Masaryk University
Faculty of Informatics

Randomized Symbolic Model Checking

Master's Thesis

Vojtěch Řehák

April 2002

Declaration

I declare that this thesis was composed by myself and the presented work is of my own if not stated otherwise. All the used sources and literature are cited with a complete reference to the corresponding source.

Acknowledgment

I would like to thank RNDr. Ivana Černá, CSc., the supervisor of my thesis, for her constant willingness to listen, discuss, and help.

I would also like to thank all members of ParaDiSe laboratory, namely Doc. RNDr. Mojmír Křetínský, CSc. for his Easter correction, Tomáš Hanžl for his help with one proof, and Jan Strejček for his continual assistance. Thanks go also to Katka for her willingness to help me with my dreadful English. Last but not least, I want to thank my parents for their constant support during my studies.

Abstract

This thesis studies improvements of the symbolic model checking algorithm. We investigated possible substitutions of the currently used OBDD data structure. Two data structures, $\oplus$-OBDDs and BEDs, are tested. The asset of this thesis consists of summarization of the theoretical knowledge, introduction of merged $\oplus$-OBDD, practical implementation of $\oplus$-OBDDs into the NuSMV model checker, and interpretation of the acquired facts.

Key words

Data structures, OBDD, $\oplus$-OBDD, BED, symbolic model checking, randomized algorithms.

Contents

Chapter 1

Introduction

Nowadays, hardware and software systems are widely used in applications where failure is unacceptable: electronic commerce, air traffic control system, medical instruments, and many others. Hence, the methods for validating this systems are in great demand. The principal validation methods for complex systems are simulation, testing, deductive verification, and model checking. We focus on the model checking approach.

Model checking [CGP99] is an automatic technique for verifying finite state concurrent systems: in this approach, properties are expressed in a temporal logic and systems are modeled as transition systems. A model checker accepts two inputs, a transition system and a temporal formula, and returns ”true” if the system satisfies the formula and ”false” otherwise. There are several approaches to solving the model checking problem, namely automata based, games, symbolic, structural, etc. In this thesis we concentrate on the symbolic approach. Symbolic model checking algorithm is based on manipulations with sets of states of the transition system where sets of states are represented by *Ordered Binary Decision Diagrams* [Bry86] (OBDDs).

The basic model checking problem is a *state explosion problem*. The state explosion problem is due to the fact that the number of states can be exponential in the size with respect to the description of the system. The primary cause of this explosion is the parallel composition of interacting processes. Hence, the basic problem is the space complexity in the model checking algorithms. We investigate possibilities how to reduce the space complexity of the model checking algorithm. This thesis investigates the possibility of using a novel data structures in symbolic algorithms.

OBDD

At first we introduce currently used data structure OBDD. It is a canonical form representation for boolean formulas. OBDD data structure is very feasible for computing boolean functions, co-factoring, and other operations which are needed for the symbolic model checking computation. OBDDs are much more succinct than conjunctive normal form and disjunctive normal form but there are still boolean formulas which have OBDDs of exponential size with respect to the number of variables.

$\oplus$-OBDD

Secondly, we discuss advantages and disadvantages of $\oplus$-OBDDs. The $\oplus$-OBDD data structure is an extension of OBDD. In addition to OBDD, there are $\oplus$-nodes available in $\oplus$-OBDD data structure. $\oplus$-OBDDs are more space-efficient (sometimes even exponentially) than OBDDs. $\oplus$-OBDDs preserve the OBDD property of efficient manipulation. Applying a boolean operation, quantification, and composition have the same complexity as in the case of OBDDs. Even better, the boolean functions *exclusive or* (XOR), the *logical equivalence*, and the *negation* can be performed in constant time. In contrast to OBDD representation, the $\oplus$-OBDD is not canonical. So, the equivalence test for $\oplus$-OBDD is much more difficult. The recently introduced deterministic equivalence test given in [Waa97] can be easily adapted to $\oplus$-OBDDs, but it performs only in high polynomial degree execution time. Hence, we focus on a probabilistic approach. The equivalence test for $\oplus$-OBDD is within co-RP [GM93b], so there is a polynomial probabilistic equivalence test with one-side error which can be amplified. $\oplus$-OBDDs are the greatest extension of OBDDs which allows a polynomial probabilistic equivalence test. According to [GM93a], the equivalence test for ω-OBDDs, $\omega \in \{\{\vee\}, \{\wedge\}, \{\vee, \wedge\}\}$ is co-NP-complete.

BED

The third discussed data structure is a Boolean Expression Diagram [AH97] (BED). This structure is appropriate because it is high compressible. BEDs have a linear size to a boolean formula. Basic problem of computation with BEDs is induced by the fact that BEDs have no canonical form. Exactly, the equivalence test for BEDs is co-NP-complete problem.

Plan of the Thesis

The rest of the thesis is structured as follows. In Chapter 2 we define the Computation Tree Logic (CTL) and the symbolic model checking algorithm. In the next chapter we introduce all three discussed data structures. Chapter 4 presents specifications of the data structures in more details. In addition, this chapter shows algorithms for all operations which are needed for the symbolic model checking. Chapter 5 compares the results achieved by using all the different data structures mentioned above. This comparison has two parts. The first one is a theoretical conclusion based on results from the previous chapter. The second comparison is based on results of a practical implementation.

Chapter 2

CTL and Symbolic Model Checking

2.1 Computation Tree Logic

Computation Tree Logic (CTL) is a restricted subset of CTL* in which each of the temporal operators X, F, G, U, and R must be immediately preceded by a path quantifier.

Syntax

CTL formulas are defined inductively. There are two types of CTL subformulas: *state formulas* (which are true in a specific state) and *path formulas* (which are true along a specific path). Whole CTL formula must be a state formula. Let AP be the set of atomic proposition names. The syntax of state formulas is given by the following rules:

- If $p \in AP$, then p is a state formula.
- If f and g are state formulas, then $\neg f$, $f \vee g$ and $f \wedge g$ are state formulas.
- If f is a path formula, then Ef and Af are state formulas.

One additional rule is needed to specify the syntax of path formulas:

- If f and g are *state* formulas, then Xf, Ff, Gf, fUg, and fRg are path formulas.

Semantics

We define the semantics of CTL with respect to a Kripke structure. Recall that a Kripke structure represents a finite-state system. A Kripke structure M is a triple $\langle S, R, L\rangle$, where S is the set of states; $R \subseteq S \times S$ is the transition relation, which must be *total*; and $L : S \to 2^{AP}$ is a function that labels each state with a set of atomic propositions which are true in that state. A *path* π in M is an infinite sequence of states, $\pi = s_0, s_1, s_2, \ldots$, such that for all $i \geq 0$ is (s_i, s_{i+1}) in the transition relation R.

We use π_i to denote the *suffix* of π starting at s_i. If f is a state formula, the notation $M, s \models f$ means that f holds at the state s in the Kripke structure M. Similarly, if f is a path formula, $M, \pi \models f$ means that f holds along the path π in the Kripke structure M. The relation $\models$ is defined inductively as follows (assuming that f_1 and f_2 are state formulas and g_1 and g_2 are path formulas):

1. $M, s \models p \Leftrightarrow p \in L(s)$.
2. $M, s \models \neg f_1 \Leftrightarrow M, s \not\models f_1$.
3. $M, s \models f_1 \vee f_2 \Leftrightarrow M, s \models f_1$ or $M, s \models f_2$.
4. $M, s \models f_1 \wedge f_2 \Leftrightarrow M, s \models f_1$ and $M, s \models f_2$.
5. $M, s \models Eg_1 \Leftrightarrow$ there is a path π from s such that $M, \pi \models g_1$.
6. $M, s \models Ag_1 \Leftrightarrow$ for every path π starting from s, $M, \pi \models g_1$.
7. $M, \pi \models f_1 \Leftrightarrow s$ is the first state of π and $M, s \models f_1$
8. $M, \pi \models \neg g_1 \Leftrightarrow M, \pi \not\models g_1$.
9. $M, \pi \models g_1 \vee g_2 \Leftrightarrow M, \pi \models g_1$ or $M, \pi \models g_2$.
10. $M, \pi \models g_1 \wedge g_2 \Leftrightarrow M, \pi \models g_1$ and $M, \pi \models g_2$.
11. $M, \pi \models Xg_1 \Leftrightarrow M, \pi^1 \models g_1$.
12. $M, \pi \models Fg_1 \Leftrightarrow$ there exists $k \geq 0$ such that $M, \pi^k \models g_1$.
13. $M, \pi \models Gg_1 \Leftrightarrow$ for all $i \geq 0$, $M, \pi^i \models g_1$.
14. $M, \pi \models g_1 U g_2 \Leftrightarrow$ there exists $k \geq 0$ such that $M, \pi^k \models g_2$ and
    for all $0 \leq j \leq k$, $M, \pi^j \models g_1$.
15. $M, \pi \models g_1 R g_2 \Leftrightarrow$ for all $j \geq 0$, if for every $i \leq j$ $M, \pi^i \not\models g_1$ then
    $M, \pi^j \models g_2$.

