the induction should become clear from Figure 5.9 which shows the OBDD for a 3-bit adder. In the diagram a 1-edge is indicated by a solid line, a 0-edge by a dashed line and a complemented 0-edge by a dotted line. □

Theorem 5.2. *The OBDD using complemented edges for the n-bit adder function w.r.t. the optimal order has size at least $5n - C$ for a fixed constant C.*

Proof. Due to the partial symmetry we can assume w.l.o.g.: For each $0 \leq i \leq n-1$, b_i appears in the order before a_i, and a_0 appears before c_{in}.

1. *Show:* For all $0 \leq i \leq n-1$ there exist nodes B_i, A_i labeled by b_i resp. a_i with the following property: The represented subfunctions have complementary cofactors w.r.t. b_i resp. a_i (i.e. the subfunction f satisfies $f_{b_i=0} = \overline{f_{b_i=1}}$).

Proof: Let $0 \leq i \leq n-1$, and consider the OBDD of the output bit s_i. As s_i depends on both b_i and a_i, there exist nodes B_i and A_i which are labeled by b_i resp. a_i. As s_i has complementary cofactors w.r.t. b_i (resp. a_i), the represented subfunction in B_i (resp. A_i) has complementary cofactors w.r.t. b_i (resp. a_i).

2. *Show:* For each $0 \leq i \leq n-1$: If a_i is not the last variable of $a_0, \ldots, a_{n-1}$ in the order, there exist three different nodes C_i, D_i and E_i which are labeled by b_i or a_i and which are different from the two nodes A_i, B_i.

Proof: Let $0 \leq i \leq n-1$, and let a_i be *not* the last variable of $a_0, \ldots, a_{n-1}$ in the order. Consider the OBDD for the outgoing carry c_n. For a variable $x \in \{a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, c_{in}\}$ let

$$
\begin{aligned}
A_x^{top} &= \{a_j : a_j <_\pi x\}, & A_x^{bot} &= \{a_j : a_j >_\pi x\}, \\
B_x^{top} &= \{b_j : b_j <_\pi x\}, & B_x^{bot} &= \{b_j : b_j >_\pi x\}, \\
C_x^{top} &= \{c_{in} : c_{in} <_\pi x\}, & C_x^{bot} &= \{c_{in} : c_{in} >_\pi x\},
\end{aligned}
$$

where π is the underlying variable order. Furthermore, for a subset $Z = \{z_1, \ldots, z_k\} \subset \{a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, c_{in}\}$, the notation $Z = 1$ (analogously $Z = 0$) is defined to denote $z_1 = 1, \ldots, z_k = 1$.

Case 1: All the variables $b_{i+1}, a_{i+1}, \ldots, b_{n-1}, a_{n-1}$ occur before b_i in the order.

As a_i is not the last variable of $a_0, \ldots, a_{n-1}$ in the order, there exists a j with $j < i$ such that a_j occurs after a_i in the order. Consider the following subfunction of c_n:

$$h_n(A_{b_i}^{bot}, B_{b_i}^{bot}, C_{b_i}^{bot}, b_i) = c_n(A, B, c_{in})|_{A_{b_i}^{top}=1, B_{b_i}^{top}\setminus\{b_j\}=0, B_{b_i}^{top}\cap\{b_j\}=1, C_{b_i}^{top}=0} = 0\,.$$

Intuitively, in this subfunction, the input bits b_i, a_i and a_j are responsible whether or not there will be a carry propagation into position $j+1$ and hence up to the output bit c_n.

$$
\begin{aligned}
h_n(A_{b_i}^{bot}, B_{b_i}^{bot}, C_{b_i}^{bot}, b_i)|_{A_{b_i}^{bot}\setminus\{a_j\}=1, a_j=0, B_{b_i}^{top}\setminus\{b_j\}=0, B_{b_i}^{top}\cap\{b_j\}=1, C_{b_i}^{bot}=0, b_i=0} &= 0 \\
&\neq \\
h_n(A_{b_i}^{bot}, B_{b_i}^{bot}, C_{b_i}^{bot}, b_i)|_{A_{b_i}^{bot}\setminus\{a_j\}=1, a_j=0, B_{b_i}^{top}\setminus\{b_j\}=0, B_{b_i}^{top}\cap\{b_j\}=1, C_{b_i}^{bot}=0, b_i=1} &= 1\,.
\end{aligned}
$$

Hence, there exists a node labeled b_i representing the subfunction h_n.

Now consider the following subfunctions h_n^0, h_n^1 of h_n:

$$\begin{aligned} h_n^0(A_{a_i}^{bot}, B_{a_i}^{bot}, C_{a_i}^{bot}, a_i) &= c_n(A, B, c_{in})|_{A_{a_i}^{top}=1, B_{a_i}^{top}\setminus\{b_i,b_j\}=0, B_{a_i}^{top}\cap\{b_j\}=1, C_{a_i}^{top}=0, b_i=0} \\ h_n^1(A_{a_i}^{bot}, B_{a_i}^{bot}, C_{a_i}^{bot}, a_i) &= c_n(A, B, c_{in})|_{A_{a_i}^{top}=1, B_{a_i}^{top}\setminus\{b_i,b_j\}=0, B_{a_i}^{top}\cap\{b_j\}=1, C_{a_i}^{top}=0, b_i=1} \end{aligned}$$

h_n^0 depends on a_i, as

$$\begin{aligned} h_n^0(A_{a_i}^{bot}, B_{a_i}^{bot}, C_{a_i}^{bot}, a_i)|_{A_{a_i}^{bot}=1, B_{a_i}^{bot}\setminus\{b_j\}=0, B_{a_i}^{bot}\cap\{b_j\}=1, C_{a_i}^{bot}=0, a_i=0} &= 0 \\ \neq& \\ h_n^0(A_{a_i}^{bot}, B_{a_i}^{bot}, C_{a_i}^{bot}, a_i)|_{A_{a_i}^{bot}=1, B_{a_i}^{bot}\setminus\{b_j\}=0, B_{a_i}^{bot}\cap\{b_j\}=1, C_{a_i}^{bot}=0, a_i=1} &= 1\ . \end{aligned}$$

