Boolean Function Representation based on disjoint-support decompositions.*

Valeria Bertacco and Maurizio Damiani
Dipartimento di Elettronica ed Informatica
Universitá di Padova, Via Gradenigo 6/A, 35131 Padova, ITALY
e-mail: brian@dei.unipd.it damiani@dei.unipd.it
Tel: +39 49 827 7829 Fax: +39 49 827 7699

Abstract

The Multi-Level Decomposition Diagrams (MLDDs) presented in this paper provide a canonical representation of Boolean functions while making explicit disjoint-support decomposition. This representation can be directly mapped to a canonical multi-level gate network of a logic function with AND/OR or NOR-only (NAND-only) gates.

Using MLDDs we are able to reduce the memory occupation with respect to traditional ROBDDs for several benchmark functions, by decomposing logic functions recursively into simpler - and more condivisible - components. Because of this property, analysis of the MLDD graphs allowed us to sometimes identify new and better variable ordering for several benchmark circuits. We expect the properties of MLDDs to be useful in several contexts, most notably logic synthesis, technology mapping, and sequential hardware verification.

1 Introduction

Reduced, Ordered Binary Decision Diagrams (ROBDDs) [1] are probably the most powerful data structure known so far for the manipulation of large logic functions, and for this reason they have become pervasive in logic synthesis and verification environments [2, 3, 4, 5]. Ongoing research is attempting to extend their applicability to other domains, such as the solution of graph problems and integer-linear programming [6, 7].

Still, some key inefficiencies (an exponential blowup for some classes of functions, the unpredictability of the ROBDD size and shape with respect to the variable ordering chosen, etc ...) motivate an increasing research activity in this area. Research directions include in particular: Efficient implementations [8, 9], development of ordering heuristics [10, 11, 12], and alternative representations altogether [13, 14, 15, 16, 17].

In [18], the authors presented an addition to the basic ROBDD representation, based on the analogy of ROBDDs with deterministic finite automata. The new representation was a counterpart of a nondeterministic automaton (hence possibly more compact), in which a function rooted at each ROBDD node was represented as a logic OR of simpler, disjoint-support components.

[0]This research was partially supported by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (Project GEPPCOM) and by CNR grant # 95.02061.CT07

In this paper, we add to the basic ROBDD representation the capability of discovering the presence of **arbitrary**, **multiple-level tree** decompositions of functions. The representation shares with ROBDDs canonicity, a directed-acyclic graph structure, and a recursive construction technique. Unlike ROBDDs, however, nodes may represent not only two-input MUXes, but also unlimited-fanin OR / AND (or NAND-only, NOR-only) gates. It is worth noting that, because of gate-like nodes, our representation is essentially a multiple-level circuit.

Through the use of a multiple-level NOR-only (or NAND-only) decomposition, we maintain constant-time complementation; and because of the tree decomposition, the representation is significantly less order-sensitive than ROBDDs. In particular, we identified a class of functions for which the representation is totally independent from the variable order chosen, and for which some difficult problems (like, Boolean NPN matching [19]) can be solved in linear time. These features represent a substantial improvement over the work [18], where a single-level OR decomposition was used, and complementation was difficult.

Experimentally, we found that the new representation is memorywise significantly more compact than ROBDDs, because decomposable functions can share components. More interestingly, however, the new representation gives us some systematic and exact insight on the role of the input variables of a logic function. This insight is deferred to special-purpose heuristics (such as dynamic reordering) in the case of ROBDDs.

The rest of the paper is organized as follows. Sections 2 and 3 introduce disjoint-support decomposition and MLDDs, respectively. Section 4 describes the procedures used for MLDD manipulation, and eventually Section 5 presents the experimental results. Proofs of Theorems are deferred to the Appendix, for the sake of readability.

2 Disjoint support decomposition

The representation presented in this paper is based on the notion of tree decomposition of a function. In this section, we introduce the basic definitions concerning this decomposition.

We consider the decomposition of functions into the NOR (NAND, OR, AND) of disjoint-support subfunctions, whenever possible. This notion will lead to a recursive (e.g. tree) decomposition style and to the definition of Multi Level Decomposition Diagrams (MLDDs).

Definition 1. *Let $f : B^n \rightarrow B$ denote a non-constant Boolean function of n variables $x_1, \cdots, x_n$. We say that f* **depends** *on x_i if $\partial f / \partial x_i$ is not identically 0. We call* **support** *of f (indicated by $S(f)$) the set of Boolean variables f depends on.* □

Definition 2. *A set of non-constant functions $\{f_1, \cdots f_k\}$, $k \geq 1$, with respective supports $S(f_i)$ is called a* **disjoint-support NOR decomposition** *of f if:*

$$\overline{f_1 + \cdots + f_k} = f; \qquad S(f_i) \cap S(f_j) = \emptyset, \;\; i \neq j \quad (1)$$

A disjoint support NOR decomposition is **maximal** *if no function f_i is further decomposable in the OR of other functions with disjoint support. We define disjoint support OR, AND, NAND decompositions in a similar fashion. We indicate by* $\mathbf{D_{NOR}(f)}$ *any such maximal decomposition.* □

Example 1. The function $f = (ab + a'c)(d + e)$ has the following disjoint-support decompositions:

- AND: $f_1 = (ab + a'c)$ and $f_2 = (d + e)$;
- NOR: $f_1 = (ab + a'c)'$ and $f_2 = (d + e)'$;
- NAND and OR: $\{f\}$.

□

In the rest of the paper, disjoint-support decompositions are referred to as decompositions, for short. Moreover, we focus only on NOR decomposition, as the results for the other decompositions can be obtained readily by standard Boolean algebra.

2.1 Tree decompositions

Decomposition can be applied recursively to logic functions. In this case, we obtain a representation of F based on a NOR tree.

Example 2. The function $F = (a + b)(c'd' + e + f'g')$

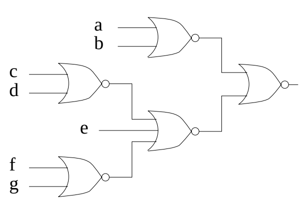

Figure 1. A recursively decomposable function.

is recursively NOR decomposable. From the first decomposition we obtain $f_1 = (a + b)'$ and $f_2 = [e + (c + d)' + (f + g)']'$. These functions are then again decomposable until reaching the input variables, as reported in Fig. (1). □

Definition 3. *A* **tree decomposition** *of a logic function f is a recursive decomposition of f into a NOR-only tree of subfunctions, where the functions at the inputs of each NOR are maximally decomposed. We indicate by* $\mathbf{TD_{NOR}}$ *the decomposition tree. Similarly we can define TD_{NAND} and $TD_{AND/OR}$.* □

Theorem (1) below states an intuitive but relevant result.

