Advanced Ordering and Manipulation Techniques for Binary Decision Diagrams

Ney Calazans, Qinhai Zhang, Ricardo Jacobi, Bruno Yernaux and Anne-Marie Trullemans

Université Catholique de Louvain - Laboratoire de Microélectronique
3, Place du Levant B-1348 Louvain-la-Neuve Belgium.

Abstract

This paper presents heuristics leading to improved ordering computation for binary decision diagrams (BDDs). An initial step, based on the topology of the network, generates a hierarchical variable ordering. This initial result is further refined by incremental manipulation governed by the stochastic evolution technique. A new property of BDDs is introduced as well, which accelerates commonly used operations. Experimental results are presented.

I. Introduction

Binary Decision Diagrams (BDDs) have been known for a long time as a means to represent the structure of switching functions [1], but only recently their use emerged in the implementation of design automation tools for digital circuits. This is owed primarily to the introduction of an effective canonical form, called Reduced Ordered BDDs (ROBDDs) [2], which provides compact as well as computationally efficient descriptions. ROBDDs are used in logic verification [3], [4], logic synthesis, circuit simulation, etc. Extensions to ROBDDs were suggested to improve their utility. One of these, Modified Binary Decision Diagrams (MBDs) [5], has been used to perform logic operations including the don't care set [6].

Before generating ROBDDs, the co-NP-complete problem of determining the best ordering of the input variables must be solved. Instead of using the exact solution, which is computationally too expensive, heuristics are the usual expedient applied to obtain satisfactory, near-optimum solutions.

The present work proposes a method to generate near-optimum orderings for even very large function descriptions. The method comprises a two-step procedure. The first step applies a set of heuristics to find the ordering of the initial diagram. This initial diagram is later refined using incremental techniques [7] to implement a stochastic evolution (SE) [8] procedure. A similar two-step approach has already been suggested in [5]. However the heuristics here are enhanced. Also, the incremental techniques, coupled with the SE procedure, provides results quite close to the exhaustive search of the best variable ordering, at least for the benchmarks where finding this best ordering was feasible.

Finally, we introduce a new property of MBDs (and ROBDDs) that can be used to reduce the complexity of large functions manipulation. This property allows the treatment of certain clusters of vertices in the graph of an MBD as a single vertex. We call these clusters supernodes.

In the next section we introduce the basic terminology. Next, we describe the new heuristics and the needed background. Then, the incremental techniques and their use in implementing the SE procedure are presented. A subsequent section details the issues connected to supernodes, while the two last sections are dedicated to experimental results and conclusions.

II. Definitions

A **variable** is a symbol representing a single coordinate of the Boolean space. A **literal** is either a variable or its complement. A **single output incompletely specified function** (SOISF) is a triplet ($\mathbf{f_{on}}$, $\mathbf{f_{off}}$, $\mathbf{f_{dc}}$) of completely specified switching functions representing the on-set, the off-set and the don't care set of the function. A **multiple output incompletely specified function** (MOISF) is a set of triplets, where each triplet is a SOISF. A **reduced ordered binary decision diagram** (ROBDD) is a directed acyclic graph (DAG) as defined by Bryant in [2]. A **modified binary decision diagram** (MBD) is an extension of the ROBDD concept [5]. In an MBD, multiple root vertices are allowed, and a third terminal vertex may appear, standing for the don't care value **X**.

III. Initial input variables ordering

A good initial ordering is necessary to avoid the exponential growth of the number of vertices in an MBD. Several heuristics to derive the ordering from a particular network topology have been suggested [3,4,7,12]. A hierarchical strategy, which is derived from the network topology, is presented in this section. This strategy involves a three-step iteration which calculates a node weight, selects an input variable, and eliminates part of the network related with the selected input.

