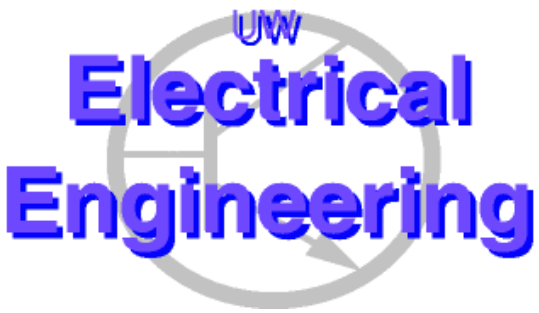
Exact and Heuristic Minimization of Determinant Decision Diagrams

Alicia Manthe and C.-J. Richard Shi

{amanthe, cjshi}@ee.washington.edu

Dept of EE, University of Washington
Seattle WA, 98195-2500



UWEE Technical Report

Number UWEETR-2002-0001

January 8, 2002

Department of Electrical Engineering
University of Washington
Box 352500
Seattle, Washington 98195-2500
PHN: (206) 543-2150
FAX: (206) 543-3842
URL: http://www.ee.washington.edu

Exact and Heuristic Minimization of Determinant Decision Diagrams*

Alicia Manthe and C.-J. Richard Shi
{amanthe, cjshi}@ee.washington.edu

Dept of EE, University of Washington
Seattle WA, 98195-2500

University of Washington, Dept. of EE, UWEETR-2002-0001
January 8, 2002

Abstract

Determinant Decision Diagram (DDD) is a variant of binary decision diagrams (BDDs) for representing symbolic matrix determinants and cofactors in symbolic circuit analysis. DDD-based symbolic analysis algorithms have time and space complexities proportional to the number of DDD vertices. Inspired by the ideas of Rudell, Drechsler, *et. al.* on BDD minimization, we present lower-bound based exact and heuristic algorithms for reordering the DDD vertices to minimize the DDD size. Our new contributions are two-folds. First, we show how vertex signs, which are specific to DDDs, can be handled during neighboring vertex reordering. Second, we develop lower bounds tailored to the DDD structures, which are much tighter than the known lower bounds for BDDs. On a set of DDD examples from symbolic circuit analysis, experimental results have demonstrated that the proposed lower-bound based reordering algorithms can effectively reduce DDD sizes. It has also been demonstrated that sifting with lower bounds uses about 55% less computation compared to sifting without using lower bounds, and sifting with the new lower bounds reduces the computation further by up to 10% compared to sifting with known lower bounds for BDDs.

1. Introduction

Symbolic circuit analysis derives analytic circuit transfer functions for analog circuits. The importance of symbolic analysis lies in its capability to provide insight into the circuit behavior and high efficiency when repetitive evaluation is required [7]. But its practical use is limited due to the high complexity of symbolic expressions, which increase exponentially with the size of the circuit.

Determinant Decision Diagrams (DDDs) are an application, and a special class, of zero-suppressed binary decision diagrams (BDDs), to represent symbolic matrix determinants and cofactors [11][12]. Being compact and canonical, DDD has demonstrated the ability to analyze symbolically analog/RF circuits that are much larger than those currently being handled by other existing techniques [13].

* This work was sponsored by DARPA/MTO NeoCAD Program under Grant No. N66001-01-1-8919, by an SRC Ph.D. Fellowship (to Alicia Manthe), and by an NSF CAREER Award (to C.-J. Richard Shi).

However, similar to BDDs, in the worse case, the number of vertices in a DDD, i.e., *size* of the DDD, can grow exponentially with the size of a circuit. With the complexities of symbolic analysis algorithms being at best linear in the size of a DDD [11], reducing as much as possible the DDD size is important for both the evaluation and manipulation of symbolic expressions.

In this paper, we present, for the first time, exact and heuristic algorithms and their implementations for reordering DDD vertices to minimize the DDD size. Previously, in the context of BDD minimization, Drechsler, *et. al.* in [5], showed a fast technique to achieve the minimum BDD by using lower bounds during the exact algorithm developed by Friedman and Supowit in [6]. For fast heuristic BDD minimization, Rudell in [10], showed sifting produced better results than other heuristics such as windowing even though the runtime can be much longer. Very recently, Drechsler, *et. al.* in [4], showed applying lower bounds to sifting can greatly improve the efficiency of the sifting algorithm for BDDs.

The proposed exact DDD minimization algorithm is based on an adaptation of exact BDD minimization algorithm of Friedman and Supowit enhanced with Drechsler's BDD lower bounds. The proposed heuristic DDD minimization algorithm combines the idea of Rudell's BDD sifting with Drechsler's BDD lower bound technique. Our novel contributions are two-folds. First, we show how vertex signs, which are specific to DDDs, can be handled during vertex reordering. Second, we develop lower bounds tailored to the DDD structures, which are much tighter than the lower bounds for BDDs.

Some preliminary results of this paper are published in [8]. The paper is organized as follows. Section 2 provides an overview of the notion of DDDs. Rules for DDD neighboring vertex reordering are introduced in Section 3. In Section 4 tight lower bounds on DDD sizes under neighboring vertex exchanging are derived. An exact DDD minimization algorithm is presented in Section 5. Section 6 presents an adapdation of the sifting algorithm due to Rudell for BDDs to DDDs enhanced with new DDD lower bounds. Experimental results are described in Section 7. Section 8 concludes this paper.

2. Determinant Decision Diagrams (DDDs)

A ***determinant decision diagram*** (DDD) is a signed, rooted, direct acyclic graph, similar in form to a binary decision diagram (BDD) [1], originally introduced for representing the determinant of a symbolic matrix in circuit analysis [11]. As illustrated in Figure 1, a DDD ***vertex*** *V* is associated with a ***label*** *(V.label)*, a positive or negative ***sign*** *(V.sign)*, and two edges namely ***1-edge*** (solid line) and ***0-edge*** (dotted line) pointing, respectively, to its ***1-child*** *(V.1child)* and ***0-child*** *(V.0child)*. A DDD has two terminals. The ***1-terminal*** represents constant 1, and the ***0-terminal*** represents constant 0. Then a non-terminal vertex V represents a ***symbolic expression*** *V.expr* defined as follows:

$$V.expr = V.sign * V.label * (V.child1).expr + (V.child0).expr. \quad (1)$$

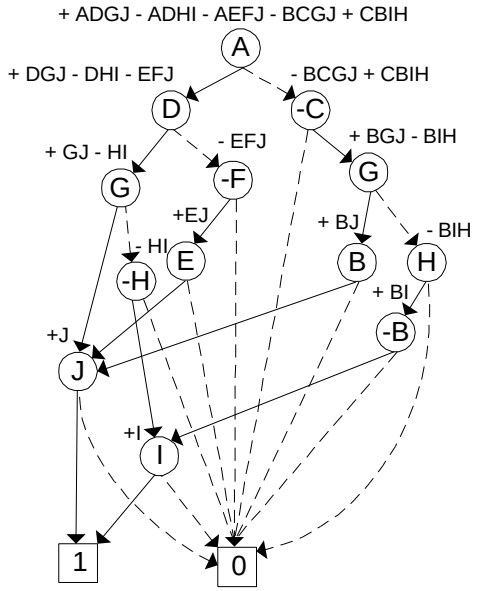


Figure 1. Example of a DDD with vertex order (A C D F G H B E I J).