It is easy to see that the operators $\neg$, $\vee$, EX, EG, and EU are sufficient to express any other CTL formula. [CGP99]

2.2 Symbolic Model Checking

The *model checking problem* is formulated as follows. Given a Kripke structure $M = \langle S, R, L \rangle$ and a temporal logic formula f expressing a desired specification, find the set S_f of all states in S that satisfy f:

$$S_f = \{s \in S \mid M, s \models f\}.$$

Model checking is called symbolic if a representation of sets of states is based on boolean formulas. We explain how this representation is performed in the next chapter. S_f is computed recursively with respect to the structure of formula f. Recall that any CTL formula can be expressed in *atomic propositions* and terms of $\neg$, $\vee$, EX, EU, and EG. So, we have to be able to handle six cases, depending on whether f is atomic or has one of the following forms:$\neg f_1$, $f_1 \vee f_2$, EXf_1, $E[f_1 U f_2]$, or EGf_1.

- If f is an atomic proposition, then we construct set S_f directly.
- If f is $\neg f_1$, then S_f is the complementary set of S_{f_1}.
- If f is $f_1 \vee f_2$, then S_f is the union of sets S_{f_1} and S_{f_2}.
- If f is EXf_1, then the set S_f is constructed as the *relational product*. Recall that relational product of a binary relation A and an unary relation B is an unary relation

  $$A \circ B = \{(x) \mid \exists y.(x, y) \in A \wedge (y) \in B\}.$$

  We employ it as the set of states can be viewed as an unary relation on S. Then we can construct relational product of the transition relation R and the set S_{f_1}.

  $$R \circ S_{f_1} = \{x \mid \exists y.(x, y) \in R \wedge y \in S_{f_1}\}.$$

  Result of the relational product $R \circ S_{f_1}$ is the required set S_{EXf_1}.
- If f is EGf_1, then we construct S_f as the *greatest fixpoint*

  $$EGf_1 = \nu Z.f_1 \wedge EXZ.$$

  A computation of the greatest fixpoint can be performed according to the following algorithm:

funct $EG(S_{f_1}$: *SetOfStates*$)$: *SetOfStates*
  $S1 := S;$ /∗ S is the set of all states ∗/
  $S2 := S_{f_1} \cap S_{EX\ S1};$
  while $(S1 \neq S2)$ **do**
    $S1 := S2;$

$S2 := S_{f_1} \cap S_{EX\ S1};$

od

$return(S1);$

end

- If f is $E[f_1 U f_2]$, then we construct S_f as the *least fixpoint*.

$$E[f_1 U f_2] = \mu Z.f_2 \vee (f_1 \wedge EXZ)$$

The algorithm for computing EU is analogous to the EG's one. There are only two differences, the initial value of $S1$ is $\emptyset$ and the transformation is performed as $S2 := S_{f_2} \cup (S_{f_1} \cap S_{EX\ S1})$.

For the correctness of algorithms just given see [CGP99].

Chapter 3

Data Structures

In this chapter we present definitions of some data structures which are suitable for storing sets of states of Kripke structure. Each Kripke structure can be amended so as its labeling function $L : S \to 2^{AP}$ is injective. If L is injective, then each state s can be represented by $L(s)$ unambiguously. $L(s)$ is a subset of AP and each subset L of AP can be represented by a boolean formula $B_L = \bigwedge_{p \in L} p \wedge \bigwedge_{p \in AP \setminus L} \neg p$ unambiguously. Hence, each state s can be represented by a boolean formula $B_{L(s)}$ unambiguously. It is easy to see that each set $\{s_1, \ldots, s_n\}$ of states can be represented by a boolean formula $B_{L(s_1)} \vee \cdots \vee B_{L(s_n)}$ unambiguously.

In the following we show how to represent a boolean function by a decision diagram. We will introduce OBDD, $\oplus$-OBDD, and BED. Unambiguous representation of boolean formula by a boolean function is obvious.

3.1 Ordered Binary Decision Diagram

In this section we define widely used data structure *Ordered Binary Decision Diagram* (OBDD). This data structure is so called function graph. OBDD is a canonical representation of boolean functions. OBDDs are often substantially more compact than normal forms such as conjunctive normal form and disjunctive normal form, and they can be manipulated very efficiently.

The following definition is taken from [Bry86].

3.1.1 Syntax

A *Binary Decision Diagram* (BDD) G over a set $X_n = \{x_1, \ldots, x_n\}$ of boolean variables is a directed acyclic connected graph $G = (V, E)$.

V consists of terminal nodes with out-degree 0 and of non-terminal nodes with out-degree 2. The two intermediate successors of node v are denoted $low(v)$ and $high(v)$, respectively. Edges from v to $low(v)$ and $high(v)$

are labeled as 0-$edge$ and 1-$edge$, respectively. In the following let $l(v)$ denote the label of the node $v \in V$.

There are two types of nodes in BDD.

- A *terminal node* v has a label $l(v) \in \{0, 1\}$
- A *variable (branching) node* v has a label $l(v) = x_i$ ($x_i \in X_n$) and two successors $low(v), high(v) \in V$.

A node with in-degree 0 is the *root*.

A BDD is *free* if each variable is encountered at most once on each path in the BDD from the root to a terminal node. A BDD is *ordered* if it is free and the variables are encountered in the same order on each path in the BDD from the root to a terminal node. A OBDD is *reduced* if it fulfills these three conditions:

- There are at most two terminal nodes, one with the label 0 and one with the label 1.
- If v is a non-terminal node, then $low(v) \neq high(v)$.
- If v and u are non-terminal nodes, then $low(u) = low(v) \wedge high(u) = high(v) \wedge l(u) = l(v)$ implies $u = v$.

Two OBDDs are shown in Figure 3.1. The first is not in the reduced form and the second is reduced.

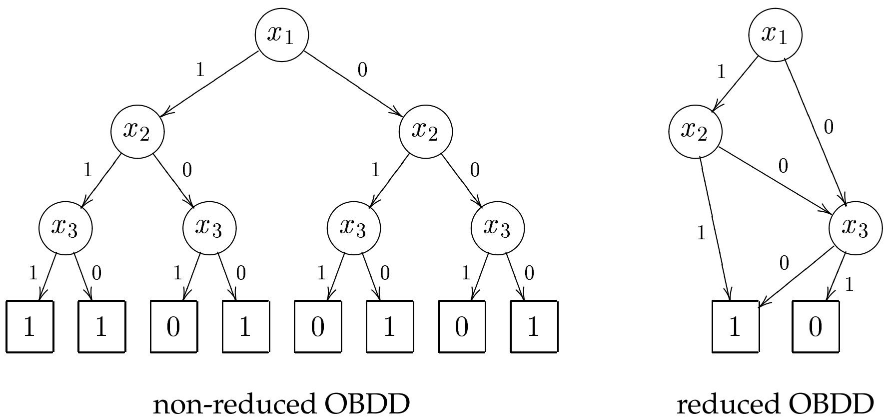


Figure 3.1: Two OBDDs representing the boolean function $(x_1 \wedge x_2) \vee \neg x_3$.

3.1.2 Semantics

Each node v of OBDD represents a boolean function $f_v : \{0, 1\}^n \to \{0, 1\}$. The definition of this function is given recursively as follows:

- If v is a *terminal node*, then $f_v(x_1, \ldots, x_n) = l(v)$.
- If v is a *variable node* with $l(v) = x_i$, then $f_v(x_1, \ldots, x_n) = (\neg x_i \wedge f_{low(v)}(x_1, \ldots, x_n)) \vee (x_i \wedge f_{high(v)}(x_1, \ldots, x_n))$.

A OBDD G with a root v represents a boolean function $f_v(x_1, \ldots, x_n)$.

There is exactly one reduced OBDD to represent each boolean function. Hence, a representation of each boolean function by a reduced OBDD is canonical.

3.2 $\oplus$-Ordered Binary Decision Diagram

A $\oplus$-OBDD [GM93b] (called also Mod2-OBDD or Parity-OBDD) is an extension of OBDD data structure. In addition to OBDD, there are $\oplus$-nodes available. This innovation can lead to more succinct representation. There are boolean functions which have exponential size optimal OBDDs but polynomial size $\oplus$-OBDDs. Size of BDD is equal to the number of nodes. There is no OBDD such that optimal $\oplus$-OBDD representation is bigger because each OBDD is a $\oplus$-OBDD too.

The basic disadvantage of $\oplus$-OBDD representation is the loss of a canonical representation.

3.2.1 Syntax

A $\oplus$-BDD P over a set $X_n = \{x_1, \ldots, x_n\}$ of boolean variables is a directed acyclic connected graph $P = (V, E)$.

V consists of terminal nodes with out-degree 0 and of non-terminal nodes with out-degree 2. The two intermediate successors of node v are denoted $low(v)$ and $high(v)$, respectively. Edges from v to $low(v)$ and $high(v)$ are labeled as 0-$edge$ and 1-$edge$, respectively. In the following let $l(v)$ denote the label of the node $v \in V$.

There are three types of nodes in $\oplus$-OBDD.

- A *terminal node* v has a label $l(v) \in \{0, 1\}$.
- A *variable (branching) node* has a label $l(v) = x_i$ ($x_i \in X_n$) and two successors $low(v), high(v) \in V$.
- A *$\oplus$-node* has a label $l(v) = \oplus$ and two successors $low(v), high(v) \in V$.

A node with in-degree 0 is the *root*.

Free and *ordered* $\oplus$-BDD are defined in the same way as free and ordered BDD.

A $\oplus$-OBDD is *reduced* if it fulfills the following four conditions:

- There are at most two terminal nodes, one with the label 0 and one with the label 1.

