ORDERED BINARY DECISION DIAGRAMS IN ELECTRONIC DESIGN AUTOMATION

Foundations, Applications and Innovations

Randal E. Bryant
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213, USA
Randy.Bryant@cs.cmu.edu

Christoph Meinel
Computer Science Department
University of Trier
D-54286 Trier, Germany
meinel@uni-trier.de

Abstract Ordered Binary Decision Diagrams (OBDDs) play a key role in the automated synthesis and formal verification of digital systems. They are the state-of-the-art data structure for representing switching functions in various branches of electronic design automation. In the following we discuss the properties of this data structure, characterize its algorithmic behaviour, and describe some prominent applications.

Keywords: Data structures for switching functions, Ordered binary decision diagrams OBDDs, Efficient algorithms with OBDDs, Optimiziation of OBDDs, Variable ordering, Formal verification of combinatorial and sequential systems, Reachablility analysis, Image computation, Model checking, Innovative BDD-structures.

1.1. INTRODUCTION

In recent years, we have seen an enormous increase in system integration of VLSI (Very Large Scale Integration) designs. Moore's law, which says that the number of transistors on a chip doubles every 18 month is still valid and an end of this growth is not within view. With modern integrated systems, traditional design techniques such as manual circuit design and simulation-based verification have become excessively time consuming and error prone. Instead, automated logic synthesis has largely replaced manual circuit design, while formal verification is increasingly used as a replacement for simulation. Efficient algorithms and data structures to represent and reason about circuit functions are a core requirement for automated synthesis and verification programs. In this area, even the most elementary tasks, such as determining whether two combinational circuits compute the same logic function, are intractable. Hence, we will most likely never find algorithms that are truly efficient, i.e., having worst case polynomial complexity. Instead, as with much of electronic design automation, we look for representations and algorithms that work reasonably well for problems of interest.

For background, proofs and technical details we refer to [36], as well as to the cited papers.

1.2. DATA STRUCTURES FOR SWITCHING FUNCTIONS

Boolean functions $f : \{0,1\}^n \to \{0,1\}$ are of central importance for describing the switching behavior of digital circuits, and hence they are also refered to as *switching functions*. By introducing a suitable 0-1 encoding, all functions over finite domains can — at least in principle — be modeled with switching functions. Many techniques have been devised for synthesizing, optimizing, and verifying switching functions. In all fields of computer-aided design, selecting suitable data structures for representing and analyzing a design is a key requirement. Data structures for representing switching functions is thus an important part of automated logic synthesis, and verification.

There are two main criteria for assessing such data structures. First, the space efficiency determines the amount of memory required for storing and manipulating switching functions. Second, the time efficiency of the algorithms operating on the data structure determine the amount time required to perform various tasks.

Since it is impossible to determine all tasks that might be performed on switching functions, one tries to identify a set of basic operations that can be combined to implement a wide variety of different tasks. To assess the efficieny of a data structure, one analyzes its ability to support these basic operations.

In the field of computer-aided design of logic circuits, the following list of basic operations on switching functions can be identified. We use the notation P_f to indicate a representation of switching function f. We let f and g denote switching functions, i.e., $f, g: \{0,1\}^n \rightarrow \{0,1\}$. Rather than using a traditional positional notation for function arguments, we label the arguments by Boolean variables. Thus, a function over $\{0,1\}^n$ is defined over a set of variables V such that $|V| = n$. For an *assignment* $a: V \rightarrow \{0,1\}$, we let $f(a)$ denote the value of function f when each argument x has value $a(x)$.

Evaluation: Compute $f(a)$ for assignment a.

Synthesis: Compute a representation P_h for the function $h = f \otimes g$ for binary Boolean operator $\otimes$ and representations P_f and P_g of f and g.

Satisfiability test (SAT): Test whether there exists an assignment $a: V \rightarrow \{0,1\}$ such that $f(a) = 1$.

Satisfiability count: Determine how many of the 2^n possible assignments $a: V \rightarrow \{0,1\}$ satisfy f.

Equivalence test: Check whether two representations P_f and P_g denote the same function, i.e., $f \equiv g$.

Functional Composition: Compute a representation P_h for $h = f_{|x=g}$ (i.e., argument x to f is replaced by the Boolean function g).

Quantification: Compute the effect of quantifying an argument, either universally ($\forall x f$) or existentially ($\exists x f$).

Once one has decided to work with a particular data structure, a very important additional operation is the *minimization* of the representation P_f of a given switching function f.

Now one is ready to evaluate the different known data structures for switching functions with respect to their fitness for use in EDA. The following list shows serious points of weakness of traditional data structures:

Combinational circuits: The elementary SAT problem is NP-complete for circuits and the minimization problem is NP-hard.

Boolean formulas: The equivalence test is co-NP-complete for Boolean formulas. Allowing quantification makes satisfiability PSPACE-complete.

Truth tables: All operations can be performed efficiently (in the length of the input), but truth tables always require exponential space.

Conjunctive Normal Form (CNF): Even simple functions often have only an exponential representation in CNF. The SAT problem is NP-complete for CNF, while satisfiability counting is #P-complete.

Disjunctive Normal Form (DNF): The problems with DNF are similar to CNF, but SAT is in P.

Branching Programs (BPs): Many functions can be represented very efficiently using branching programs but tasks such as SAT (NP-complete), satisfiability count (#P-complete), and quantified SAT (PSPACE-complete) are intractable.

Hence, all these data structures fail for use in electronic design automation either for reasons of space complexity (e.g., truth tables) or time complexity of the basic tasks (e.g., branching programs).

1.3. OBDDS – ORDERED BINARY DECISION DIAGRAMS

1.3.1. DEFINITIONS

The introduction of *Ordered Binary Decision Diagrams (OBDDs)* in 1986 was a major step in the search for suitable data structures for switching functions in circuit design [5, 7]. In contrast to conventional descriptions based on computation rules, OBDDs are based on a decision process. OBDDs combine two advantages: the data structure is reasonably space efficient, and the algorithms that operate on them are generally time efficient. OBDDs can be viewed as a restricted class of binary decision diagrams [1] and branching programs [32] that provides a good balance between the size of the representation and the efficiency of the algorithms implementing basic operations.