A DDD is constructed with rules due to Bryant [3] and Minato [9] in the contexts of ordered binary decision diagram (OBDD): ordered, shared and zero-suppressed. Each label will appear no more than once in any 1-path of the graph; labels that appear will be in the same order for all the 1-paths (known as ***vertex order***). Note that a 1-path is a path to the 1-terminal. Common subgraphs are shared. Finally, vertices with 1-edges pointing to the 0-terminal are eliminated and subgraphs of the 0-edges are used. Similar to OBDDs, which are canonical representations for Boolean functions, DDDs are canonical representations for the determinants of symbolic matrices.

Figure 1 actually represents the determinant of the following matrix:

$$\det\begin{pmatrix} A & B & 0 & 0 \\ C & D & E & 0 \\ 0 & F & G & H \\ 0 & 0 & I & J \end{pmatrix} = ADGJ - ADHI - AEFJ - BCGJ + CBIH.$$

It corresponds to the expansion of the matrix in the order of (A C D F G H B E I J). This is the DDD vertex order. The ***size*** of a DDD, i.e., the number of vertices, is 13. There are 10 different vertex labels.

Clearly the size of a DDD depends critically on the vertex order. In [11], a vertex-ordering heuristic is developed that guarantees the optimum for ladder structure circuits. However, it is unknown how to find the best vertex order for general-structure circuits.

3. DDD Neighboring Vertex Exchanging

In this section, we first present, using an example, how to minimize a DDD through a sequence of DDD neighboring vertex exchanges. We then formulate general rules for DDD neighboring vertex exchanges; these rules will be used in the proposed exact and heuristic minimization algorithms.

We consider first the change of the order of labels in Figure 1 to (*A C D F G B H E I J*); i.e., exchanging only the order of labels *H* and *B*. Since the order of all other labels stays the same, only one local section of the DDD in Figure 1 needs to be updated; this is illustrated in small solid circle in Figure 2. Note that these exists already a subgraph with the root vertex labeled H with the minus sign, and this step reduces the number of vertices by 1 through subgraph sharing. Then we exchange the order of *G* and *B*, again only one local section of the DDD needs to be changed, as illustrated in large solid circle in Figure 2. Note that two vertices are reduced in this step. First, two B vertices becomes one B. Second, we note that the resulting subgraph rooted at vertex G is exactly the same as the one marked at the left side of the diagram. So they can be shared, and this saves one vertex (G). The resulting DDD has only 10 vertices and is minimal. The obtained order (A C D F *B G* H E I J) is thus optimal.

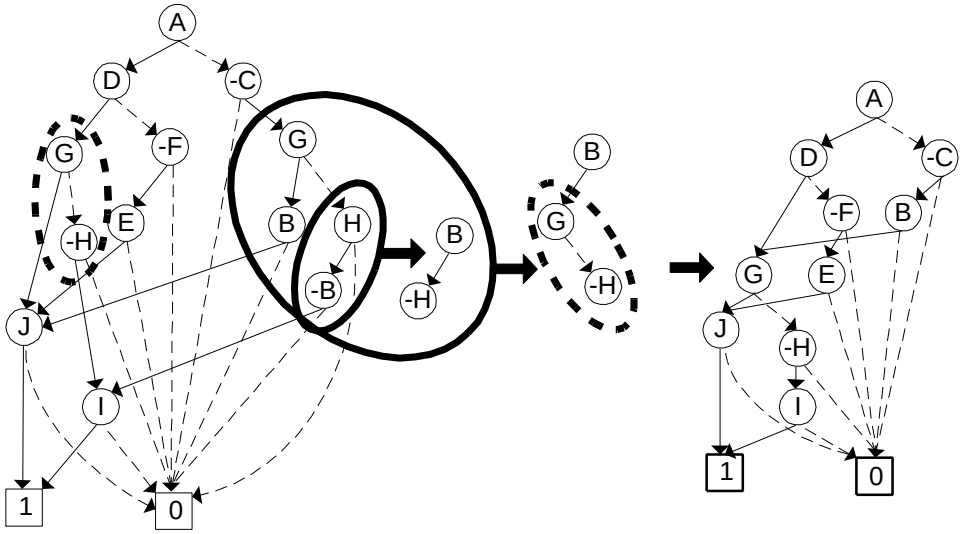
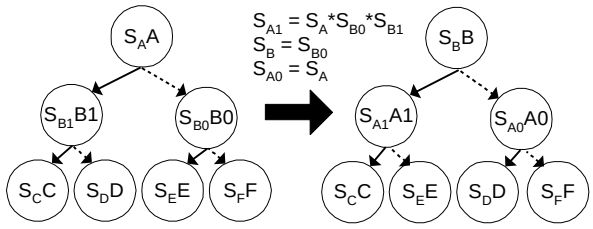


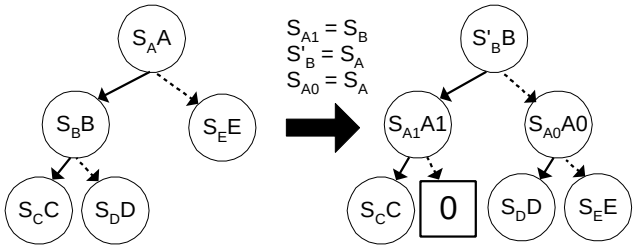
Figure 2. Reordering of DDD in Fig. 1

A key operation used in the example above is the *exchange of neighboring vertices*. Similar to BDDs, due to the canonicity of DDDs, neighboring vertex exchanges only affect local vertices, i.e., when two neighboring vertices are exchanged, only some local updating of DDD structures are needed.