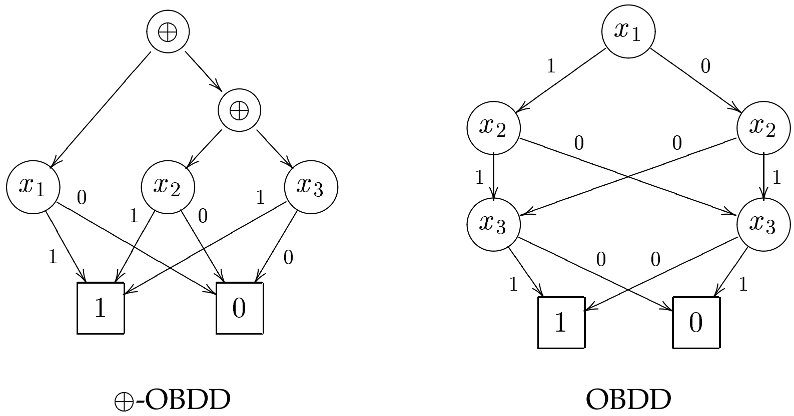


Figure 3.2: Two diagrams representing the boolean function $x_1 \oplus x_2 \oplus x_3$.

- If v is a non-terminal node, then $low(v) \neq high(v)$.
- If v and u are non-terminal nodes, then $low(u) = low(v) \wedge high(u) = high(v) \wedge l(u) = l(v)$ implies $u = v$.
- If v is a $\oplus$-node, then neither $low(v)$ nor $high(v)$ is the terminal node with the label 0.

In contrast to OBDD, a representation of each boolean function by a reduced $\oplus$-OBDD is not canonical. These four rules serve only for a reduction of the size and do not provide any canonicity. There are four equivalent $\oplus$-OBDDs in Figure 3.3 and each of these four $\oplus$-OBDDs is reduced.

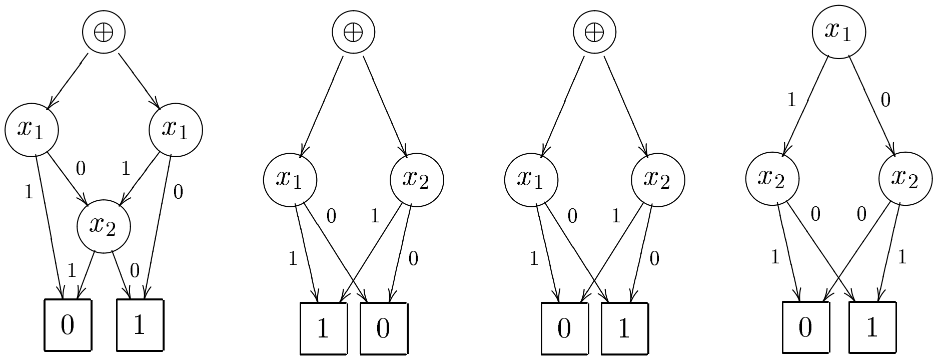


Figure 3.3: Four reduced $\oplus$-OBDDs representing the boolean function $x_1 \oplus x_2$.

3.2.2 Semantics

Each node v of $\oplus$-OBDD represents a boolean function $f_v : \{0,1\}^n \to \{0,1\}$. The definition of this function is given recursively as follows:

- If v is a *terminal node*, then $f_v(x_1, \ldots, x_n) = l(v)$.
- If v is a *branching (variable) node* with $l(v) = x_i$, then $f_v(x_1, \ldots, x_n) = (\neg x_i \wedge f_{low(v)}(x_1, \ldots, x_n)) \vee (x_i \wedge f_{high(v)}(x_1, \ldots, x_n))$.
- If v is a $\oplus$*-node*, then $f_v(x_1, \ldots, x_n) = f_{low(v)}(x_1, \ldots, x_n) \oplus f_{high(v)}(x_1, \ldots, x_n)$, where $\oplus$ is a boolean exclusive or (XOR) operator.

A $\oplus$-OBDD with a root v represents a boolean function $f_v(x_1, \ldots, x_n)$.

3.3 Boolean Expression Diagram

This section presents another extension of OBDDs called *Boolean Expression Diagrams* (BEDs) [AH97]. In addition to OBDD, there are operator nodes for each of 16 binary boolean operations in BED. The size of BED is linear with respect to the size of boolean formula.

3.3.1 Syntax

A BED B over a set $X_n = \{x_1, \ldots, x_n\}$ of boolean variables is a directed acyclic connected graph $B = (V, E)$.

V consists of terminal nodes with out-degree 0 and of non-terminal nodes with out-degree 2. The two immediate successors of node v are denoted $low(v)$ and $high(v)$, respectively. Edges from v to $low(v)$ and $high(v)$ are labeled as 0-*edge* and 1-*edge*, respectively. In the following let $l(v)$ denote the label of the node $v \in V$.

There are three types of nodes in BED.

- A *terminal node* v has a label $l(v) \in \{0,1\}$.
- A *variable node* v has a label $l(v) = x_i$ ($x_i \in X_n$) and two successors $low(v), high(v) \in V$.
- An *operator node* v has a label $l(v)$ equal to a binary boolean operator, and two successors $low(v), high(v) \in V$.

A node with in-degree 0 is the *root*.

Free and *ordered* BED are defined in the same way as free and ordered BDD.

A BED is *reduced* if it fulfills these four conditions:

- There are at most two terminal nodes, one with the label 0 and one with the label 1.
- If v is a non-terminal node, then $low(v) \neq high(v)$.
- If v and u are non-terminal nodes, then $low(u) = low(v) \wedge high(u) = high(v) \wedge l(u) = l(v)$ implies $u = v$.
- If v is an operator node, then neither $low(v)$ nor $high(v)$ is the terminal node.

Like $\oplus$-OBDD, reduced BED does not provide a canonical representation.

3.3.2 Semantics

Each node v of BED represents a boolean function $f_v : \{0,1\}^n \to \{0,1\}$. The definition of this function is given recursively as follows:

- If v is a *terminal node*, then $f_v(x_1, \ldots, x_n) = l(v)$.
- If v is a *variable node* with $l(v) = x_i$, then $f_v(x_1, \ldots, x_n) = (\neg x_i \wedge f_{low(v)}(x_1, \ldots, x_n)) \vee (x_i \wedge f_{high(v)}(x_1, \ldots, x_n))$.
- If v is an *operator node*, then $f_v(x_1, \ldots, x_n) = f_{low(v)}(x_1, \ldots, x_n)\ "l(v)"\ f_{high(v)}(x_1, \ldots, x_n)$, where $"l(v)"$ is a boolean operation which corresponds with the label $l(v)$.

A BED with a root v represents a boolean function $f_v(x_1, \ldots, x_n)$.

Chapter 4

Operations on Data Structures and Their Implementation

In each section of this chapter we give details of an implementation, which lead to more economical storing of data structures. Then we show how to implement operations which have been used in Section 2.2, namely:

- *creation* `CREATE` of a new set S_f of states, where f is an atomic proposition,
- *complementation* `NEG` of a set,
- *union* `UNION` of two sets,
- *relational product* `RELP` for computation of EX, and
- *equivalence test* `EQU` for fixpoint computation. `EQU` tests if the representations of sets are equivalent. It means that it is an equality test of the represented sets.

4.1 OBDD

4.1.1 Specifications

The basic idea of reducing the size of OBDD even more is to use the so-called *complemented edges*. Each edge has an extra bit, the compl-bit. Edge is *regular* if the compl-bit is not set on. If the compl-bit is set on, then the function represented by the following node is negated. Pointers to the roots representing outputs may carry a compl-bit because we need to complement whole OBDDs, too. If we want to preserve canonicity, we restrict the use of complemented edges by the following two rules:

- OBDD contains only one terminal node. This node has the label 1.

- 1-$edges$ must be regular.

Starting from an OBDD with arbitrary compl-bits, we can easily obtain an OBDD fulfilling given restrictions. The algorithm works bottom-up.

The terminal node with the label 0 is obtained by complementation of the terminal node with the label 1. If the compl-bit of the 1-*edge* is set on, then this compl-bit, compl-bit of the 0-*edge*, and the compl-bits of all edges reaching this node are complemented.

In what follows we employ OBDDs only in the reduced form with complemented edges.

4.1.2 Algorithms and Their Complexities

In practice we need to guarantee reduced form of all used OBDDs. Therefore each creation of a new node must be performed carefully. At first we must check all restrictions given in the definition of reduced OBDD and restrictions on use of complemented edges. Then we must determine whether this node already exists. A widely used way how to solve this problem is to use a *unique table*. The unique table is a hash table of all used nodes. The new node is created only if it already does not exist in the unique table. We denote this careful creation of a new node as `NewNode`. The time of procedure `NewNode` is in $O(1)$ because correction of an OBDD node observes reduced form of its successors. We suppose that looking for a node in the unique table takes constant time. An algorithm to transform arbitrary OBDD into a reduced OBDD is not needed because `NewNode` keeps the reduced representation.

Procedures `CREATE`, `NEG`, `UNION`, `RELP`, and `EQU` are needed to implement symbolic model checking algorithm.

There is a boolean variable x_p corresponding to each atomic proposition p in symbolic model checking algorithm. `CREATE(p)` returns OBDD G where root r of G has the label x_p, 0-*edge* of r points to the terminal node 0 (complemented edge to terminal node 1), and 1-*edge* of r points to the terminal node 1.

Negation `NEG` of OBDD is easy because we use a mechanism of complemented edges.