h_n^1 depends on a_i, as

$$\begin{aligned} h_n^1(A_{a_i}^{bot}, B_{a_i}^{bot}, C_{a_i}^{bot}, a_i)|_{A_{a_i}^{bot}=0, B_{a_i}^{bot}=0, C_{a_i}^{bot}=0, a_i=0} &= 0 \\ \neq& \\ h_n^1(A_{a_i}^{bot}, B_{a_i}^{bot}, C_{a_i}^{bot}, a_i)|_{A_{a_i}^{bot}=0, B_{a_i}^{bot}=0, C_{a_i}^{bot}=0, a_i=1} &= 1\ . \end{aligned}$$

h_n^1 and h_n^0 are different because

$$\begin{aligned} h_n^0(A_{a_i}^{bot}, B_{a_i}^{bot}, C_{a_i}^{bot}, a_i)|_{A_{a_i}^{bot}=0, B_{a_i}^{bot}=0, C_{a_i}^{bot}=0, a_i=1} &= 0 \\ \neq& \\ h_n^1(A_{a_i}^{bot}, B_{a_i}^{bot}, C_{a_i}^{bot}, a_i)|_{A_{a_i}^{bot}=0, B_{a_i}^{bot}=0, C_{a_i}^{bot}=0, a_i=1} &= 1\ . \end{aligned}$$

Hence, there are three nodes whose corresponding subfunctions are monotone increasing in b_i resp. a_i and therefore different from A_i, B_i.

Case 2: There exists a $j > i$ such that a_j occurs after a_i in the order.

As c_n depends on a_j, there are at least two nodes C_i, D_i labeled by b_i, a_i whose corresponding subfunctions are monotone increasing functions. We will now show the existence of another node labeled b_i by inspecting the OBDD for the function s_j.

Consider the following subfunction of s_j:

$$h_n(A_{b_i}^{bot}, B_{b_i}^{bot}, C_{in}^{bot}, b_i) = s_j(A, B, c_{in})|_{A_{b_i}^{top}\cap\{a_{i+1},\ldots,a_{n-1}\}=1, B_{b_i}^{top}=0, A_{b_i}^{top}\cap\{a_0,\ldots,a_{i-1}\}=0, C_{b_i}=0}$$

h_n depends on b_i as

$$\begin{array}{rcl} h_n(A^{bot}_{b_i}, B^{bot}_{b_i}, C^{bot}_{in}, b_i)|_{A^{bot}_{b_i}\cap\{a_{i+1},\ldots,a_{n-1}\}=1, B^{bot}_{b_i}=0, A^{bot}_{b_i}\cap\{a_0,\ldots,a_{i-1}\}=0, C_{b_i}=0, a_i=1, b_i=0} & = & 1 \\ \neq & & \\ h_n(A^{bot}_{b_i}, B^{bot}_{b_i}, C^{bot}_{in}, b_i)|_{A^{bot}_{b_i}\cap\{a_{i+1},\ldots,a_{n-1}\}=1, B^{bot}_{b_i}=0, A^{bot}_{b_i}\cap\{a_0,\ldots,a_{i-1}\}=0, C_{b_i}=0, a_i=1, b_i=1} & = & 0. \end{array}$$

h_n is not monotone increasing in b_i. Hence, the corresponding node is different from C_i. As h_n obviously depends on a_j (namely, it has complementary cofactors w.r.t. a_j), it is different from B_i (whose corresponding subfunction does not depend on a_j). □

The following lower bound on the size of the linearly transformed adder follows immediately from the fact that the transformed adder must depend on all $2n+1$ input variables.

Lemma 5.3. *The transformed adder has OBDD-size (using complemented edges) at least* $2n+2$.

The following does not only show that linearly transformed adders nearly meet this lower bound, but (as the transformation is found by our implementation of the Linear Sifting algorithm) shows that such a transformation is also found in practice.

Lemma 5.4. *For* $n \geq 1$*, there exists a linear transformation* τ_n *such that the OBDD (using complemented edges) of the transformed* n*-bit adder has exactly* $3n+1$ *nodes.*

The transformation in this lemma is exactly the one that is found in our implementation of Linear Sifting.

Proof. The proof is also done by induction. The idea for the induction should become clear from Figure 5.10 which can be extended to arbitrary n. On the left side of the figure the functional relation between the original and the transformed variables a'_i, b'_i, c'_{in} is given. In particular, we have for the example case $n = 3$

$$\begin{array}{rcl} s_0 & = & c'_{in} \equiv b'_0 \equiv b'_1 \equiv b'_2 \\ & = & (a_0 \equiv c_{in}) \equiv (b_1 \equiv b_0) \equiv (b_2 \equiv b_1) \equiv b_2 \\ & = & a_0 \oplus b_0 \oplus c_{in}\,, \end{array}$$

$$\begin{array}{rcl} s_1 & = & a'_1 \equiv (a'_0 \cdot b_0 \oplus \overline{a'_0} \cdot \overline{(a_0 \oplus b_0 \oplus c_{in})}), \\ & = & a'_1 \equiv ((b_0 \equiv c_{in}) \cdot b_0 \oplus (b_0 \oplus c_{in}) \cdot (a_0 \oplus b_0 \oplus \overline{c_{in}})) \\ & = & a'_1 \equiv (b_0 c_{in} \oplus a_0 b_0 \oplus b_0 \oplus b_0 \overline{c_{in}} \oplus a_0 c_{in} \oplus b_0 c_{in}) \\ & = & (b_1 \equiv a_1) \equiv (b_0 c_{in} \oplus a_0 b_0 \oplus a_0 c_{in}) \\ & = & a_1 \oplus b_1 \oplus c_1\,. \end{array}$$

