Electronic Colloquium on Computational Complexity - Reports Series 1996 - available via:

ECCC
TR96-010

FTP: ftp.eccc.uni-trier.de:/pub/eccc/
WWW: http://www.eccc.uni-trier.de/eccc/
Email: ftpmail@ftp.eccc.uni-trier.de with subject 'help eccc'

An Adequate Reducibility Concept for Problems Defined in Terms of Ordered Binary Decision Diagrams

Christoph Meinel, Anna Slobodová*†
FB IV – Informatik
Universität Trier
54286 Trier, Germany

Abstract

Reducibility concepts are fundamental in complexity theory. Usually, they are defined as follows: A problem Π is reducible to a problem Σ if Π can be computed using a program or device for Σ as a subroutine. However, in the case of such restricted models as ordered binary decision diagrams (OBDDs), this approach is very limited in its power and leads necessarily to concepts which are quite meaningless for complexity theoretic considerations.

In the following, we propose a new reducibility concept for OBDDs. We say that Π is reducible to Σ if an OBDD for Π can be constructed by applying a sequence of computationally elementary operations to an OBDD for Σ instead of using the unmodified OBDD for Σ as a subroutine. Hence, Π is reducible to Σ means that it is somewhat clear how to obtain a program for Π from a program for Σ without insisting on an almost unmodified use of the original program.

Although well-motivated, defining reducibility in terms of sequences of elementary operations has the disadvantage that it is very difficult to handle dynamically changing structures. The main purpose of this paper is to establish an algorithmically based static description of this dynamical reducibility notion which makes adaquate complexity theoretic investigations possible.

1 Introduction

Reducibility is one of the most basic notions in complexity theory. It provides a fundamental tool for comparing the computational complexity of different problems. The key idea is to use a program for a device that solves one problem Σ as a subroutine within the computation of another problem Π. If this is possible, Π is said to be reducible to Σ. Reductions provide the possiblity to conclude upper bound results on the computational complexity of problem Π and lower bounds for Σ, if one insists that the program for Π designed around the subroutine for Σ respects certain resource complexity bounds of interest.

In the past, a great variety of different reducibility notions have been investigated in order to get a better understanding of the different computational paradigms and/or resource bounds. Here we only mention polynomial-time Turing reducibility, log-space reducibility, polynomial projection reducibility, and NC^1-reducibility (see, e.g., [Lee90], [BDG88]). This great variety of different reducibility notions is a consequence of the fact that the computational power implemented within a reducibility notion must not be stronger than the computational power of the complexity classes under consideration. Otherwise, it would be possible to hide some essential computations within the reduction, instead of respecting the

*Supported by DFG grant Me 1077/2-2
†We are grateful to DAAD ACCIONES INTEGRADAS, grant Nr. 322-ai-e-dr

computational paradigma and/or resource bounds under consideration and, consequently, no relevant complexity theoretic results can be obtained. For example, polynomial-time reducibility does not give any relevant insight into the computational complexity of, say, logarithmic-time bounded computations. Another example provides NC^1-reducibility which is not the right tool for the investigation of classes like AC^1.

This fact causes some troubles if one considers very restricted models, like eraser Turing machines [KMW88], real-time branching programs [KW87], or ordered binary decision diagrams [Bry86]. The consideration of the complexity classes defined by such restricted models is interesting and important since, on one hand, they occur in connection with (our merely minor abilities in) lower bound considerations, and, on the other hand, they provide a complexity theoretic framework for the investigation of data structures used in practical applications. The difficulty in defining adapted reducibility notions for such restricted classes lies in the fact that the frame for using subroutines becomes extremely restrictive, since it devours almost all of the computational resources of the complexity class under consideration. Hence, the computational power implemented in the reducibility notion has to be very restricted. As a result, one obtains a reducibility notions that relate merely highly similar problems, and are, hence, only of limited use.

Let us consider complexity classes defined by ordered binary decision diagrams (OBDDs), i.e., reduced oblivious read-once branching programs. In [BW95], Bollig and Wegener tried to introduce a reducibility notion for these classes. In order to obtain a reducibility notion whose computational power does not exceed the computational power of the underlying OBDDs, they successively restricted the projection [SV81] (which fits well to complexity classes defined by branching programs [Mei89]) up to the point that it respects the read-once property. Although the resulting *read-once projection* formally defines a reducibility for OBDD-based complexity classes, there are some properties which cast doubts whether this notion is meaningful from the viewpoint of complexity theory. For example, with respect to read-once projections, even constant functions are not reducible to each other although they have almost identical OBDD realizations. The reason for these difficulties lies not in a bad definition – Bollig and Wegener used the broadest reducibility notion that can be obtained on this line – it lies in the fact that almost all computational resources of OBDD-classes are devoured by the OBDDs used as subroutines. Hence, in the case of very restricted classes, the usual reducibility approach to construct a program for a problem 'around' the more or less unchanged program for the other problem does not give the desired results.

In the paper, we propose a new reducibility concept that overcomes the difficulties which arise if one applies the traditional reduction concept in the context of very restricted complexity classes like those defined in terms of OBDDs. The new reducibility concept is based on the idea that a problem Π is reducible to a problem Σ if a program (in our case an OBDD) for Π can be constructed from a given program for Σ by applying a sequence of elementary (i.e., performable in constant time) operations. Hence, Π is reducible to Σ means that it is somewhat clear how to obtain a program for Π from a program for Σ without insisting on an almost unmodified use of the original program.

Although well-motivated, defining reducibility in terms of sequences of elementary operations has the disadvantage that it is very difficult to handle dynamically changing structures. The main purpose of this paper is to establish an algorithmically based static description of this dynamical reducibility notion. This simplifies the handling of the reducibility concept in complexity theoretic investigations, e.g. for deriving lower bounds on the size of OBDD representation for a function, as well as in practical applications, where the estimation of the OBDD-sizes may be crucial.

2 Notations and Preliminaries

Let X_n denote the set $\{x_1, x_2, ..., x_n\}$ of Boolean variables. A variable ordering on X_n is meant as a total order and is described by a permutation of the index set $I_n = \{1, ..., n\}$, i.e. $x_i < x_j$ iff $\pi^{-1}(i) < \pi^{-1}(j)$.

Identity is a trivial permutation. Two orderings are said to be *consistent* if there is no pair (x_i, x_j) such that x_i precedes x_j in one ordering and x_j precedes x_i in the other.

Speaking about functions, we mean Boolean functions $\{false, true\}^n \longrightarrow \{false, true\}$. The standard representation of *false* and *true* is 0 and 1, respectively. However, it will be more convenient for us to use -1 for *false* and $+1$ for *true*. Speaking about labeled graphs, we write $\cong$ for the isomorphism.

Definition.
An *ordered binary decision diagram (OBDD)* over X_n is a connected acyclic directed graph with the following properties:

- There is one distinguished node (called *root*) without in-coming edges;
- Nodes without out-going edges (*sinks*) are labeled by -1 or $+1$;
- All non-sink nodes are labeled by variables from X_n and have two out-going edges (called *true*- and *false-edge*) labeled by $+1$ and -1, respectively;
- Each node has a *negation mark* -1 or $+1$;
- All variable orderings defined by the occurence of variables on root-to-sink paths are consistent.

In order to make figures more clear, we use solid lines for the 1-edges and dotted lines for the -1-edges of an OBDD. The negation mark 1 is usually omitted in the figures.

The nodes that are labeled by the same variables form a *level*. Let π be a variable ordering. An ordered binary decision diagram is a πOBDD if all variable orderings defined by the occurence of variables on root-to-sink paths are consistent with π. The size of an OBDD P is defined as the number of non-sink nodes and is denoted by $size(P)$.

Each node v of an OBDD *represents* a Boolean function as follows: Each input assignment $\alpha = (a_1, \ldots, a_n)$ uniquely determines a v-to-sink path $p(\alpha)$ according to the following rule: At an inner node with label x_i, the outgoing edge with label a_i is chosen. Let $\mathrm{sgn}(\alpha)$ denote the product of the negation marks on the nodes on $p(\alpha)$. Now v represents the function f if, for each input α, $p(\alpha)$ terminates in a sink labeled with $f(\alpha)\mathrm{sgn}(\alpha)$. The function represented by an OBDD is the function represented by its root. We will not distinguish between an OBDD and its root (as long as it do not introduce an ambiguity). In this sense, the successors of a node are the nodes reachable via edges starting in the node as well as the subOBDDs rooted in these nodes. An OBDD (respectively, a πOBDD) for a function f is denoted by OBDD(f) (respectively, πOBDD(f)).