Theorem 1. *For a given function f, the following properties hold:*

1. *there is a unique D_{NOR} ;*
2. *there is a unique TD_{NOR} .*

□

2.2 Tree-decomposable functions.

When decomposing a function, it may be possible that the leaves of the decomposition reduce only to primary inputs or their complements. This is the case, for example, of the function $F = (a+b)(c+d+e) = [(a+b)'+(c+d+e)']'$.

Definition 4. *A logic function $f(x_1, \ldots, x_n)$ is* **tree-decomposable** *if the input subfunctions of its TD_{NOR} belong to the set $\{x_1, \ldots, x_n, x'_1, \ldots, x'_n\}$, i.e. the set of inputs variables and their complements.* □

If a function is tree decomposable, then Theorem (1) indicates that its decomposition tree TD_{NOR} is a canonical representation. Not every function,however, is tree-decomposable. For instance, the function $F = a'b' + c'd'e'$ cannot be represented as the NOR of any disjoint-support subfunctions. Hence, NOR decomposition trees are not a universal representation style. We can enlarge, however, the set of tree-representable functions as follows. Sometimes the complement of a non-decomposable function may be decomposable. In this case the complement $F' = (a + b)(c + d + e)$ is indeed tree-decomposable. We can thus exploit the decomposability of F' in representing the D_{NOR} of F by simply appending a NOT gate at the root of the NOR tree. Fig. (2) shows the representation.

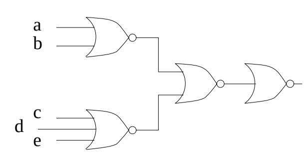

Figure 2. Recursive decomposition with a NOT at root.

The only remaining question is whether the introduction of NOT gates at the root preserves canonicity, that is, whether a tree-decomposable function can have two decomposition trees, one with a NOT at the root and another without it. To this regard, the following result holds:

Theorem 2. *If a logic function F is tree-decomposable, then its complement $\overline{F}$ is not.* □

Because of Theorem (2), NOT gates can appear only at the root or the leaves of the TD_{NOR} . Suppose, by contradiction, that a topology like in Fig. (3) were possible. In this case, we could merge the NOR gate $N2$ with $N1$. This would indicate that that we had not decomposed maximally the function represented by the NOR N1.

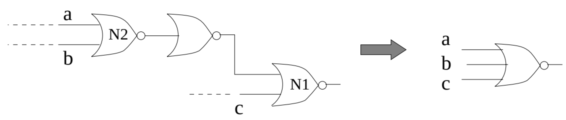

Figure 3. An impossible topology for NOR decomposition trees.

Based on recursive D_{NOR} , we have now partioned logic functions into three classes:

1. tree-decomposable functions;
2. functions tree-decomposable with the use of NOT gates;
3. functions not tree-decomposable.

In the next section, we describe how the notion of tree decomposition and decomposability is used for obtaining a hybrid representation style for arbitrary logic functions. This representation will contain NOR trees and BDD nodes.

We conclude this section with some observations on tree-decomposable functions:

Canonicity and variable orderings.

As mentioned, for tree-decomposable functions, the tree decomposition is canonical. Moreover, unlike ROBDDs, the tree representation is trivially **independent** from the variable ordering. Indeed, even if a function F is not entirely tree-decomposable, the knowledge of a partial decomposition indicates ordering strategies for the input variables of F. If F is decomposable as, say, $F = (f_1 + f_2)'$, then optimal orderings will place all the variables of f_1 on top of those of f_2 (or viceversa), and the size of the ROBDD of F will be the sum of those of f_1 and f_2. Hence, it follows in particular that the ROBDD of any tree-decomposable function, with an optimal variable ordering, is linear in the number of inputs.

Boolean matching.

Boolean matching is an important step of technology mapping [19]. It consists of finding whether two functions $f(x)$ and $g(y)$ coincide after replacing the input variables y with a permutation Px of the input variables x. Variations of the problem include matching modulo complementation of some inputs and of one of the functions, and it is named NPN-matching (Negation, Permutation, Negation matching). In general, this is a difficult problem, as it entails enumerating the permutations of the input vector x and checking the equivalence of $f(x)$ with $g(Px)$. For tree-decomposable functions, clearly a match exists if and only the trees representing f and g are equal, except possibly for the presence of input/ output inverters. Tree isomorphism can be carried out in time linear in the size of the tree [20]. More generally, a matching can exist only if g can be decomposed in a fashion similar to f. Even a partial decomposability of f is thus helpful.

Boolean manipulation routines.

ROBBD manipulation routines are based on a recursive visit of the ROBDD functions f. At each recursion, a variable x is selected and the cofactors $f_x, f_{x'}$ are evaluated. Recursion is made fast because, by their very nature, ROBDDs allow constant-time cofactoring. If a function is represented by a NOR tree, instead, then cofactoring requires assigning the value to the tree input and then propagating the effect (*i.e.* , simulating) towards the output. This simulation takes time proportional to the tree depth. In the implementation section it will be seen how the knowledge of a decomposition, however, helps compensating this more difficult cofactoring.

3 Multi-Level Decomposition Diagrams

In this section we exploit tree decompositions to derive a new hybrid model for representing logic functions. We will represent functions of the first and second class by a tree of NOR gates. Functions in the third class will be represented through the use of Shannon expansion with respect to some variable x, leading to BDD nodes. We then apply tree decomposition and Shannon expansion in order to each cofactor f_x and $f_{\overline{x}}$ recursively.

Example 3. The function $f = (a'b + ac'd' + e'f')'$ has

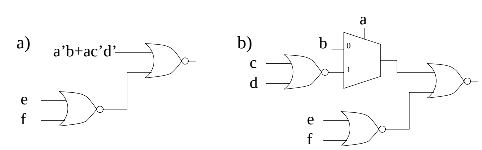

Figure 4. a) Tree decomposition of the function in Example (3). b) The same function with the addition of BDD nodes.

TD_{NOR} as in Fig. (4.a). Note that in no case we could further decompose $ac'd' + a'b'$ because of the disjoint support constraint. Applying Shannon expansion, in Fig. (4.b) we obtain a TD_{NOR} for each input of MUX. □

The new structure we present in this paper explores tree decompositions of a given function. Because of its purpose we called it Multi-Level Decomposition Diagrams, MLDD. We now define MLDDs based on TD_{NOR} . In our drawings of graphs, circles represent MUX vertices, while arrays of squares represent NOR vertices.

Definition 5. *A MLDD is a directed acyclic graph, with leaf vertices labeled by a Boolean constant or variable and two kinds of internal vertices:*

NOR vertices *have a non-empty set of outgoing edges, each pointing to a MLDD.*

MUX vertices *have two outgoing edges, 0 and 1, and are labeled by a Boolean variable.*

A MLDD defines recursively a logic function with the following rules:

- *A terminal vertex t labeled by Boolean variable or constant x denotes the function x.*
- *A MUX vertex m labeled by Boolean variable x defines the function:*

$$F_m = \overline{x}F_{\mathbf{0}}(m) + xF_{\mathbf{1}}(m) \tag{2}$$

- *A NOR vertex n with k outgoing edges defines the function:*

$$F_n = \overline{f_1 + \cdots + f_k} \tag{3}$$

where f_i, $i = 1, \ldots, k$ is the function defined by the MLDD pointed by edge i.