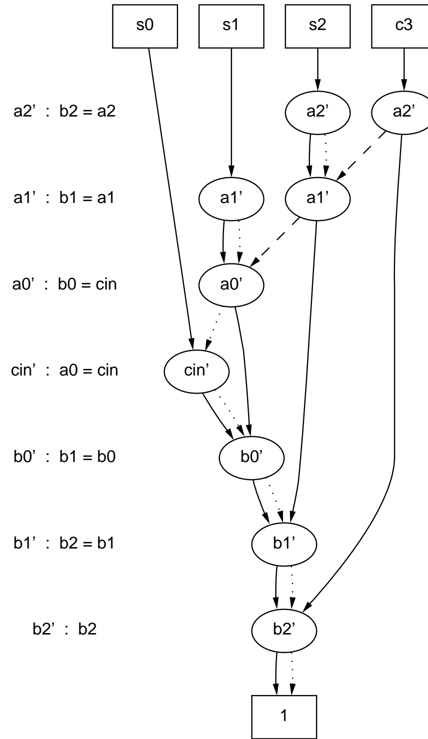


Figure 5.10: Transformed 3-bit adder

□

5.4.2 The ISOP Algorithm

Modern symbolic synthesis algorithms that are for example integrated in the VIS system, use symbolic representations of two-level forms. In particular, ZDD-representations of **disjunctive normal forms** (**DNF**) are of great importance. In this context, one also speaks of **sum of products** or **cube sets**.

Definition 5.5. *A disjunctive normal form is called*

- **prime**, *if each term is a prime implicant; that is, no literal can be eliminated without changing the function.*
- **irredundant**, *if there are no redundant terms; that is, no term can be eliminated without changing the function.*

The transformation from an OBDD to a ZDD for a prime-irredundant sum of products is achieved by using Morreale's algorithm [Mor70] which has been adapted to the decision diagram environment by Minato [Min96a]. This algorithm is shown in Figure 5.11 for an incompletely function f with one output. If f is a function with multiple outputs, the algorithm is performed successively on each output bit.

The body of the **ISOP algorithm** (**Irredundant Sum of Products**) contains three recursive calls. The parameter of the third call strongly depends on the results of the first two recursive calls. Due to this dependency a detailed analysis of the resulting ZDD function quickly becomes quite tedious.

Property 5.6. *For a completely specified Boolean function f the ISOP algorithm has the following properties, where v, f_0, f_1, $isop_0$, $isop_1$ and $isop_d$ are defined as in the description of the algorithm:*

1. *If $f_0 \cdot f_1 = 0$ (i.e. f_0 and f_1 do not have common minterms or don't cares) then*
$$ISOP(f) = \overline{v} \cdot ISOP(f_0) + v \cdot ISOP(f_1) + 0$$
2. *If $f_0 \Rightarrow f_1$ then $isop_d = f_0$ and*
$$ISOP(f) = 0 + v \cdot isop_1 + isop_d.$$
3. *If $f_1 \Rightarrow f_0$ then $isop_d = f_1$ and*
$$ISOP(f) = \overline{v} \cdot isop_0 + 0 + isop_d.$$

Proof. 1. If f_0 and f_1 do not have common minterms or don't cares, then f_0'' and f_1'' do not have common minterms, common don't cares or common don't cares/minterms. Hence, $f_d = 0$.

2. If $f_0 \Rightarrow f_1$ then f_0' only consists of 0's and don't cares, and hence $isop_0 = ISOP(f_0') = 0$. This also implies $f_0'' = f_0$. Hence, in order to show that $isop_d = f_1$ we only have to show that in the definition diagram of f_d, the combination $f_0'' = 1, f_1'' = 0$ is not possible. If for a particular input $f_0'' = f_0 = 1$ then $f_1 = 1$, and therefore $f_1'' = 1$ or $f_1'' = d$, i.e. $f_1'' \neq 0$.

3. Analogous to 2. □

The two-level form of a function $f(x_1, \ldots, x_n)$ is represented by a ZDD in the following way: For each input variable x_i two ZDD-variables x_i, x_i^c ("c" for "complement") are created: one for the positive literal x_i and one for the negative literal $\overline{x_i}$.

Lemma 5.7. *For $n \geq 1$, the resulting ZDD (which depends on $2 \cdot (2n+1)$ variables) when applying Minato's algorithm on the adder of Lemma 5.1 has size $11n + 2$.*

ISOP($f(x)$) {
/* Input: OBDD for $f(x) : \{0,1\}^n \rightarrow \{0,1,d\}$ (d: don't care) */
/* Output: $isop$: prime-irredundant cube sets being consistent with f */
If ($\forall x \in \{0,1\}^n : f(x) \neq 1$) { $isop \leftarrow$ 0;}
Else If ($\forall x \in \{0,1\}^n : f(x) \neq 0$) { $isop \leftarrow$ 1;}
Else {
$v \leftarrow$ top variable in the OBDD for f;
$f_0 \leftarrow f(x)|_{v=0}$;
$f_1 \leftarrow f(x)|_{v=1}$;
Compute f_0', f_1' as defined by the following rules:

f_0' :

f_1 \ f_0	0	1	d
0	0	1	d
1	0	d	d
d	0	d	d

f_1':

f_1 \ f_0	0	1	d
0	0	0	0
1	1	d	d
d	d	d	d

$isop_0 \leftarrow$ ISOP(f_0'); /* recursively generates cubes including $\overline{v}$ */
$isop_1 \leftarrow$ ISOP(f_1'); /* recursively generates cubes including v */
Let g_0, g_1 be the Boolean functions which are defined by the cube sets $isop_0$, $isop_1$, respectively;
Compute f_0'', f_1'' as defined by the following rules:

f_0'' :

g_0 \ f_0	0	1	d
0	0	1	d
1	-	d	d

f_1'':

g_1 \ f_1	0	1	d
0	0	1	d
1	-	d	d

Compute f_d as defined by the following rule:

f_d :

f_1'' \ f_0''	0	1	d
0	0	0	0
1	0	1	1
d	0	1	d

$isop_d \leftarrow$ ISOP(f_d); /* recursively generates cubes excluding $\overline{v}, v$ */
$isop \leftarrow (\overline{v} \cdot isop_0) \vee (v \cdot isop_1) \vee isop_d$;
}
}