The defined OBDD model slightly differs from the one usually used. The introduction of the negated nodes is motivated by the existing OBDD-implementations [BRB90, Lon93, Ros95], where the use of negated edges allows to save up to half the size of the representation. In these implementations, the root of an OBDD has one possibly negated in-coming edge (reference). In order to model this situation, we assign the negation marks to nodes and avoid the introduction of dummy nodes for the origins of the reference edges.

If P is an OBDD, then $\overline{P}$ denotes an OBDD obtained from P by multiplying the negation mark of the root by -1. Two OBDDs are *(functionally) equivalent* (denoted by $\equiv$) if they represent the same function. An OBDD is called *reduced* if all nodes have negation mark $+1$ and no two subgraphs represent equivalent OBDDs. We remark that an OBDD can be reduced in linear time [SW93]. For a fixed variable ordering, the representation of a Boolean function in terms of reduced OBDD is uniquely determined [Bry86].

The main interest of this paper is the development of a complexity theoretic reducibility concept for OBDDs. Unfortunately, in the context of OBDDs, the notions 'reduce' and 'reduction' have a fixed meaning in the abovementioned sense. Speaking about reductions in the complexity theoretical sense, we avoid the terminological ambiguity by using the term OBDD-transformation instead of OBDD-reduction.

The non-uniformity of OBDDs and the sensitivity of the structure and the size of an OBDD to a variable ordering implicitly specify the notion of 'problem' in the context of OBDDs.

Definition.
A *problem* (f, π) is a sequence of pairs $((f_n, \pi_n))_{n=1}^{\infty}$, where $f_n : \{false, true\}^n \longrightarrow \{false, true\}$ is a Boolean function and π_n is a permutation on $I_n = \{1, ..., n\}$. For each n, (f_n, π_n) is called an *instance* of the problem.

For each problem (f, π), there is a uniquely determined sequence of reduced OBDDs $(\mathrm{OBDD}(f_n, \pi_n))_{n=1}^{\infty}$.

3 OBDD-Transformations

Our aim is the introduction of a reducibility notion that compares the problems with respect to a 'similarity' in their OBDD-representations. What the 'similarity' means can be intuitively described as a possibility to construct a target OBDD from another one by performing a sequence of 'elementary' operations. An operation is considered as elementary if it can be performed in constant time (under unit cost measure). We define elementary operations via schemes instead of a long formal description. The original node always appears on the left side and the transformed one on the right. Marks and labels that are not significant for the operation are omitted. The label (the negation mark) of a node is positioned on the left (respectively, right) of the node.

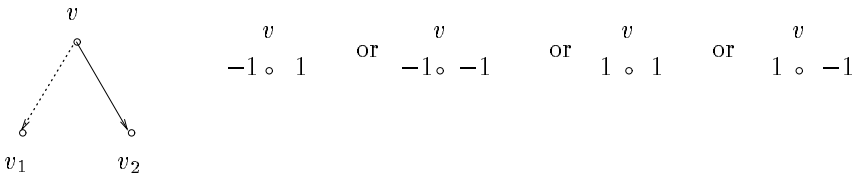
Definition.
Elementary operations are:

Operation 1. Setting/deletion of the negation mark of a node.

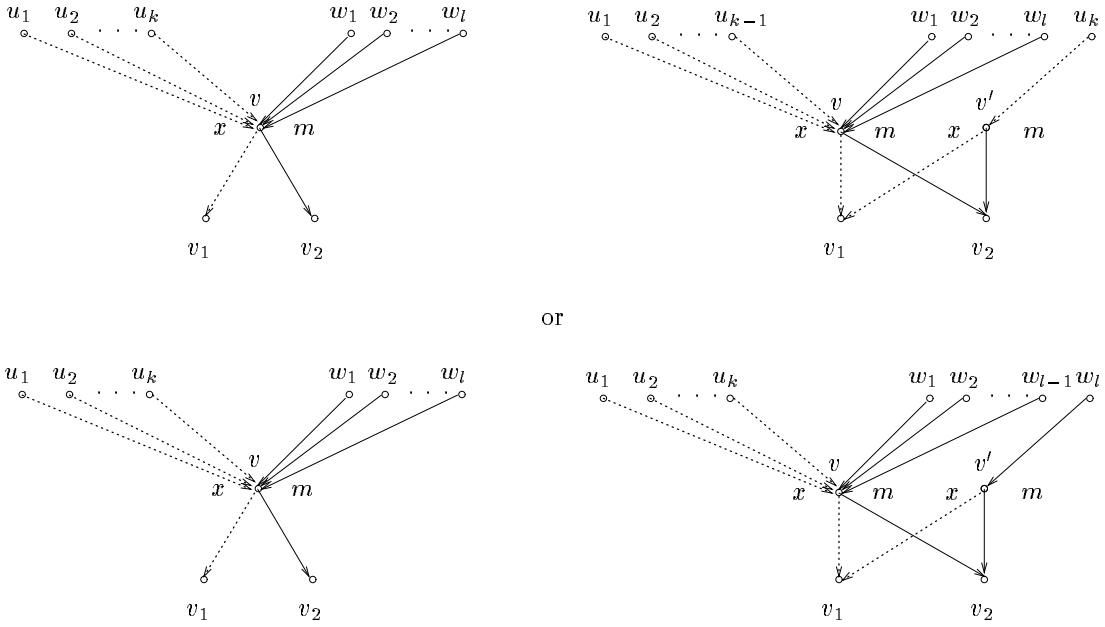
Operation 2. Exchange of out-going edges of a node, i.e. negation of their labels.

Operation 3. Redirection of one out-going edge towards the second one.

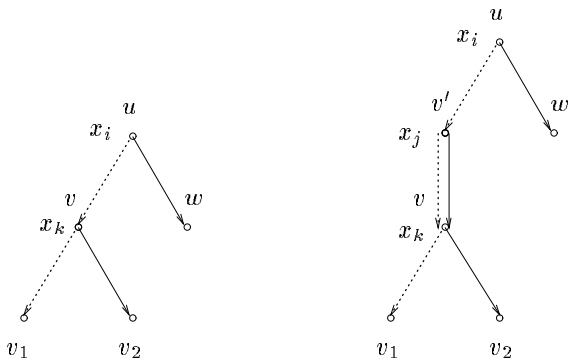
Operation 4. Conversion of a node to a sink.



Operation 5. Node splitting, i.e. redirection of an edge to a newly created equivalent node.



Operation 6. Introduction of a dummy node labeled with a variable that is consistent with the considered variable ordering (let $x_i < x_j < x_k$ in the picture).



The abovedefined operations have clear semantics. The first operation negates the subfunction defined in a node, the second operation negates the evaluation of the checked literal, the third one corresponds

to the restriction, and the fourth operation provides a replacement of a subfunction by a trivial function (constant). Unlike the first four operations that change the functionality of the OBDD, the last two allow to work with an unreduced OBDD for a function, and hence, to perform the subsequent operations merely on some distinguished subfunctions.

There are several operations that look like trivial, e.g. redirection of an edge to another node. However, being in a node, we only have local information about the OBDD. Another observation is that not every elementary operation has an inverse counterpart, e.g. Operation 4. Anyway, as it is discussed later, also such operations can be simulated by a sequence of elementary operations. However, the length of the sequence reflects the complexity of the simulated operation.

A natural requirement by a transformation of OBDDs is the possibility of a renaming variables. Among others, it allows to move from one variable ordering to another. In order to preserve the 'read-once property', we have to insist that no two different variables are identified by renaming.

Now we are ready to define the reducibility/transformability notion formally.

Definition.
A problem (f, π) is an *(OBDD-)transformation* of a problem (g, σ) (written as $(f, \pi) \leq_{OBDD} (g, \sigma)$), if for each n, there is an m such that the reduced π_nOBDD(f_n) can be obtained from the reduced σ_mOBDD(g_m) by a sequence of elementary operations completed by the reduction and a renaming of variables (i.e., application of a bijective mapping on the set of variables that occur in the reduced OBDD).

OBDD-transformations are more powerful than read-once projections introduced in [BW95].

Proposition 1.
Read-once projections are special OBDD-transformations.