□

In a MLDD, while MUX vertices correspond to ROBDD nodes, NOR vertices are a new feature of this model which emphasizes the tree decompositions of the function.

Just like ROBDDs, we impose **reduction rules** and **ordering rules** to MLDDs in order to obtain a more compact canonical representation:

- There are no two identical subgraphs in the same MLDD.
- There are no vertices with two or more outgoing edges pointing at the same MLDD .
- We impose a total ordering between variables labeling internal and terminal vertices of a MLDD. Each path from root to a terminal must traverse subsequent MUX nodes in respect of this ordering and each variable is evaluated at most once on each path.

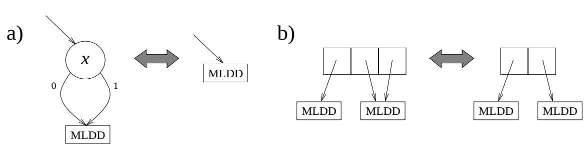

Figure 5. Second reduction rule. a) Mux vertices. b) NOR vertices.

It is worth noting that, unlike ROBBDs, the second reduction rule bears different consequences on the two kinds of internal vertices. As sketched in Fig. (5), a reduction of a MUX vertex implies the deletion of the node. This is not the case for NOR vertices.

In addition to ROBDD-like rules, in order to grant canonicity we must impose decomposition rules:

- the subfunctions pointed by a NOR vertex must have disjoint support. None of them can be decomposed by a D_{OR};
- a function is represented by a MUX iff it is not decomposable, nor its complement.

The following result is a direct consequence of the canonicity of tree decompositions and reduction rules. We thus state it without proof, for the sake of conciseness:

Theorem 3. *Reduced Ordered Decomposed MLDDs are canonical.* □

The MLDD of a function matches a multi-level logic circuit in the obvious way. In Fig. (6.a) and (6.b) we reported a MLDD and the corresponding gate-level network. Due to this evident correspondence, hereafter we will call graph nodes indifferently vertices or gates.

Example 4. Fig. (6.a) represents the canonical MLDD

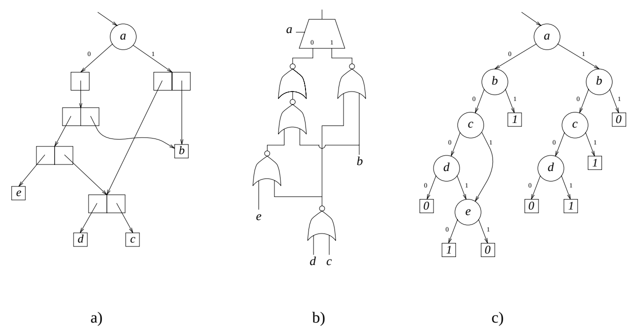

Figure 6. a) The MLDD of the function in Example (4). b) A logic circuit view of the MLDD. c) Its ROBDD.

for the function $f = (c + d)(ab' + a'e') + a'b$ with a lexicographical ordering of the variables.

In this case two distinct subgraphs share the NOR gate representing $c'd'$. This is not often the case for simple functions. For more complex functions it is more likely to happen.

In Fig. (6.c) shows the corresponding ROBDD. The MLDD has decomposed both cofactors f_a and $f_{a'}$ until reaching the input variables. □ .

3.1 Properties of MLDDs

We conclude the section by pointing some results on D_{NOR}s that are useful for the construction of MLDD manipulation routines.

Theorem 4. *Suppose $\{f_1, \cdots, f_k\}$ is a D of some function. Then, by erasing elements from the set, the new set is also a D.* □

Theorem 5. *If D_{NOR} $(f) = \{f_1, \cdots, f_k\} \cup \{p_1, \cdots, p_h\}$ and D_{NOR} $(g) = \{g_1, \cdots, g_l\} \cup \{p_1, \cdots, p_h\}$, where $g_i \neq f_j, i = 1, \cdots l, j = 1, \cdots k$, then:*

1. *D_{NOR} $(f \cdot g) = \{p_1, \cdots, p_h\} \cup$ D_{NOR} $([(f_1 + ... + f_k)' \cdot (g_1 + ... + g_l)']')$.*
2. *D_{NOR} $(f + g) = \{p_1, \cdots, p_h\} \cup$ D_{NOR} $([(f_1 + ... + f_k)' + (g_1 + ... + g_l)']')$*
3. *Let x denote a variable not in the support of f or g. Then:*

$$D_{NOR}\,(x'f + xg) = \{p_1, \cdots, p_h\} \cup D_{NOR}\,([x'(f_1 + \ldots + f_k)' + x(g_1 + \cdots + g_l)']')$$

□

Theorem 6. *Let x denote a variable, $x \notin S(g)$, and suppose $f = x + g'$. Then,*

$$D_{NOR}\,(f) = \{x\} \cup D_{NOR}\,(g) \tag{4}$$

□

3.1.1 Complementation

The MLDD of a tree-decomposable function is trivially a NOR tree, possibly with a NOT gate at the root. This allows constant time and space complementation.

It is well known [8] that for MUX nodes, the insertion of NOT gates (*i.e.* complement edges), can arise canonicity problems. To get around this problem we use NOT gate reduction rules similar to those of [8]. These are depicted in Fig. (7).

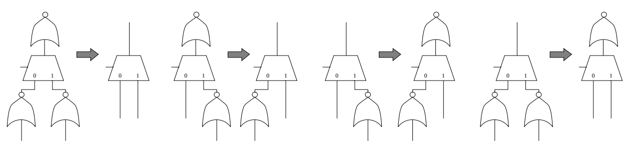

Figure 7. Equivalent MLDDs

4 MLDD manipulation routines

As we have seen, this model has some of the ROBDD features. Among these, a data structure that can be manipulated through recursive procedures.

The data structure we implemented realizes vertices uniformly with n-tuples, the first element being an integer, all the others being pointers to other MLDDs. In the first element we encode the type of node (*i.e.* , MUX or NOR vertex), the number of elements in the n-tuple (for MUX nodes it is always 2) and the top variable of the function represented.

We maintain the structure in strong canonical form, *i.e.* , two equivalent functions are identified by the same pointer, by the familiar hashing mechanism.

We have then implemented Boolean operation routines. As an example, Fig. (8) reports the pseudo-code for the logic OR of two functions.

Rows 1, 2 and 3 are the application of Theorem 5, case 2. We seek common elements in the operands and remove them from the recursive operation. This removal can result in faster execution because we have simpler operands.

`D(op)` indicates the set of elements of the decomposition of `op`. In a NOR vertex `op`, it is the set of all outgoing pointers (\ indicate set operation of difference).