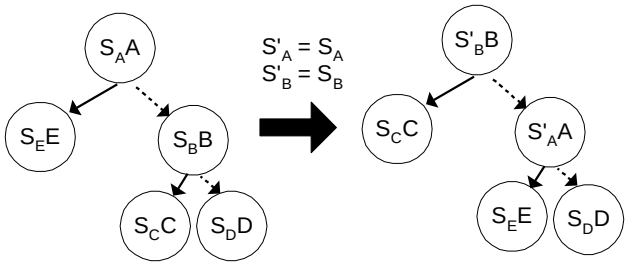
Figure 3 summarizes the vertex updating rules for all the three potential cases of neighboring vertex exchanges. On the left side are the original segments of DDDs, and on the right side are the new DDD segments after exchanging or swapping. Each vertex is labeled by a label (*A* to *F*) and a sign *S* with the subscript indicating its label. Labels *A1* and *A0*, *B1* and *B0* are actually the same label but appear in different places. The three cases are (a) vertex labeled by *A* is exchanged with its 1 child and 0 child (both are labeled by *B*, shown as *B1* and *B0* in the diagram), (b) vertex labeled by *A* is exchanged with its 1-child, and (c) vertex labeled by *A* is exchanged with its 0-child. The new signs are determined by the sign transformation rules illustrated above the big solid arrows.



(a) Vertex A exchanging with its both children



(b) Vertex A exchanging with its 1-child



(c) Vertex A exchanging with its 0-child

Figure 3. Rules for DDD neighboring vertex exchanges.

It can be verified that for all the cases in Figure 3, the DDDs shown on the right side are equivalent to the DDDs on the left side. That means a DDD segment with incoming edges pointing to its top and outgoing edges leaving at its bottom vertices can be replaced by the one on the right side so that the overall DDD still represents the same symbolic expressions, and vice versa.

Note that the sign transformation rules are new to DDDs, which are not needed for BDD neighboring vertex exchanges. It turns out that the sign transformation may destroy the DDD canonicity. This is due to the fact that Figure 3(a) has many different sign scenarios that can lead to the same DDD segment after exchange. This is illustrated in Figure 4. For each case in Figure 4, if we apply the sign transformation rule in Figure 3(a), then both the top left and bottom left DDD segments lead to the same DDD segment on the right. However, if we apply the sign transformation rule in Figure 3(b) to the DDD segment on the right side, then we can obtain only the DDD segment on the top left side. Therefore, the DDD segment in the bottom left side after the exchange of label *A* and *B* twice will lead to the DDD segment on the top left side.

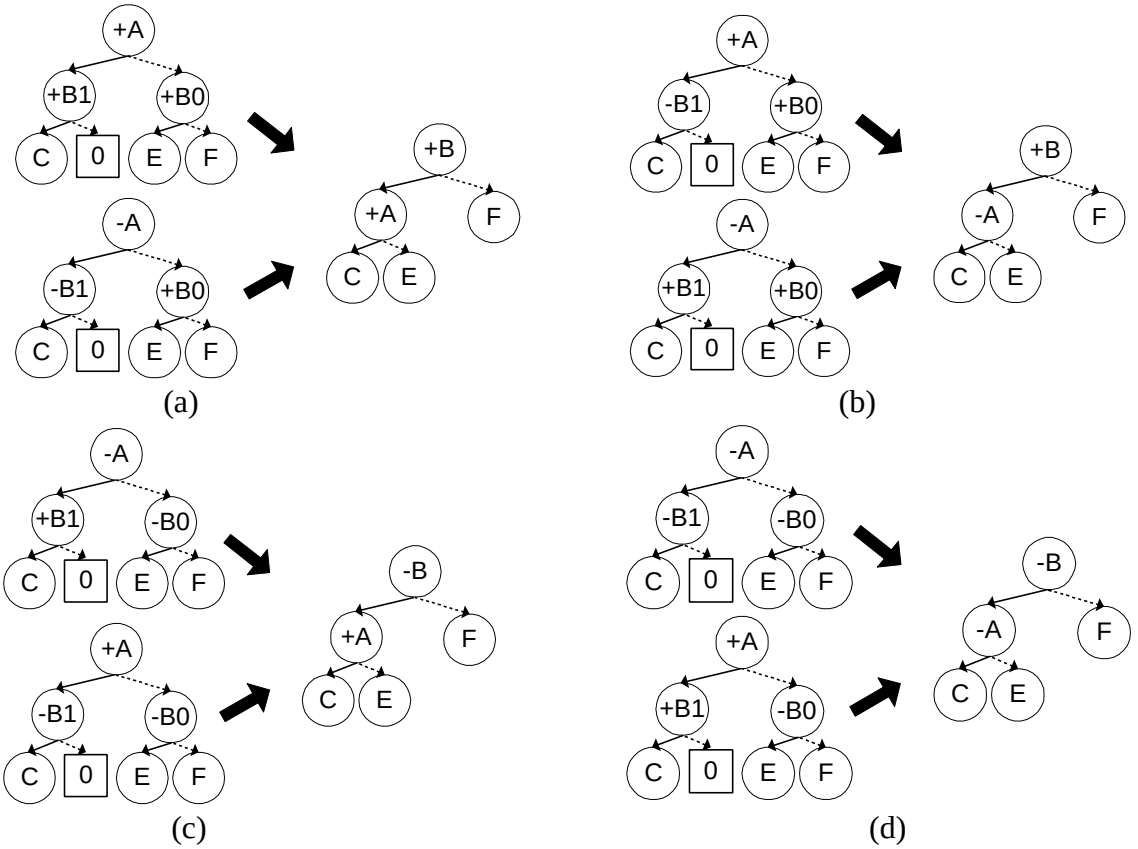


Figure 4. Four sign scenarios of exchange rule in Figure 3a.

This has two implications. First, there will be multiple DDDs that represent the same symbolic expression, and therefore theoretically the DDD is no longer canonical. Second, this transformation non-uniqueness may lead to the blow up of the DDD size when neighboring vertex exchanging is used to minimize the DDD size in practice. For example, in Figure 5 say *B1* is shared between *A* and *Q* parents and an exchange of *B* and *A* has occurred. In memory *B1* still exists for parent *Q*. During the local exchange back of *A* and *B*, the sign rule of Figure 3(b) is applied. As shown in the far right of Figure 5 this is not the original DDD tree that we started with. In memory *B1* still exists and the previously shared vertices are no longer shared resulting in an increase of the size of the DDD for that particular order. If the lost of sharing is continued through further exchanges an explosion of the number of DDD vertices can happen.

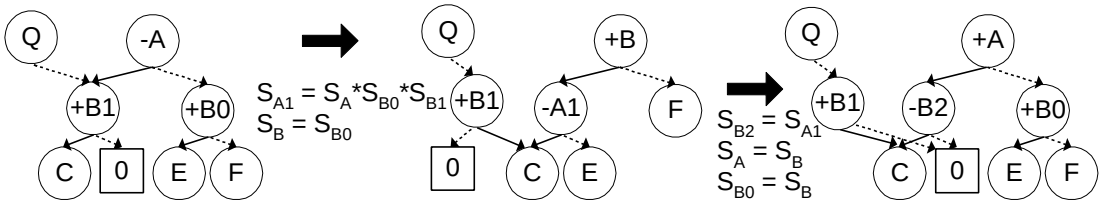


Figure 5. Local exchanging *A* and *B* using sign rule Fig. 3a and exchanging back *B* and *A* using sign rule Fig. 3b.