`UNION` of two OBDDs $G1$ and $G2$ results in an OBDD G, where represented boolean function fulfills $f_G = f_{G1} \vee f_{G2}$. An algorithm for disjunction is based on *Shannon expansion rule*:

$$f = (\neg x \wedge f|_{x \leftarrow 0}) \vee (x \wedge f|_{x \leftarrow 1}).$$

Each boolean function $\star$ can be solved recursively as:

$$f \star g = (\neg x \wedge (f|_{x \leftarrow 0} \star g|_{x \leftarrow 0})) \vee (x \wedge (f|_{x \leftarrow 1} \star g|_{x \leftarrow 1})),$$

where $f|_{x\leftarrow 0}$ and $f|_{x\leftarrow 0}$ are co-factors of f for $x = 0$ and $x = 1$, respectively. If x is a label of the root of an OBDD F, then co-factors $F|_{x\leftarrow 0}$ and $F|_{x\leftarrow 1}$ are equal to $low(F)$ and $high(F)$, respectively. Hence, the best variable for co-factoring is the topmost variable. In addition, we use a hash table *ResultCache* to improve the performance of `UNION`. The *ResultCache* maps input OBDDs F, G to the result OBDD returned by `UNION`(F, G) once this result has been computed.

`UNION` is implemented as follows:

funct `UNION`$(F, G : \mathit{OBDD}) : \mathit{OBDD}$
  if $F = 1 \vee G = 1$ **then** $return(1)$; **fi**
  if $F = 0$ **then** $return(G)$; **fi**
  if $G = 0$ **then** $return(F)$; **fi**
  $/*$ Result cache checking $*/$
  if $(F, G, R) \in \mathit{ResultCache}$
    then $return(R)$;
    else
      $/*$ Co-factor based computing $*/$
      $z :=$ `Topmost(TopVariable(`F`), TopVariable(`G`));`
      $R_0 :=$ `UNION`$(F|_{z\leftarrow 0}, G|_{z\leftarrow 0})$;
      $R_1 :=$ `UNION`$(F|_{z\leftarrow 1}, G|_{z\leftarrow 1})$;
      $R :=$ `NewNode`(z, R_1, R_0);
      `InsertInResultCache`(F, G, R);
      $return(R)$;
  fi
end

With the assumption of constant time lookup and insert in the unique table and *ResultCache*, all operations in `UNION` takes constant time. `UNION` can be called at most once for each combination of nodes in the input OBDDs F and G. So the time complexity of this algorithm is $O(|F| * |G|)$, where $|F|$ is the number of nodes of the OBDD F. In practice, the typical performance is closer to the size of the resulting OBDD.

`RELP` algorithm is an implementation of the relational product

$$F \circ G = \{x \mid \exists y.(x, y) \in F \wedge y \in G\}.$$

There are three input parameters of `RELP`: a set E of common variables and two OBDDs F and G. Result of `RELP` is an OBDD R representing the boolean function f_R,

$$f_R = \exists x_1, \ldots, x_n . f_F \wedge f_G \text{ , where } x_1, \ldots, x_n \text{ are in } E.$$

Conjunction is computed in a similar way as the disjunction. A computation of an existentially quantified boolean formulas is performed according to the following rule:

$$\exists x . f = f|_{x\leftarrow 0} \vee f|_{x\leftarrow 1}.$$

The implementation of `RELP` is performed as follows.

```
funct RELP(F, G : OBDD; E : SetOfVariables) : OBDD
  if F = 0 ∨ G = 0 then return(0); fi
  if F = 1 then return(G); fi
  if G = 1 then return(F); fi
  z := Topmost(TopVariable(F), TopVariable(G));
  /* Omitting idle variables from E */
  while z ≠ Topmost(z, TopVariable(E)) do
        E := E ∖ {TopVariable(E)};
  od
  /* Result cache checking */
  if (F, G, E, R) ∈ ResultCache
    then return(R);
     else
          /* Co-factor based computing */
          R0 := RELP(F|z←0, G|z←0, E);
          R1 := RELP(F|z←1, G|z←1, E);
          if z ∈ E
            then R := UNION(R1, R0);
             else R := NewNode(z, R1, R0);
          fi
          InsertInResultCache(F, G, E, R);
          return(R);
  fi
end
```

The time complexity of `RELP` is $O(|F|^2 * |G|^2)$, because "co-factor based computing" is executed at most $|F| * |G|$ times and the time complexity of each pass is $O(|F| * |G|)$.

There is a need for testing equivalence `EQU` between two OBDDs in checking formulas EU and EG. We must compare Q and Q' after each pass through the while-loop in the fixpoint algorithm. Because of canonical representation and unique table implementation, equivalence test `EQU` can be performed in the constant time $O(1)$. The time complexity of each pass through the while-loop is polynomial.

4.2 $\oplus$-OBDD

4.2.1 Specifications

We may use complemented edges to achieve more compact $\oplus$-OBDD representation, too.

We may furthermore introduce any other rules and heuristics to make $\oplus$-OBDD representation more succinct. These rules and heuristics are based on features of the boolean operation XOR. We introduce several possible approaches. As reduced $\oplus$-OBDDs do not provide a canonical representation, there are no strictly defined rules how to obtain the optimal representation.

Complemented edges

At first we allow complemented edges. Hence, we must add the following four rules:

- $\oplus$-OBDD contains only one terminal node. This node has the label 1.
- If v is a variable node, then 1-$edge$ of v must be regular.
- If v is a $\oplus$-node, then both 0-$edge$ and 1-$edge$ must be regular.
- If v is a $\oplus$-node, then both successors are non-terminal nodes.

The first two rules are taken from OBDD representation. See previous section for more information.

If compl-bit of any outgoing edge of $\oplus$-node is set on, then we complement this compl-bit and compl-bits of all edges reaching this node.

If $\oplus$-node has a terminal successor with the label 1, then we destroy this $\oplus$-node, redirect all edges reaching this $\oplus$-node to the second successor, and complement compl-bits of all the reaching edges. Terminal successors with the label 0 are realized as a complement of the terminal node with the label 1. So, we can debug the 0 terminal successor of a $\oplus$-node according to the previous rules.

In [MS00], the third rule is presented in a different way. They allow complementation of 1-$edge$ and disallow complementation of 0-$edge$ and all edges reaching this node. We present our rule because then can be correcting algorithm performed strictly bottom-up and $\oplus$-meta-nodes can be added without changing this rule.

$\oplus$-meta-nodes

A *$\oplus$-meta-node* is a $\oplus$-node which can have more than two successors. This innovation is based on commutativity and associativity of XOR. $\oplus$-meta-nodes are introduced in [MS99]. But they demonstrate it only as an algorithmic consideration for $\oplus$-OBDD reordering. We present compact collection of rules and correcting instructions. In addition, we present partial ordering on successors of $\oplus$-meta-node.

As $\oplus$-meta-nodes are allowed, we must stiffen up the fourth rule of the rules which are added for complemented edges using.

- If v and u are adjacent successors of $\oplus$-meta-node, then v and u are different variable nodes and $l(v) \leq l(u)$.

If any successor of a $\oplus$-meta-node is a $\oplus$-meta-node, then we join these two nodes into one $\oplus$-meta-node.

If a $\oplus$-meta-node with more than two successors has a terminal node with the label 1 as a successor, then we remove this outgoing edge and complement compl-bits of all edges reaching the $\oplus$-meta-node. Reduction for $\oplus$-node with exactly two successors is written above. Two equal successors are the same as the terminal node with the label 0 for XOR operation, and are deleted according to the previous rules.

If u and v are adjacent successors of $\oplus$-meta-node and $l(v) > l(u)$, then we exchange the two successors u and v.

Merged $\oplus$-OBDD

Now we define our own restriction which leads to more succinct and almost canonical representation. We present own innovation of the fourth rule:

- If v and u are adjacent successors of $\oplus$-meta-node, then v and u are variable nodes and $l(v) < l(u)$.

We must explain how to correct $\oplus$-OBDD where $l(v) \geq l(u)$.

If $l(v) > l(u)$ then we exchange the two successors v and u.

If $l(v) = l(u)$ then successors v and u are supplanted by the new variable node w. The label $l(w)$ is equal to $l(v)$. The successor $low(w)$ is a new $\oplus$-meta-node with successors $low(v)$ and $low(u)$. The successor $high(w)$ is a new $\oplus$-meta-node with successors $high(v)$ and $high(u)$. It is easy to see that this correction can be done together for all successors with the same label.

This innovation may lead to more succinct representation. Indeed, if there are n successors with the same label, then we delete these n nodes and replace them only by three nodes. If we represent $\oplus$-meta-node as a chain of $\oplus$-nodes, then the number of nodes is preserved. Furthermore, we can find and remove more redundant subtrees, make representation more canonical, and perform some operations more easily.

The basic disadvantage of this innovation is the loss of strictly bottom-up implementation of the correcting algorithm. Correction must be performed top-down. Hence, the careful creation of a new node `NewNode` is not $O(1)$ but it has a linear time complexity. The careful creation algorithm for merged $\oplus$-OBDD is shown in the next subsection.

Because of this, we use all three types of implementation in the following: $\oplus$-OBDD with complemented edges, $\oplus$-OBDD with $\oplus$-meta-nodes, and merged $\oplus$-OBDD.