Figure 5.11: The ISOP algorithm

In the proof of the lemma, we will use the following don't care notation:

Definition 5.8. *Let $f, g : \{0,1\}^n \rightarrow \{0,1\}$ be Boolean functions satisfying $f \cdot g = 0$. Then $f \;/\; g$ denotes the incompletely specified Boolean function $\{0,1\}^n \rightarrow \{0,1,d\}$ that is defined by*

$$(f \;/\; g)\;(x) = \begin{cases} 0 & \text{if } \; f(x) = 0 \text{ and } g(x) = 0 \\ 1 & \text{if } \; f(x) = 1 \text{ and } g(x) = 0 \\ d & \text{if } \; f(x) = 0 \text{ and } g(x) = 1 \end{cases} .$$

Proof. The case $n = 1$ can be checked separately. We prove the following three statements for the ZDD of ISOP($adder_n$), $n \geq 2$, by one induction:

1. The ZDD for ISOP(s_{n-1}), ISOP(c_n) has the structure as shown in Figure 5.12 (a) with sub-OBDDs rooted in A and B which represent ISOP(c_{n-1}) resp. ISOP($\overline{c_{n-1}}$).
2. The ZDD for the n-bit adder has $11n + 2$ nodes.

Induction base: The case $n = 2$ can easily be checked.

Induction step: From the induction hypothesis it follows that the shared ZDD for ISOP(s_{n-2}), ISOP(c_{n-1}) of the $(n-1)$-bit adder has the structure that is shown in Figure 5.12 (b). The sub-OBDDs rooted in A, B represent ISOP(c_{n-2}) and ISOP($\overline{c_{n-2}}$), respectively. We will now extend this ZDD to construct a ZDD for ISOP(s_{n-2}, s_{n-1}, c_n). The construction which will be explained in the following is shown in Figure 5.13.

First we show how to construct the ZDD for ISOP($\overline{c_{n-1}}$). Later on, we will see that indeed this ZDD will be necessary to construct the ZDD of ISOP($adder_n$).

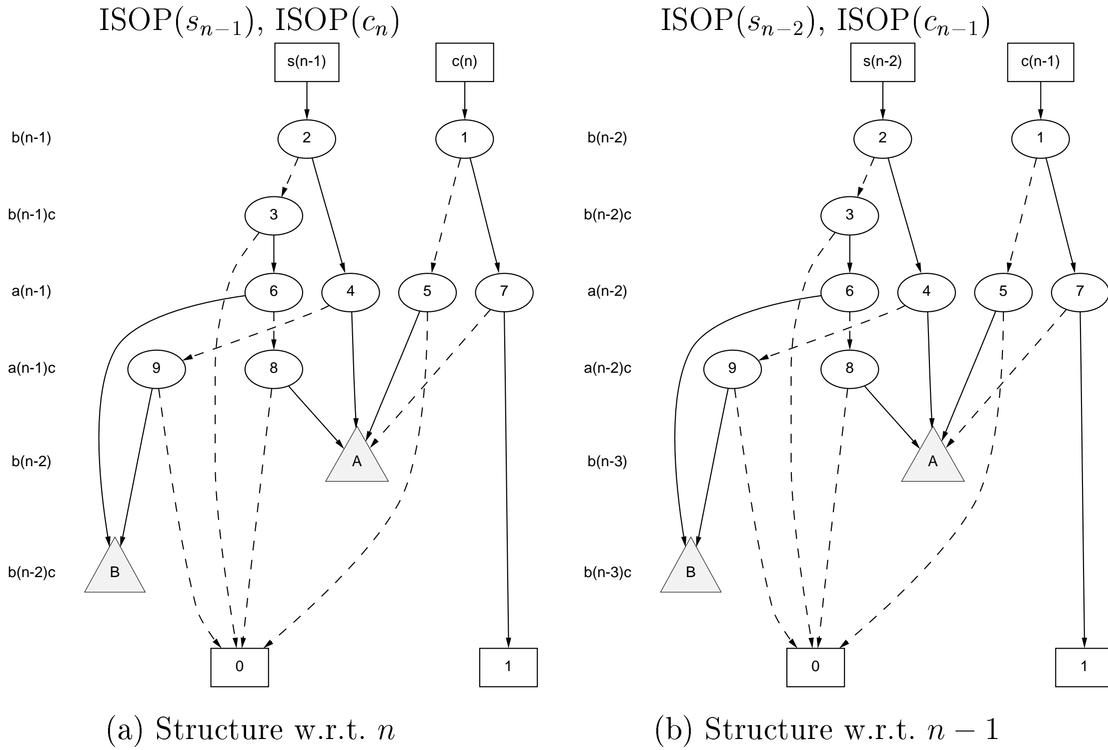
ISOP($\overline{c_{n-1}}$), splitting variable b_{n-2}:

In the stage of the ISOP algorithm where b_{n-2} is the splitting variable, we have the following assignments to f_0, f_1:

$$\begin{aligned} f_0 &= (\overline{c_{n-1}})_{b_{n-2}=0} = \overline{c_{n-2}} + \overline{a_{n-2}} c_{n-2}, \\ f_1 &= (\overline{c_{n-1}})_{b_{n-2}=1} = \overline{a_{n-2}}\;\overline{c_{n-2}}. \end{aligned}$$

It holds $f_1 \Rightarrow f_0$. Due to Property 5.6, $isop_1 = 0$. Then, using the don't care notation from Definition 5.8,

$$f_0' = a_{n-2}\overline{c_{n-2}} + \overline{a_{n-2}} c_{n-2} \;/\; \overline{a_{n-2}}\;\overline{c_{n-2}}.$$



(a) Structure w.r.t. n (b) Structure w.r.t. $n-1$

Figure 5.12: Inductive structure

- ISOP($a_{n-2}\overline{c_{n-2}} + \overline{a_{n-2}}c_{n-2}$ / $\overline{a_{n-2}}\ \overline{c_{n-2}}$), splitting variable a_{n-2}:

  $f_0 = c_{n-2}$ / $\overline{c_{n-2}}$, $f_1 = \overline{c_{n-2}}$. Consequently, $isop_0 = 1$, $isop_1 = 0$, $isop_d =$ ISOP($\overline{c_{n-2}}$), and $isop = \overline{a_{n-2}} +$ ISOP($\overline{c_{n-2}}$).