Fortunately, this potential explosion problem can be solved by keeping track of whether the top-left or bottom-left segment leads to the right side of the transformation for each case in Figure 4. . This can be accomplished by storing a flag in the new *B* parent. When exchanging back is performed this flag is checked. In this way, exchanging the same pair of labels twice will lead to the original DDD representation.

4. Lower Bounds on DDD Sizes under Neighboring Vertex Exchanging

Let position i be within range $1 \le i < n$, where n is the number of unique labels. In this section, we first derive tight lower bounds on the DDD size if the two neighboring labels at position i and $i+1$ are to be exchanged. We then derive lower bounds on the DDD size if the label at position i is to be *sifted* to position j, *i.e.* the label at position i is moved to position j while the relative positions of all the other labels remain unchanged. These lower bounds can be computed efficiently.

Let $X_j^i = \{x_i, x_{i+1}, \ldots, x_{j-1}, x_j\}$ denote a set of labels. Let #vertices(x_i) and #vertices(X_j^i) respectively represent the number of vertices with label x_i and a set of labels X_j^i **before** reordering. Adding an apostrophe to these expressions leads to, #vertices'(x_i) and #vertices'(X_j^i), which respectively denote the number of vertices with label x_i and a set of labels X_j^i **after** reordering.

We assume that the symbolic determinant expressions depend on all the symbols (labels). Then we have the following lemma.

Lemma 1. The following bound holds for all labels:

$$1 \le \#\text{vertices}'(x_i). \tag{2}$$

If each label was not represented at least once, then the DDD no longer represents the determinant expression. This same bound holds for BDDs, as described in [4].

Theorem 1. If a label at position i is exchanged with an adjacent label at position $i+1$, then

$$\lceil \tfrac{1}{2} \cdot \#\text{vertices}(x_i) \rceil \le \#\text{vertices}'(x_i) \le 2 \cdot \#\text{vertices}(x_i) \tag{3}$$

and

$$\lceil \tfrac{1}{2} \cdot \#\text{vertices}(x_{i+1}) \rceil \le \#\text{vertices}'(x_{i+1}) \le \min\{\#\text{vertices}(x_i) + \#\text{vertices}(x_{i+1}),\ 2 \cdot \#\text{vertices}(x_{i+1})\}. \tag{4}$$

Proof: We consider the following four cases.

Case 1: $\lceil \tfrac{1}{2} \cdot \#\text{vertices}(x_i) \rceil \le \#\text{vertices}'(x_i)$: We need to show when a label at position i is exchanged with a label at position $i+1$, *at most*, half of the parent vertices with the label originally at position i can disappear. If a parent vertex with label at position i does not have any children vertices with labels at position $i+1$, then the parent vertex stays the same after the label exchange. There are three situations as shown in Figure 3 (a)-(c) when a parent vertex (with label at position i, say A) has a child labeled with the label at position $i+1$, say label B. After the exchange, a vertex labeled A can disappear only if there exists a subgraph in the DDD identical to the subgraph rooted at the new A vertex after the exchange. To show at most half of the vertices labeled by A can disappear, we need to show that for all the potential local sections as shown in Figure 3(a)-(c), there is no two subgraphs rooted at a new A vertex that are identical. This is true by verifying that there are no identical two subgraphs rooted at A for all the right sides of Figure 3(a)-(c), unless the original DDD segments on the left side are the same.

The ceiling is taken of $\tfrac{1}{2} \cdot \#\text{vertices}(x_i)$ because if #vertices(x_i) is odd, then taking half of an odd number leads to a fractional number of vertices, which is not possible. This new lower bound is tighter than 1, the lower bound given in [2][4].

Case 2: $\#\text{vertices}'(x_i) \le 2 \cdot \#\text{vertices}(x_i)$: Refer to Figure 3, a vertex in level i (label A) can lead to at most two new vertices, if both of its 1-child and 0-child are labeled B at level $i+1$, and remains unchanged if only its 0-child or no child is labeled B. This has been observed in [2][4].

Case 3: $\lceil \tfrac{1}{2} \cdot \#\text{vertices}(x_{i+1}) \rceil \le \#\text{vertices}'(x_{i+1})$: The inequality is clear since only the half of the vertices labeled B can disappear as in Figure 3(a). For the same reason in Case 1 the ceiling is taken of $\tfrac{1}{2} \cdot \#\text{vertices}(x_{i+1})$.

Case 4: $\#\text{vertices}'(x_{i+1}) \le \min\{\#\text{vertices}(x_i) + \#\text{vertices}(x_{i+1}),\ 2 \cdot \#\text{vertices}(x_{i+1})\}$: Referring to the local cases shown in Figure 3 if all the children labeled by B at position x_{i+1} have another parent with different labels other than A, then during the moving process the original B vertex is kept because of the other parent and a new B vertex is created for the B and A exchange. This leads to the local upper bound, #vertices(x_i) + #vertices(x_{i+1}), for cases Figure 3(a)-(c).

For Figure 3(b)-(c), #vertices(x_i) + #vertices(x_{i+1}) = 2·#vertices(x_{i+1}), and for Figure 3(a) #vertices(x_i) + #vertices(x_{i+1}) < 2·#vertices(x_{i+1}). Considering #vertices(x_i) + #vertices(x_{i+1}) on a global scale, labels at position x_i could be independent of x_{i+1}, which could lead to #vertices(x_i) + #vertices(x_{i+1}) > 2·#vertices(x_{i+1}). Therefore the minimum of #vertices(x_i) + #vertices(x_{i+1})

and 2·#vertices(x_{i+1}) is taken. In [2][4][6], only #vertices'(x_{i+1})≤ #vertices(x_i) + #vertices(x_{i+1}) was observed with slightly complicated reasoning. ð

Theorem 1 thus gives the lower bounds tighter than the ones known in [2][4][6]. With this, we can obtain tighter lower bounds on the DDD sizes when a label at position *i* is sifted to position *j*, as stated below.