4.2.2 Algorithms and Their Complexities

The time complexity of $\texttt{NewNode}_{compl}$ for $\oplus$-OBDD with complemented edges is $O(1)$. The time complexity of $\texttt{NewNode}_{meta}$ for $\oplus$-OBDD with $\oplus$-meta-nodes is $O(|F|)$ because of the ordering on successors of $\oplus$-meta-nodes. These two procedures are simple compositions of the correcting instructions presented in previous subsection. $\texttt{NewNode}_{merged}$ for merged $\oplus$-OBDD is implemented as follows:

```
funct NewNode_merged(l : Label; 𝓕 : SetOfSuccessors) : ⊕-OBDD
  if l ≠ ⊕
    then /* Create a variable node */
         R := NewNode_compl(l, 𝓕_1, 𝓕_0);
    else /* Create a ⊕-node */
         /* Take ⊕-successors off */
         for F ∈ 𝓕 do
             if l(F) = ⊕ then 𝓕 := (𝓕 ∪ {successors of F}) ∖ {F}; fi
         od
         /* Take off variable successors with the same label */
         Z := GetSetOfTopVars(𝓕);
         for z ∈ Z from the TopMost to the BottomMost do
             𝓕_z := successors from 𝓕 with the label z;
             if |𝓕_z| ≥ 2 then R_1 := NewNode_merged(⊕, high successors of 𝓕_z);
                               R_0 := NewNode_merged(⊕, low successors of 𝓕_z);
                               R_z := NewNode_compl(z, R_1, R_0);
                               𝓕 := (𝓕 ∪ {R_z}) ∖ 𝓕_z;
             fi
         od
         /* All special corrections of merged ⊕-OBDD are done */
         R := NewNode_meta(⊕, 𝓕);
  fi
  return(R);
end
```

The time complexity of $\texttt{NewNode}_{merged}$ is linear in the sum of sizes of successors because taking off a variable successors with the same label may damage correctness of its successors.

Procedures `CREATE` and `NEG` are the same for $\oplus$-OBDD representation as for OBDD one.

A widely used way how to realize computation of `UNION` is the implementation of a `ITE` algorithm. In [MS97], there is introduced `ITE-`$\oplus$ algorithm, the `ITE` algorithm for $\oplus$-OBDDs. But there are two basic mistakes in their pseudocode. The first one is confusion between `ITE` algorithm [BRB90] and synthesis algorithm. In `ITE` algorithm, there are three decision diagrams F, G, H as input parameters and resulting diagram is equal to $(F \wedge G) \vee (\neg F \wedge H)$. It means *if* F *then* G *else* H. In [BRB90], there is shown how to compute each boolean operation on decision diagrams by ITE algorithm. For example, $\mathtt{ITE}(F, \neg G, G)$ results in $F \oplus G$. In synthesis algorithm, there are three input parameters too, but first is a boolean operation and other two parameters F and G are decision diagrams. Output of this algorithm is result of applying engaged boolean operation on diagrams F and G. In $\mathtt{ITE}$-$\oplus$ algorithm [MS97], there is written:

funct $\mathtt{ITE}$-$\oplus(F, G, H, Result)$
  ⋮
  if $F = \oplus$ **then** $Result = \mathtt{NewNode}(\oplus, G, H)$; **fi**
  ⋮
end

This does not correspond with the facts written in this paragraph.

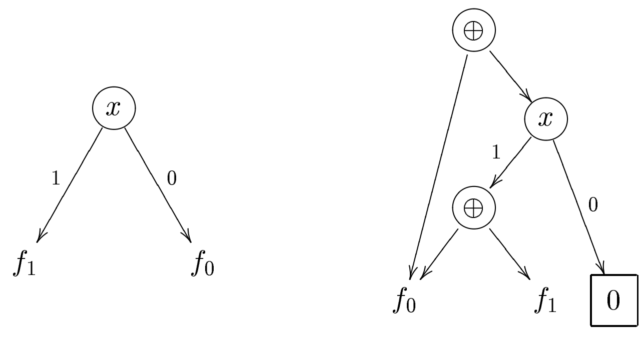
The second mistake is that this line is the only difference between ITE algorithm [BRB90] and $\mathtt{ITE}$-$\oplus$ algorithm [MS97]. It means that there is no $\oplus$-node in the result of this $\mathtt{ITE}$-$\oplus$ algorithm. This mistake is not fatal because result is correct, but it is an OBDD.

Because of mistakes in $\mathtt{ITE}$-$\oplus$ algorithm [MS97], we present our own algorithm.

funct $\mathtt{UNION}(F, G : \oplus\text{-}OBDD) : \oplus\text{-}OBDD$
  if $F = 1 \vee G = 1$ **then** $return(1)$; **fi**
  if $F = 0$ **then** $return(G)$; **fi**
  if $G = 0$ **then** $return(F)$; **fi**
  /∗ Result cache checking ∗/
  if $(F, G, R) \in ResultCache$
    then $return(R)$;
    else
      /∗ Co-factor based computing ∗/
      $z := \mathtt{Topmost}(\mathtt{TopVariable}(F), \mathtt{TopVariable}(G))$;
      $R_0 := \mathtt{UNION}(F|_{z \leftarrow 0}, G|_{z \leftarrow 0})$;
      $R_1 := \mathtt{UNION}(F|_{z \leftarrow 1}, G|_{z \leftarrow 1})$;
      $R := \mathtt{Davio}(z, R_1, R_0)$;   /∗ new node based on Davio rule ∗/
      $\mathtt{InsertInResultCache}(F, G, R)$;
      $return(R)$;
  fi
end

Where Davio is a substitution of NewNode. Davio returns a $\oplus$-OBDD equivalent to the OBDD returned by NewNode. For illustrating the concept of Davio see Figure 4.1. Nodes are composed according to the *positive Davio expansion rule* [MS97]:

$$f = f|_{x \leftarrow 0} \oplus x(f|_{x \leftarrow 1} \oplus f|_{x \leftarrow 0}).$$



`NewNode`(x, f_1, f_0) `Davio`(x, f_1, f_0)

Figure 4.1: Two equivalent $\oplus$-OBDD returned by the different procedures for creation of a new node.

Hence, `Davio` is implemented as follows:

funct `Davio`$(x$: *variable*; F, G : $\oplus$-*OBDD*$)$: $\oplus$-*OBDD*
  $P :=$ `NewNode`$(\oplus, F, G)$;
  $Q :=$ `NewNode`$(x, P, 0)$;
  $R :=$ `NewNode`$(\oplus, Q, G)$;
  $return(R)$;
end

The time complexity of `Davio` is asymptotically equal to the time complexity of `NewNode`. Hence, it is $O(1)$ for $\oplus$-OBDD with complemented edges. But it is $O(|F|)$ for $\oplus$-OBDD with $\oplus$-meta-nodes and for merged $\oplus$-OBDD.

Finding *TopVariable(F)* is more complicated in $\oplus$-OBDD than in OBDD because top node of $\oplus$-OBDD may be a $\oplus$-node. The time complexity of this procedure depends on chosen type of $\oplus$-OBDD. It is $O(|F|)$ for $\oplus$-OBDD with complemented edges. But it is $O(1)$ for $\oplus$-OBDD with $\oplus$-meta-nodes and for merged $\oplus$-OBDD.

The complexity of $F|_{z \leftarrow 0}$, where z is a top variable, is different too. It is $O(|F|)$ for $\oplus$-OBDD with complemented edges and for $\oplus$-OBDD with $\oplus$-meta-nodes. But it is $O(1)$ for merged $\oplus$-OBDD.

The time complexity of `UNION` is $O(|F| * |G|)$ for $\oplus$-OBDD with complemented edges. But it is $O(|F|^2 * |G|^2)$ for $\oplus$-OBDD with $\oplus$-meta-nodes and for merged $\oplus$-OBDD because of the time complexity of `NewNode`.

`RELP` is a modification of `UNION`. It is built in the same way as in OBDD representation.

funct `RELP`$(F, G$: $\oplus$-*OBDD*; E : *SetOfVariables*$)$: $\oplus$-*OBDD*
  if $F = 0 \vee G = 0$ **then** $return(0)$; **fi**
  if $F = 1$ **then** $return(G)$; **fi**
  if $G = 1$ **then** $return(F)$; **fi**

```
    z := Topmost(TopVariable(F), TopVariable(G));
    /* Omitting idle variables from E */
    while z ≠ Topmost(z, TopVariable(E)) do
          E := E ∖ {TopVariable(E)}
    od
    /* Result cache checking */
    if (F, G, E, R) ∈ ResultCache
      then return(R);
       else
          /* Co-factor based computing */
          R_0 := RELP(F|_{z←0}, G|_{z←0}, E);
          R_1 := RELP(F|_{z←1}, G|_{z←1}, E);
          if z ∈ E
            then R := UNION(R_1, R_0);
             else R := Davio(z, R_1, R_0);
                                        /* new nodes based on Davio rule */
          fi
          InsertInResultCache(F, G, E, R);
          return(R);
    fi
end
```

The time complexity of `RELP` is $O(|F|^2 * |G|^2)$ for $\oplus$-OBDD with complemented edges. But it is $O(|F|^3 * |G|^3)$ for $\oplus$-OBDD with $\oplus$-meta-nodes and for merged $\oplus$-OBDD because of the time complexity of `UNION`.

Deterministic equivalence test for $\oplus$-OBDD is in time $O(n*(|F|+|G|)^3)$ and in space $O((|F|+|G|)^2)$, where n is the number of variables. These complexities are proven in [Waa97]. The probabilistic equivalence test [GM93a] for $\oplus$-OBDD needs only linear time. We concentrate on the probabilistic approach. It is based on a probabilistic equivalence test for read-once branching programs (BP1) which was originally introduced in [BCW80]. Equivalence of two $\oplus$-OBDDs is determined by an algebraic transformation of the $\oplus$-OBDDs into terms of polynomials over a finite field.