Proof. Let $f = (f_n)$ be a read-once projection of $g = (g_n)$. According to the definition of read-once projection, for each n there is a p_n and a mapping $\tau : X_n \mapsto Y_{p(n)}$ such that:
1) $y_i \in \{-1, +1\} \cup X_n \cup \{\overline{x_j} \mid x_j \in X_n\}$ for each $1 \leq i \leq p(n)$.
2) For any i and j, $i \neq j$, if $y_i \in \{x_r, \overline{x_r}\}$ then $y_j \notin \{x_r, \overline{x_r}\}$. (read-once property)
3) $f_n(x_1, ..., x_n) = g_{p(n)}(y_1, ..., y_{p(n)})$.

An OBDD(f_n) can be obtained from an OBDD$(g_{p(n)})$ by applying elementary operations of one type on each particular level of nodes according to the rules described below, completed by the reduction of equivalent nodes and by a certain renaming. The operations as well as the renaming depend on the projection τ as follows:

- If $y_i = +1$, or -1, then Operation 3 is performed on each node v labeled by y_i.
- If $y_i = x_j$, then y_i will be renamed to x_j in all occurences.
- If $y_i = \bar{x_j}$, then Operation 2 is performed on each node labeled by y_i, and y_i is renamed to x_j

The nodes modified by Operation 3 are redundant and are removed in the reduction. The read-once property assures the injectiveness of the renaming. It is easy to see that, for each π, (f, π) is an OBDD-transformation of (g, π). □

Proposition 2.
There are OBDD-transformations which can be derived by means of one elementary operation, but cannot be obtained via any read-once projection.

Proof. Let us consider the constant functions 0 and 1. Obviously, none of the functions is a read-once projection of the other. On the other side, independently from the variable ordering, the OBDD for 0 can be obtained from an OBDD for 1 (and vica versa) by applying Operation 1. □

4 OBDD-Transformer and the Algorithms *Derive* and *Compose*

The definition of the OBDD-transformation via a sequence of elementary operation matches our intuition about an OBDD-reducibility concept. However, the fact that the operations are applied to a dynamically changed OBDD complicates the description of an OBDD-transformation.

In this section, we develop an algorithmical framework for efficient representation and handling of OBDD-transformations. We will show that every sequence of elementary operations can be described by a structure that has the same nature as the considered OBDD-model. This can be seen as an evidence that the introduced reducibility notion reflects the properties of OBDD-representations adequately.

4.1 OBDD-Transformer

Definition.
Let $\tilde{X_n} = X_n \cup \{\bar{x} | x \in X_n\} \cup \{1_x | x \in X_n\} \cup \{-1_x | x \in X_n\}$. An *OBDD-transformer* (or simply *transformer*) over X_n is a connected directed acyclic graph with the following properties:

- There is one distinguished node (called *root*) without in-coming edges.
- Nodes without out-going edges (*sinks*) are labeled by $-1, +1$, or $\perp$ (the last symbol has the meaning of no label assigned).
- All non-sink nodes are labeled by symbols in $\tilde{X_n}$ and have two out-going edges (called *true-* and *false-edge*) labeled by $+1$ and -1, respectively.
- Each node has a negation mark -1 or $+1$.
- All variable orderings defined by the occurence of variables on root-to-sink paths are consistent.

For each label $x \in \tilde{X_n}$, $|x|$ denotes the associated variable, i.e. $|x_i| = |\bar{x}_i| = |1_{x_i}| = |-1_{x_i}| = x_i$. A πOBDD-transformer is an OBDD-transformer where the orderings of the variables associated with the labels of the nodes on root-to-sink paths are consistent with π. Obviously, each πOBDD is a πOBDD-transformer. The size of a transformer T is defined as the number of its non-sink nodes and is denoted by $size(T)$.

4.2 The Algorithm *Derive*

Next, we prove that an OBDD-transformer uniquely describes an OBDD-transformation and, vice versa, every OBDD-transformation can be described by some OBDD-transformer. In order to do this, we construct an algorithm *Derive* that realizes an OBDD-transformation defined by a given OBDD-transformer. *Derive* is applicable on any pair consisting of a π_1OBDD and a π_2OBDD-transformer as long as π_1 and π_2 are consistent. In that case, handling π_1 and π_2 as relations, we can define an ordering π (over the union of variables that appear in the OBDD and the transformer) as a transitive closure of the union $\pi 1 \cup \pi 2$ and all comparisons are meant w.r.t. π.

The result of $Derive(P, T)$ is denoted by $P \diamond T$. If the variable orderings in T and P are not consistent, $P \diamond T$ remains undefined. The algorithm starts in the roots of P and T, scans the graphs in parallel, and creates the result recursively. The information in the root of T describes the required changes in the root of P. Instead of modifying P, we create a new graph as the result. The mark -1 corresponds to Operation 1, the negative literal to Operation 2, labels 1_x and -1_x to Operation 3, and the sink node to Operation 4.

We explain this idea more deeply. If x is a label (of the root) of T, then (the root of) $T_{|x|=\delta}$ is the δ-successor (of the root) of T. Similarly, $P_{x=\delta}$ is the δ-successors of P.

Trivial case:
– If T is a sink labeled by $\perp$, then $P \diamond T$ will be a graph isomorphic to P but its mark is inverted if the mark of T is -1.
– If T is a sink labeled by a constant, then $P \diamond T$ is a sink isomorphic to T.

Nontrivial case:
Let x be the label of P and y the label of T. The program splits according to the positions of x and $|y|$ in the ordering.
– If $x < |y|$, then a node with the label and mark of P is created and with true-, and false-successor $P_{x=1} \diamond T$, and $P_{x=-1} \diamond T$, respectively.
– If $x > |y|$, then a node labeled by $|y|$ with the same mark as T is created, with the true-, and false-successor $P \diamond T_{|y|=1}$, and $P \diamond T_{|y|=0}$, respectively.
– If $x = |y|$, then a node labeled by x is created and the mark of the node will be the product of the marks of P and T. The true- and false-successor of the node depend on y:

		true-successor	false-successor
$y = x$	:	$P_{x=1} \diamond T_{x=1}$	$P_{x=-1} \diamond T_{x=-1}$
$y = \bar{x}$	:	$P_{x=-1} \diamond T_{x=1}$	$P_{x=1} \diamond T_{x=-1}$
$y = 1_x$	:	$P_{x=1} \diamond T_{x=1}$	$P_{x=1} \diamond T_{x=-1}$
$y = -1_x$	:	$P_{x=-1} \diamond T_{x=1}$	$P_{x=-1} \diamond T_{x=-1}$

Particularly, if T is an OBDD, then for each OBDD P holds $P \diamond T = T$.

For details, see the pseudocode of the algorithm in the appendix. The algorithm is presented in a form that is close to the respective transformation and produces an OBDD that is not reduced. The reduction running in parallel is easy to implement in the algorithm without changing its assymptotic time and space performance.

Proposition 3.
The time as well as the space complexity of the algorithm Derive *on an input* (P, T) *is bounded by* $\mathcal{O}(size(P) \cdot size(T))$. *The size of the output is bounded by* $size(P) \cdot size(T)$.

Proof. The main observation in the complexity analysis of the algorithm is that each pair of nodes (u, v), where $u \in P$ and $v \in T$, generates at most two recursive calls to *Derive_step*. □

4.3 The Algorithm *Compose*

An important property of the OBDD-transformation is its transitivity. This property should be expressible in terms of transformers, too. In the following, we give an algorithm *Compose* for the composition of two transformers that will reflect the 'concatenation' of two OBDD-transformations. The result of $Compose(T_1, T_2)$ is denoted by $T_1 \circ T_2$. *Compose* is closely related to *Derive*. It is designed in such a way that $(P \diamond T_1) \diamond T_2$ is isomorphic to $P \diamond (T_1 \circ T_2)$ which corresponds to the idea of the composition of two transformations.

For details, see the pseudocode of the algorithm given in the appendix.

Proposition 4.
The time as well as the space complexity of the algorithm Compose *on an input* (T_1, T_2) *is bounded by* $\mathcal{O}(size(T_1) \cdot size(T_2))$. *The size of the output is bounded by* $size(T_1) \cdot size(T_2)$.