III.1 Preliminaries

In an acyclic network, each net(or node) has different influences on the outputs. The measure of this influence is called **node weight**. The symbol $|\mathrm{Fanin}|_{v_i}$ ($|\mathrm{Fanout}|_{v_i}$) denotes the number of fan-ins (fan-outs) of node v_i. The node weight of each node v_i with regard to output y_l is defined recursively as follows:

$$1).\ DF_{vi}\Big|_{yl} = \begin{cases} 0 & \text{if output node vi} \neq \text{yl} \\ 1 & \text{if output node vi = yl} \end{cases}$$

$$2).\ DF_{vi}\Big|_{yl} = \sum_{k=1}^{|Fanout|_{vi}} \frac{DF_{vj}\big|_{yl}}{|Fanin|_{vj}} \quad \text{if vj is a fan-out of vi}$$

If there is a sub-network with a set of **input nodes** $\{x_1, x_2, \dots x_k\}$ and one **output node** v_p, and if the input nodes are not used as fan-ins of any node out of this sub-network, we say that v_p **covers** completely the input nodes; it is called an intermediate node for these nodes. The intermediate nodes play an important role in finding the variable ordering and also in building the MBD graph since we can directly replace the set of input variables $\{x_1, x_2, \dots x_k\}$ by the intermediate variable v_p. The intermediate variable is a dynamic concept related to the current structure of the network. If an intermediate variable can be obtained directly from the initial network, we call it a **static intermediate variable**. Otherwise we call it a **dynamic intermediate variable**. Dynamic intermediate variables appear depending on the sequence of variable selections.

The **cover degree** (CD) of node v_i with regard to input x_l denotes how much of the signal propagation from the input xl is covered (or received) by the node vi. The cover degree of node v_i with regard to input x_l is defined recursively as follows:

$$1).\ CDvi\Big|_{xl} = \begin{cases} 0 & \text{if input node vi} \neq \text{xl} \\ 1 & \text{if input node vi = xl} \end{cases}$$

$$2).\ CDvi\Big|_{xl} = \sum_{k=1}^{|Fanin|_{vi}} \frac{CDvj\big|_{xl}}{|Fanout|_{vj}} \quad \text{if vj is a fan-in of vi}$$

For instance, in the network shown in Figure 1(a), the cover degrees computed for nodes **m0**, **m1**, **m2** and **out** for the inputs {x1, x2, x3, x4, x5, x6} are [0, 0, 1, 1, 0, 0], [1, 1, 0.5, 0.5, 0, 0], [0, 0, 0.5, 0.5, 1, 1] and [1, 1, 1, 1, 1, 1], respectively. It is easy to see that a node v_p is an intermediate node iff $CD_{vp}|xl = 1$ or $CD_{vp}|xl = 0$ for every input $xl \in \{x1, x2, x3, \dots, xn\}$. Therefore, in Figure 1(a), the node **m0** is a static intermediate node for inputs {x3, x4}, **out** is a static intermediate node for inputs {x1, x2, x3, x4, x5, x6}, but the nodes **m1** and **m2** are not.

III.2 Dynamic Variable Selection

The basic idea of the technique is to select the input variable x_i that has the largest influence on the outputs and to add it to the tail of the variable ordering list under construction. Due to the MBD (ROBDD) structure, choosing a specific place in the ordering for a variable x_i means to partition the function into two sub-functions independent of x_i. Accordingly, the part of the network related to x_i will be eliminated after the variable choice, so as to reflect this characteristic. The node weight and the intermediate variable distributions are dynamically changed during the elimination process. Figure 1(a) to 1(d) illustrates this process for a single output network.

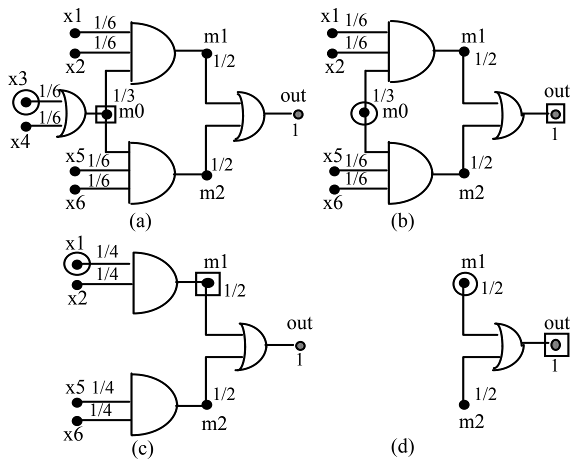

Figure 1. Variable selection.

Table 1. Record of variable selections.