Let $f : \{0,1\}^n \rightarrow \{0,1\}$ be a Boolean function where the arguments to f are denoted by a set of variables V, such that $|V| = n$. Let $\pi : V \rightarrow \{1, \ldots, n\}$ be a bijection indicating a total ordering of these variables. That is we consider variables x and y to be ordered $x < y$ when $\pi(x) < \pi(y)$. An *Ordered Binary Decision Diagram* (OBDD) P

for f with respect to ordering π is a directed acyclic graph consisting of nonterminal nodes labeled by the variables in V and terminal nodes labeled by the Boolean constants 1 and 0. Each nonterminal node has two outgoing edges: the 1-edge and the 0-edge. The OBDD has a starting node called the *root*. The computation of $f(a)$ follows a path from the root to a terminal node, where at a node labeled by x, if $a(x) = 1$, the path follows the 1-edge, and otherwise it follows the 0-edge. The value of the reached terminal node determines the value of $f(a)$. On a path from the root to the sink, each variable occurs at most once. The variables on every path from the root to a terminal node respect ordering π. That is, for an edge leading from a node labeled by x to one labeled by y, we must have $\pi(x) < \pi(y)$.

Figure 1.1 gives two examples for OBDDs for the Boolean function $f = bc + a\overline{b}\overline{c}$ under the variable ordering $a < b < c$. In this figure and others, we use dashed lines to indicate 0-edges and solid lines to indicate 1-edges.

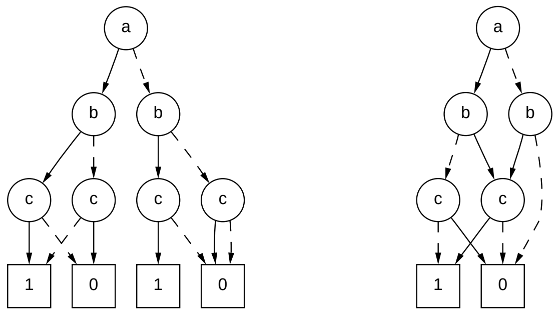

Figure 1.1. Two OBDDs of $f = bc + a\overline{b}\overline{c}$.

The *Shannon decomposition* (which should actually be attributed to Boole) states that a function f can be expressed in terms of its cofactors with respect to any of its variables x:

$$f \;=\; xf|_{x=1} \;+\; \overline{x}f|_{x=0},$$

where $f|_{x=\delta}$ denotes the subfunction that results from f after fixing the argument denoted by the variable x to the constant δ. From the Shannon decomposition, one can derive the first important property of OBDDs:

Property 1.3.1 (Universality) *Any switching function can be represented by an OBDD for any variable order.*

The proof of this property follows by induction on $n = |V|$. For $n = 0$, there can only be constant functions 0 and 1. For a function f over n

variables, and an ordering π of these variables we let x be the element of V such that $\pi(x) = 1$. We then recursively synthesize OBDDs for functions $f|_{x=1}$ and $f|_{x=0}$ for an ordering π', where $\pi'(y) = \pi(y) - 1$. We then create a new root node labeled by x having its 1-edge directed to the OBDD for $f|_{x=1}$ and its 0-edge directed to the OBDD for $f|_{x=0}$.

In general, the OBDD representation of a function is not unique, and this makes algorithms such as equivalence testing difficult. By using a surprisingly simple reduction mechanism , an arbitrary OBDD for a function can be transformed into a standard, or *canonical* form . Obviously, the following two reduction rules keep the represented function invariant:

Elimination rule If the 1- and 0-edges of a node v point to the same node u, then eliminate v, and redirect all incoming edges of v to u.

Merging rule All terminal nodes with a given label are merged to one node, redirecting all incoming edges to this node. If the nonterminal 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 remaining node.

The elimination rule and the merging rule are illustrated in Figure 1.2.

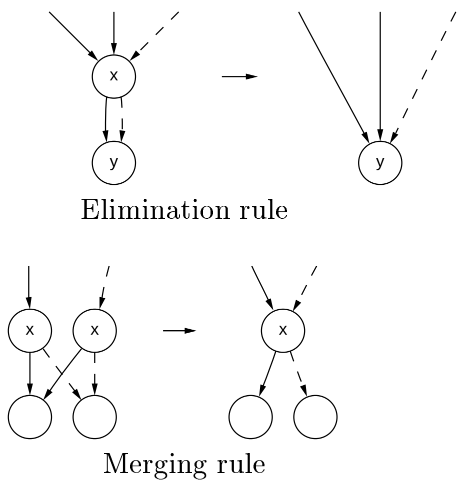

Figure 1.2. Reduction rules.

An OBDD is called *reduced* , if neither of the two reduction rules can be applied. It is easy to see that the right OBDD in Figure 1.1 is reduced. Regarding the algorithmic properties of reduced OBDDs, the following property [5] of canonicity is of basic importance:

Property 1.3.2 (Canonicity) *For any Boolean function f and any variable ordering π, there is a unique reduced OBDD for f with respect to π.*

Thus, no matter how we initially construct an OBDD representation of a function, we can transform it into a canonical form (with respect to a given variable ordering) by simply repeatedly applying the two reduction rules. Since this reduction is so simple, we will henceforth assume that all OBDDs we consider are in reduced form.

In addition to universality and canonicity, OBDDs have third fundamental property, which makes OBDDs such a successful data structure for representation of Boolean functions: efficiency of algorithmic manipulation.

1.3.2. OPERATIONS ON AND WITH OBDDS

Besides truth tables, the OBDD representation of switching functions is the only data structure among those listed above that has deterministic polynomial algorithms for all of the basic operations listed in Section 1.2. Since truth tables are always exponential in the number of arguments, they are practical only for representing over a small number of arguments. By contrast, an OBDD may be exponentially smaller than the truth table of the function it represents.