Proof. The main observation in the complexity analysis of the algorithm is that each pair of nodes (u, v), where $u \in T_1$ and $v \in T_2$, generates at most two recursive calls to *Compose_step*. □

Proposition 5.
For any πOBDD P and πOBDD-transformers T_1, T_2 holds:

$$P\Diamond(T_1 \circ T_2) \cong (P\Diamond T_1)\Diamond T_2. \qquad (*)$$

Proof. The construction of *Compose* was based on (*). Its correctness can be easily shown by induction on the top variable of the triple (P, T_1, T_2) (i.e., the top most label w.r.t. the given variable ordering). The basis of the induction is the case when all arguments are sinks. *Derive_step* and *Compose_step* always generate recursive calls with a smaller top variable that allows to use the inductive hypothesis. The proof is then reduced to the consideration of all possible cases which are characterized by different relative positions of the top variables. □

5 Algorithmic Description of OBDD-Transformations

The well-motivated concept of defining OBDD-transformations via sequences of elementary operations has the big disadvantage that it is very difficult to handle the dynamically changing structures occuring in the course of an OBDD-transformation. In the following section, we prove that the application of any sequence of elementary operations to a given OBDD P can be simulated by the algorithm *Derive* applied to P and a suited transformer T. Hence, we obtain a static description of the dynamical transformation process in terms of the OBDD-like structure of transformers. Moreover, the size and the shape of the transformer gives all information needed to understand the transformation itself. For example, it is possible to estimate the size of the target OBDD from the sizes of the original OBDD and the transformer.

In fact, we prove even more. We show that the concepts of transforming OBDDs by means of applying sequences of elementary operations is *equivalent* to the concept of computing OBDDs by means of the *Derive* algorithm and certain transformers.

The second part of this section, contains several examples of OBDD-transformations that illustrate the use of transformers.

5.1 Transformers vs. Sequences of Elementary Operations

Theorem 6.
(f, π) is an OBDD-transformation of (g, σ), $(f, \pi) \leq_{OBDD} (g, \sigma)$, if and only if for every n, there is an m and an OBDD-transformer T_n such that $\pi_n OBDD(f_n)$ and $\sigma_m OBDD(g_m)\Diamond T_n$ are equivalent up to a renaming of the variables.

The **proof** of Theorem 6 is the consequence of the next two lemmas.

Lemma 7.
Let T be a πOBDD-transformer and let P be a πOBDD. Then $P\Diamond T$ is a transformation of P, i.e., it can be obtained from P by applying a sequence of elementary operations.

Proof.
In order to prove the statement, we show that, for each P and T, there is a P' derivable from P by a sequence of elementary operations such that $P' \equiv P\Diamond T$. The proof is done by induction on the depth of T (i.e., on the length of the longest path in T).

Basis: depth(T)=1.
T consists of a sink labeled by $-1, +1$,or $\perp$. Let the negation mark of T be 1. In the first two cases, $P \diamond T \cong T$ and we obtain T from P by applying Operation 4. In the last case, when T consists of the sink labeled by $\perp$, $P \diamond T \cong P$. If the negation mark of T is -1, then we use Operation 1 as well.

Inductive step: Let the statement hold for all transformers of depth less than k, $k > 1$.

The information in a node is represented by a tuple [*var,true-successor, false-successor, negation mark*]. Let $P = [x_P, P_t, P_f, m_P]$ and $T = [x_T, T_t, T_f, m_T]$.

a) $x_P = |x_T| = x$
There are several subcases that correspond to x_T. If $m_T = -1$, we change the negation mark of P. Applying Operation 5, we separate the subgraphs P_t and P_f, i.e., we derive from P an OBDD $Q = [x, Q_t, Q_f, m_P * m_T]$ such that $Q_t \cong P_t$, $Q_f \cong P_f$ and Q_t and Q_f are disjunct.

i) $x_T = x$
According to *Derive*, $P \diamond T = [x, P_t \diamond T_t, P_f \diamond T_f, m_P * m_T]$. $P_t \diamond T_t \cong Q_t \diamond T_t$ and $P_f \diamond T_f \cong Q_f \diamond T_f$. Since T_t and T_f are transformers of depth less than k, we can use the inductive hypothesis on $Q_t \diamond T_t$ and $Q_f \diamond T_f$. There is a P'_t (P'_f) derivable from Q_t (Q_f) and, hence, from P_t (P_f) by applying elementary operations such that $P'_t \equiv P_t \diamond T_t$ ($P'_f \equiv P_f \diamond T_f$).

ii) $x_T = \bar{x}$
According to *Derive*, $P \diamond T = [x, P_f \diamond T_t, P_t \diamond T_f, m_P * m_T]$. Applying Operation 2 to Q, we obtain an OBDD $R = [x, Q_f, Q_t, m_P * m_T]$. Since T_t and T_f are transformers of depth less than k, we can use the inductive hypothesis on $Qf \diamond T_t$ and $Q_t \diamond T_f$. There is a P'_t (P'_f) derivable from Q_f (Q_t) and, hence, from P_f (P_t) by applying elementary operations such that $P'_t \equiv P_f \diamond T_t$ ($P'_f \equiv P_t \diamond T_f$).

iii) $x_T = 1_x$
According to *Derive*, $P \diamond T = [x, P_t \diamond T_t, P_t \diamond T_f, m_P * m_T]$. Applying Operation 3 on Q we modify Q to $R = [x, Q_t, Q_t, m_P * m_T]$. Since T_t and T_f are transformers of depth less than k, we can use the inductive hypothesis on $Q_t \diamond T_t$ and $Q_t \diamond T_f$. There is a P'_t and P'_f derivable from Q_t and, hence, from P_t by applying elementary operations such that $P'_t \equiv P_t \diamond T_t$ ($P'_f \equiv P_t \diamond T_f$).

iv) $x_T = -1_x$
According to *Derive*, $P \diamond T = [x, P_f \diamond T_t, P_f \diamond T_f, m_P * m_T]$. Q is modified by Operation 3 to $R = [x, Q_f, Q_f, m_P * m_T]$. Then we proceed similarly as in the previous case.

In all subcases we obtain $P' = [x, P'_t, P'_f, m_P * m_T]$ which is equivalent to $P \diamond T$.

b) $x_P > |x_T|$
P can be expanded by multiple application of Operation 6 to an equivalent OBDD P' with the variable x_T on the top. Then we have the case a).

c) $x_P < |x_T|$
Let $T' := [x_P, T, T, 1]$. According to *Derive*, $P \diamond T' = [x_P, P_t \diamond T, P_f \diamond T, m_P] = P \diamond T$. A similar "unfolding" can be done recursively in $P_t \diamond T$, resp. $P_f \diamond T$, until the restrictions of P with top variables greater or equal to x_T are reached. Thus, the top part of $P \diamond T$ will be isomorphic to P and the bottom part consists of OBDDs $P_i \diamond T$, $1 \leq i \leq r$, for respective restrictions $P_1, ..., P_r$ of P. Each of them fulfils the assumption of case a) or b). □

Lemma 8.
Let P be a πOBDD. If P' is a πOBDD obtained from P by applying a sequence of elementary operations, then there is a πOBDD-transformer T such that P' is equivalent to $P \diamond T$.

Proof.
The corresponding transformers are constructed inductive, with respect to the number of elementary operations that transform P to P'.

Basis: $k = 1$.
Let v be a node of P which is transformed via an elementary operation. Since Operations 5 and 6 do not change the functionality of P, the trivial transformer (a sink labeled by $\perp$ with negation mark $+1$) is the solution. In the other cases, the transformer T is constructed as follows. We start with a graph T_{top} isomorphic with the subgraph of P that consists of all root-to-v paths in P. Then we change all negations marks in T_{top} to 1. The node v' in T that corresponds to v is changed according to the used elementary operation:

Let t be a sink node labeled by $\perp$ with negation mark equal 1.

Operation 1: v' has the inverse negation mark of v and its true- and false-edges enter t.

Operation 2: v' is labeled by the negated variable of v and the true- and false-edges enter t.

Operation 3: If the true- (false-) edge of v should be redirected to the false- (true-) successor of v, then v' is labeled by -1_x (respectively, 1_x), and both successors are t.

Operation 4: v' will be the respective sink.

All other edges (according to definition each non-sink node of a transformer has two out-going edges) enter t.

The correctness of the construction can be shown by induction on the index of the top variable of T.