  As ISOP($\overline{c_{n-2}}$) is represented by node B, the node n_{11} is constructed.

Now the computation of ISOP($\overline{c_{n-1}}$) proceeds as follows: $f_d = (\overline{c_{n-1}})_{b_{n-2}=1} = \overline{a_{n-2}}\ \overline{c_{n-2}}$.

- ISOP($\overline{a_{n-2}}\ \overline{c_{n-2}}$), splitting variable a_{n-2}:

  $f_0 = \overline{c_{n-2}}$, $f_1 = 0$. Consequently, $isop_1 = 0$, $isop_0 = \overline{c_{n-2}}$, $isop_d = 0$, $isop = \overline{a_{n-1}}\ \overline{c_{n-2}}$. Hence, due to the elimination rule for ZDDs, $isop$ is represented by a node labeled a^c_{n-2} with 1-successor B and 0-successor 0. Such a node already exists in the ZDD for ISOP($adder_{n-1}$), namely node 9.

Now the subresults within the computation of ISOP($\overline{c_{n-1}}$) can be combined: As ISOP(f_0') is represented by node n_{11} and ISOP($\overline{f_d}$) is represented by node 9, the node n_{10} is constructed.

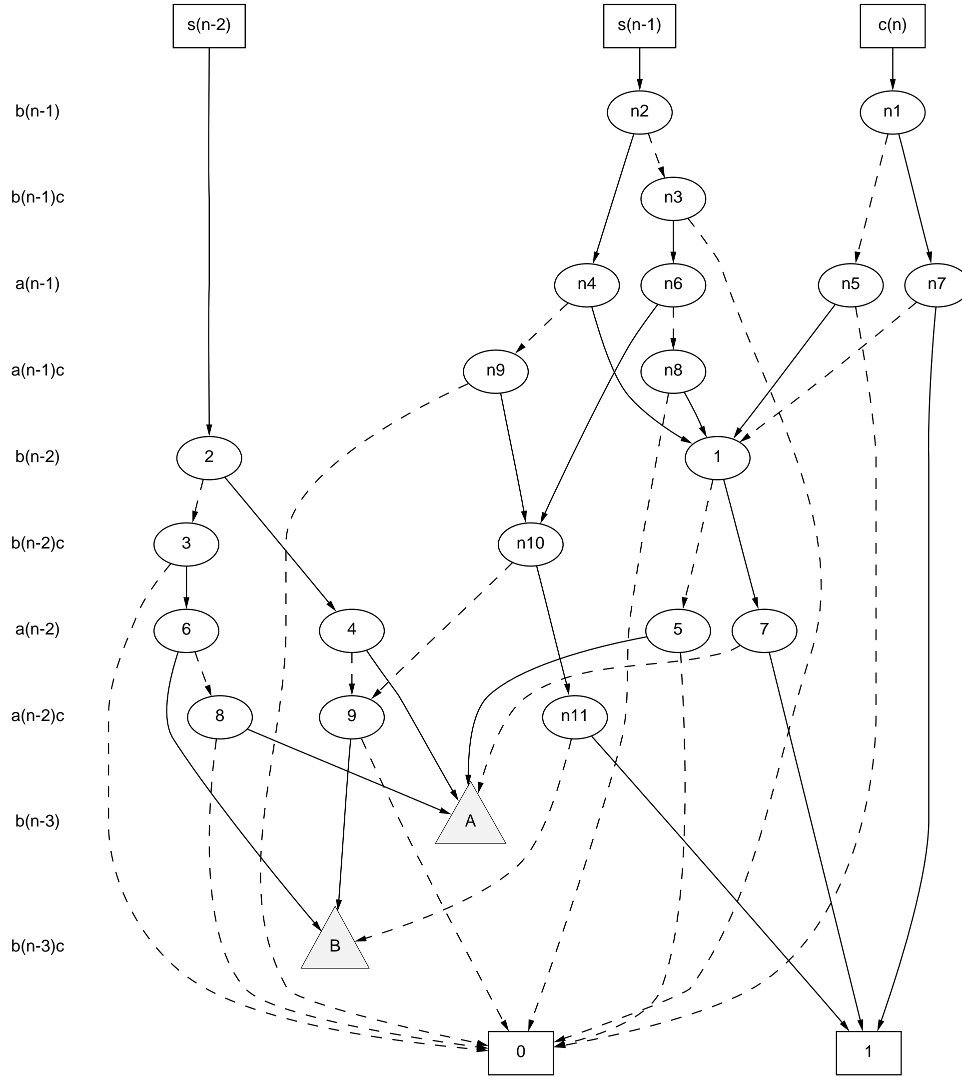


Figure 5.13: Induction step

ISOP(s_{n-1}), *splitting variable* b_{n-1}*:*

$f_0 = (s_{n-1})_{b_{n-1}=0}$, $f_1 = (s_{n-1})_{b_{n-1}=1}$. Due to Property 5.6 we have $f_0' = f_0$, $f_1' = f_1$.

- ISOP$((s_{n-1})_{b_{n-1}=0})$, splitting variable a_{n-1}:

  $f_0 = (s_{n-1})_{b_{n-1}=0,a_{n-1}=0} = c_{n-1}$. $f_1 = (s_{n-1})_{b_{n-1}=0,a_{n-1}=1} = \overline{c_{n-1}}$. Due to Property 5.6 we have

$$isop = \overline{a_{n-1}} \cdot c_{n-1} + a_{n-1} \cdot \overline{c_{n-1}}.$$

As ISOP(c_{n-1}) is represented by node 1 and ISOP($\overline{c_{n-1}}$) is represented by node n_{10}, the nodes n_6, n_8 are constructed.