Step	1	2	3	4	5
Anc.	m0	out	m1	m2	out
Desc.	{x3,x4}	{m0}	{x1,x2}	{x5,x6}	{m1,m2}

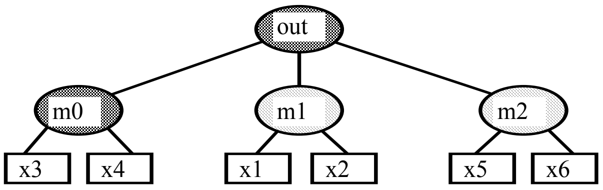

Figure 2. Hierarchical variable ordering.

In the first step, the lowest level intermediate variable, marked with a small box, is **m0**. Suppose that the current input selection, marked with a small circle, is **x3** (1(a)). We observe that two input nodes x_i and x_j have the same node weight if their fan-outs are the same. All the equivalent input variables can be selected simultaneously. The node **x4** is equivalent to **x3**, thus selecting **x4** directly is a good choice. After elimination of **x3** and **x4**, node **m0** becomes a 0-fan-in node and will be selected next in the input variable list. Now, the lowest level intermediate variable is **out**, and the current selection is **m0**(1(b)). After deleting **m0**, the node weight distribution is changed. Also, **m1** and **m2** become intermediate variables. The above select-eliminate-reassign process is repeated until the network vanishes. The sequence selection as well as the hierarchical ancestor-descendant relationship is kept for further use. For this example, the records are shown in Table I and a hierarchical variable ordering tree is shown in Figure 2. Here, **m0** and **out** are static intermediate variables, and **m1**, **m2** are dynamic intermediate variables. If the variable ordering is required, we simply decode these records and the final variable ordering {x3, x4, x1,x2, x5, x6} is deduced.

III.3 Implementation of MBDs

MBDs are implemented using the algorithms described in [10]. In order to reduce the size of the MBD, we add one extension to our implementation: some internal parts of the MBD are replaced by static intermediate variables, and an additional complete BDD is used to present the subfunction associated to each static intermediate variable. This technique is illustrated in Figure 3. Assume that p0 is a duplicated subgraph in the MBD and all the variables in p0 are covered by a static intermediate variable VS(this condition results from the structure of the hierarchical order tree). If the static intermediate variable appears k times in the MBD and if the size of the corresponding BDD is sp, [k * sp - (sp + k)] vertices can be saved as long as we substitute the covered inputs by the intermediate variable VS.

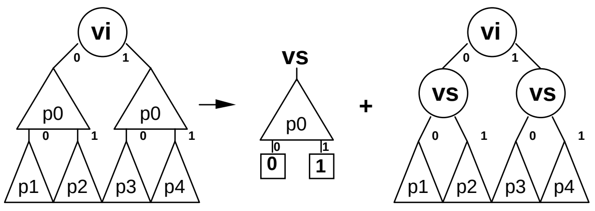

Figure 3. Static Intermediate Variable.

The use of static intermediate variables into the MBD destroys the canonical form, since these are only extracted from a particular network. However, for some applications, e.g. test pattern generation, this hierarchical MBDs can be used more efficiently than the flattened version.

IV. MBD compaction using SE

IV.1 Stochastic Evolution

Stochastic evolution (SE), a technique dedicated to combinatorial optimization, was proposed by Saad and Rao in [8]. The method is similar to Simulated Annealing (SA). The main difference is that SE accepts initially only positive gains. Hill climbing starts only when a local minimum is reached. This approach allows SE to converge faster than SA with similar or better results [8].

The basic idea is to seek a global minimum of a cost function defined over a discrete domain D, called **state space**. Each **state** S is a mapping of a set of **movable elements** M into a set of **locations** L: S: M -> L. A new state S' is generated by **moving** some elements in S. A move m can be **simple** or **composed** and must generate a unique new state S'. The gain of a move m is: **gain**(m) = **cost**(S) - **cost**(S'). Each move is accepted if the gain > random(-p), where p is the parameter that allows negative gains in order to perform hill climbing. R controls the number of iterations, which is an estimation of the time needed to improve the current solution. Each time a better solution is found, the counter is decremented by R, providing more steps to the SE algorithm.

The generation and acceptation of new states are done by the function **perturb_SE**. The **update** function modifies the control parameter p according to some heuristics depending on the kind of problem. These functions are described in subsection IV.3.