Inductive step: Let $P_0, P_1, \ldots, P_k, P_{k+1}$ be the sequence of OBDDs such that for $1 \leq i \leq k$ holds that P_{i+1} is obtained from P_i via one elementary operation (applied to one node), and let $P_0 = P$. According to the inductive hypothesis, there are transformers T_k and T' such that $P \diamondsuit T_k = P_k$ and $P_k \diamondsuit T' = P_{k+1}$. Applying Proposition 5, we get $P \diamondsuit (T_k \circ T') \cong (P \diamondsuit T_k) \diamondsuit T' \cong P_k \diamondsuit T' = P_{k+1}$, i.e., $T_k \circ T'$ is a transformer that transforms P into P_{k+1}. $\square$

We say that the sequence of OBDD-transformers $(T_n)_{n=1}^{\infty}$ from the theorem realizes an OBDD-transformation $(f, \pi) \leq_{\text{OBDD}} (g, \sigma)$. We remark that there is nothing said about the number of variables in T_n yet (e.g., see examples at the end of this section).

5.2 Examples

We have proved in Theorem 6 that transformers provide an alternative formalism for OBDD-transformations which is supported by the output-efficient algorithms *Derive* and *Compose*. The effect of this formalism is demonstrated by the following examples.

5.2.1 Redirection of an Edge

In Section 3, we discussed the operation of redirecting an edge of an OBDD P to another node of P. The situation is sketched in the Figure 1. The above mentioned non-elementarity of this operation follows

from the fact that each node stores only local information.

Let u, v and w be nodes of P, v be δ-successor of u. We want to redirect the δ-edge of u to w. A transformer T that simulates this operation on P is constructed as follows. In order to describe the operation, we have to specify the node to which the edge should be redirected. The idea is the same as in the basis case of the proof of Lemma 8. The top part of T is constructed as an isomorphic counterpart of all root-to-u paths in P. All other edges enter a sink labeled by $\perp$ with negation mark equal 1. The δ-edge of u enters a root of an OBDD that is isomorphic with the OBDD rooted in w.

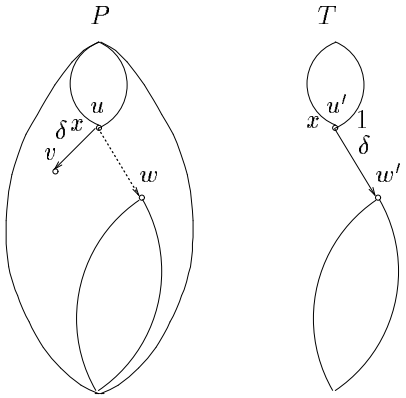


Figure 1

5.2.2 $(\mathbf{MOD\text{-}2}, \pi) \leq_{OBDD} (\mathbf{MOD\text{-}4}, \pi)$

Another example, describes the transformers that realize the transformation of MOD-4 to MOD-2 function with respect to trivial variable ordering (the change to any other ordering is straightforward) Formally, MOD-$2^i_n(x_1, ..., x_n) = 1$ iff $\sum_{j=1}^{n} x_j$ MOD-$2^i = 0$. All negation marks are equal to 1 and are omitted in the figures.

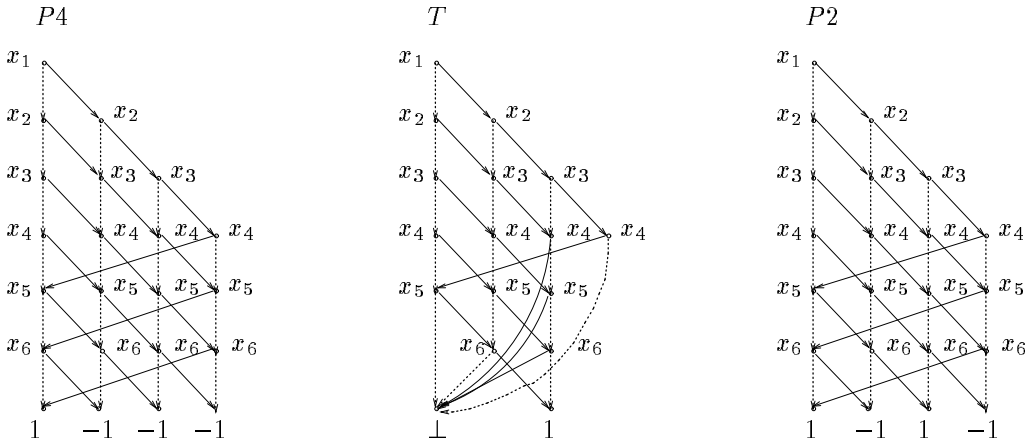


Figure 2 $P4 \diamond T = P2$

$P4$ is an OBDD for MOD-4_6 and $P2$ is an OBDD for MOD-2_6. T is a transformer that modifies $P4$ to $P2$, i.e., $P4 \diamond T = P2$. We leave the OBDDs unreduced since it is easier to see what happens. The idea behind the construction of T is based on the observation that MOD-$2_n(x_1, ..., x_n) = 1$ iff $\sum_{i=1}^{n} x_i$ MOD 4 is equal 0 or 2.

Another transformer which realizes an OBDD-transformation of (MOD-2, π) to (MOD-4, π) can be constructed based on the observation that MOD-$2_n(a_1, a_2, ..., a_n)$ = MOD-$4_{2n}(a_1, a_1, a_2, a_2, ..., a_n, a_n)$ holds for all $a \in \{0,1\}^n$, $a = (a_1, a_2, ..., a_n)$. S is a transformer that modifies OBDD $Q4$ (for MOD-4_6) into OBDD $Q2$ (for MOD-2_3). After reduction of redundant nodes from $Q2$ (i.e., those nodes that have two equal successors), we obtain an OBDD for MOD-2 over the variables $\{x_1, x_3, x_6\}$.

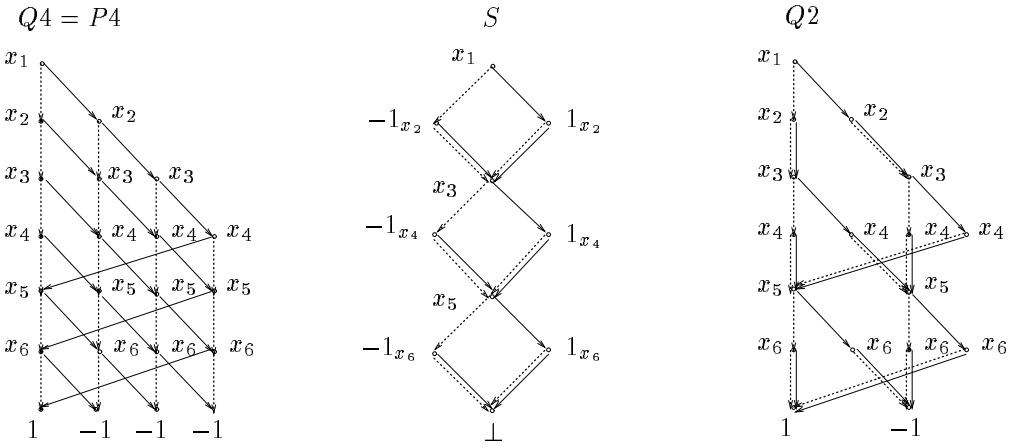


Figure 3 $Q4 \diamond S = Q2$

It is easy to see that MOD-2 $\leq_{OBDD}$ MOD-2^i, for any i.

5.2.3 (MOD-4, π) $\leq_{OBDD}$ (MOD-2, π)

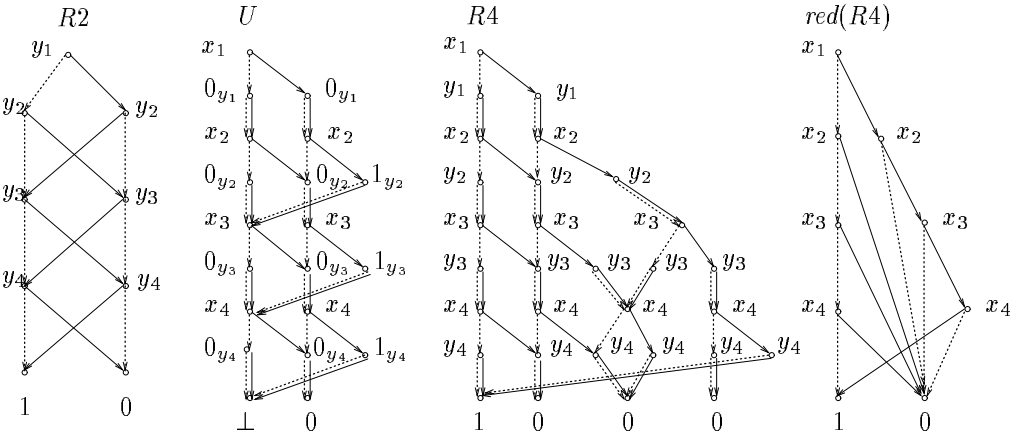


Figure 4 $R4 = R2 \diamond U$

$R2$ is an OBDD for MOD-2_4, U is a transformer and $R4 = R2 \diamond U$. red($R4$) is obtained from $R4$ by elimination of the redundant nodes.