The situations for which `op` is a MUX is a special case. For a uniform management of the structure and functions represented, we indicate as `D(op)` of a MUX vertex `op`, a pointer to the complement of the function rooted at `op`.

```
OR (mldd op1, mldd op2)
{
1    D(opc) = D(op1) ∩ D(op2);
2    D(op1) = D(op1) \ D(opc);
3    D(op2) = D(op2) \ D(opc);
4    if (terminal case) return (D(opc) ∪ terminal value);
5    res = comp_lookup(op1, op2);
6    if (res != NULL) return (D(opc) ∪ res);
7    x = top_var(op1, op2);
8    if (top_var(op1) == x)
     {
9      op1_l = evaltop(op1, 0); op1_r = evaltop(op1, 1);
10   } else {
11     op1_l = op1; op1_r = op1;
     }
12   if (top_var(op2) == x) { symmetric case }
13   left=OR (op1_l, op2_l); right=OR (op1_r, op2_r);
14   res = mldd_find (left, right, x);
15   comp_insert (op1, op2, res);
16   return (D(opc) ∪ res);
}
```

Figure 8. Pseudocode of `OR()`

We also maintain a computed table, like that of standard ROBDD procedures, where we store partial results. The removal of common subfunctions also helps avoiding the overfill of this table because we can exploit the generic single entry of the table $F' + G' = H'$ for retrieving results of every operation $(F + f)' + (G + f)' = (H + f)'$ when f varies, which consequently needs not be stored.

If the search in the computed table fails, we start recursion. First of all we find the top variable of the operands, which is immediate due to its encoding in the first element of the data structure.

Procedure `evaltop(f, value)` returns the MLDD of function $f_{x=value}$ assuming x is the top var. of f. This step corresponds to taking cofactors in ROBDDs. After recursion, `mldd_find()` creates a MLDD from a top var. and its cofactors.

We now analyze in more detail these three steps, namely, terminal cases, cofactoring, and MLDD creation.

Terminal cases and values depend on the operation we are applying. For the Boolean `OR` we recognize the following situations:

terminal case	return value
`op1=1, op2=1`	**1**
`op1=0, op2=0`	`op2, op1`
`op1 = op2`	`op1`
$\exists x, x \in$ `DSD(op1')`; $x' \in$ `DSD(op2')`	**1**

Procedure `evaltop()` is responsible for cofactoring. Its pseudo-code is reported in Fig. (9, and Fig. (10) shows its operation. `evaltop()` recursively goes down the tree decomposition until it reaches the MUX node labeled with the top variable of the MLDD, and it takes its cofactor. In Fig. (10.a), this is the MUX labeled by a. Returning up from recursion, it substitutes NOR vertices with newly generated ones, while maintaining canonicity (the shaded gates of Fig. (10.b)).

```
evaltop (mldd op, boolean value)
{
1   if (op is a MUX node) return (op.value);
2   i = element of op such that op.topvar = op.i.topvar;
3   opr = evaltop(op.i, value);
4   D(op) = D(op) \ op.i;
5   D(res) = D(opr) ∪ D(op);
6   return (res);
}
```

Figure 9. Pseudocode of `evaltop()`

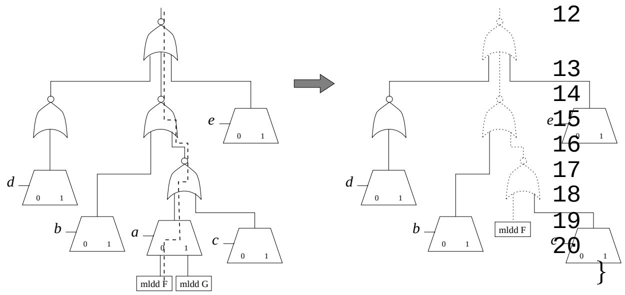

Figure 10. An example of `evaltop()` **application**

```
mldd_find(mldd left, mldd right, top_var x)
{
1   if (left == right) return (left);
2   if (right == 0) {
3     new_vertex = find_or_create(0, 1, x);
4     D(res) = new_vertex ∪ right;
5     return(res);
    }
6   if (right == 1) { similarly }
7   if (left == 1 or left == 0) { symmetric case }
8   if (left ⊂ D(right)) {
9     D(right) = D(right) \ left;
10    new_vertex = mldd_find(1, right, x);
11    D(res) = new_vertex ∪ left;
12    return(res);
    }
13  if (right ⊂ D(left)) { symmetric case }
14  D(opc) = D(left) ∩ D(right);
15  if ( D(opc) = ∅ ) return( find_or_create(left, right, x) );
16  D(left) = D(left) \ D(opc);
17  D(right) = D(right) \ D(opc);
18  new_vertex = mldd_find(left, right, x);
19  D(res) = D(opc) ∪ new_vertex;
20  return(res);
}
```

Figure 11. Pseudocode of `mldd_find()`

The code of `evaltop()` works as follows: Line 1 checks for end-of-recursion-case, *i.e.* reaching of a MUX node from which we can take the requested cofactor. Otherwise we have to find the critical element in our NOR vertex list to use for going down one level. Line 3 makes the recursive call with this critical element.

After recursion we substitute the critical element in the list with the returned graph. For example if the critical element was a MUX vertex we substitute it with its cofactor. While doing this work we may have to merge list and/or check for special cases (for example if the returned graph is the constant 1, we simply return the constant 0) and maintain canonicity (reduction rules).

Procedure `mldd_find()` is sketched in Fig. (11). It builds a MLDD trying to discover every possible 'common term' from the two cofactors. First of all, it checks for simple cases (rows 2 to 8). They are application of Theorem 6. For example, rows 2 to 5 examine the situation for *right* = **0**, *i.e.* , the function to generate is $f = x' \cdot left$. With NOR MLDD such a function is given by $f = (x + l_1 + \cdots + l_n)'$ (l_i are the components of *left*).

We have represented these terminal cases in Fig. (12). `find_or_create()` provides the creation or retrieval of a MUX or a terminal vertex while keeping up to date a unique table similar to that of ROBDD.

In rows 8 - 13 we check for one of the two general cases, where none of the cofactors is a constant. If the complement of one cofactor is contained in the other as a unique element, then there is a tree decomposition.

$$x'r_1' + x(r_1 + r_2 + \cdots + r_n)'$$

where $r_1, r_2, \ldots, r_n$ are the components of the right MLDD. This is equivalent to:

$$r_1 + [x' + x(r_2 + \cdots + r_n)']'$$

We have reported this case in Fig. (13.a).

Lines 14 - 20 deal with the other general case. Here we have to search for common elements between left and right MLDD and to factor them out. This applies case 3 of Theorem 5. These steps are sketched in Fig. (13.b).

As mentioned, `evaltop()` and `mldd_find()` replace cofactoring and the basic `find_or_create()` operations in ROBDDs. While operations are trivial constant-time in ROBDDs, they may take $O(d)$ time in MLDDs, where d denotes the tree depth. To this regard, we observe that d is bound by the number of variables and it is rather small in practice (always 3 or less for the synthesis benchmarks).