Theorem 2. Let *1* £ *i* < *j* £ *n* and the original label order is (x_1 … x_{i-1} x_i x_{i+1} … x_j x_{j+1} … x_n). If the label the label in position *i* is sifted **down** to position *j* resulting in the label order (x_1 … x_{i-1} x_{i+1} … x_j x_i x_{j+1} … x_n), or if the label in position *j* is sifted **up** to position *i* resulting in the label order (x_1 … x_{i-1} x_j x_i … x_{j-1} x_{j+1} … x_n), then the size *S'* of the resulting DDD satisfies the following inequality:

$$S' \geq \#vertices(X_{i-1}^{1}) + \max\left\{\left\lceil \frac{1}{2^{j-i}}\#vertices(x_i)\right\rceil + \sum_{k=i+1}^{j}\left\lceil \frac{1}{2}\#vertices(x_k)\right\rceil, (j-i+1)\right\} + \#vertices(X_n^{j+1}) \quad (5)$$

Proof. Since the cases of sifting up and down are symmetrical, it is sufficient to consider the case of sifting down. Clearly the labels before *i* or after *j* are undisturbed in the DDD, this leads to the first and last terms in the right side of inequality (5). The middle term in (5) is due to moving the label at position *i* down to *j*. Let us first consider moving the label at position *i* down one position. The lower bound in Theorem 1, ⌈½·#vertices(x_i)⌉ and ⌈½·#vertices(x_{i+1})⌉, is applied to both labels at position *i* and position *i+1*. The new size of the DDD would be greater or equal to:

$$S' \geq \#vertices(X_{i-1}^{1}) + \left\lceil \frac{1}{2}\#vertices(x_i)\right\rceil + \left\lceil \frac{1}{2}\#vertices(x_{i+1})\right\rceil + \#vertices(X_n^{i+2}), \quad (6)$$

where the first and last terms are the undisturbed labels that reside either before *i-1* or after *i+2*. Now move the label at position *i* down another slot to *i+2*. The formula would read as follows:

$$S' \geq \#vertices(X_{i-1}^{1}) + \left\lceil \frac{\left\lceil \frac{1}{2}\#vertices(x_i)\right\rceil}{2}\right\rceil + \left\lceil \frac{1}{2}\#vertices(x_{i+1})\right\rceil + \left\lceil \frac{1}{2}\#vertices(x_{i+2})\right\rceil + \#vertices(X_n^{i+3}). \quad (7)$$

Here the new number of vertices for x_i and x_{i+1} is equivalent to the last exchange and are, respectively, equal to the second and third terms in (6) and are seen in (7). Once again Theorem 1 is applied to the sifted labels, *i* and *i+2*. If the second through forth terms are combined the expression reads as follows:

$$S' \geq \#vertices(X_{i-1}^{1}) + \max\left\{\left\lceil \frac{1}{2^{(i+2)-i}}\#vertices(x_i)\right\rceil + \sum_{k=i+1}^{i+2}\left\lceil \frac{1}{2}\#vertices(x_k)\right\rceil, ((i+2)-i+1)\right\} + \#vertices(X_n^{(i+2)+1}) \quad (8)$$

Lemma 1 has to be considered when combining the terms, so the maximum is taken between the combined terms and the number of labels affected by sifting, (*i+2*)-*i+1*, have to be represented at least once. Whether you take the ceiling of the second term in (7) recursively or just once does not change the result of that term. However the ceiling of each label sifted by has to be taken at each step, thus the ceiling is shown within the sum term in (8). The theorem is proved simply by mathematical induction. ð

We note that the lower bounds in Theorem 2 are tighter than the one derived in [4]. Further the lower bounds for both the cases of sifting up and sifting down are the same, where in [4] they are different.

5. Exact DDD Minimization with Pruning

Starting with a given DDD constructed with a specific vertex order, the proposed DDD minimization algorithm invokes a sequence of local vertex exchanging, adapting ideas from a series of previous work on exact BDD minimization [5][6]. The algorithm enumerates all the label order possibilities by using the sifting operation on a single DDD in memory.

The basic idea of exact minimization is to first generate all potential vertex orderings of two labels to create a subset of size 2. Assuming *n* unique labels, there are *n*n-1* subsets of size 2 created. Labels in a subset are sifted into the *upper* part of the DDD according to their position in the subset (all other labels can be in any arbitrary order in the *bottom* part of the DDD).

Note that only some of the subsets of size 2 can lead to smaller DDDs. So before proceeding to subsets of size 3, we prune out those subsets of size 2 that would certainly lead to sizes larger than the current DDD size. This can be accomplished by noting that the minimum DDD size after labels 1 to *i* (*i* = 2 for subset of size 2) are sifted to the upper part cannot be smaller than

$$Exact\_Lower\_bound = \#vertices(X_i^1) + (n - i), \tag{9}$$

where $\#vertices(X_i^1)$ represents the number of vertices that are labeled from 1 to *i*.

After examining all the subsets of size 2 and pruning unwanted subsets according to the *Exact_Lower_bound*, then another label will be added to the subsets of size 2 that are left after pruning to create subsets of size 3. Figure 6 shows a segment of the subsets created and pruned. Once again the chosen labels are sifted into the upper part of the DDD for analysis and pruning. The process continues until the subset size is *n* (all the labels are considered). The vertex order with the minimum size DDD is recorded. Then the DDD is sifted to the recorded vertex order.

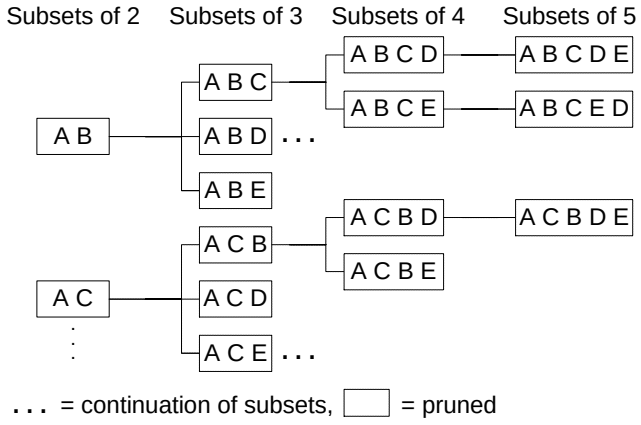


Figure 6. Creation of subsets for symbols in set {A B C D E}. Square shows order of symbols in known upper part of DDD and the number of symbols represents the amount of symbols in that subset.

Pruning allows an originally computationally exhaustive algorithm to be more manageable. Those subsets that are eliminated earlier have a larger effect on the time complexity. If a subset of size 2 is eliminated then subsets of size 3, … , *n* consisting of this subset of size 2 are also eliminated. Whereas, if a subset of size *n-1* is eliminated then subsets of size n consisting of this subset of size *n-1* are only eliminated. See Figure 6 above and notice eliminating subset {*A B E}* also eliminates subsets that are created from that branch. Thus four subsets are not even created, just from eliminating at subsets of size 3. Pruning subset {*A C B E}* leads to only 1 subset not created. As shown in Figure 6 it is much more beneficial to prune subsets as early as possible. Also, by reducing the number of subsets analyzed will allow for larger circuits to be analyzed due to memory requirements.