- ISOP($(s_{n-1})_{b_{n-1}=1}$), splitting variable a_{n-1}:

  $f_0 = (s_{n-1})_{b_{n-1}=0,a_{n-1}=1} = \overline{c_{n-1}}$. $f_1 = (s_{n-1})_{b_{n-1}=1,a_{n-1}=1} = c_{n-1}$. As ISOP($\overline{c_{n-1}}$) is represented by node n_{10} and ISOP(c_{n-1}) is represented by node 1, the nodes n_4, n_9 are constructed.

Now the subresults within the computation of ISOP(s_{n-1}) can be combined: As ISOP(f_0') is represented by node n_6 and ISOP(f_1') is represented by node n_4, the nodes n_2, n_3 are constructed.

ISOP(c_n), splitting variable b_{n-1}:

Analogous to our reduction of ISOP(c_{n-1}) to ISOP(c_{n-2}), the computation of ISOP(c_n) can be reduced to the computation of ISOP(c_{n-1}). This results in the construction of the nodes n_1, n_5, n_7.

Now the induction step can be completed: Node 1 and node n_{10} are the nodes A resp. B from the induction claim, and 11 new nodes have been added.

□

In contrast to the linear ZDD-size resulting from the original adder, the ZDD-size resulting from the *transformed* adder grows much faster. When starting from the linear transformed OBDD that has been described in Lemma 5.4 the OBDD satisfies the recurrence equation

$$size(n) = 2 \cdot size(n-1) + 4n + 5, \quad size(2) = 31, \tag{5.1}$$

for all $n \geq 3$ in our corresponding experiments (in particular $n \leq 16$) and hence seems to grow exponentially in n. The explicit solution of the recurrence equation (5.1) can be computed by the substitutions $size(n) = b_n - 4n$, and $b_n = c_n - 13$, which leads to the homogeneous equation $c_n = 2c_{n-1}$, $c_2 = 52$ and hence to $c_n = 13 \cdot 2^n$. Substituting this result backwards yields

$$size(n) = 13 \cdot 2^n - 4n - 13.$$

5.4.3 Open Questions

We have given some theoretical results in the presence of OBDDs, ZDDs for sum of products and linear transformations. In general, we know only very few suitable techniques for proving lower bounds in the case of linear transformations. Some theoretical problems in this area are therefore still open.

Open questions in connection with adders:

1. Is the transformation from Lemma 5.4 the best linear transformation for adders ?
2. How to prove the above conjecture of the ISOP-behavior for the transformed adders (see Equation (5.1)) ?

More general open questions concerning lower bounds in the presence of linear transformations are:

1. Do the OBDDs for multipliers remain exponential for all transformations ? Conjecture: Yes.
2. Can linear transformations be exponentially better than variable reordering ? How to construct such a function ?
3. What is the exact complexity of the problem of finding the best linear transformation ?
4. Typically, in practice the size of the input-OBDDs is a good measure for the expected running time of an algorithm. The presented material has given examples where this assumption fails. For a given OBDD-algorithm like ISOP, therefore the following principle question is of particular importance: How can (heuristically) be checked in advance whether a given input-OBDD is an easy or a hard instance for the algorithm ?

Chapter 6

Related Approaches and Final Remarks

6.1 Related Approaches

In this section we want to mention some recent research results that are directly related to our work and therefore give some additional ideas on further research directions.

Canonical TBDDs. The concept of transformed BDDs that has been introduced in Section 1.3 and which formed the basis for our Linear Sifting algorithms works with *bijective* cube transformations. In this case the canonicity of the representation is preserved.

In [GKB97] the authors tackle the problem of working with less restricted *injective* mappings onto an image space of possibly larger dimension. They show how to preserve canonicity by identifying the image set of the transformation function precisely. For this reason, ternary OBDDs are considered whose third edge represent don't care values. Don't care values are denoted by d.

Definition 6.1. *Let $f \in \mathbb{B}_n$ a Boolean function and $\tau : \mathbb{B}_n \to \mathbb{B}_m$, $m \geq n$, be an injective mapping.*

The ternary OBDD of an incompletely specified function $g : \mathbb{B}_m \to \{0, 1, d\}$ is called **canonical TBDD** *of f by the transformation function τ if g satisfies the following*

properties:

$$g(y) = \begin{cases} f(x) & \text{if there exists an } y \in \mathbb{B}_m \text{ with } y = \tau(x). \\ d & \text{otherwise.} \end{cases}$$

Based on this concept the authors investigate the larger optimization space compared with standard TBDDs, examples of suitable transformations and techniques for applying these concepts to combinational verification.

BDD Input Encoding Problem. The recent work [GVSS97] is related to the presented work on state encodings. The authors address the so-called **BDD input encoding problem**. This problem starts from a given function with arbitrary values, e.g. the next-state function of a finite state machine, and tries to minimize the size of the resulting multi-valued BDD if the values of the function are binary encoded.

Input: Functions $f_i : D \to R$, $1 \leq i \leq r$, where D and R are finite sets and $|D| = 2^n$ for some n.

Output: A bijection $e : D \to \mathbb{B}^n$ such that the size of the multi-valued (shared) BDD for $f_i' : \mathbb{B}^n \to R$, $1 \leq i \leq r$, is minimal, where the functions f_i' are defined by

$$f_i'(x) = y \iff f_i(e^{-1}(x)) = y.$$

In the mentioned paper, conditions for finding an optimal encoding in this context are discussed, as well as heuristic algorithms.

Re-encoding Techniques Using High-Level Descriptions. In [STB96, STB97] state re-encoding techniques are investigated from a different point of view. Our approach was to minimize the OBDD-size when starting from a given OBDD. In contrast to this, the authors of [STB96] start from a high-level language like Esterel. By using re-encoding techniques they try to optimize the designs generated from such high-level specifications. In particular, they aim at minimizing the number of state bits which has also a strong effect on the OBDD-sizes.

6.2 Final Remarks

We have analyzed transformation techniques for BDD-based representations and proposed new techniques for using this optimization potential. Within this framework, we have covered a variety of aspects: from theoretical investigations over

optimization algorithms, fully automated techniques like Linear Sifting, implementations issues for efficient realizations, up to applications in VLSI design.