It is easy to see that MOD-2^i $\leq_{OBDD}$ MOD2, for any i.

6 Complexity-bounded OBDD-Transformations

It is easy to see that the relation $\leq_{OBDD}$ defined via OBDD-transformations is reflexive and transitive. Even more, we can show that $\leq_{OBDD}$ is symmetric, too, and, hence, that all problems are equivalent with respect to *unbounded* OBDD-transformations. This is not surprising since we allow to apply any sequence of elementary operations, no matter how long it is.

Proposition 9.
Each problem (f, π) is transformable to problem $(1, \sigma)$, where σ is any variable ordering and, vice versa, problem $(1, \sigma)$ is transformable to any problem (f, π).

Proof. The proof is based on the algorithm *Derive*. Every OBDD can be seen as an OBDD-transformer. Let $\mathbf{1}$ be the reduced OBDD for the constant function 1. Each OBDD P fulfills the relation $\mathbf{1} \diamond P = P$ and $P \diamond \mathbf{1} = \mathbf{1}$. □

Corollary 10.
Any two problems are transformable to each other. □

In order to use OBDD-transformations for the investigation of complexity theoretic properties of OBDDs, we have to restrict their computational resources in an appropriate way. As already mentioned, the complexity measure of main interest is OBDD-size. Since, due to Propositions 3 and 4 together with Theorem 6, the size of the OBDD constructed by the transformation can be estimated in terms of the size of the applied transformer it suffices to restrict the size of the transformers in order to obtain results of complexity theoretic relevance.

Definition.
Let $r(n)$ be a function on the natural numbers. A problem (f, π) is called a *$r(n)$-bounded OBDD-transformation* of the problem (g, σ) (denoted by $(f, \pi) \leq^{r(n)}_{OBDD} (g, \sigma)$) if there is a sequence of OBDD-transformers (T_n) such that

- for each n there is an m such that $\pi_n\text{OBDD}(f_n)$ and $\sigma_m\text{OBDD}(g_m) \diamond T_n$ are equivalent up to a renaming of the variables, and
- the sequence of transformers $(T_n)_{n=1}^{\infty}$ is *$r(n)$-bounded*, i.e., $size(T_n) \leq r(n)$ for each n.

If $\mathcal{C}$ is a class of functions, then the problem (f, π) is a *$\mathcal{C}$-OBDD-transformation* of the problem (g, σ) (denoted by $(f, \pi) \leq^{C}_{OBDD} (g, \sigma)$) if there is a function $r \in \mathcal{C}$ such that $(f, \pi) \leq^{r(n)}_{OBDD} (g, \sigma)$.

To give some **examples**, we mention that the transformations described in Sections 5.2.2 and 5.2.3 are examples of linearly bounded transformations. Unlike that, the operation in the example 5.2.1 provides a transformation that may be exponential, depending on the position of the redirected edge.

No matter how much we restrict the sizes of transformers, the transformations remain reflexiv in any case. The transitivity is a more sensitive property. According to Propositions 4 and 5, transitivity is fulfilled in the cases of, e.g., constantly, polylogarithmically or polynomially bounded transformations. The transformation might not be transitive e.g. in the case of linearly bounded ones. Generally, we have

Proposition 11.
For every function class $\mathcal{C}$ it holds that the $\mathcal{C}$-OBDD-transformations are transitive if and only if for every $r_1, r_2 \in \mathcal{C}$ there is an $r \in \mathcal{C}$ such that $r \geq r_1 r_2$. □

A noteworthy phenomenon is that the symmetry property of unbounded transformations stated in Proposition 9 is no more true if one considers bounded transformations. For example, if one cuts out an exponentially large subOBDD by Operation 4, then an exponentially large transformer is needed to 'repair' this. Fig. 5 below shows an example.

Example: Operation 4 and its reversion

Let P be a OBDD and u be a node of P. The replacement of the δ-successor of u by the 0-sink can be realized by means of a transformer T whose top part is isomorphic to the subgraph of P which is defined by all root-to-u paths. The δ-successor ($-\delta$-successor) of node u' in T that corresponds to u in this isomorphism is the sink with the negation mark equal to 1 and is labeled by 0 (respectively, by $\perp$). All omitted edges enter the sink labelled by $\perp$ with negation mark 1.

The transformer T^{-1} realizes the inverse operation as T on P. The whole OBDD P' rooted in v has to be reconstructed. This OBDD P' is a part of T^{-1}. Hence, it can be exponential in the number of variables, if this is true for P'.

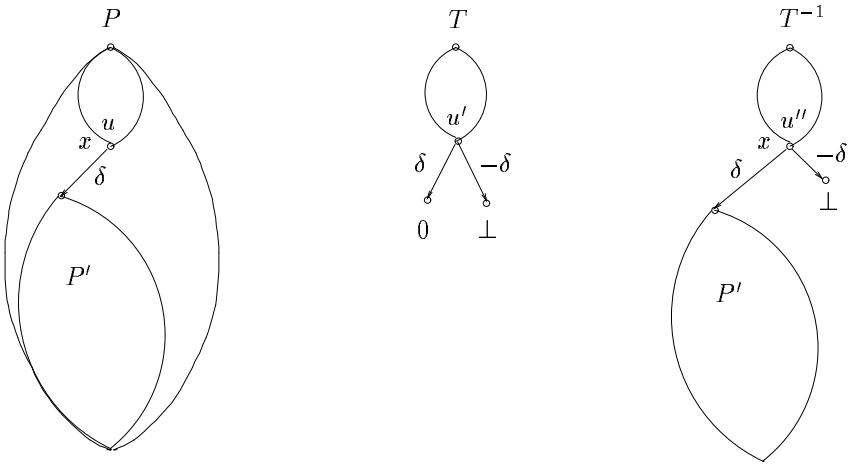


Figure 5 $(P \diamond T) \diamond T^{-1} = P$

6.1 Polynomial OBDD-Transformations

Size bounded transformers provide a relevant reduction tool for the investigation of complexity classes defined in terms of OBDDs. Remember, if $\mathcal{C}$ is a class of functions, then the complexity class $\mathcal{C}_{OBDD}$ is defined to consist of all problems (f, π) for which, for every n, the size of the reduced π_nOBDD(f_n) is bounded by $r(n)$ for some $r \in \mathcal{C}$. Particularly, $\mathcal{P}_{OBDD}$ denotes the set of problems representable by means of polynomially bounded OBDDs, i.e.,

$$\mathcal{P}_{OBDD} = \{(f, \pi) \,|\, \text{there is a polynomial } p(n) \text{ with } \mathit{size}(\pi_n\text{OBDD}(f_n)) \leq p(n) \text{ for all } n\}.$$

Now, let $\leq^p_{OBDD}$ denotes the relation defined by means of polynomially bounded OBDD-transformation. Two problems Π and Σ are said to be *equivalent w.r.t. polynomial OBDD-transformation* (denoted by $\Pi \equiv^p_{OBDD} \Sigma$) if $\Pi \leq^p_{OBDD} \Sigma$ and $\Sigma \leq^p_{OBDD} \Pi$.

Proposition 12.
The relation $\leq^p_{OBDD}$ is transitive.

Proof. The transitivity follows from Propositions 4 and 5. In order to apply *Compose*, we have to use the respective renamings, too. □

$\mathcal{P}_{OBDD}$ is the basic class in the hierarchy of complexity classes defined by polynopmially bounded OBDD-transformations

Proposition 13.
Let Π be a problem from $\mathcal{P}_{OBDD}$ and σ any variable ordering. Then $\Pi \equiv^p_{OBDD} (1, \sigma)$.