Let $GF(2^m)$ denote a Galois field with 2^m elements of characteristic 2, where $m \in \mathbb{N}$, n is the number of variables, and $m > (log(n)) + 1$. An example of $GF(2^m)$ is $\mathbb{F}_2[x]/p(x)$, where $p(x)$ is an irreducible polynomial of degree m.

We assign a polynomial $p_v : (GF(2^m))^n \to GF(2^m)$ to each node v of a $\oplus$-OBDD P:

- If v is a *terminal node*, then $p_v(x_1, \ldots, x_n) = l(v)$.
- If v is a *variable NODE* with $l(v) = x_i$, then $p_v(x_1, \ldots, x_n) = (1 - x_i) * p_{low(v)}(x_1, \ldots, x_n) + x_i * p_{high(v)}(x_1, \ldots, x_n)$.

- If v is a $\oplus$-*node*, then
  $p_v(x_1, \ldots, x_n) = p_{low(v)}(x_1, \ldots, x_n) + p_{high(v)}(x_1, \ldots, x_n)$.

The polynomial associated with the $\oplus$-OBDD F is the polynomial associated with the root of F. Resemblance between a polynomial p_F associated with the $\oplus$-OBDD F and a boolean function f_F represented by the $\oplus$-OBDD F is feasible. Operations $+$ and $*$ on $GF(2^m)$ correspond to $\oplus$ and $\wedge$ on boolean functions, respectively. The operation $+$ fulfills all properties of the logical operation $\oplus$ exactly. The operation $*$ fulfills all properties of the logical operation $\wedge$ with the except for idempotency. But there is no associated polynomial with a variable powered by 2 or more because each essential $\oplus$-OBDD is a free decision diagram. Hence, the associated polynomial remains unchanged for different representations of the same boolean function.

In [GM93a], there is presented this probabilistic algorithm for checking equivalence of two BP1s:

funct EQU$(F, G : BP1)$: *Boolean*;
  choose independently and uniformly $a_1, \ldots, a_n$ from $GF(2^m)$;
  if $p_F(a_1, \ldots, a_n) = p_G(a_1, \ldots, a_n)$
    then $return(TRUE)$;
    else $return(FALSE)$;
  fi
end

EQU is a probabilistic algorithm with one-side error. EQU returns always $TRUE$ if F and G are equal. If F and G are different, EQU may return both results. We prove that probability of an error result is $< \frac{n}{|GF|}$ in the following. $|GF|$ is the number of elements the Galois filed GF and n is the number of variables. $|GF|$ is 2^m, where $m > 1 + log(n)$. Then probability of error is $< \frac{n}{2^m} < \frac{n}{2^{1+log(n)}} = \frac{1}{2}$. Hence, EQU has bounded probability of its one-side error.

In [MS00], there is used this algorithm for $\oplus$-OBDDs. The error estimation presented in [MS00] is:

$$error < \frac{size(P)^2 * n^s}{2 * |GF|^s}$$

Where $size(P)$ denotes the number of nodes of the $\oplus$-OBDD P, n is the number of variables, $|GF|$ is the number of elements in the finite field, and s is the number of executions of the equivalence test. There is no proof of the error estimation in [MS00]. Hence, we present our own error estimation which is more precise.

Theorem 4.1. *Let n is the number of variables, $|GF|$ is the number of elements in the finite field, and s is the number of executions of the equivalence test then*

$$\textit{Prob(error)} < \frac{n^s}{|GF|^s} \quad \textit{for } n \geq 2.$$

Proof. Checking the equivalence of two polynomials can be performed as checking if the difference of these polynomials is a zero polynomial. Hence, our error estimation is equal to the error estimation of checking if a polynomial is a zero polynomial.

At first we compute error for exactly one execution of the equivalence test ($s = 1$). We get incorrect result when we evaluate a polynomial by the root of this polynomial.

$$\text{Prob(error)} \leq \frac{\text{the number of roots of a non-zero polynomial}}{\text{the number of evaluations of a polynomial}}$$

The number of evaluations of a polynomial is $|GF|^n$. We prove that the number of roots is $< n * |GF|^{n-1}$ in the next paragraph. Thus, our error estimation is:

$$\text{Prob(error)} < \frac{n * |GF|^{n-1}}{|GF|^n} = \frac{n}{|GF|}$$

When we use the equation test s times then the estimation is:

$$\text{Prob(error)} < \frac{n^s}{|GF|^s}$$

We prove the upper bound of the number of roots of a non-zero polynomial by an induction with respect to the number of variables.

Let p be a polynomial in n variables. Let $\|p\|$ denote the number of roots of the polynomial p. Let $p_1(a)$ be a polynomial in $n-1$ variables equal to $p(a, x_2, \ldots, x_n)$.

Lemma 4.2. *$p \not\equiv 0$ implies $\exists$ at most one $a \in GF$ such that $p_1(a) \equiv 0$.*

Proof. We prove the contraposition of the implication. If $\exists a, b \in GF$ such that $a \neq b \wedge p_1(a) \equiv p_1(b) \equiv 0$, then for each affinity combination c of a and b is $p_1(c) \equiv 0$. But each $c \in GF$ is an affinity combination of a and b because $c = \frac{c-b}{a-b} * a + \frac{a-c}{a-b} * b$. Then $p \equiv 0$ because $\forall c \in GF.p_1(c) \equiv 0$. □

Induction: We must prove that $\forall n \in \mathbb{N}$: if p is a polynomial in n variables and $p \not\equiv 0$ then $\|p\| \leq n * |GF|^{n-1}$.

Basis step: For $n = 1$: $p \not\equiv 0 \overset{Lemma\ 4.2}{\Longrightarrow} \|p\| \leq 1$.
For $n = 2$:

$$p \not\equiv 0 \overset{Lemma\ 4.2}{\Longrightarrow} \begin{cases} \forall a \in GF.p_1(a) \not\equiv 0 \text{ then} \\ \quad \begin{array}{rcl} \|p\| & = & \sum_{a \in GF} \|p_1(a)\| \\ & \leq & \sum_{a \in GF} 1 \\ & = & |GF| \\ & < & 2 * |GF|^{2-1} \end{array} \\ \\ \text{or} \\ \\ \exists! a \in GF.p_1(a) \equiv 0 \text{ then} \\ \quad \begin{array}{rcl} \|p\| & = & \|p_1(a)\| + \sum_{b \in GF \wedge b \neq a} \|p_1(b)\| \\ & = & |GF| + \sum_{b \in GF \wedge b \neq a} \|p_1(b)\| \\ & \leq & |GF| + \sum_{b \in GF \wedge b \neq a} 1 \\ & = & |GF| + (|GF| - 1) * 1 \\ & = & 2 * |GF| - 1 \\ & < & 2 * |GF|^{2-1} \end{array} \end{cases}$$

Inductive step: *Induction hypothesis:* The number of roots of a non-zero polynomial in $n - 1$ variables is $< (n - 1) * |GF|^{n-2}$.

$$p \not\equiv 0 \overset{Lemma 4.2}{\Longrightarrow} \begin{cases} \forall a \in GF.p_1(a) \not\equiv 0 \text{ then} \\ \quad \begin{array}{rcl} \|p\| & = & \sum_{a \in GF} \|p_1(a)\| \\ & <^{\text{by } IH} & \sum_{a \in GF} (n-1) * |GF|^{n-2} \\ & = & |GF| * (n-1) * |GF|^{n-2} \\ & = & (n-1) * |GF|^{n-1} \\ & < & n * |GF|^{n-1} \end{array} \\ \\ \text{or} \\ \\ \exists! a \in GF.p_1(a) \equiv 0 \text{ then} \\ \quad \begin{array}{rcl} \|p\| & = & \|p_1(a)\| + \sum_{b \in GF \wedge b \neq a} \|p_1(b)\| \\ & = & |GF|^{n-1} + \sum_{b \in GF \wedge b \neq a} \|p_1(b)\| \\ & <^{\text{by } IH} & |GF|^{n-1} + \sum_{b \in GF \wedge b \neq a} (n-1) * |GF|^{n-2} \\ & = & |GF|^{n-1} + (|GF| - 1) * (n-1) * |GF|^{n-2} \\ & < & |GF|^{n-1} + (n-1) * |GF|^{n-1} \\ & = & n * |GF|^{n-1} \end{array} \end{cases}$$

Hence, the number of roots of a non-zero polynomial is $\leq n * |GF|^{n-1}$, where n is the number of variables. If $n \geq 2$, then the number of roots of a non-zero polynomial is $< n * |GF|^{n-1}$. So, the probability of an error result is $< \frac{n^s}{|GF|^s}$ and the proof of Theorem 4.1 is complete. $\square$

4.3 BED

4.3.1 Specifications

Complemented edges are not necessary because operator node for negation of the first argument and negation of the second argument is available.

There are many simple rules for reducing of the size of BED. They are based on commutativity, associativity, idempotency, distributivity, and other features of the used boolean operators.

We recall a more elaborate reduction introduced in [AH97]. First of all, we must present some definitions.