The following list gives the time and space requirements for the basic operations ($\#P_f$ denotes the number of nodes in the OBDD P for the function f):

Evaluation: Time: $O(n)$

Synthesis: Time and space: $O(\#P_f \cdot \#P_g)$

Satisfiability test: Time: $O(\#P_f)$

Satisfiability count: Time: $O(\#P_f)$

Equivalence test: Time: $O(\min(\#P_f, \#P_g))$

Functional Composition: Time and space: $O((\#P_f)^2 \cdot \#P_g)$

Variable Quantification: Time and space: $O((\#P_f)^2)$

Minimization: (w.r.t. given variable order) Time and space: $O(\#P_f)$

Observe that many of operations have time and space requirements linear in the size of the OBDD, and that all of them have polynomial time and space conmplexity relative to their argument sizes.

1.3.3. EFFICIENT SYNTHESIS OF OBDDS

To give an example an algorithm for one of the basic operations on switching functions using OBDDs, we consider synthesis. To this end let $\otimes$ denote an arbitrary binary Boolean operation, such as AND or OR. We have as arguments the OBDD representations of functions f and g with respect to a variable ordering π. In order to compute the OBDD of $f \otimes g$ with respect to π, one can use Shannon's decomposition w.r.t. the leading variable x in the variable order π:

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

Since x is the first variable in the ordering, the OBDDs for the subfunctions $f|_{x=1}$, $f|_{x=0}$, $g|_{x=1}$, and $g|_{x=0}$ are simply the children of the two roots. By recursively applying this decomposition, we can derive an OBDD representation of the function $f \otimes g$.

In order to perform this operation efficiently, multiple calls with the same argument pairs should to be avoided. This is implemented by a form of dynamic programming known as *memorization.* The computed result for each recursive application is stored in a *computed table.* Successive applications check this table before continuing the recursion. In this way, the originally exponential number of decompositions is now bounded by the product of the two OBDD sizes.

To increase the usage of the computed table all synthesis operations are mapped to a single operation, the so called if-then-else operator (ITE):

$$ITE(f, g, h) = f \cdot g + \overline{f} \cdot h.$$

e.g., $h = f \cdot g$ maps to $h = ITE(f, g, 0)$. Because of the huge number of ITE operations during synthesis the computed table is usually implemented as a cache to reduce memory consumption.

Most OBDD packages use a *shared* representation , in which multiple functions share common nodes in the data structure [45]. In effect, this involves applying the merging rule on the global data structure. It is implemented via a *unique-table* , mapping keys consisting of a label and the destinations of the 0- and 1-edges to an existing BDD node. Before creating a new node, its existence is checked in the unique-table. Together with an immediate check for the elimination rule the constructed OBDDs are always reduced and the reduction operation becomes obsolete. With a shared representation, OBDDs become a *strong canonical* *form*, where two functions f and g are equivalent if and only if they have the same root OBDD node. Thus, equivalence testing has complexity $O(1)$ in this representation.

1.3.4. CONSTRUCTING OBDDS BY SYMBOLIC EVALUATION

One method for constructing the OBDD representation of the function computed by a combinational logic circuit is to perform a *symbolic evaluation* of the circuit. Symbolic evaluation is based on the iterated application of the synthesis algorithm described in the last section.

Starting with the (trivial) OBDD representations of the primary inputs of the circuit one constructs, in topological order, OBDDs for the function at each gate of the circuit from the OBDDs of the corresponding predecessor gates, applying the synthesis operation corresponding to the operation performed by the gate under consideration.

Although each step of symbolic evaluation has polynomial complexity, the final OBDD generated may be exponentially larger than the original circuit. Fortunately, this potential "blow-up" does not occur for many of the circuits encountered in real-world applications. In addition, the basic algorithms are sufficiently simple that they can be applied to OBDDs with several 100.000 nodes. As a consequence OBDDs have proved to be quite practical in a variety of applications.

1.3.5. IMPORTANCE OF THE VARIABLE ORDERING

The size of an OBDD and hence the complexity of its manipulation heavily depends on its variable order. An example is shown in Figure 1.3. With respect to the variable order $x_1, x_2, \ldots, x_{2n-1}, x_{2n}$ the function

$$x_1x_2 + x_3x_4 + \ldots + x_{2n-1}x_{2n}$$

has an OBDD representation of linear size. For the variable order $x_1, x_3, \ldots,\ x_{2n-1}, x_2, x_4, \ldots, x_{2n}$ however, the size of the OBDD grows exponentially in n.

The same effect occurs in the case of the functions for n-bit binary addition. Depending on the variable order, the OBDD size varies from linear to exponential in the number of input bits. Unfortunately, for some important functions, including those for n-bit binary multiplication, the OBDDs representations are of exponential size for all possible variable orders [6].

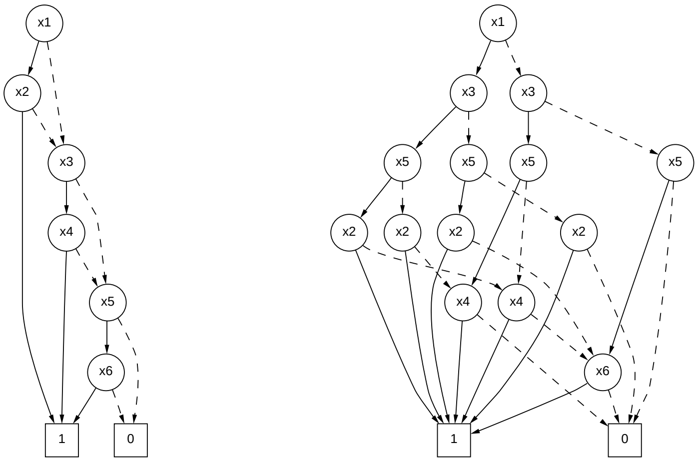

Figure 1.3. Influence of the variable order.

Due to the uniqueness of the OBDD representation w.r.t. a given variable order, finding a suitable variable order is the only way to optimize the size of the OBDD representation of a Boolean function.

In Section 1.5 we will discuss this problem in more detail.

1.4. PARADIGMATIC APPLICATIONS OF OBDDS

In the following we will discuss three paradigmatic applications of OBDDs: (1) formal verification of combinational circuits , (2) analysis of sequential systems and (3) symbolic model checking.

1.4.1. FORMAL VERIFICATION OF COMBINATIONAL CIRCUITS

Formal hardware verification involves proving that a circuit C satisfies its specification S. With combinational circuit verification, both C and S are acyclic networks of logic gates, and the task is to prove that they compute identical Boolean functions. For circuits of more than 30 inputs, it becomes totally impractical to test this equivalence by exhaustive simulation.