Proof. The construction of the transformers is the same as in the proof of Proposition 9. □

Corollary 14.
For any two problems Π and Σ from $\mathcal{P}_{OBDD}$ it holds $\Pi \equiv^p_{OBDD} \Sigma$.

Polynomially bounded OBDD-transformations are the adequate reduction tool for investigating the membership in the class $\mathcal{P}_{OBDD}$. This property can be very useful in practical applications of OBDD transformations.

Corollary 15.
If $\Pi \leq^p_{OBDD} \Sigma$ then $\Sigma \in \mathcal{P}_{OBDD}$ implies $\Pi \in \mathcal{P}_{OBDD}$ and $\Pi \notin \mathcal{P}_{OBDD}$ implies $\Sigma \notin \mathcal{P}_{OBDD}$. □

In the following we sketch an example which shows how to use OBDD-transformations in order to derive exponential lower bounds on the OBDD-size.

Example: *Transfer of exponential lower bounds.*

Let $PERM_n$ be the test whether an $n \times n$ matrix M over $\{0,1\}$ is a permutation matrix, i.e., if there is exactly one 1 in each row and each column. Let MAG^*_n denote the test whether an $n \times n$ matrix M over $\{0,1\}$ is a magic square, i.e. whether the sum of the elements in each rows and each column agree. MAG_n differs from MAG^*_n in the point that the matrix has entries from $\{0, 1, ..., n\}$. It holds

$$PERM \leq^p_{OBDD} MAG^* \qquad PERM \leq^p_{OBDD} MAG$$

In [KMW88] it was proven that the function $PERM$ requires exponentially large OBDDs for any variable ordering. (This lower bound holds even for FBDDs that generalize OBDDs by allowing different variable orderings on different paths). Due to this fact we derive exponential lower bounds for the two magic square problems MAG^* and MAG.

First, we construct, for any variable ordering π, an OBDD-transformation from $(PERM, \pi)$ to (MAG^*, π). Then we discuss the extension of this solution to the transformation of $(PERM, \pi)$ to (MAG, π). In any case, the existence of these transformations prove the intractability of the problems MAG and MAG^*.

W.l.o.g, the functions $PERM_n$ and MAG^*_n are assumed to be defined over the same set of variables $\{m_{ij} \mid 1 \leq i, j \leq n\}$. The transformation is based on the observation that a permutation matrix is a magic matrix whose row and column sum is equal to 1. Let m_{ij} be the first variable in the considered ordering π. Let the variables in the i-th row occur in the ordering $m_{ij} = m_{ij(1)} < m_{ij(2)} < m_{ij(3)} < \cdots < m_{ij(n)}$ and the variables in the j-th column in the ordering $m_{ij} = m_{i(1)j} < m_{i(2)j} < m_{i(3)j} < \cdots < m_{i(n)j}$. The transformer T^1_n (Fig. 6) excludes from the solutions of MAG^*_n those matrices which row sums are destinct from 1 (it suffices to check the sum in one row). The transformer T^2_n that can be obtained from T^1_n by replacing each $m_{ij(k)}$ by $m_{i(k)j}$ excludes from the solutions of MAG^*_n those matrices which column sums are destinct from 1 (it suffices to check the sum in one column).

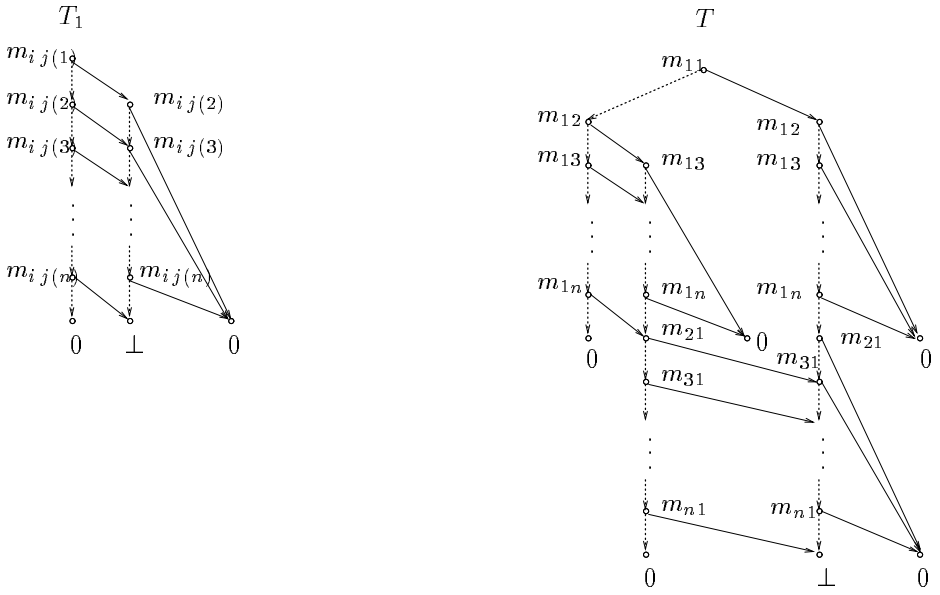


Figure 6

T_n^1 and T_n^2 are both π_n-transformers and we can build a transformer $T = T_n^1 \triangle T_n^2$, which excludes those matrices that have either a row sum or a column sum distinct from 1. The size of T_n is bounded by the product of the sizes of T_n^1 and T_n^2 and is hence $O(n^2)$. Transformer T_n on the figure is constructed with respect to the natural ordering, i.e. for all $1 \leq i, j, k, l \leq n$, such that $i < j$ or $(i = j)\&(k < l)$ holds $m_{ij} < m_{kl}$. Let MAG_n be defined over variables $\{p_{ijk} \mid 1 \leq i, j \leq n, 0 \leq k \leq N\}$, where $N = \lfloor \log n \rfloor + 1$ and $(p_{ij0}, p_{ij1}, ..., p_{ijN})$ is a binary code of the element in the i-th row and j-th column in the input matrix. The transformers for the transformation of *Perm* to *MAG* can be obtained from the transformers T for the transformation of *Perm* to MAG^* described above by setting $p_{ij0}, p_{ij1}, ..., p_{ijN-1}$ to 0 and by replacing $m_{i,j}$ by p_{ijN}.

Corollary 16.
For any variable ordering π, πOBDDs for the functions MAG_n^* and MAG_n have size in $2^{\Omega(n)}$.

Now we show that read-once projections correspond to polynomially bounded OBDD-transformations which are realizable by means of very simple structured transformers.

Corollary 17.
The read-once projection defined by a mapping $\tau_n : (x_1, ..., x_n) \mapsto (y_1, ..., y_m)$ can be realized by means of an OBDD-transformer of size m.

Proof. Following the proof of Proposition 1, we construct the transformer T_m which consists of a sequence of $n+1$ nodes labeled by the literals consistently with the natural order. Depending on τ_n, the following cases occur: $1 \leq i \leq n$

- $y_i = 1$(respectively, 0): The $i-$th node is labeled by 1_{y_i} (resp., -1_{y_i}) and both its outgoing edges enter the $(i+1)$-st node.
- $y_i = x_j$(respectively, $\overline{x_j}$): The $i-$th node is labeled by y_i (resp., $\overline{y_i}$) and has two outgoing edges, both enter the $(i+1)$-st node.

The $(n+1)$-st node is a sink labelled by $\perp$ with the negation mark 1. □

However, polynomially bounded OBDD transformations are definitively stronger than read-once-reductions.

Corollary 18.
There are even constantly bounded OBDD-transformations that can not be obtained via any read-once projection. □

Generalizing OBDDs to multi-rooted OBDDs, we obtain an appropriate representation of multivalued Boolean functions, i.e., mappings $\{true, false\}^n \mapsto \{true, false\}^m$, where all output bits are represented at once. OBDD-transformations for this generalised type of OBDDs can be obtained in a similar way as explained for OBDDs: A problem Π is reducible to problem Σ if, for each instance Π_n, its OBDD-representation can be obtained from the OBDD-representation of some instance Σ_m. Formally, $(f, \pi) \leq_{OBDD} (g, \sigma)$ iff, for each n, there is an m and a transformer T such that for all i there are j, k with

$$F_n^i \equiv T^k \diamond G_m^j$$

(up to a renaming of the variables), where F_n^i is the πOBDD for the i-th output bit of f_n, $T_{m'}^k$ is the k-th root of the transformer T, and G_m^j is the σOBDD of the $j-$th output bit of g_m.