A set $\{w1, w2\}$ of two nodes is a *2-cut* for node $u \in V \setminus \{w1, w2\}$ if each path p from u to a terminal node can be decomposed into two parts $u \rightsquigarrow^{p_1} w \rightsquigarrow^{p_2} u'$ such that $w \in \{w1, w2\}$. A 2-cut is called an *operator 2-cut* if it for all paths p, p_1 contains only operator nodes. The cut $\{low(u), high(u)\}$ is called a *trivial 2-cut* for the node u.

If the BED rooted at u has an operator 2-cut $\{w_1, w_2\}$, then there exists a binary boolean operator *op* such that

$$f_u = f_{w_1} \; op \; f_{w_2}.$$

It can be used for reduction of the size of BED. So, we need an efficient algorithm to find an operator 2-cut. In [AH97], there is the following lemma:

Lemma 4.3. *Let $u \in V$ be an operator node, $l = low(u)$ and $h = high(u)$. If l and h have only trivial operator 2-cut and u has any non-trivial operator 2-cut $\{w_1, w_2\}$, then $w_1, w_2 \in \{l, h, low(l), high(l), low(h), high(h)\}$.*

Proof. By contradiction: Assume that u has a non-terminal operator 2-cut $\{w_1, w_2\}$ and $w_1, w_2 \notin \{l, h, low(l), high(l), low(h), high(h)\}$.

Observe that either l or h is not in $\{w_1, w_2\}$ (otherwise the 2-cut would be trivial). Assume without any loss of generality that $l \notin \{w_1, w_2\}$. Because $\{w_1, w_2\}$ is a 2-cut of u, each path from u to a terminal node contains w_1 or w_2. Each path from l to a terminal node also contains w_1 or w_2 because the path is a postfix of a path from u. Thus, $\{w_1, w_2\}$ is an operator 2-cut of l but this operator 2-cut must be trivial. So, $w_1, w_2 \in \{low(l), high(l)\}$. This is a contradiction. $\square$

It means that non-trivial cuts exist only among the children and grandchildren of u. Thus, we may reduce BED with reduction rules for at most three boolean operations.

4.3.2 Algorithms and Their Complexities

Procedures `CREATE`, `NEG`, `UNION` are $O(1)$. They are easily performed as an adding of a new node.

Procedure `RELP`(F, G, E) connects F and G by AND operator node and then existentially abstracts the variables in E. The abstraction of each variable is performed according to this rule:

$$\exists x.f = f|_{x \leftarrow 0} \vee f|_{x \leftarrow 1}.$$

The time complexity of each abstraction is $O(|F| + |G|)$ because the time complexity of co-factoring $F \wedge G$ is $O(|F| + |G|)$. Hence, the time complexity of `RELP`(F, G, E) is $O(n * (|F| + |G|))$, where n is the number of variables in E.

Procedure `EQU` is co-NP-complete [GJ79]. It can be easily proven by the reduction on NON-SAT problem because conversion from boolean function into BED is a linear algorithm. Thus, we do know neither polynomial deterministic algorithm nor feasible probabilistic algorithm.

Chapter 5

Comparison of Effectivity

This chapter compares properties of the presented data structures, namely OBDD, $\oplus$-OBDD with complemented edges, $\oplus$-OBDD with $\oplus$-meta-nodes, merged $\oplus$-OBDD, and BED. We are interested in applicability in the symbolic model checking. At first we concentrate on the theoretical aspects. This comparison is based on the time complexities presented in the previous chapter. The second comparison is based on the results of a practical implementation. We do not implement BED representation because `EQU` is a co-NP-complete problem. Practical implementation is performed for OBDD, $\oplus$-OBDD with complemented edges, $\oplus$-OBDD with $\oplus$-meta-nodes, and merged $\oplus$-OBDD.

5.1 Comparison Based on Theoretical Results

In this chapter we compare theoretical results presented in the previous chapter. This comparison is based on upper bounds of the time complexities. Hence, we compare the worst times of the procedures. It means that there is still a chance that comparison of the average time falls out well. The time complexities from the previous chapter are presented in Table 5.1.

$\oplus$-OBDD Comparing the time complexities, we consider $\oplus$-OBDD to be worse. But there is a possibility for $\oplus$-OBDD to be applicable. The basic model checking problem is the state explosion problem. The state explosion problem is due to the fact that the number of states can be exponential in the size of the description of the system. Hence, the basic problem is the space complexity in the model checking algorithms. Model checking may benefit from $\oplus$-OBDD if the $\oplus$-OBDD representation is really more succinct than OBDD representation. It is possible because there are functions such as the hidden weighted bit function (HWB) which are more succinct in $\oplus$-OBDD representation than in OBDD representation. It has been shown in [Bry91] that HWB has an exponential size of each OBDD representation.

	OBDD	⊕-OBDD			BED
procedure	**compl. edges**	**compl. edges**	**meta nodes**	**merged**	
`CREATE`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`NEG`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`TopVar`	$O(1)$	$O(\|F\|)$	$O(1)$	$O(1)$	$O(\|F\|)$
$F\|_{x_{top}\leftarrow 0}$	$O(1)$	$O(\|F\|)$	$O(\|F\|)$	$O(1)$	$O(\|F\|)$
`UNION`	$O(\|F\|*\|G\|)$	$O(\|F\|*\|G\|)$	$O(\|F\|^2*\|G\|^2)$	$O(\|F\|^2*\|G\|^2)$	$O(1)$
`RELP`	$O(\|F\|^2*\|G\|^2)$	$O(\|F\|^2*\|G\|^2)$	$O(\|F\|^3*\|G\|^3)$	$O(\|F\|^3*\|G\|^3)$	$O(n*(\|F\|+\|G\|))$
`EQU`	$O(1)$	$O(\|F\|+\|G\|)$	$O(\|F\|+\|G\|)$	$O(\|F\|+\|G\|)$	???

Table 5.1: Comparison of the time complexities.

In [MS00], there is HWB represented by an ⊕-OBDD which has only cubic size. Applicability of ⊕-OBDD crucially depends on its succinctness.

BED The basic problem of BED representation is an efficient implementation of the equivalence test `EQU`. Possible solution of this problem is presented in [AH97]. It is based on transformation of BED into OBDD. After this transformation the equivalence test is easy because OBDD representation is canonical. This solution is useful when the comparing BEDs are expected to have a small OBDD representation. The comparison based on transformation into OBDDs is inefficient in case of frequent comparing or comparing the BEDs which have large OBDD representations. Performing fixpoint iterations using OBDDs, several researchers have observed that the intermediate results are often much larger than the final result. Then BED representation may be useful to implement into the symbolic model checking algorithm. In fact, some improvements of the SMC algorithm are based on BED's features. For example, *partitioned transition relation* [BCL+94] is the transition relation represented as a BED with conjunction or disjunction nodes at the top which connect OBDDs representing partial transition relations.

5.2 Comparison Based on Experimental Results

In the previous section we wrote that the applicability of ⊕-OBDD crucially depends on the succinctness of ⊕-OBDD. If we want to do the comparison credibly, we need to compare the real space requirements of the SMC algorithms based on different representations. So, we decided to implement introduced representations into the SMC algorithm. We did not implement BED representation because `EQU` is a co-NP-complete problem. This thesis is focused on the probabilistic approach to SMC and there is no known feasible probabilistic algorithm for a co-NP-complete problem. We have implemented ⊕-OBDD with complemented edges, ⊕-OBDD with ⊕-

meta-nodes, and merged $\oplus$-OBDD representation into the SMC algorithm. OBDD representation is in current use.

The reason for implementation is the comparison with OBDDs. So, we do not implement our own $\oplus$-OBDD package. We change implementation of the widely used OBDD package CUDD [Som98]. The Colorado University Decision Diagrams (CUDD) package of Fabio Somenzi has been developed by the Department of Electrical and Computer Engineering University of Colorado at Boulder. The CUDD package provides many functions to manipulate OBDDs, Algebraic Decision Diagrams (ADDs), and Zero-suppressed Binary Decision Diagrams (ZDDs). We have added the ability to manipulate $\oplus$-OBDDs.

The main reason to choose CUDD package is its compatibility to the symbolic model checker NuSMV [CCGR00]. The NuSMV (new symbolic model checker) is the result of the reengineering and implementations of the CMU SMV symbolic model checker. NuSMV is developed as a joint project between Carnegie Mellon University and Istituto per la Ricerca Scientifica a Tecnologica. NuSMV is distributed by an open source licence that allows free academic and commercial usage. NuSMV uses CUDD package for manipulating OBDDs and ADDs.

We have added a node counter *MaxUsedKey* in CUDD because we are interested in comparison of memory requirements. The *MaxUsedKey* keeps the maximal number of active nodes during the computation. A node is active if it is created and it is not intended to be erased. It means that *MaxUsedKey* corresponds to the maximal size of memory which is occupied during the computation of NuSMV.

We have implemented all three introduced types of $\oplus$-OBDD into the CUDD package. All new nodes are created by `Davio` and so many $\oplus$-nodes may be unnecessary. We did not implement $\oplus$-meta-nodes as single special nodes because the CUDD package is originally an OBDD package and there is no support for meta-nodes. So, we have implemented $\oplus$-meta-node as a chain of $\oplus$-nodes. It means that low successor of $\oplus$-node must be the terminal node or a variable node and only $high$ successor of $\oplus$-node may be a $\oplus$-node. A $\oplus$-meta-node with many successors occupies larger part of memory than $\oplus$-node in each implementation. Hence, if we implement $\oplus$-meta-node as a chain, then the number of nodes in use corresponds to the size of used memory much more better. We did not add any heuristic improvements on reordering and $\oplus$-node placement.