In our opinion, the area of transformation techniques for BDD-based computer-aided design still bears additional potential. The presented open questions and related approaches have given some ideas for further investigations.

Bibliography

[ATB94] A. Aziz, S. Taziran, and R. K. Brayton. BDD variable ordering for interacting finite state machines. In *Proc. 31st ACM/IEEE Design Automation Conference (San Diego, CA)*, pages 283–288, 1994.

[BBK89] F. Brglez, D. Bryan, and K. Koźmiński. Combinational profiles of sequential benchmark circuits. In *Proc. IEEE International Symposium on Circuits and Systems (Portland, OR)*, pages 1929–1934, 1989.

[BC95] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proc. 32nd ACM/IEEE Design Automation Conference (San Francisco, CA)*, pages 535–541, 1995.

[BCL+94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13:401–424, 1994.

[BD95] B. Becker and R. Drechsler. How many decomposition types do we need ? In *Proc. European Design and Test Conference*, pages 438–443, 1995.

[BF85] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran. In *Proc. IEEE International Symposium on Circuits and Systems (Portland, OR)*, 1985.

[BFG+93] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. IEEE International Conference on Computer-Aided Design (Santa Clara, CA)*, pages 188–191, 1993.

[BHS+96] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, et al. VIS: A system for verification and synthesis. In *Proc. Computer-Aided Verification '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer, 1996.

[BLW95] B. Bollig, M. Löbbing, and I. Wegener. Simulated annealing to improve variable orderings for OBDDs. In *International Workshop on Logic Synthesis (Granlibakken, CA)*, 1995.

[BMS95] J. Bern, Ch. Meinel, and A. Slobodová. OBDD-based Boolean manipulation in CAD beyond current limits. In *Proc. 32nd ACM/IEEE Design Automation Conference (San Francisco, CA)*, pages 408–413, 1995.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. 27th ACM/IEEE Design Automation Conference (Orlando, FL)*, pages 40–45, 1990.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C–35:677–691, 1986.

[Bry91] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, C–40:205–213, 1991.

[Bry92] R. E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[Bry95] R. E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proc. IEEE International Conference on Computer-Aided Design (San José, CA)*, pages 236 – 243, 1995.

[BW96] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45:993–1002, 1996.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proc. Workshop on Automatic Verification Methods for Finite State Machines*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer, 1989.

[CCC+92] G. Cabodi, P. Camurati, F. Corno, S. Gai, S. Prinetto, and M. Sonza Reorda. A new model for improving symbolic product machine traversal. In *Proc. 29th ACM/IEEE Design Automation Conference (Anaheim, CA)*, pages 614–619, 1992.

[CM95] O. Coudert and J. C. Madre. The implicit set paradigm: A new approach to finite state system verification. *Formal Methods in System Design*, 6(2):133–145, 1995.

[Cou94] O. Coudert. Two-level logic minimization: An overview. *INTEGRATION, the VLSI journal*, 17:97–140, 1994.

[DBG95] R. Drechsler, B. Becker, and N. Göckel. A genetic algorithm for variable ordering of OBDDs. In *International Workshop on Logic Synthesis (Granlibakken, CA)*, 1995.

[DST+94] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient representation and manipulation of switching functions based on ordered

Kronecker functional decision diagrams. In *Proc. 31st ACM/IEEE Design Automation Conference (San Diego, CA)*, pages 415–419, 1994.

[FKB95] M. Fujita, Y. Kukimoto, and R. K. Brayton. BDD minimization by truth table permutations. In *International Workshop on Logic Synthesis (Granlibakken, CA)*, 1995.

[FMK91] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proc. European Design Automation Conference (Amsterdam)*, pages 50–54, 1991.

[FS90] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, 39:710–713, 1990.

[GDN92] A. Ghosh, S. Devadas, and A. R. Newton. *Sequential Logic Testing and Verification.* Kluwer Academic Publishers, 1992.

[GJ78] M. R. Garey and M. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1978.

[GKB97] E. I. Goldberg, Y. Kukimoto, and R. K. Brayton. Canonical TBDDs and their application to combinational verification. In *International Workshop on Logic Synthesis (Granlibakken, CA)*, 1997.

[GVSS97] W. Gosti, T. Villa, A. Saldanha, and A. L. Sangiovanni-Vincentelli. An exact formulation of the BDD input encoding problem. In *International Workshop on Logic Synthesis (Granlibakken, CA)*, 1997.

[HMM85] S. L. Hurst, D. M. Miller, and J. C. Muzio. *Spectral Techniques in Digital Logic.* Academic Press, 1985.

[HS96] J. P. Hansen and M. Sekine. Synthesis by spectral translation using Boolean decision diagrams. In *Proc. 33th ACM/IEEE Design Automation Conference (Las Vegas, NV)*, pages 248–253, 1996.

[ISY91] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on the exchanges of variables. In *Proc. IEEE International Conference on Computer-Aided Design (Santa Clara, CA)*, pages 472–475, 1991.

[Kar88] K. Karplus. Representing Boolean functions with if-then-else dags. Technical Report UCSC-CRL-88-28, Computer Engineering, University of California at Santa Cruz, 1988.

[Koh78] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw Hill, 1978.

[Kri94] S. Krischer. *Méthodes de vérification de circuits digitaux.* PhD thesis, Institut Nationale Polytechnique de Lorraine, Nancy, 1994.

[KSR92] U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. In *Proc. European Design Automation Conference*, pages 43–47, 1992.

[LL92] H.-T. Liaw and C.-S. Lin. On the OBDD-representation of general Boolean functions. *IEEE Transactions on Computers*, 41:661–664, 1992.

[Lon92] D. Long. Efficient implementation of an OBDD package, 1992.

[LPV94] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula. EVBDD-based algorithms for integer linear programming, spectral transformation and function decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13:959–975, 1994.

[LS92] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proc. 29nd ACM/IEEE Design Automation Conference (Anaheim, CA)*, pages 608–613, 1992.

[MB88] J.-C. Madre and J.-P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proc. 25th ACM/IEEE Design Automation Conference (Anaheim, CA)*, pages 205–210, 1988.