IV.2 Incremental Manipulation

The incremental manipulation of MBDs is a technique that allows to generate new orderings by exchanging pairs of adjacent variables v_i, v_{i+1} in the ordering while updating the MBD accordingly. The operation is divided into two steps: first, update the subgraphs defined by the vertices with indices i, i+1 to reflect the new ordering; second, delete the redundant vertices that can arise from the previous step. A more detailed description of these functions can be found in [7], together with two greedy heuristics for MBD compaction.

IV.3 Compaction using SE

The MBD ordering problem can be modelled as a permutation problem. We choose M={x_0,..,x_{n-1}}, the set of input variables, and L = {1,..., n}. The state space is the set of all permutations of M. A state is a bijection S: M->L, i.e., a permutation of the input variables. A move in the state space is generated by swapping two consecutive indexes. Let S = <x_0,..., x_i, x_{i+1},..., x_{n-1}>, be a given ordering of M. S' is a neighbour of S if S' = <x_0,..., x_{i+1}, x_i,..., x_{n-1}> for some i in {0, ..., n-2}. The cost function **cost**(S) is the number of vertices or size of the MBD. R and p0 where determined experimentally. R = 15 and p0 = 3 have produced the best results. The main function is **MBD_SE**, sketched in figure 4.

The function **perturb_SE** (Figure 5) performs a composed move. Each simple move is accepted if the gain is positive or if it is smaller than a random negative value between 0 and -p, the controlling value. Variables can thus be arbitrarily displaced in the ordering. A potential problem with the SE algorithm is the lack of large hill climbing steps. The solution can cycle around a local minimum if the **valley** is too deep. There is no mechanism to prevent the process to return to the same local minimum after some steps. To avoid this problem a random state is generated if the best solution is not improved after R steps. R is chosen as the expected number of iterations to improve the best solution.

```
function MBD_SE (mbd, costf) : state;
begin
     initialize data;
     while (r < R) begin
        $C_{old}$ = costf (mbd);
        mbd = perturb _SE(mbd,p,costf);
        $C_{cur}$ = cost (S);
        update (p, $C_{cur}$, $C_{old}$);
        if ($C_{cur}$ < cost (mbd$_{best}$)) then
           begin
                mbd$_{best}$ = mbd;
                r = r - R;
           end
        else r = r + 1;
     end;
return (mbd$_{best}$);
end;
```

Figure 4. MBD_SE algorithm.

```
function perturb_SE (mbd, p, costf): mbd;
begin
if solution not improved after R steps then
     return(random order);
foreach index i
     swap indices i, i+1.
     Accept the swapping if gain > random(-p),  otherwise
swap back indices i, i+1.
return (mbd);
end;
```

Figure 5. Perturb_SE algorithm.

```
function update (p, $C_{cur}$, $C_{old}$, Queue)
begin
     compute the average of last four values
     if ((Ccur - average) < Threshold)
       then return(p + 1,S)   /* increase p */
       else return (Pse,S); /* else return default */
end;
```

Figure 6. Update algorithm.

The function **perturb_SE** generates states that oscillate around a local minimum if when updating p as in [8]. To circumvent this problem we update p as shown in figure 6. The parameter threshold is set to 3 and graduates the amount of oscillation allowed.

V. Supernodes

To reduce the complexity of MBD manipulation, we introduce the concept of supernodes. Supernodes are a means to group together locally connected vertices in an MBD so as to form clusters that can then be treated as a single vertex of the MBD.

V.1 Definition

Informally, we define a supernode as a subgraph of an MBD containing two special non-terminal vertices, unique but not necessarily distinct, called **initial** and **final**. Any arc of the MBD beginning in the initial vertex must pass through the final vertex before reaching a terminal, and the longest arc in the supernode must visit every vertex in the subgraph. A more formal definition follows.