Moreover, as `OR` is applied to pairs of nodes down in the graph, the support set of subfunctions will have fewer elements and so the number of calls to `evaltop()`.

Example 5. We have reported in Fig. (14) a maximal depth tree decomposable function $f = (((x_1 + x_2)\overline{x_3} + x_4)\overline{x_5} + \ldots$. □

5 MLDDs versus ROBDDs

In this section we present some comparisons in representing functions with MLDDs and ROBDDs.

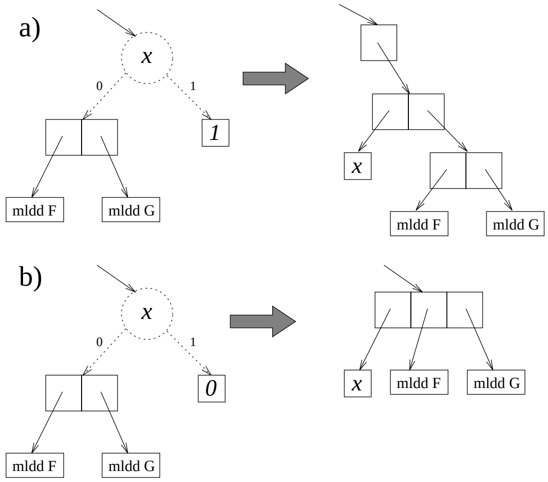

Figure 12. Identification of D during traversal - terminal cases

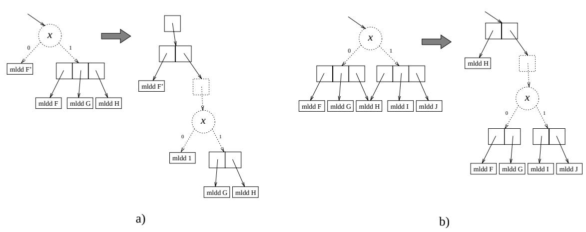

Figure 13. Identification of D during traversal - general cases

5.1 Exponential growth

In this subsection we contrast MLDDs with ROBDDs with respect to a particular class of order-sensitive functions, namely, the functions:

$$F_n = (x_1 + x_2)(x_3 + x_4) \cdots (x_{2n-1} + x_{2n}) \qquad (5)$$

It is well known that with an improper ordering of the variables (for example, placing the odd-labeled variables up top) results in a ROBDD for F_n with over 2^n nodes [1]. Moreover, in spite of the simplicity of the function, most variable orderings for F_n can be proved bad.

The MLDD for the function is shown in Fig. (15). It consists of a two-level NOR circuit, **regardless of the order chosen** for the variables x_i and it is always linear.

Example 6. Consider the function $f = (aA + bB)c' + (ab + AB)c$, with an ordering of variables placing c on top. Since $f_{c=0} \neq f_{c=1}$, any ROBDD has the aspect shown in Fig. (16.a). In general, we may think of a case where the two cofactors look like a function f_n of Eq. (5), but with a different combination of products. Any ordering of a, A, b, B which optimizes one branch is bound to be sub-optimal for the other branch of the ROBDD. Fig. (16.b) illustrates the MLDD for the same function. Both branches are automatically decomposed optimally. □

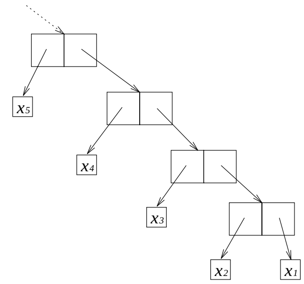

Figure 14. A maximal tree decomposable function

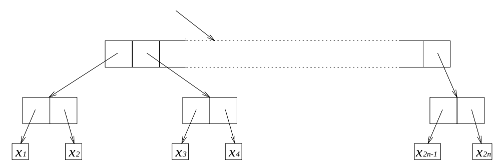

Figure 15. The PAD for the functions F_n

5.2 Tests on benchmark circuits

We have compared our new model with ROBDDs in a number of benchmark circuits in terms of memory occupation and CPU time needed to build the output function graphs.

The benchmarks are divided in three sections: multi-level circuits, two-level and a third section testing the combinational part of synchronous circuits. All these benchmarks come from the IWLS91 benchmark suite [21].

The variable ordering chosen for these circuits was obtained by applying the Berkeley ordering [3]. No variable reordering took place, however, during the execution of any package.

We have implemented our model and tested it against the Carnegie-Mellon ROBDD package. We carried out comparisons on the actual memory occupation. We assumed bare-bone implementations, in which in particular each ROBDD node takes three machine words. Moreover ROBDDs have complement edges. With regards to MLDD vertices, we assumed an implementation where each node con-

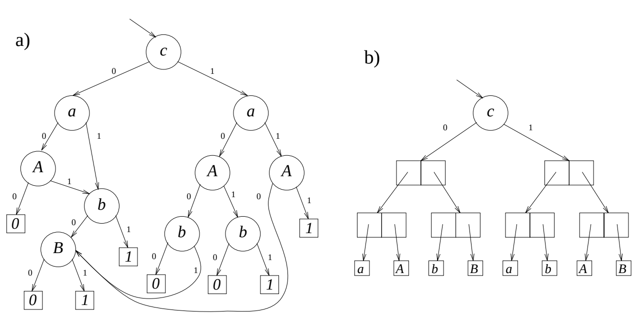

Figure 16. a) ROBDD structure for the function of Example (9). b) MLDD structure for the same function.

sists of an array. As mentioned, the first element stores informations about the node, while other elements are pointers. This model also implements NOT gates through complement edges.

CPU-time was taken on a HP Vectra 5/133 with 48Mbytes of RAM.

From Table 1, MLDDs turn out to be more compact on average of 18%. Some benchmarks give particularly good results, for example *comp* and *pair*, benchmarks which TD_{NOR} is very effective in decomposing output functions until reaching input variables or very simple functions.

The CPU time is always better for ROBDDs. Empirically we have found the following three reasons:

- We make internal recursions in the constructions of MLDDs (`evaltop()` and `mldd_find()`). Thus the number of calls to key procedures for each computation is higher.
- We have to manage arrays that in general have more elements than ROBDDs. For example, hash functions are more complex and also storing and retrieving from computed table and unique table needs more time.
- The structure we use allows multiple paths from a certain node (NOR). On the other hand, with ROBDD the path is unique. This is similar to simulation through a NFA opposite to a DFA.

We have also implemented dynamic reordering in our model with a sifting-based algorithm [12]. Over ROBDDs, we have the advantage to know more about a 'good variable order' directly from the data structure.

In table 2 we make comparisons using for each benchmark the order given by sifting (interestingly, it is different for the two models). Variable ordering took place only at the end of execution.

Results show that, after sifting, MLDDs improve slightly further over ROBDDs. We think this is because during sifting we exploit our better knowledge of the function's structure and can avoid to go through orderings that give a small advantage but block further improvements.

6 Conclusions and future work

MLDDs have proved themselves efficient in making explicit the Ds of logic functions.