OBDD based combinational verification proceeds in two steps:

1 Construct the OBDDs P_C and P_S for C and S by symbolic simulation.

2 Check the equivalence of the OBDDs P_C and P_S.

In case of a shared OBDD representation, the equivalence test itself consists of a single pointer comparison. Each step of symbolic simulation can be performed efficiently with respect to the OBDD sizes of the predecessor gates. This shows that the difficulty of the NP-complete equivalent test [20] has now been shifted into the representation size. Unfortunately, since each synthesis step can potentially produce an OBDD that is the product of its argument sizes, the OBDD representation of either C or S can be of exponential size.

If C and S are not equivalent, the operation $P_C \oplus P_S$ gives the inputs where implementation and specification differ. This may be used for debugging.

A number of companies have developed equivalence checking tools, and these are in widespread use. Most use some combination of OBDDs and combinatorial search (e.g., using algorithms develped for automatic test pattern generation.) They also try to exploit any structural similarity between C and S. This mostly involves finding *cutpoints*, i.e., pairs of nodes within circuits C and S having identical circuit functions.

1.4.2. FORMAL VERIFICATION OF SEQUENTIAL SYSTEMS

A central task in the formal verification of sequential systems such as controllers or protocols is to test the equivalence of two given finite state machines (FSMs). For example, one might wish to make sure that a sequential circuit optimizer does not inadvertently alter the circuit function by testing the equivalence of the original and optimized circuits. This, along with many other tasks for FSMs, can be reduced to one of computing the set of reachable states for a state machine.

Sequential equivalence checking. Testing two finite state machines M_1 and M_2 for equivalence can be reduced to a reachability analysis by using the construction in Figure 1.4: Let M denote the *product machine* whose state space is the Cartesian product of the state spaces of M_1 and M_2. The output of M for a given state and a given input is 1 if and only if for this configuration the outputs of M_1 and M_2 agree. M_1 and M_2 have the same input/output behavior if and only if the output of M is 1 for all reachable states. This demonstrates an important application of reachability analysis.

Reachability analysis. Reachability analysis involves determining the set of states R that can be reached by a finite state machine, after

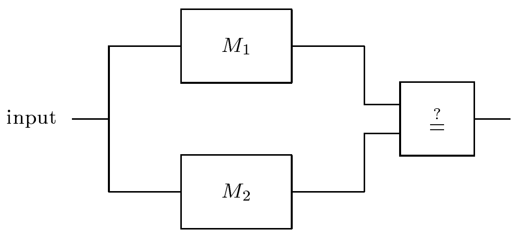

Figure 1.4. Product machine for FSM verification.

an arbitrary number of transitions, given that it starts from a state in initial state set Q_0. For large state machines, it is not practical to explicitly enumerate the set of reachable states. Coudert, Berthet and Madre [11, 13] developed the first reachability algorithms based on OBDD representations of state sets. For a digital circuit, the state is encoded as a set of Boolean latch values $s = \langle s_1, s_2, \ldots, s_n \rangle$, and hence a set of states A can be represented by a *characteristic function* $\chi_A(s)$, yielding 1 for any state $s \in A$ and yielding 0 otherwise. With such a representation, Boolean operations AND, OR, and NOT correspond to set operations Intersection, Union, and Complement.

The set of reachable states can be computed by a form of breadth-first traversal, where all operations are performed symbolically. The complexity of this *symbolic breadth-first traversal* depends on the sizes of the OBDDs representing the reached state sets. Figure 1.5 gives an outline of the algorithm. The key step is the *image computation*, giving the set of states that can be reached by one transition of the state machine.

```
traverse(TR, S0) {
/* Input: Transition relation TR, initial set S0 */
/* Output: Set of reachable states */
    Reached = From = S0;
    Do {
        To = Img(TR, From);
        New = To \ Reached;
        From = New;
        Reached = Reached ∪ New;
    } While (New ≠ ∅);
    Return Reached;
}
```

Figure 1.5. Basic algorithm for reachability analysis based on breadth-first traversal.

Image computation. Given a set of states R, its *image* is the set of states that can be reached by one step of the state machine. That is, suppose we represent the state machine by a transition function $\delta\colon B^n \times B^m \rightarrow B^n$, where $B = \{0, 1\}$, and n and m are the number of bits encoding the state and the inputs, respectively. For state s and input x, $\delta(s, x)$ yields a new state s'.

We can convert this next state function into a *transition relation* TR, where $\mathrm{TR}(s, s')$ holds when there is some input x such that $\delta(s, x) = s'$. This relation is defined by existentially quantifying the inputs from δ:

$$\mathrm{TR}(s, s') = \exists x \left[\bigwedge_i \delta_i(s, x) \equiv s'_i \right], \tag{1.1}$$

where δ_i represents the transition function for the ith state bit. This transition relation can be represented by a Boolean characteristic function χ_{TR}, where $\chi_{\mathrm{TR}}(s, s')$ equals 1 when $\mathrm{TR}(s, s')$ holds.

Given a set of states R with characteristic function χ_R, its *image under TR*, denoted $Img(\mathrm{TR}, R)$, is a set R' having characteristic function:

$$\chi_{R'}(s') = \exists s (\chi_{\mathrm{TR}}(s, s') \wedge \chi_R(s))$$

Reachability analysis involves repeatedly applying this image operation, as shown in Figure 1.5. On each iteration, the procedure has a set of newly encountered states *From* and performs an image computation to determine the set of states *To* that can be reached by one more transition. The new value of *From* is simply the states in *To* that have not already been encountered.

Since we are dealing with a finite state machine, these iterations will eventually reach a point where no new states are encountered, and hence the accumulated set of states *Reached* represents the set of all reachable states. In general, this convergence may require an exponential (in the number of latches) number of iterations. An example of such a case would be an n-bit counter starting at a reset state. In many applications, however, the analysis converges after a reasonably small (e.g., less than 20) number of iterations.