The pseudo code for exact DDD minimization is illustrated in Figure 7, where *ELB* is the *Exact_Lower_bound*, *NumVertices* is the current size of the DDD, m represents the current size of the subset, $Subset_m$ represents the labels in the subset of size *m*, *n* is the number of unique labels in the DDD, and *Bound* represents the minimum bound found so far.

```
Bound = NumVertices;
m =2;
while (m <=n){
    while (Subset_m exist){
        for (j = 1 to n){
            if (j not in Subset_m){
                sift j to order m+1;
                ELB_j = Exact_Lower_bound(j);
                if (ELB_j <= Bound){
                    Bound = ELB_j;
                    Store Subset_j;
        }}} Subset_m = Subset_m→next;
    } m=m+1;
} sift DDD to represent order of Subset_n;
```

Figure 7. Exact DDD minimization.

6. Heuristic DDD Minimization with Lower Bound based Sifting

In this section, we adapt the sifting algorithm developed by Rudell in [10] for OBDDs to reduce the DDD sizes as much as possible, but not necessarily minimum. The resulting heuristic minimization algorithm runs more efficiently and uses less memory space than the exact minimization algorithm, and can be applied to solve much larger problems. We then apply the DDD lower bounds derived in Section 4 to further improve the efficiency of sifting.

This algorithm is based on sifting a vertex label through all other vertex labels until a locally optimal position is found, while keeping all other vertex labels fixed. The sifting order is based on the number of vertices in each *level* of a DDD. Vertex labels found in the largest levels are sifted first. A chosen vertex label is swapped with each predecessor until the chosen vertex label is sifted to the beginning. Next the vertex label is swapped with each successor until it reaches the last position. The best sized DDD found is noted. Once the sifted vertex label is finished swapping up and down, the label is swapped back to the order where the best-sized DDD was noted.

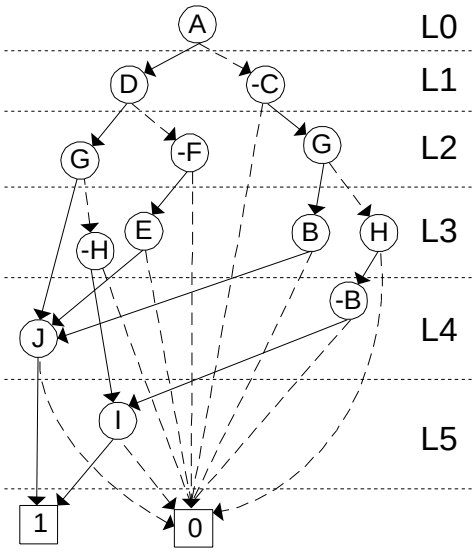


Figure 8. Figure 1 broken down into levels, level 0 (L0) to level 5 (L5).

For DDDs the sifting order is based on label occurrences and number of vertices in each level. Precedence is given to the label occurrence and then the level size. For example, consider the DDD in Figure 1. The original order of this DDD is: (A C D F G H B E I J). Figure 8 above shows the DDD structure broken to respective *levels.* The number of times a label appears is shown in Table 1. Using the level information found in Figure 8 and label occurrences found in Table 1, a sifting order can be established: H B G E F D C J A I.

Table 1. Label occurrence for the DDD in Figure 1.

Label	*#vertices(label)*
A	1
B	2
C	1
D	1
E	1
F	1
G	2
H	2
I	1
J	1

Table 2 gives an example of the sifting algorithm applied to the DDD in Figure 1. The first column shows the current vertex label order. The second column shows the size of the best-sized DDD found so far. The current number of vertices in the DDD for the order found in column 1 is displayed in column 3. The sift type can be either sift up (*Up*) or sift down (*Dn*) and is shown in column 4. Best position is given in the last column and represents the number of swaps away from the best-sized DDD found. Just sifting H has 17 swaps or exchanges performed. After H is sifted the rest of the labels in the sifting order are sifted through.

Table 2. Sifting Algorithm Example

Order	**Best Size**	**# Vertex**	**Sift Type**	**Best Pos.**
A C D F G **H** B E I J	13	13	*Up*	0
A C D F **H** G B E I J	13	13	*Up*	1
A C D **H** F G B E I J	13	13	*Up*	2
A C **H** D F G B E I J	13	14	*Up*	3
A **H** C D F G B E I J	13	15	*Up*	4
H A C D F G B E I J	13	15	*Dn*	5
A **H** C D F G B E I J	13	15	*Dn*	4
A C **H** D F G B E I J	13	14	*Dn*	3
A C D **H** F G B E I J	13	13	*Dn*	2
A C D F **H** G B E I J	13	13	*Dn*	1
A C D F G **H** B E I J	13	13	*Dn*	0
A C D F G B **H** E I J	12	12	*Dn*	**0**
A C D F G B E **H** I J	12	12	*Dn*	1
A C D F G B E I **H** J	12	12	*Dn*	2
A C D F G B E I J **H**	12	12	*Dn*	3
A C D F G B E I **H** J	12	12	*Up*	2
A C D F G B E **H** I J	12	12	*Up*	1
A C D F G B **H** E I J	12	12	-	0

The efficiency of sifting can be improved. The basic idea is to stop moving a label in a direction as early as possible if no further reduction in the DDD size can be obtained. For this purpose, the lower bounds formulated for DDDs in the previous section are incorporated. Suppose we want to sift the label at position *i* down one slot. From Theorem 2, we know the minimum DDD size is bounded by the following relationship:

$$S' \geq \min_{j=i+1,\cdots,n} (\#vertices(X_{i-1}^{1}) + \max\left\{\left\lceil \frac{1}{2^{j-i}} \#vertices(x_i) \right\rceil + \sum_{k=i+1}^{j} \left\lceil \frac{1}{2} \#vertices(x_k) \right\rceil, (j-i+1)\right\} + \#vertices(X_{n}^{j+1})). \quad (10)$$

The minimum is taken from $j = i+1$ to n because we need to look at every position the label at position i could move into and see whether to sift the one slot further or not. The minimalism holds when j = n. Therefore we have the lower bound:

$$S' \geq LBD = \#vertices(X_{i-1}^{1}) + \max\left\{\left\lceil \frac{1}{2^{n-i}} \#vertices(x_i) \right\rceil + \sum_{k=i+1}^{n} \left\lceil \frac{1}{2} \#vertices(x_k) \right\rceil, (n-i+1)\right\}. \quad (11)$$

This lower bound expression can be calculated in *O(n)* time. Pseudo code is given in Figure 9 for sifting down with lower bounds and sifting back to the best DDD size found. The smallest DDD found so far is denoted by *BestSize* and the position of a label where the *BestSize* was found is represented by *BestPosition*. The current size of the DDD is *NumVertices* and the current position the label is at is denoted as *Position*. For a label at position m, LB_m represents the result of the lower bound formula. To exchange two labels at positions *x* and *y* the function Exchange2Labels(x, y) is called.

```
k = sifting label;
LB_k = LBD( k);
BestSize = NumVertices;
BestPosition = k;
for(i = k to n){
    if(LB_i < BestSize){
        Exchange2Labels( i, i+1);
        if (NumVertices < BestSize){
            BestSize = NumVertices;
            BestPosition = i;
        }
        LB_i = LBD( i );
    }
    else{
        Position = i;
        break;
    }
}
for (i = Position to BestPosition){
    Exchange2Labels( i, i-1 );
}
```

Figure 9. Pseudo code of sifting down with lower bounds and sifting back to best sized DDD.