This property allows us to reach a more compact, flexible and robust graph-based representation.

Moreover, this representation is closely related to a multiple level circuit, and is more informative on the role of the support variables of a function.

We expect these properties to be useful in diverse applications, most notably technology mapping for combinational circuits and Boolean matching /reachability analysis for verification / ATPG in sequential circuits.

7 Appendix

Proof of Theorem 1. The proof of the first assertion follows by contradiction: We assume the existence of two distinct D_{NOR} , namely, $\{f_1, \cdots, f_p\}$ and $\{g_1, \cdots, g_q\}$, and show that this leads

Benchmark	ROBDD		MLDD		RATIOS	
	nodes	mem	nodes	mem	nodes	mem
MultiLevel						
alu2	205	615	126	519	62.7%	18.5%
alu4	685	2055	511	1771	34.1%	16.0%
apex6	1171	3513	903	3377	29.7%	4.0%
apex7	555	1665	231	979	140.3%	70.1%
b9	181	543	75	452	141.3%	20.1%
C1355	45922	137766	44156	150231	4.0%	-8.3%
C432	31178	93534	16147	82676	93.1%	13.1%
C499	45922	137766	44156	150231	4.0%	-8.3%
c8	133	399	96	388	38.5%	2.8%
C880	12841	38523	9173	31476	40.0%	22.4%
cht	150	450	86	421	74.4%	6.9%
CM151	511	1533	285	1066	79.3%	43.8%
CM152	383	1149	284	1060	34.9%	8.4%
comp	5476	16428	434	1459	1161.8%	1026.0%
count	204	612	187	703	9.1%	-12.9%
DES	31508	94524	28185	90660	11.8%	4.3%
example2	869	2607	223	1362	289.7%	91.4%
frg1	204	612	53	458	284.9%	33.6%
frg2	3714	11142	3149	10472	17.9%	6.4%
k2	28336	85008	27437	86341	3.3%	-1.5%
lal	138	414	63	284	119.0%	45.8%
Adderfds	457	1371	456	1372	0.2%	-0.1%
pair	41128	123384	8053	26641	410.7%	363.1%
pcler8	144	432	98	392	46.9%	10.2%
rot	12537	37611	7796	27463	60.8%	37.0%
sct	118	354	49	239	140.8%	48.1%
term1	638	1914	154	540	314.3%	254.4%
too_large	7096	21288	4876	18153	45.5%	17.3%
ttt2	205	615	115	565	78.3%	8.8%
vda	4345	13035	4203	13235	3.4%	-1.5%
x1	1297	3891	223	1671	481.6%	132.9%
x4	683	2049	477	1825	43.2%	12.3%
TwoLevel						
alu4.pla	1197	3591	801	3294	49.4%	9.0%
apex5.pla	2679	8037	1088	5259	146.2%	52.8%
clip.pla	226	678	148	664	52.7%	2.1%
e64.pla	1441	4323	66	2404	2083.3%	79.8%
misex2.pla	137	411	34	294	302.9%	39.8%
misex3.pla	1301	3903	814	3929	59.8%	-0.7%
sao2.pla	155	465	48	319	222.9%	45.8%
vg2.pla	1044	3132	520	2429	100.8%	28.9%
F.S.M.						
ex1	769	2307	118	1785	551.7%	29.2%
ex2	375	1125	44	729	752.3%	54.3%
ex3	129	387	27	317	377.8%	22.1%
ex4	248	744	39	497	535.9%	49.7%
ex5	119	357	23	251	417.4%	42.2%
ex7	144	432	28	308	414.3%	40.3%
s1196	3387	10161	2216	9523	52.8%	6.7%
s1238	3087	9261	2018	8998	53.0%	2.9%
s1423	12708	38124	10153	33116	25.2%	15.1%
s344	168	504	97	454	73.2%	11.0%
s420	152	456	76	340	100.0%	34.1%
s526	189	567	98	482	92.9%	17.6%
s713	903	2709	228	1480	296.1%	83.0%
s838	300	900	148	668	102.7%	34.7%
s953	474	1422	201	1302	135.8%	9.2%

Table 1. ROBDD vs. MLDD in size and performance with Berkeley ordering

Benchmark	ROBDD		MLDD		RATIOS	
	nodes	mem	nodes	mem	nodes	mem
MultiLevel						
alu2	164	492	118	474	39.0%	3.8%
alu4	429	1287	349	1251	22.9%	2.9%
apex6	669	2007	537	2147	24.6%	-6.5%
apex7	480	1440	157	760	205.7%	89.5%
b9	165	495	58	388	184.5%	27.6%
C1355	30460	91380	30043	95622	1.4%	-4.4%
C432	1300	3900	2478	11054	-47.5%	-64.7%
C499	30460	91380	30043	95622	1.4%	-4.4%
c8	100	300	64	238	56.2%	26.1%
C880	6969	20907	4294	14392	62.3%	45.3%
cht	91	273	90	376	1.1%	-27.4%
CM151	17	51	17	52	0.0%	-1.9%
CM152	19	57	16	46	18.8%	23.9%
comp	152	456	66	381	130.3%	19.7%
count	82	246	87	339	-5.7%	-27.4%
DES	9515	28545	8058	28898	18.1%	-1.2%
example2	646	1938	183	1061	253.0%	82.7%
frg1	186	558	41	337	353.7%	65.6%
frg2	1957	5871	790	3809	147.7%	54.1%
k2	1426	4278	617	3796	131.1%	12.7%
lal	94	282	42	232	123.8%	21.6%
Adderfds	502	1506	456	1372	10.1%	9.8%
pair	6032	18096	4283	15485	40.8%	16.9%
pcler8	130	390	64	324	103.1%	20.4%
rot	7069	21207	3708	14185	90.6%	49.5%
sct	53	159	33	187	60.6%	-15.0%
term1	107	321	53	283	101.9%	13.4%
too_large	1113	3339	578	2529	92.6%	32.0%
ttt2	158	474	60	381	163.3%	24.4%
vda	534	1602	289	1648	84.8%	-2.8%
x1	682	2046	122	1004	459.0%	103.8%
x4	682	2046	216	1128	215.7%	81.4%
TwoLevel						
alu4.pla	790	2370	515	2675	53.4%	-11.4%
apex5.pla	1440	4320	881	3935	63.5%	9.8%
clip.pla	170	510	66	276	157.6%	84.8%
e64.pla	732	2196	66	2404	1009.1%	-8.7%
misex2.pla	111	333	32	286	246.9%	16.4%
misex3.pla	760	2280	185	1170	310.8%	94.9%
sao2.pla	133	399	45	289	195.6%	38.1%
vg2.pla	414	1242	59	403	601.7%	208.2%
FSM						
ex1	639	1917	104	1568	514.4%	22.3%
ex2	370	1110	44	711	740.9%	56.1%
ex3	129	387	27	316	377.8%	22.5%
ex4	239	717	36	477	563.9%	50.3%
ex5	108	324	23	249	369.6%	30.1%
ex7	128	384	28	301	357.1%	27.6%
s1196	806	2418	318	1862	153.5%	29.9%
s1238	813	2439	311	1812	161.4%	34.6%
s344	164	492	80	395	105.0%	24.6%
s420	179	537	75	337	138.7%	59.3%
s526	147	441	69	387	113.0%	14.0%
s713	747	2241	173	1110	331.8%	101.9%
s838	293	879	147	665	99.3%	32.2%
s953	498	1494	158	1081	215.2%	38.2%

Table 2. ROBDD vs. MLDD in size and performance after dynamic reordering

necessarily to the violation of some properties of the functions f_i or g_i.