Partitioned transition relation. Equation 1.1 defines a *monolithic* represention of the transition relation as a single OBDD. This is often much too large to allow an efficient computation of the reachable states. Therefore, more sophisticated reachable states computation methods are needed. One idea that has been proved sucessful is to consider a *partitioned* representation of the transition relation [9]. That is, we divide the state latches into subgroups $L_1, \ldots, L_k$, and represent the transition relation by a set of subrelations $\mathrm{TR}_1, \ldots, \mathrm{TR}_k$, where subrelation TR_j describes the next state of the latches in L_j. Subrelation TR_j is defined as $\mathrm{TR}_j(x, s, s') = \bigwedge_{i \in L_j} [\delta_i(s, x) \equiv s'_i]$. The overall transition relation is then given by

$$\mathrm{TR}(s, s') = \exists x \left[\bigwedge_{j=1,k} \mathrm{TR}_j \right].$$

In the extreme, we can consider each latch to be in its own subgroup.

Early quantification. By carefully partitioning and ordering the latches, we can partition the Boolean variables encoding the current state s and the input x into sets $T_1, T_2, \ldots, T_k$, such that no next state value of a latch in partition L_j depends on a current state or input variable in T_i for $j < i$. We can then move existential quantifiers inward in the image computation:

$$\chi_{R'} = \exists T_1(\chi_{\mathrm{TR}_1} \wedge \exists T_2(\chi_{\mathrm{TR}_2} \wedge \cdots \wedge \exists T_k(\chi_{\mathrm{TR}_k} \wedge \chi_R) \cdots)).$$

The so called *AndExist* [9] or *AndAbstract* operation $(\exists_x(f \wedge g))$ may prevent a blow-up of BDD-size during the image computation. Another important problem is finding an optimal schedule of the partitions for the AndExist operation. Geist and Beer [21] presented a heuristic for the ordering of partitions each representing a single state variable.

OBDD size explosion during image computation. Even with partitioned transition relations and early quantification, the size of the intermediate OBDDs and the OBDDs representing the reachable state set tend to behave as shown in Figure 1.6. This problem of explosive memory consumption in reachability analysis is a major subject for current research.

1.4.3. MODEL CHECKING

Since a complete formal verification of a sequential system is often too complex, methods are of interest that guarantee at least correctness of certain properties. One of them is *model checking.*

A model checker decides whether an implementation satisfies a specification given as a temporal logic formula. There are many different varieties of temporal logic, but *computation tree logic* (CTL) has proved the most successful in this context. The idea of combining model checking with symbolic OBDD algorithms was first introduced by McMillan [31] and Coudert/Madre [12]. Using *symbolic* model checking, real-life systems up to 10^{100} states can be verified.

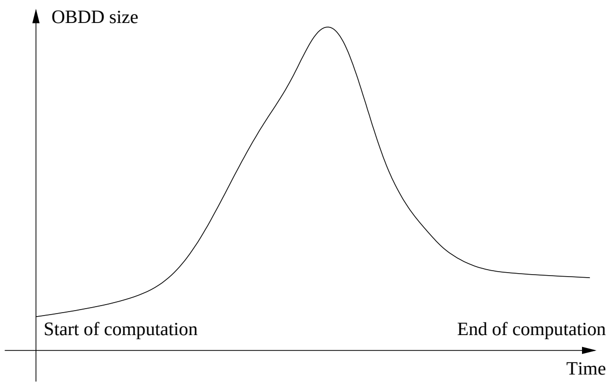

Figure 1.6. Memory consumption during image computation.

The formulas of CTL describe properties of infinite paths of states that are traversed during the computation.

Computation Tree Logic (CTL). The operation of a finite state machine can be viewed as an infinite sequence of states, with each state derived from its predecessor by one state transition. For a nondeterministic state machine, a state may have multiple possible successors, and hence there are many *paths*, or sequences, emanating from any single state. A formula in CTL describes some property of states along the paths emanating from a given state. First, an *atomic formula* defines some restriction on the current state. To these we can add *path formulas* describing some properties of a path p that starts at the given state:

X The next operator describes a condition that is true in the next state along p.

G The global operator describes a condition that is true for states along p.

F The future operator describes a condition that is true at some state along p.

U The until operator $a\mathbf{U}b$ holds on path p if a is true at every state up to one at which b holds.

In CTL, all path operators must be quantified with either **A** (on all paths emanating from the current state) or **E** (on some path emanating from the current state). As an example, suppose we have a synchronization protocol that makes sure that two processes P_1 and P_2 are never simultaneously in their critical sections, as indicated by the flags cs_1 and cs_2. We could verify this property by performing model checking of the protocol with the CTL formula $\mathbf{AG}\neg(cs_1 \wedge cs_2)$. As another example, suppose that P_1 has a flag t_1 indicating that it wants to enter its critical section. We could check that the protocol does not starve P_1 by model checking with the CTL formula $\mathbf{AG}(t_1 \rightarrow \mathbf{AF}cs_1)$.

With symbolic model checking, the set of states satisfying a temporal formula is computed by a set of iterative computations similar to those used in reachability analysis. In general, each occurence of a temporal operator, other than $\mathbf{X}$, requires an iterative computation. The main idea is that we can perform all of these operations by OBDD manipulation. We never enumerate individual states.

1.5. OPTIMIZATION OF VARIABLE ORDERING

Since an OBDD is uniquely determined with respect to a given variable ordering, and its size can depend greatly on the chosen ordering, one of the key problems in using OBDDs to find "good" orders. However, the problem to construct an optimal order of a given OBDD is known to be NP-hard [50, 4]. The currently best known exact procedure is based on dynamic programming and has running time $O(n^2 \cdot 3^n)$ [17]. For real-life applications, this method cannot scale beyond around 16 variable functions. To make the problem even worse, Sieling [48] has shown that there is no polynomial time approximation scheme for the variable ordering problem unless P=NP.

1.5.1. STATIC TECHNIQUES

There exist a variety of heuristics to determine a variable ordering before building the OBDD of a function. These heuristics utilize various information specific to the application, e.g., the structure of the combinational circuit being evaluated symbolically [30, 18, 45]. Such approaches do not work well in applications where much of the processing occurs after the symbolic evaluation is complete, e.g., with symbolic reachability analysis and symbolic model checking. Nevertheless these heuristics are often successful at determining good initial variable orders.