The most surprising result obtained for read-once projections in [BW95] was the non-reducibility of SQUARING to MULTIPLICATION. We show that this result is not a witness of the higher complexity of the SQUARING. In contrary, it is an additional argument for the need of a more adequate notion of reducibility in the context of restricted complexity classes.

Example: (SQU,π) $\leq^p_{OBDD}$ (MUL,π'), where
$\pi'_{2n} = (\pi(1), \pi(1)+n, \pi(2), \pi(2)+n, ..., \pi(n), \pi(n)+n)$

For every n, we can construct a π'_{2n}OBDD-transformer that transforms a π'_{2n}OBDD for MUL^i_{2n} into a πOBDD for SQU^i_n, for each $i, 1 \leq i \leq n$. For each j, $1 \leq j \leq n$, the π'_{2n}OBDD-transformer contains one node labeled by x_{2j-1}. Each node labeled by x_{2j-1} has two successor nodes labelled by x_{2j}. The true-successor (respectively, the false-successor) has one out-going edge labeled by $+1$ (resp., by -1). The transformer for the naturaly ordered variables is illustrated on Fig. 7. □

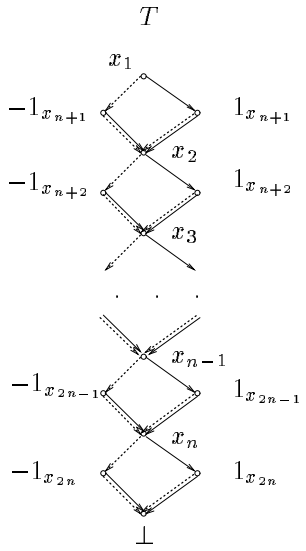


Figure 7

References

[BRB90] K. S. Brace, R. L. Rudell, R. E. Bryant: Efficient Implementation of a BDD Package, Proc. of 27th ACM/IEEE Design Automation Conference, 1990, 40-45.

[Bry86] R. E. Bryant: Graph-based Algorithms for Boolean Function Manipulation. IEEE Trans. Comput. c-35, 6 (1986), 677-691.

[BW95] B. Bollig, I. Wegener: Read-once Projections and Formal Circuit Verification with Binary Decision Diagrams. Proc. STACS'95, to appear.

[BDG88] J. L. Balcázar, J. Díaz, J. Gabarró: Structural Complexity I., Springer Verlag, 1988.

[KW87] K. Kriegel, S. Waack: Exponential Lower Bounds for Real-time Branching Programs. Proc. FCT'87, LNCS 278 (1987), 263–367.

[KMW88] M. Krause, Ch. Meinel, S. Waack: Separating the Eraser Turing Machine Classes $\mathcal{L}_\epsilon, \mathcal{NL}_\epsilon, co\text{–}\mathcal{L}_\epsilon$ and $\mathcal{P}_\epsilon$. Proc. MFCS'88, LNCS 324 (1988), 405-413.

[Lon93] D. Long: BDD-Package, CMU.

[Mei89] Ch. Meinel: Modified Branching Programs and Their Computational Power. LNCS 370, Springer Verlag, 1989.

[SV81] S. Skyum, L. G. Valiant: A Complexity Theory Based on Boolean Algebra. Proc. 22^{nd}IEEE FOCS (1981), 244-253.

[SW93] D. Sieling, I. Wegener: Reduction of BDDs in linear time. Information Processing Letters 48 (1993) 139-144.

[Lee90] J. van Leeuwen (edit.): Handbook of Theoretical Computer Science. The MIT Press 1990.

[Ross95] J. Rossmann: BDD-Package, University of Trier.

Appendix

Pseudo-code descriptions for the algorithms *Derive* and *Compose*

We use the following notions and denotations: OBDDs and OBDD-transformers are referred to by their root. A node stores the information about its label (*var*), its two successors (*true* and *false*), and its negation mark (*mark*). Sinks have no successors (i.e., *true* and *false* are set to $\perp$ which has a meaning of NIL) and store only their labels ($0, 1, \perp$) that are (for sake of uniformity) denoted as *var*, too. The labels of the variables are less than the labels of the sinks. Command 'select' performs only the commands that follows after the first true condition.

```
Derive:
input: πOBDD G, πOBDD-transformer T.
output: πOBDD H.
begin
   input(G);input(T);
   ROOT(H)=Derive_step(ROOT(G),ROOT(C));
   output(H);
end

Derive_step(u, v)
begin
if computed(u, v) then return result;
if SINK(v) { (* SINK(v) is true if v is a sink. *)
  if (v.var ==⊥){
     [u, v].var=u.var;
     [u, v].mark=u.mark*v.mark;
     [u, v].true=u.true;
     [u, v].false=u.false;
  } else{[u, v].var=v.var;
         [u, v].mark=v.mark;
         [u, v].true=v.true;
         [u, v].false=v.false;
  }
} else {
  if (u.var < |v.var|) {
     [u, v].var=u.var;
     [u, v].mark=u.mark;
     [u, v].true=Derive_step(u.true,v);
     [u, v].false=Derive_step(u.false,v)
  } else {
     [u, v].var= |v.var|;
     select {
      u.var > |v.var|:
       [u, v].mark=v.mark;
       [u, v].true=Derive_step(u, v.true);
       [u, v].false=Derive_step(u,v.false)
      |u.var| == |v.var|:
       [u, v].mark=u.mark*v.mark;
       x = |v.var|;
       select {
        v.var == x:
         [u, v].true=Derive_step( u.true,v.true);
         [u, v].false=Derive_step( u.false,v.false);
        v.var == x̄:
         [u, v].true=Derive_step(u.false,v.true);
         [u, v].false=Derive_step(u.true,v.false);
        v.var == 1_x:
         [u, v].true=Derive_step(u.true,v.true);
         [u, v].false=Derive_step(u.true,v.false);
        v,var == −1_x:
         [u, v].true=Derive_step(u.false,v.true);
         [u, v].false=Derive_step(u.false,v.false);
     } (* end of select *)
       } (* end of select *)
} } store_and_return([u, v]);
end

Compose:
input: πOBDD-transformer C_1, C_2.
output: πOBDD-transformer T.
begin
   input(C_1);input(C_2);
   ROOT(C)=Compose_step(ROOT(C_1),ROOT(C_2));
   output(T);
end

Compose_step(u, v)
begin
if computed(u, v) then return result;
if SINK(v) { (* SINK(v) is true if v is a sink. *)
  if (v.var ==⊥){
     [u, v].var=u.var;
     [u, v].mark=u.mark*v.mark;
     [u, v].true=u.true; [u, v].false=u.false;
  } else{ [u, v].var=v.var;
          [u, v].mark=v.mark;
          [u, v].true=v.true; [u, v].false=v.false;
  }
} else { select
     |u.var| < |v.var|: {
      [u, v].var=u.var;
      [u, v].mark=u.mark;
      [u, v].true=Compose_step(u.true,v);
      [u, v].false=Compose_step(u.false,v) }
     |u.var| > |v.var|:
      [u, v].var=v.var;
      [u, v].mark=v.mark;
      [u, v].true=Compose_step(u,v.true);
      [u, v].false=Compose_step(u,v.false); }
     |u.var| == |v.var|:
      [u, v].mark=u.mark*v.mark;
      x = |v.var|;
      select {
       v.var == x: {
        [u, v].var=u.var;
        [u, v].true=Compose_step(u.true,v.true);
        [u, v].false=Compose_step(u.false,v.false);
       }
       v.var == x̄:
        {select {
         u.var == x: [u, v].var = v.var;
         u.var == x̄: [u, v].var = x;
         true: [u, v].var = u.var; }
        [u, v].true=Compose_step(u.false,v.true);
        [u, v].false=Compose_step(u.true,v.false); }
       }
       v.var == 1_x:
        {select {
         u.var == x: [u, v].var = v.var;
         u.var == x̄: [u, v].var = −1_x;
         true: [u, v].var = u.var; }
        [u, v].true=Compose_step(u.true,v.true);
        [u, v].false=Compose_step(u.true,v.false);
       }
       v.var == −1_x:
        {select {
         u.var == x: [u, v].var = v.var;
         u.var == x̄: [u, v].var = 1_x;
         true: [u, v].var = u.var; }
        [u, v].true=Compose_step(u.false,v.true);
        [u, v].false=Compose_step(u.false,v.false);
       }
     }
} store_and_return([u, v]);
end
```