[Mei89] Ch. Meinel. *Modified Branching Programs and Their Computational Power*, volume 370 of *Lecture Notes in Computer Science*. Springer, 1989. Reprinted by World Publishing Corporation, Beijing, 1991.

[MGS97] S. Manne, D. C. Grunwald, and F. Somenzi. Remembrance of thing past: Locality and memory in BDDs. In *Proc. 34th ACM/IEEE Design Automation Conference (Anaheim, CA)*, 1997.

[Mic94] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[Min93] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. 30th ACM/IEEE Design Automation Conference (Dallas, TX)*, pages 272–277, 1993.

[Min96a] S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD.* Kluwer Academic Publishers, 1996.

[Min96b] S. Minato. Fast factorization method for implicit cube representation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15:377–384, 1996.

[Mor70] E. Morreale. Recursive operators for prime implicant and irredundant normal form determination. *IEEE Transactions on Computers*, C–19:504–509, 1970.

[MS94] Ch. Meinel and A. Slobodová. On the complexity of constructing optimal ordered binary decision diagrams. In *Proc. Mathematical Foundations in Computer Science*, volume 841 of *Lecture Notes in Computer Science*, pages 515–524, 1994.

[MS97] Ch. Meinel and A. Slobodová. A unifying theoretical background for some BDD-based data structures. *Formal Methods in System Design*, 11:1–15, 1997.

[MST97] Ch. Meinel, F. Somenzi, and T. Theobald. Linear sifting of decision diagrams. In *Proc. 34th ACM/IEEE Design Automation Conference (Anaheim, CA)*, pages 202–207, 1997.

[MT96] Ch. Meinel and T. Theobald. Local encoding transformations for optimizing OBDD-representations of finite state machines. In *Proc. International Conference on Formal Methods in Computer-Aided Design (Palo Alto, CA)*, volume 1166 of *Lecture Notes in Computer Science*, pages 404–418. Springer, 1996.

[MT97a] Ch. Meinel and T. Theobald. Geordnete binäre Entscheidungsgraphen und ihre Bedeutung im rechnergestützten Entwurf hochintegrierter Schaltkreise (Ordered binary decision diagrams and their significance in computer-aided design of highly integrated circuits). To appear in Informatik-Spektrum (in German), 1997.

[MT97b] Ch. Meinel and T. Theobald. On the influence of the state encoding on OBDD-representations of finite state machines. To appear at Mathematical Foundations in Computer Science (Bratislava, Slovakia), 1997.

[MWBS88] S. Malik, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. IEEE International Conference on Computer-Aided Design (Santa Clara, CA)*, pages 6–9, 1988.

[NJFS96] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partioned OBDDs – a compact, canonical and efficiently manipulable representation for Boolean functions. In *Proc. IEEE International Conference on Computer-Aided Design (San José, CA)*, 1996.

[PS95] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Proc. IEEE International Conference on Computer-Aided Design (San José, CA)*, pages 74–77, 1995.

[PSP94] S. Panda, F. Somenzi, and B. F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *Proc. IEEE International Conference on Computer-Aided Design*, pages 628–631, 1994.

[RAB+95] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *International Workshop on Logic Synthesis (Granlibakken, CA)*, 1995.

[RS95] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proc. IEEE International Conference on Computer-Aided Design (San José, CA)*, 1995.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. IEEE International Conference on Computer-Aided Design (Santa Clara, CA)*, pages 42–47, 1993.

[Sen96] E. M. Sentovich. A brief study of BDD package performance. In *Proc. International Conference on Formal Methods in Computer-Aided Design (Palo Alto, CA)*, volume 1166 of *Lecture Notes in Computer Science*, pages 389–403. Springer, 1996.

[Sha49] C. E. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28:59–98, 1949.

[Sha93a] A. Shamir. Efficient signature schemes based on birational permutations. In *Advances in Cryptology: Proceedings of CRYPTO '93 (Santa Barbara, CA)*, volume 773 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1993.

[Sha93b] A. Shamir. On the generation of multivariate polynomials which are hard to factor. In *25th ACM Symposium on Theory of Computing*, pages 796–804, 1993.

[Som96] F. Somenzi. *CUDD: Colorado University Decision Diagram Package.* ftp://vlsi.colorado.edu/pub/, 1996.

[SRBS96] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. High performance BDD package by exploiting menory hierarchy. In *Proc. 33rd ACM/IEEE Design Automation Conference (Las Vegas, NV)*, pages 635–640, 1996.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, et al. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, Electronics Research Labs, Univ. of California, Berkeley, 1992.

[SSM+92] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proc. International Conference on Computer Design (Cambridge, MA)*, pages 328–333, 1992.

[STB96] E. M. Sentovich, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. In *Proc. IEEE International Conference on Computer-Aided Design (San José, CA)*, pages 428–435, 1996.

[STB97] E. M. Sentovich, H. Toma, and G. Berry. Efficient latch optimization using exclusive sets. In *Proc. 34th ACM/IEEE Design Automation Conference (Anaheim, CA)*, pages 8–11, 1997.

[SW93] D. Sieling and I. Wegener. NC-algorithms for operations on binary decision diagrams. *Parallel Processing Letters*, 3:3–12, 1993.

[The95] T. Theobald. How to break Shamir's asymmetric basis. In *Advances in Cryptology: Proceedings of CRYPTO '95 (Santa Barbara, CA)*, volume 963 of *Lecture Notes in Computer Science*, pages 136–147. Springer, 1995.

[THY93] S. Tani, K. Hamaguchi, and S. Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In *Proc. International Symposium on Algorithms and Computation '93*, volume 762 of *Lecture Notes in Computer Science*, pages 389–398. Springer, 1993.

[Weg87] I. Wegener. *The Complexity of Boolean Functions.* John Wiley & Sons and Teubner-Verlag, 1987.

[Yan91] S. Yang. Logic synthesis and optimization benchmarks user guide, version 3.0. Technical report, Microelectronics Center of North Carolina, Research Triangle Park, NC, 1991.