From Theorem 2, the case of sifting up is symmetrical to the case of sifting down. The lower bound formula for sifting up is similar to (11). The pseudo code of the algorithm for sifting up can be written analogous to Figure 9.

As an example, Table 3 shows steps for sifting label *H* with lower bounds implemented. As in Table 2, the first column shows the current vertex label order. The second column shows the size of the best-sized DDD found so far. The current number of vertices in the DDD for the order found in column 1 is displayed in column 3. Column 4 gives the lower bound calculation for swapping H with its neighbor for it's respective sift type shown in the next column. The sift type can be either sift up or sift down. Best position is given in the last column and represents the number of swaps away from the best-sized DDD found. In comparison to Table 2, we can see that using lower bounds reduced the number of swaps to sift H by 59%.

Table 3. Steps of sifting label H up and down using lower bounds (LBs).

Order	Best Size	# Vertex	LB vs. Best Size	Sift Type	Best Pos.
A C D F G **H** B E I J	13	13	11 < 13	*Up*	0
A C D F **H** G B E I J	13	13	12 < 13	*Up*	1
A C D **H** F G B E I J	13	13	12 < 13	*Up*	2
A C **H** D F G B E I J	13	14	13 = 13 stop	*Dn*	3
A C D **H** F G B E I J	13	13	-	*Dn*	2
A C D F **H** G B E I J	13	13	-	*Dn*	1
A C D F G **H** B E I J	13	13	11 < 13	*Dn*	0
A C D F G B **H** E I J	12	12	12 = 12 stop	-	**0**

The process is continued until all labels have been sifted. The resulting vertex label order is: (A C D F B G H E I J) and its corresponding DDD is shown on the right hand side of Figure 2. This reordered DDD now has a total number of vertices equal to 10. This is a reduction of 3. It can be verified that this is the optimum order.

7. Experimental Results

The proposed exact and heuristic DDD minimization algorithms have been implemented, integrated with a symbolic analysis program [11], and tested on a set of circuits, including filters and operational amplifiers and some RLC circuits [11]. For each test case, DDD is first constructed from the MNA circuit matrix using the method in [11], and then reordered to minimize its size using the exact minimization algorithm proposed here. The correctness of the algorithms and their implementation have been experimentally confirmed by observing the same frequency-domain responses from reduced DDDs and original DDDs for all the test circuits in [11]. Similar to other work on BDD minimization [4][5], we compare the efficiencies of different algorithms based on the number of neighboring label exchanges, *swaps*, used, instead of absolute CPU time.

Table 4 summarizes the exact DDD minimization results. Columns 1 and 2 list the test case name and the number of labels (*n*) for each test case. Columns 3, 4 and 5 are, respectively, the number of DDD vertices before minimization, after minimization, and the percentage size reduction. Columns 6, 7 and 8 list, respectively, the number of swaps without using pruning, the number of swaps using pruning with the DDD lower bound, and the percent of swaps when with pruning compared to that without. For the last 4 examples, exact minimization without pruning ran out the memory.

We note that the original DDD sizes are based on the orderings derived by heuristics in [11]. We can observe that the proposed exact minimization algorithm can further reduce the number of vertices by up to 38%. For example, the original DDD representation for test case *opamp* has 8 vertices, and it is reduced to 6. We can further observe that for large examples, pruning with bounds uses significantly less number of swaps than, typically a few percent of, than without pruning.

Table 4. Exact DDD minimization results.

ckt	*n*	#Vertices			#Swaps		
		before	after	% Δ	No Prune	With Prune	% Δ
E1	2	2	2	0	2	2	0
E4	3	3	3	0	18	30	167
2x2	4	4	4	0	144	264	183
E5	5	6	5	17	1201	1161	97
E8	6	6	6	0	10800	19500	181
Opamp	6	8	6	25	12139	5308	44
RC	7	9	7	22	105848	31843	30
E9	7	9	7	22	105841	31671	30
E10	7	8	7	13	105849	32657	31
E11	8	12	8	33	1128967	48068	4
E16	9	12	9	25	13063696	95752	1
E13	9	12	9	25	13063686	168111	1
3x3	9	13	12	8	13063682	527118	4
Fig.1	10	13	10	23	X	347270	X
TwoStage	10	16	10	38	X	545452	X
E15	11	15	12	20	X	14430117	X
E14	12	22	13	41	X	5889044	X

X -- without results after memory consumed.

Table 5. Results of DDD reordering using sifting and sifting with lower bounds.

Circuit	*n*	# Vertex before reorder	Did not use sign transformation rules							Used sign transformation rules						
			# Vertex after reorder	Size reduced	# Swap No LB	# Swap BDD LB	# Swap DDD LB	Δ #Swap DDD/ No LB	Δ #Swap DDD/ BDD	# Vertex after reorder	Size reduced	# Swap No LB	# Swap BDD LB	# Swap DDD LB	Δ #Swap DDD/ No LB	Δ #Swap DDD/ BDD
big	117	1612	1410	12.5%	26975	12593	11443	57.6%	9.1%	1528	5.2%	27100	12674	11444	57.8%	9.7%
ccstest	35	109	93	14.7%	2391	1369	1307	45.3%	4.5%	101	7.3%	2378	1354	1294	45.6%	4.4%
folded	67	2379	1908	19.8%	8888	4762	4458	49.8%	6.4%	2173	8.7%	8832	4572	4212	52.3%	7.9%
pino	19	24	21	12.5%	680	290	290	57.4%	0.0%	21	12.5%	680	290	290	57.4%	0.0%
rlcmix	39	130	115	11.5%	2669	1617	1461	45.3%	9.7%	124	4.6%	2668	1630	1492	44.1%	8.5%
rlctest	39	119	105	11.8%	2971	1617	1523	48.7%	5.8%	111	6.7%	2970	1592	1510	49.2%	5.2%
s741m	55	304	215	29.3%	5983	3149	2999	49.9%	4.8%	270	11.2%	5934	3202	3010	49.3%	6.0%
statvar	49	43	39	9.3%	1802	736	736	59.2%	0.0%	40	7.0%	1808	704	704	61.1%	0.0%
u741m	90	7431	3112	58.1%	16209	7295	6887	57.5%	5.6%	7075	4.8%	15994	7120	6592	58.8%	7.4%
ua741	103	7228	4800	33.6%	21074	8890	8284	60.7%	6.8%	6448	10.8%	20972	8330	7814	62.7%	6.2%
vcstest	46	121	95	21.5%	3630	1864	1776	51.1%	4.7%	116	4.1%	3616	1846	1728	52.2%	6.4%