It is not restrictive to assume that the two sets $\{f_i\}, \{g_i\}$ differ because $g_1 \neq f_i, i = 1, \cdots, p$. Since $\{f_i\}, \{g_i\}$ are both decompositions of f, it must be :

$$\overline{f_1 + \cdots + f_p} = \overline{g_1 + \cdots + g_q} \tag{6}$$

or equivalently,

$$f_1 + \cdots + f_p = g_1 + \cdots + g_q \; . \tag{7}$$

Since all functions g_i have disjoint support, it is possible to find an assignment of the variables in $S(g_2), S(g_3), \cdots, S(g_q)$ such that $g_i = 0, i = 2, \cdots, q$. Notice that the variables in $S(g_1)$ have not been assigned any value. Corresponding to this partial assignment, Eq. (7) becomes:

$$f_1^* + \cdots + f_p^* = g_1 \tag{8}$$

In Eq. (8), f_i^* denotes the residue function obtained from f_i with the aforementioned partial assignment.

We need now distinguish several cases, depending on the assumptions on the structure of the left-hand side of Eq. (8).

Case 1). The left-hand side reduces to a constant. Hence, g_1 is a constant, against the assumptions.

Case 2). The left-hand side contains two or more terms. Since these terms must have disjoint support, g_1 is further decomposable, against the assumptions.

Case 3). The left-hand side reduces to a single term. It is not restrictive to assume this term to be f_1^*. If $f_1 = f_1^*$, then we have $g_1 = f_1$, against the assumption that g_1 differs from any f_i. Hence, it must be $f_1^* \neq f_1$, and

$$S(g_1) = S(f_1^*) \subset S(f_1) \; strictly. \tag{9}$$

We now show that also this case leads to a contradiction.

Consider a second assignment, zeroing all functions $f_j, j \neq 1$. Eq. (7) now reduces to

$$f_1 = g_1^* + \cdots + g_q^* \; . \tag{10}$$

By the same reasonings carried out so far, the r.h.s. of Eq. (10) can contain only one term. We now show that this term must be g_1^*. If, by contradiction, $f_1 = g_j^*, j \neq 1$, then by Eq. (9) one would have

$$S(f_1) = S(g_j^*) \supset S(g_1) \tag{11}$$

against the assumption of g_i, g_j being disjoint-support. Hence, it must be $f_1 = g_1^*$. In this case, by reasonings similar to those leading to Eq. (9), we get

$$S(f_1) = S(g_1^*) \subset S(g_1) \;\; strictly \tag{12}$$

which contradicts Eq. (9). Hence, g_1 cannot differ from any f_i, and the first point is proved.

The proof of the second statement follows by applying recursively a D_{NOR} to each of f_i. Since each D is unique, the tree decomposition is also unique and the Theorem is proved. □

Proof of Theorem 2. By contradiction. Suppose we have a function F that is decomposable as $F = (f_1 + f_2)'$ with $S(f_1) \cap S(f_2) = \emptyset$ and such that F' is also decomposable as $F' = (g_1 + g_2)'$ with $S(g_1) \cap S(g_2) = \emptyset$. We have to prove a contradiction in the equivalence:

$$(f_1 + f_2)' = g_1 + g_2 \tag{13}$$

For sake of readability, we define $a = f_1', b = f_2', c = g_1, d = g_2$ and contradict:

$$a \cdot b = c + d \; . \tag{14}$$

We partition the supports of these functions in this way:

$$\begin{aligned} S_{ac} &= S(a) \cap S(c) \\ S_{ad} &= S(a) \cap S(d) \\ S_{bc} &= S(b) \cap S(c) \\ S_{bd} &= S(b) \cap S(d) \end{aligned}$$

Some of the S_{ij} can be empty. In the rest of the proof we show that Eq. (14) implies that the support of at least one of a, b, c, d is empty, against the assumptions.

To this end, we will rewrite Eq. (14) under different partial assignments of the variables in S_{ij}. For instance, by selecting an assignment of $S(a)$ such that $a = 1$, we obtain:

$$b = c_a + d_a \tag{15}$$

where c_a indicates the function obtained by assigning in c the variables of S_{ac} with values satisfying $a = 1$. The support of c_a is then S_{bc}.

Similarly, we can choose another assignment in $S(b)$ so that $b = 1$ and obtain:

$$a = c_b + d_b \ . \tag{16}$$

From Eqs. (15) and (16), we have:

$$c + d = a \cdot b = (c_b + d_b)(c_a + d_a) \ . \tag{17}$$

We now find expressions for c and d. We evaluate d to zero, reducing the above equation to:

$$c = (c_b + d_{bd'})(c_a + d_{ad'}) \ . \tag{18}$$

$d_{ad'}$ is obtained by assigning first the variables in S_{ad} and then those in S_{bd}. Due to the complete assignment, $d_{ad'}$ is a constant (not necessarily 0). Similarly for $d_{bd'}$. So, in reducing the last equation, we face four cases:

1. $d_{ad'} = d_{bd'} = 1$. Then $c = 1$, *i.e.* its support set is empty against the assumptions.
2. $d_{ad'} = 0$ and $d_{bd'} = 1$. Then $c = c_a$.
3. $d_{ad'} = 1$ and $d_{bd'} = 0$. Then $c = c_b$.
4. $d_{ad'} = d_{bd'} = 0$. Then $c = c_a \cdot c_b$.

Repeating the same procedure to Eq. (17) with the evaluation of c to 0, we have the symmetric cases:

1. $c_{ac'} = c_{bc'} = 1$. Then $d = 1$.
2. $c_{ac'} = 0$ and $c_{bc'} = 1$. Then $d = d_a$.
3. $c_{ac'} = 1$ and $c_{bc'} = 0$. Then $d = d_b$.
4. $c_{ac'} = c_{bc'} = 0$. Then $d = d_a \cdot d_b$.

Now we have to prove the contradiction using Eq. (17) for all the possible combinations of these cases.