1.5.2. DYNAMIC TECHNIQUES

Dynamic Variable Reordering is the process of improving the variable order and hence the size of an already built OBDD. Many different optimization paradigms have been applied to variable reordering, ranging from simple greedy methods to genetic techniques and simulated annealing. One of the most successful strategies is a local search algorithm proposed by Rudell 1993. The so-called *Sifting* [46] algorithm is based on the swap of two variables [19] in the order, which can be carried out locally and very efficiently. That is, variable orderings π and π' differ from each other only in that two adjacent variables x and y are swapped, i.e., $\pi(x) = \pi'(y) = \pi'(x) + 1 = \pi(y) + 1$, then we can convert an entire shared OBDD structure from variable ordering π to π' by just modifying the nodes labeled by variables x and y. No external pointers or pointers from other nodes need to be changed.

With dynamic variable reordering, the storage manager periodically attempts to reduce the number of nodes in the shared OBDD structure by altering the current variable ordering. A simple way is to select some variable (e.g., the one labeling the most nodes) move it to all possible positions via sifting, and leave it at the position that minimizes the total number of nodes.

In the following we describe some state-of-the-art dynamic variable reordering techniques based on sifting that improve on this simplistic approach:

Group Sifting: In this method mutually attracting variables are united into a group. Then, within the sifting algorithm instead of single variables the whole group is moved through the order. There are two different possible sifting based reordering strategies:

1 The individual blocks are reordered by means of a sifting strategy.

2 The variables within each block are reordered by means of a sifting strategy.

Recently, a method called *lazy group sifting* [24] has been introduced. This method relaxes the grouping of present and next state variables during the process of reachable states computation.

Lower Bound Sifting: The *lower bound sifting* algorithm[14] utilizes the theoretical lower bounds of the swap operation [4] to determine when to stop sifting of a variable if there is no improvement in size possible.

Linear Sifting: , The *linear sifting* algorithm [40, 41] combines the efficiency of Sifting and the power of linear transformations . Linear transformations replace one variable x_i by a linear combination of this variable with another. A linear combination is obtained by taking the EXCLUSIVE-OR of the arguments or its complement.

Linear transformations considerably enlarge the search space for the OBDD optimization, when compared to variable reordering, still they enjoy efficient manipulation.

Block Restricted Sifting: The idea behind this method [33] is to move the variables during reordering only within fixed blocks instead of moving them through the complete order. From theory it is known that changing the variable order of a block does not affect the size of the other blocks. The determination of the block boundaries follows from a communication complexity argument. A small information flow between two parts of an OBDD indicates a good candidate for a block boundary. If there is little information flow between two blocks the distribution of variables to these blocks is well chosen. Improving the variable order inside the blocks might lead to a significant reduction of the OBDD size. The information flow is best indicated by the number of subfunctions that cross one level.

Sample Sifting: Sampling is a common heuristic technique applied to optimization problems with huge search spaces. The idea behind the sampling strategy is to choose a relevant sample from the given problem instance to solve the optimization problem for the chosen subset and to generalize the solution to the complete instance.

Applied to the problem of reordering OBDD variables the sampling strategy can be described as follows [49, 25]:

1 Choose some OBDDs or subOBDDs from the common shared OBDD.

2 Copy these OBDDs to a different location.

3 Reorder only the Sample.

4 Shuffle the variables of the original BDD to the newly computed order of the sample.

In sequential verification the time needed for reordering often is unacceptably long. The lazy group sifting algorithm and adaptions of block restricted sifting and sample sifting [39, 42] address this problem.

1.6. VARIOUS IMPROVEMENTS OF THE BDD DATA STRUCTURE

To further improve the efficiency of the BDD data structure, several variants and extensions have been proposed. For the requirements in

specific application fields, these refined models are better suited than the classical OBDD. We mention only
- *ordered functional decision diagrams* (OFDD) [27],
- *transformed binary decision diagrams* (TBDD) [3],
- *parity decision diagrams* (*oplus*-BDD) [22],
- *multi-terminal binary decision diagrams* (MTBDD) [10], [2],
- *edge valued binary decision diagrams* (EVBDD) [28],
- *binary moment diagrams* (BMD) [8],
- *zero-suppresed binary decision diagrams* (ZBDD) [44],
- *partitioned BDD* (POBDD) [26],

The search for more compact representations of switching functions, which preserve the valuable algorithmic properties of OBDDs, is still ongoing.

Due to a lack of space, here we discuss merely the first three of the listed BDD variants.

1.6.1. ORDERED FUNCTIONAL DECISION DIAGRAMS (OFDDS)

If f denotes the function being represented by an OBDD node with label x_i, and g, h denote the functions being represented in the two sons, then Shannon's decomposition holds:

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

However, it is also possible to perform other decompositions in the nodes, e.g. the so-called Davio decomposition

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

These *ordered functional decision diagrams* (OFDDs), introduced in [27], are particularly suited for problems employing large numbers of EXCLUSIVE-OR operations, e.g., the minimization of AND-XOR-polynomials. Taking these ideas one step further, in [15] it is shown that different decomposition types can be combined within the same subgraph while preserving good algorithmic properties (*ordered Kronecker functional decision diagrams*, OKFDDs).

1.6.2. TRANSFORMED BINARY DECISION DIAGRAMS (TBDDS)

Another variant of the OBDD concept converts Boolean functions into functions with easier representations [3], similar to classic transformation concepts like Fourier transformation.

More precisely, let $\mathbb{B}_n$ denote the set of all n-variable Boolean functions $f : \{0,1\}^n \to \{0,1\}$. The transformation approach is based on *cube transformations* τ which are bijective mappings from $\{0,1\}^n \to \{0,1\}^n$. A cube transformation τ induces a mapping $\Phi_\tau : \mathbb{B}_n \to \mathbb{B}_n$ 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:

Theorem 1.6.1 If $\tau : \{0,1\}^n \to \{0,1\}^n$ 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 $\otimes$ be any binary operation on $\mathbb{B}_n$. If $f = f_1 \otimes f_2$, then $\Phi_\tau(f) = \Phi_\tau(f_1) \otimes \Phi_\tau(f_2)$.

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 re-transform the functions.