Table 5 summarizes the heuristic DDD minimization results. The first three columns list, respectively, the test case name, the number of symbols (n), and the number of DDD vertices before reordering. The next seven columns describe the results without considering the vertex signs. Columns 4 and 5 are, respectively, the numbers of DDD vertices after reordering and the percentage size reduction with reordering. Columns 6 to 8 list, respectively, the number of swaps used by three reordering algorithms: sifting without using lower bounds, sifting using Drechsler's BDD lower bounds, and sifting using the new DDD lower bounds. The percent reduction of the number of swaps used by sifting with new DDD lower bound with respect to reordering without lower bounds and reordering with Drechsler's lower bound are computed in Columns 9 and 10. The last 7 columns describe the results of reordering when the vertex signs are considered. The number of swaps performed gives you insight on computation time.

We can observe that for the set of test cases, reordering reduces the DDD sizes (Column 5 and 12, respectively) by 9% up to 58% without sign and 4% up to 12% with sign. Typically, the bigger the case, the more reduction achieved. This is significant considering that the original DDDs were constructed with a provably good ordering heuristic that guarantees the optimality for ladder-structure circuits, and yields DDDs orders of magnitude smaller than that of random ordering [11].

We can also observe that sifting with the new lower bounds developed in this paper reduces the number of swaps used by sifting without lower bounds by about 55% (Column 9 and 16). In comparison to Drechsler's lower bounds developed for BDDs, the new lower bounds introduced here for DDDs lead up to about 10% reduction in the number of swaps (Column 10 and 17). Furthermore, we can observe that the reduction of the number of swaps due to pruning with the new lower bounds increases with the size of a problem compared to pruning with the known lower bounds.

8. Conclusions

In this paper, we studied how to reorder DDD vertices to minimize the DDD sizes. We presented new DDD neighboring exchanging and sign transformation rules that enable us to adapt the previous work on BDD minimization to DDD minimization. We derived tighter lower bounds than the previous known on DDD sizes under neighboring vertex exchanges and sifting. Both exact and heuristic DDD minimization algorithms are then developed, implemented, and applied to symbolic circuit analysis. Experimental results have indicated that our exact minimization algorithm can handle the problems with up to 10 variables. This enables its use to minimize the complexity of symbolic expressions for human interpretation and visualization, since in such a case that the number of variables are only a few. Experimental results have further shown that vertex reordering can reduce substantially the sizes of DDDs even for some already compact DDD representations. Further, lower-bound based sifting reduces the computational cost by about 55% in comparison with sifting without exploiting lower bounds.

With the complexities of symbolic circuit analysis algorithms at best linear in the size of a DDD, the proposed reordering algorithms will reduce significantly both the space and CPU time requirements for evaluating and manipulating symbolic expressions for circuit analysis. Similar to BDDs, which provides an architecture for hardware implementation of Boolean functions, DDD also provides a canonical hardware architecture with adders and multiplier for linear equation solving, thus minimizing the number of DDD vertices can reduce the hardware cost for such an application.

Acknowledgement

The authors thank Dr. Rolf Drechsler of Siemens AG Corporate Technology, Germany for several helpful discussions on decision diagram minimization.

References

[1] S. B. Akers, "Binary decision diagrams", IEEE Transactions on Computers, vol. C-27, pp. 509-516, June 1976.

[2] B. Bollig, M. Lobbing, and I. Wegener, "On the effect of local changes in the variable ordering of ordered decision diagrams", *Information Processing Letters*, vol. 59, pp. 233-239, Oct. 1996.

[3] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation”, *IEEE Transactions on Computer-Aided Design*, vol. 35, no. 8, pp. 677-691, August 1986.

[4] R. Drechsler, W. Gunther, and F. Somenzi, “Using lower bounds during dynamic BDD minimization”, *IEEE Transactions on Computer-Aided Design*, pp. 51-57, Jan. 2001.

[5] R. Drechsler, N. Drechsler, and W. Gunther, “Fast exact minimization of BDD’s” *IEEE Transactions Computer-Aided Design*, vol. 19, no. 3, pp. 384-389, Mar. 2000.

[6] S. J. Friedman and K. J. Supowit, “Finding the optimal variable ordering for binary decision diagrams”, *IEEE Transactions on Computers*, vol. 39, no. 5, pp. 710 – 713, May 1990.

[7] G. Gielen and W. Sansen, *Symbolic Analysis for Automated Design of Analog Integrated Circuits.* Norwell, MA: Kluwer Academic, 1991.

[8] A. Manthe and C.-J. R. Shi, “Lower bound based DDD minimization for efficient symbolic circuit analysis”, in *Proc. IEEE International Conf. on Computer Design, (ICCD),* Austin, TX, Sept. 2001, pp. 374-379.

[9] S. Minato, “Zero-suppressed BDDs for set manipulation in combinatorial problems”, in *Proc. 30th IEEE/ACM Design Automation Conf.*, Dallas, TX, June 1993, pp. 272-277.

[10] R. Rudell, “Dynamic variable ordering for ordered binary decision diagrams”, in *Proc. IEEE International Conference on Computer-Aided Design*, Santa Clara, CA, 1993, pp. 42-47.

[11] C.-J. R. Shi and X.-D. Tan, “Canonical symbolic analysis of large analog circuits with determinant decision diagrams”, *IEEE Transactions on Computer-Aided Design*, vol. 19, pp. 1-18, Jan. 2000.

[12] C.-J. R. Shi and X.-D. Tan, “Compact representation and efficient generation of s-expanded transfer functions for computer-aided analog circuit design”, *IEEE Transactions on Computer-Aided Design*, vol. 20, pp. 813-827, Jul. 2001.

[13] W. Verhaegen and G. Gielen, ``Efficient DDD-based symbolic analysis of large linear analog circuits”, in *Proc. IEEE/ACM Design Automation Conference*, Las Vegas, NE, June 2001, pp. 139-144.