1. $c = c_a \cdot c_b$ and $d = d_a \cdot d_b$.
$$c + d = (c_b + d_b)(c_a + d_a) = c_a \cdot c_b + d_a \cdot d_b \tag{19}$$
The contradiction becomes evident if, for example, we assign $c_b = 0$, $c_a = 1$ and $d_a = 0$, which leads to $d_b = 0$, *i.e.* $S_{ad} = \emptyset$. A second assignment, $c_b = 1$, $c_a = 0$ and $d_b = 0$, leads to $d_a = 0$, so that also $S_{bd} = \emptyset$. Thus $S_d = S_{ad} \cup S_{bd} = \emptyset$; d would have to be a constant, a contradiction.
2. $c = c_a$ and $d = d_a \cdot d_b$.
$$c + d = (c_b + d_b)(c_a + d_a) = c_a + d_a \cdot d_b \tag{20}$$
Since $c = c_a$ we know that $S_{ac} = \emptyset$ and c_b is 0 or 1. We consider both cases. If $c_b = 0$ the equation above reduces to:
$$d_b(c_a + d_a) = c_a + d_a \cdot d_b$$
and evaluating $d_a = 0$ and $d_b = 0$ we find $c_a = 0$, hence $S_{bc} = \emptyset$, and therefore $S(c) = S_{ac} \cup S_{bc} = \emptyset$.
If, instead, $c_b = 1$ we have:
$$c_a + d_a = c_a + d_a \cdot d_b$$
Assigning $c_a = 0$ and $d_a = 1$ we find $d_b = 1$, *i.e.* $S_{ad} = \emptyset$, and then $S(a) = S_{ad} \cup S_{ac} = \emptyset$, against the assumptions.
3. $c = c_a$ and $d = d_a$. Then, $S_{ac} = \emptyset$, $S_{ad} = \emptyset$ and $S(a) = S_{ac} \cup S_{ad} = \emptyset$, *i.e.* a is a constant.
4. $c = c_a$ and $d = d_b$.
$$c + d = (c_b + d_b)(c_a + d_a) = c_a + d_b \tag{21}$$
and also $S_{ac} = \emptyset$ and $S_{bd} = \emptyset$. Since c_b and d_a are constants, we consider two cases:
$c_b = 1$. Then
$$c_a + d_a = c_a + d_b$$
and evaluating $c_a = 0$ we find that d_b is a constant, so that $S(a) = S_{ad} \cup S_{ac} = \emptyset$, against the assumptions.
If, instead, $c_b = 0$, we have
$$d_b(c_a + d_a) = c_a + d_b \ .$$
Evaluating $d_b = 0$ we find $c_a = 0$. Then $S_{bc} = \emptyset$ and $S(c) = S_{bc} \cup S_{ac} = \emptyset$.

All other situations are resolved by applying the same reasoning as in last cases. □

Proof of Theorem 4. Consider removing a single element, say, f_1, from the set. The new set, $\{f_2, \cdots, f_k\}$, is still a decomposition. It is also maximal, for if any term were further decomposable, then the same term would be decomposable in $\{f_1, \cdots, f_k\}$, and $\{f_1, \cdots, f_k\}$ would not be a D. □

Proof of Theorem 5. We prove only the third result, the other cases being conceptually similar. Clearly, the right-hand side of the third equation is a NOR decomposition. Therefore, the only issue is its maximality. None of $p_1, \cdots, p_h$ can be further decomposed, or else we would contradict the assumption that $p_1, \cdots p_h$ appear in, say, $\mathrm{D_{NOR}}\ (f)$. The only candidate for further decomposition is then $p_{h+1} = [x'(f_1 + ... + f_k)' + x(g_1 + \cdots + g_l)']'$. Suppose, by contradiction, that p_{h+1} has a $\mathrm{D_{OR}}$ $\{z_1, \cdots, z_q\}$ with more than one element. In this case, x appears in the support of only one function z_j, say, z_q. Hence,

$$\begin{aligned} f &= (x'f + xg)_{x=0} = \\ &= (p_1 + \cdots + p_h + z_1 + \cdots + z_{q-1} + z_{q,x=0})' \\ g &= (x'f + xg)_{x=1} = \\ &= (p_1 + \cdots + p_h + z_1 + \cdots + z_{q-1} + z_{q,x=1})' \end{aligned}$$

Since the terms $z_1, \cdots, z_{q-1}$ appear in f and g, they cannot coincide with any of f_i, g_j. But then f and g would have two distinct $\mathrm{D_{NOR}}$ s, already proved impossible. □

Proof of Theorem 6. The right-hand side is a disjoint-support decomposition. Its maximality follows from the impossibility of breaking down x or any term in $\mathrm{D_{NOR}}\ (g)$ into a sum of other terms. □

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, August 1986.

[2] O. Coudert and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. ICCAD*, pages 126–129, November 1990.

[3] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. ICCAD*, pages 6–9, November 1988.

[4] Y. Matsunaga and M. Fujita. Multi-level logic optimization using binary decision diagrams. In *Proc. ICCAD*, pages 556–559, November 1989.

[5] H. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *Proc. ICCAD*, pages 130–133, November 1990.

[6] F. Corno, P. Prinetto, and M. Sonza Reorda. Using symbolic techniques to find the maximum clique in very large sparse graphs. In *Proc. EDAC*, pages 320–324, March 1995.

[7] Y-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proc. DAC*, pages 240–243, June 1992.

[8] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. DAC*, pages 40–45, June 1990.

[9] H. Ochi, K. Yasuoka, and S. Yajima. Bredth-first manipulation of very large binary decision diagrams. In *Proc. DAC*, pages 48–55, November 1993.

[10] K. M. Butler, D. E. Ross, R. Kapur, and M. R. Mercer. Heuristics to compute variable orderings for the efficient manipulation of binary decision diagrams. In *Proc. DAC*, pages 417–420, June 1991.

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

[12] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. ICCAD*, pages 42–47, November 1993.

[13] U. Kebschull, E. Schubert, and W. Rosentiel. Multilevel logic based on functional decision diagrams. In *EuroDAC, Proceedings of the European Conference on Design Automation*, pages 43–47, September 1992.

[14] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient representation and manipulation of of switching functions based on ordered kronecker functional decision diagrams. In *Proc. DAC*, pages 415–419, June 1994.

[15] S.-I. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proc. DAC*, pages 272–277, June 1993.

[16] Y.-T.Lai, M. Pedram, and S. B. K. Vrudhula. Evbdd-based algorithms for ilp, spectral trnasform and function decomposition. *IEEE Trans. on CAD/ICAS*, 13(8):959–975, August 1994.

[17] R. E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In *Proc. ICCAD*, pages 236–243, 1995.

[18] V. Bertacco and M. Damiani. Boolean function representation using parallel-access diagrams. In *Sixth Great Lakes Symposium on VLSI*, March 1996.

[19] F. Mailhot and G. DeMicheli. Algorithms for technology mapping based on binary decision diagrams and on boolean operations. *IEEE Trans. on CAD/ICAS*, pages 599–620, May 1993.

[20] A. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison Wesley, 1974.

[21] S. Yang. Logic synthesis and optimization benchmark user guide, version 3.0. *MCNC*, January 1991.