The realization of this general framework requires to find good techniques for finding suitable cube transformations τ which lead to small OBDD sizes for the transformed versions of the relevant functions. Suitable transformations that have already demonstrated their optimization power include graph-driven transformations based on complete types [3] and linear transformations [35, 34].

1.6.3. PARITY-OBDDS ($\oplus$-OBDDS)

Besides relaxing the ordering restriction [22] or the usage of different decomposition types for Boolean functions, we are focusing on the extension of OBDDs with functional operator nodes, esp. $\oplus$-OBDDs (Mod2-OBDDs), i.e. OBDDs with additional operator nodes computing the Boolean parity of their successors [23]. By introducing parity nodes the representation has the potential of being more compact while on the other hand giving up canonicity. More precisely, a $\oplus$-OBDD P over a set $X_n = \{x_1, \ldots, x_n\}$ of Boolean variables is a rooted directed acyclic connected graph $P = (V, E)$. V is the set of nodes, consisting of non-terminal nodes with out-degree 2 and of terminal nodes with out-degree 0. The two terminal nodes with no outgoing arcs are labeled by the

Boolean constants 0 and 1. The remaining nodes are either labeled with Boolean variables $x_i \in X_n$ (*branching nodes*), or with the binary Boolean function $\oplus$ (EXOR) ($\oplus$*-nodes, functional nodes*). On each path, every variable is permitted to occur at most once.

The function f_P associated with the $\oplus$-OBDD P is determined in the following way: For a given input assignment $a = (a_1, \ldots, a_n) \in \{0,1\}^n$, the function f_P associated with the $\oplus$-OBDD P is determined by extending the Boolean values assigned to the leaves to all other nodes of P as follows:

- Let v_0 and v_1 be the successors of v, carrying the Boolean values $\delta_0, \delta_1 \in \{0,1\}$.
- If v is a branching node, $l(v) = x_i \in X_n$, then v is associated with δ_{a_i}.
- If v is a $\oplus$-node, then v is associated with $\oplus(\delta_0, \delta_1) = (\delta_0 + \delta_1) \bmod 2$.

The function $f_P(a)$ takes the value associated with the root of P.

$\oplus$-OBDD are not a canonical representation for Boolean functions, i.e. there can be different distinct $\oplus$-OBDD representations for the same Boolean function. Therefore, the identification of two $\oplus$-OBDDs representing the same Boolean function becomes an essential operation.

The fastest known deterministic equivalence test for $\oplus$-OBDDs requires time cubic in the number of nodes [51], and therefore, it is not suited for an application within a practical working environment. Equivalence of $\oplus$-OBDDs can also be tested probabilistically [23]. The $\oplus$-OBDDs are algebraically transformed to polynomials over a Galois field and finally the equivalence of two polynomials is tested [26]. Thus, based on an efficient implementation of this technique, the probabilistic equivalence test can be performed in constant time [37].

For the minimization of $\oplus$-OBDDs not only the variable order has to be taken under consideration. Also the number and the placement of the $\oplus$-nodes within the $\oplus$-OBDD has a major influence on the size [37]. Therefore, sophisticated heuristics including all these requirements based on an efficient implementation, ranging from the exchange of adjacent variables to implanting and moving $\oplus$-nodes have to be developed [38].

1.7. WWW-PORTAL FOR BDD RESEARCH

Today the Internet offers the possibility of standardized and global communication without a need for special hardware or expensive infrastructure. To address the needs of the BDD research community, a specialized portal site (`www.bdd-portal.org`) [43] was created that provides easy access to researchers, methods, literature, and other relevant material for BDDs. Furthermore this portal includes a system for providing online BDD functionality by allowing the use of heuristics and tools via a WWW interface. The system is aimed at facilitating the process of determining which heuristics are best suited for different special applications and at allowing easy comparison between different approaches and heuristics. It is open to all researchers and users. By providing a standardized benchmarking environment the system greatly facilitates the evaluation of research results in BDD heuristics.

References

[1] S. B. Akers, "Binary decision diagrams", *IEEE Trans. on Computers*, No. 27, pp. 509-516. 1978.

[2] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Machi, A. Pardo, F. Somenzi, "Algebraic decision diagrams and their application", *Prof. IEEE International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 188-191, 1993.

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

[4] B. Bollig, I. Wegener, "Improving the variable ordering of OBDDs is NP-complete", *IEEE Transactions on Computers*, No. 45, pp. 993–1002, 1996.

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

[6] 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, pp. 205–213, 1991.

[7] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary decision diagrams", *ACM Computing Surveys*, No. 24(3), pp. 293–318, 1992.

[8] R. E. Bryant, Y. A. Chen, "Verification of arithmetic circuits with binary moment diagrams", *Proc. 32th ACM/IEEE Design Automation Conference*, (San Francisco, CA), pp. 535-541, 1995.

[9] J. R.. Burch, E. M. Clark, D. E. Long, "Symbolic model checking with partitioned transition relations", *Proc. of Int. Conf. on VLSI*, 1991.

[10] E. M. Clarc, K. L. McMillen, X. Zhao, M. Fujita, J. C.-Y. Yang, "Spectral transforms for large Boolean functions with application to technology mapping", *Proc. 30th ACM/IEEE Design Automation Conference* (Dallas, TX), pp. 54-60, 1993.

[11] O. Coudert, C. Berthet, J. C. Madre, "Verification of synchronous sequential machines using symbolic execution", *Proc. Workshop on Automatic Verification Methods for Finite State Machines*, Lecture Notes in Computer Science vol. 407, pp. 365–373, 1989.

[12] O. Coudert, J. C. Madre, "A unified framework for the formal verification of sequential circuits", *Proc.IEEE International Conference on Computer-Aided Design*, pp. 126–129, 1990.

[13] O. Coudert, J. C. Madre, "The implicit set paradigm: A new approach to finite state system verification", *Formal Methods in System Design*, No. 6(2), pp. 133–145, 1995.

[14] R. Drechsler, W. Günther, "Using lower bounds during dynamic BDD minimization", *Proc of. IEEE International Conference on Computer-Aided Design* , pp. 29-32 ,1999.

[15] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, M. A. Perkowski, "Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams", *Proc. 31st ACM/IEEE Design Automation Conference*, (San Diego, CA), pp. 415–419, 1994.