The probabilistic equivalence test `EQU` is performed on a Galois field with 2^{30} elements. Each test `EQU` is performed eight times. It means that probability of an error result is smaller than $1/2$ for 2^{239} variables. Each of verified examples has less than 250 variables. Hence, it is obvious that all results of our experiments are correct.

We verified some examples by NuSMV and compared the *MaxUsedKeys* with respect to the used representation. Results of this comparison are pre-

example	OBDD compl. edges	$\oplus$-OBDD compl. edges	meta nodes	merged
dartes	-	-	-	-
counter	47	54	54	51
dme1	-	-	-	-
mutex	104	145	131	127
mutex1	306	895	573	567
ring	124	252	175	170
semaphore	237	490	376	358
short	22	22	22	22
gigamax	52633	-	127864	191084
hwb6	789	3274	1955	1878
newring	60	97	82	73
p_error	8182446	-	-	-
p	5194783	-	-	-
philo	5543735	-	8491390	-
robot	33346	-	44723	-

Table 5.2: The maximal number of active nodes during the computation of NuSMV.

sented in Table 5.2. The column *example* contains the names of examples which have been verified. Next columns contain the *MaxUsedKeys*. A dash indicates that the computation did not finish in twelve hours. We decided to stop long time computation because the most of computations terminate in one second and the others finished computations terminated in half an hour.

Examples have been executed on an Intel Pentium III 450MHz Linux workstations with 384MB RAM. Making fair data source for the comparison, we did not allow any OBDD based heuristics such as reordering techniques, partial transition relation, and others. Hence, same examples did not finish for OBDD representation, too. We switched off generating of the counter example because we do not discuss any algorithm for finding the true evaluation of a $\oplus$-OBDD.

The examples **dartes**, **counter**, **dme1**, **mutex**, **mutex1**, **ring**, **semaphore**, **short**, **gigamax**, and **robot** are taken from the standard NuSMV distribution. The examples **p_error**, **p**, and **philo** are taken from the comparison made by Tomáš Brázdil. The examples **hwb6** and **newring** have been created by the author of the thesis.

Concentrating on small examples, we can observe that restrictions on the use of $\oplus$-nodes leads to more succinct representation. Computing large examples such as **philo** and **robot**, NuSMV for the merged $\oplus$-OBDD representation did not terminate. It is a conclusion of the fact that the procedure `NewNode` is a linear time algorithm.

The $\oplus$-OBDD representation with complemented edges is discursive. It is induced by many unnecessary $\oplus$-nodes created by `Davio`. Hence, this representation is not practically usable because there are no rules reducing unnecessary $\oplus$-nodes. Introduction of $\oplus$-OBDD representation with $\oplus$-meta-nodes leads to more economic representation. Constructing $\oplus$-meta-nodes, we can find and remove more redundant subtrees. It is the first improvement of $\oplus$-OBDD representation. Merged $\oplus$-OBDD brings another improvements but the basic disadvantage of merged $\oplus$-OBDD is the linear time complexity of `NewNode`.

In spite of this improvements, $\oplus$-OBDD representation is not more succinct than primary OBDD representation.

Chapter 6

Conclusion

We have investigated the possibility of a probabilistic approach to the SMC algorithm based on exchange of the OBDD representation. We inspected two possible representations, $\oplus$-OBDD and BED.

BEDs can be seen as an intermediate form between the compact circuits and the canonical OBDDs. All standard OBDD operations can be performed on BEDs as well or better. But BED is not a proper data structure for randomized SMC because the equivalence test `EQU` is a co-NP-complete problem and there are no known feasible probabilistic algorithms for co-NP-complete problems.

The deterministic equivalence test can be performed by transforming the BED into an equivalent OBDD. Hence, the use of BED is a lazy computation on OBDD. It means that if we want to perform any binary operation, we easily create new node with a label corresponding to this operation. When we need to compare two BEDs, we transform them into OBDDs. It is the same as performing the boolean operations which label the operation nodes. Hence, operations are performed lazily. The advantage of this lazy algorithm is induced by many reductions which may reduce the number of operation nodes before its performing.

BEDs are particularly useful in applications where the end-result has a small OBDD representation. The tautology checking is a feasible example of this applications (see [HWA99] for more information on this topic).

$\oplus$-OBDDs seem to be a promising alternative to OBDDs because they admit a more compact representation of boolean functions. However, our comparison indicate that the $\oplus$-OBDDs are not so good for SMC as OBDDs.

$\oplus$-OBDD is an immature data structure. Hence, we had to revise and complete rules for reductions and correct mistakes in `ITE-`$\oplus$ algorithm. In addition, we have introduced our own modification of $\oplus$-OBDDs, merged $\oplus$-OBDDs.

The adverse result of our comparison may be induced by incompleteness of our implementation such as representing $\oplus$-meta-node as a chain of $\oplus$-nodes and absence of heuristic algorithms for reordering and $\oplus$-node placement.

$\oplus$-meta-nodes are internally implement as chains of $\oplus$-nodes in our implementation. It means that each $\oplus$-meta-node is encountered $n - 1$ times, where n is the number of its successors. Representing $\oplus$-meta-node as a single special node, the merged $\oplus$-OBDD representation may bring markedly more reductions. If there is a $\oplus$-meta-node with n successors with the same label, then merged $\oplus$-OBDD reduction exchange those n nodes only for three nodes. Hence, the implementation of $\oplus$-meta-nodes as single special nodes may lead to better results in case of $\oplus$-OBDD representation with $\oplus$-meta-nodes and markedly better results in case of merged $\oplus$-OBDD representation.

The basic insufficiency of $\oplus$-OBDDs is the absence of heuristic algorithms for reordering and $\oplus$-node placement. These innovations may lead to the wide applicability of $\oplus$-OBDDs. Specifically oriented usage is positive for creating effective heuristic algorithms. But there are no particularities in the SMC algorithm which may be utilized because boolean functions used in SMC are very multifarious. Though, some elementary heuristic algorithms are presented in [MS01a, MS01b].

$\oplus$-OBDDs are very suitable for performing boolean operation XOR. In SMC, there is verification performed according to the CTL formula which is made by a person. And people are accustomed to use boolean operations such as conjunction, disjunction, negation, and implication. So, it is better to apply $\oplus$-OBDDs into algorithms where the need for XOR operations spring up naturally.

Bibliography

[AH97] Henrik R. Andersen and Henrik Hulgaard. Boolean Expression Diagrams. In *IEEE Symposium on Logic in Computer Science*, 1997.

[BCL+94] Jerry R. Burch, Edmund M. Clarke, David E. Long, Ken L. MacMillan, and David L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[BCW80] Manuel Blum, Ashok K. Chandra, and Mark N. Wegman. Equivalence of Free Boolean Graphs Can be Decided Probabilistically in Polynomial Time. *Information Processing Letters*, 10(2):80–82, March 1980.

[BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, Florida, June 1990. ACM/IEEE, IEEE Computer Society Press.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, volume C-35-8, pages 677–691, August 1986.

[Bry91] Randal E. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

[CCGR00] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A New Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. San Francisco : W. H. Freeman, 1979.

[GM93a] Jordan Gergov and Christoph Meinel. Frontiers of Feasible and Probabilistic Feasible Boolean Manipulation with Branching Programs. In *10th Annual Symposium on Theoretical Aspects of Computer Science*, volume 665 of *Lecture Notes in Computer Science*, pages 576–585, Würzburg, Germany, 25–27 February 1993. Springer.

[GM93b] Jordan Gergov and Christoph Meinel. Mod-2-OBDD's: A Generalization of OBDD's and EXOR-Sum-of-Products. Technical Report 93–21, Universität Trier, 1993. ISSN 0944–0488; FTP; WWW.

[HWA99] Henkik Hulgaard, Poul Williams, and Henrik R. Andersen. Equivalence Checking of Combinational Circuits using Boolean Expression Diagrams. In *IEEE Transactions of Computer-Aided Design*, volume 18(7), July 1999.

[MS97] Christoph Meinel and Harald Sack. Case Study: Manipulating $\oplus$-OBDDs by Means of Signatures. In *Proc. of the 3rd International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, Oxford, UK, 1997.

[MS99] Christoph Meinel and Harald Sack. Algorithmic Considerations of $\oplus$-OBDD Reordering. In *Proc. of the 4th International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, Victoria, BC, Canada, 1999.

[MS00] Christoph Meinel and Harald Sack. Parity-OBDDs - a BDD Structure for Probabilistic Verification. In *Electronic Notes in Theoretical Computer Science*, volume 22. Elsevier Science Publishers, 2000.

[MS01a] Christoph Meinel and Harald Sack. A Heuristic for $\oplus$-OBDD Minimization. Technical report, Universität Trier, 2001.

[MS01b] Christoph Meinel and Harald Sack. Improving XOR-Node Placement for $\oplus$-OBDDs. Technical report, Universität Trier, 2001.

[Som98] Fabio Somenzi. CUDD: CU Decision Diagram Package Release, 1998.

[Waa97] Stephan Waack. On the Descriptive and Algorithmic Power of Parity Ordered Binary Decision Diagrams. In *Proc. of 14th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1200 of *Lecture Notes in Computer Science*, pages 201–212, Lübeck, Germany, 27 February–March 1 1997. Springer.

[Weg00] Ingo Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Society for Industrial and Apllied Mathematics, 2000.