[16] E. Dubrova, H. Sack, "Probabilistic verification of multiple-valued functions", *Proc. of the 30th Int. Symp. on Multiple Valued Logic (ISMVL 2000)*, 2000.

[17] S. J. Friedman, K. J. Supowit, "Finding the optimal variable ordering for binary decision diagrams", *IEEE Transactions on Computers*, No. 39, pp. 710–713, 1990.

[18] M. Fujita, H. Fujisawa, N. Kawato, "Evaluation and improvements of Boolean comparision method based on binary decision diagrams", *Proc. of International Conference on Computer-Aided Design (Santa Clara, CA)*, pp. 2–5, 1988.

[19] M. Fujita, Y. Matsunaga, K. Kakuda, "On variable ordering of binary decision diagrams", *Proc. of European Design Automation Conference EDAC*, 1991.

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

[21] D. Geist, I. Beer, "Efficient model checking by automated ordering of transition relation partitions", *Proc. Computer-Aided Verification*, vol. 818, pp. 299–310, 1994.

[22] J. Gergov, Ch. Meinel, "Frontiers of feasible and probabilistic feasible Boolean manipulation with branching programs", *Symposium on Theoretical Aspects in Computer Science*, Springer-Verlag, volume 665 of Lecture Notes in Computer Science, pp. 576–585, 1993.

[23] J. Gergov, Ch. Meinel, "MOD-2-OBDDs - a data structure that generalizes EXOR-sum-of-products and ordered binary decision diagrams", *Formal Methods in System Design*, No. 8, pp. 273–282, 1996.

[24] H. Higuchi, F. Somenzi, "Lazy group sifting for efficient symbolic state traversal of FSMs", *Proc. of IEEE International Conference on Computer-Aided Design*, 1999.

[25] J. Jain, W. Adams, M. Fujita, "Sampling schemes for computing OBDD variable orderings", *Proc. IEEE International Conference on Computer-Aided Design*, pp. 331–638, 1998.

[26] J. Jain, J. Bitner, D. S. Fussell, J. A. Abraham, "Probabilistic verification of Boolean functions", *Proc. Formal Methods in System Design* 1, Kluwer Academic Publishers, pp. 65-116, 1992.

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

[28] Y. T. Lai, M. Pedram, S. B. K. Vrudhula, "EVBDD-based algorithms for integer linear programmin, spectral transformation and function decomposition *IEEE Trans. on Computer-Aided Design*, 13: pp. 959-975, 1994.

[29] S. Lawrence, C. L. Gilles, "Accessibility of information on the web", *Nature, Volume 400, Number 6740*, 1999.

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

[31] K. L. McMillan, *"Symbolic Model Checking"*, Kluwer Academic Publishers, 1993.

[32] Ch. Meinel, *"Modified Branching Programs and Their Computational Power"*, Springer-Verlag, Lecture Notes in Computer Science Vol. 370, 1998.

[33] Ch. Meinel, A. Slobodová, "Speeding up variable reordering of OBDDs", *Proc.of the International Conference on Computer Design*, pp. 338–343, 1997.

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

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

[36] Ch. Meinel, T. Theobald, "Algorithms and Data Structures in VLSI-Design", Springer-Verlag, Berlin, New York, 1998.

[37] Ch. Meinel, H. Sack, "$\oplus$-OBDDs - a BDD structure for probabilistic verification", *Proc. Workshop on Probabilistic Methods in Verification*, pp. 141-151, 1998.

[38] Ch. Meinel, H. Sack, "Algorithmic considerations for parity-OBDD reordering", *Proc. of the 1999 IEEE/ACM Int. Workshop on Logic Synthesis (IWLS99)*, pp. 71–74, 1999.

[39] Ch. Meinel, K. Schwettmann, A. Slobodová, "Application driven variable reordering and an example implementation in reachability analysis", *Proc. of ASP-DAC'99*, Hongkong, pp. 327-330, 1999.

[40] Ch. Meinel, F. Somenzi, T. Theobald, "Linear sifting of decision diagrams", *Proc. of 34th IEEE International Conference on Computer-Aided Design*, (Anaheim, CA), IEEE Computer Society Press, pp. 202-207, 1997.

[41] Ch. Meinel, F. Somenzi, T. Theobald, "Function decomposition and synthesis using linear sifting", *Proc. ASP-DAC'98*, (Yokohama, Japan),pp. 81-86, 1998.

[42] Ch. Meinel, Ch. Stangier, "Speeding up symbolic model checking by accelerating dynamic variable reordering", *Proc. of 10th ACM Great Lake Symposium on VLSI*, (Chicago, USA), pp. 39-42, 2000.

[43] Ch. Meinel, A. Wagner, "WWW.BDD-Portal.Org - an electronic basis for cooperative research in EDA", *Proc. CRIS 2000*, (Espoo, Finnland), pp. 1–7, 2000.

[44] S, Minato, "Zero-supressed BDDs for set manipulation in combinatorial problems", *Proc. 3oth ACM/IEEE Design Automation Conference*, (Dallas, TX), pp. 272-277, 1993.

[45] S. Minato, N. Ishiura, S. Yajima, "Shared binary decision diagrams with attributed edges", *Proc. 27th ACM/IEEE Design Automation Conference* (Florida, FL), pp. 52–57, 1990.

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

[47] H. Sack, E. Dubrova, Ch. Meinel, "Mod-p decision diagrams: A data-structure for multiple-valued functions", *Proc. of the 2000 IEEE/ACM Int. Workshop on Logic Synthesis IWLS*, pp. 341–348, 2000.

[48] D. Sieling, "On the existence of polynomial time approximation schemes for OBDD Minimization", *Proc. of Symposium on theoretical aspects in Computer Science*, pp. 205–215, 1998.

[49] A. Slobodová, Ch. Meinel, "Sample method for minimization of OBDDs", *Proc. of the International Workshop on Logic Synthesis*, pp. 311–316, 1998.

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

[51] S. Waack, "On the descriptive and algorithmic power of parity ordered binary decision diagrams", *Proc. 14th Symp. on Theoretical Aspects of Computer Science*, Springer-Verlag, LNCS 1200, pp.201-212,1997.