Definition - A **supernode** S of an MBD M =(V, E), where V is a set of vertices and E is a set of edges (a set of ordered pairs of vertices), is a subgraph of M recursively defined by the following axioms:

(a) Every single vertex **s** in M determines a supernode $S_1 = (V_1, E_1)$, where V_1 is the set containing **s** as single element, and E_1 is the empty set, **s** being both initial and final;

(b) Let $S_1 = (V_1, E_1)$, $S_2 = (V_2, E_2)$ be two component supernodes of M, and $p \in V_1$, $q \in V_2$, and $r \in V - V_1 - V_2$, three vertices, where p and q are either initial and final or final and initial vertices, respectively. Thus, if $\exists$a, b, c$\in E - E_1 - E_2$, such that

either c = (p, q) or c = (q, p),

and a = (p, r), b = (q, r),

then $S_3 = (V_1 \cup V_2, E_1 \cup E_2 \cup$

{ c | c = (p, q), if p is final and q is initial

or c = (q, p), otherwise}) is a supernode.

The initial and final vertices of this new supernode are either the initial vertex of V_1 and the final vertex of V_2, if p is final and q is initial or the initial vertex of V_2 and the final vertex of V_1, otherwise.

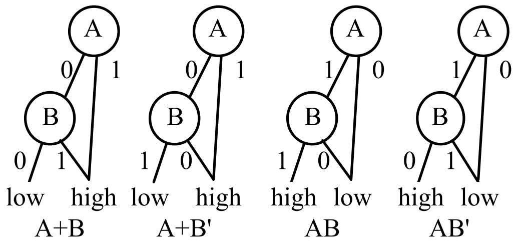

Figure 7. Simple supernode cases.

The above definition says that a supernode S comprises a finite set of vertices of M, each of these vertices connected to another supernode completely contained in S (or to a terminal vertex), either as a source or as a sink, such that they share the sink vertex for one of their output edges. Note that neither the common sink vertex nor the edges pointing to it need be part of S. Figure 7 shows all configurations of 2-vertices supernodes, and the associated factored forms. The general case can be inferred from these.

Supernode S_2 **is contained** in a supernode S_1 if $V_2 \subseteq V_1$. If $V_2 \subset V_1$, we say that S_2 **is properly contained** in S_1 or that S_2 is a **proper** supernode of S_1. The **size** of a supernode is the cardinality of its set of vertices.

Another useful concept is the **level** of a supernode S = (V, E), a positive integer such that: (a) If S is a single-vertex supernode, it has level 0, and (b) If this is not the case, we determine the level of S using a recursive procedure: partition V into two sets V_1 and V_2, such that V_1 is a singleton containing the initial vertex of S and V_2 contains all other vertices of S. This procedure determines two supernodes: $S_1 = (V_1, E_1)$, $S_2 = (V_2, E_2)$, by the supernode definition axioms. If we know the level of S_2 to be **n-1**, the level of S will be **n-1**, if there is a level-**0** supernode in S_2 that forms a supernode with S_1. Otherwise, the level of S is **n**.

Special supernodes can be defined. A supernode is **maximum** if no supernode in the MBD properly contains it. A supernode is **maximal** if no supernode with the same initial vertex in the MBD properly contains it. Let M = (V, E) be an MBD and S = (V_1, E_1) be a supernode of M. Let also **c** = (**p**, **q**) be an edge of E - E_1, i. e. an edge of M but not of S. S is a **closed** supernode iff, for all **c** such that **q** $\in V_1$, **q** is always the same vertex and, at the same time, no vertex but **q** is the initial vertex of any SOISF of the MOISF represented by M. This implies that a closed supernode is one which interacts the least possible with the surrounding vertices of M. Stated otherwise, if S is a closed supernode, every edge going from a vertex not in S to a vertex of S must have as sink the initial vertex of S. The last requirement prevents that some internal vertex of a closed supernode be some root of M.

V.2 Properties

The main property of supernodes is that any closed maximum supernode behaves like a single vertex in the MBD. Yet, any supernode is connected to the rest of the MBD through exactly two output edges, called **low** and **high** of S, although these are not really part of S. These edges coincide with the outputs edges of the final vertex of S. In practice, the low and high of S may not correspond to the low and high edges of the final vertex of S. Figure 7 shows how we define low and high for all cases of 2-vertices supernodes. Again, inference from these lead to the general case.

Supernodes contain neither terminal vertices nor more than one vertex associated with the same variable in its set of vertices. Also, there is exactly one arc in a supernode which visits every vertex of V. Second, any supernode S can be represented by a factored form containing only as many literals as there are vertices in V. This provides better ways for doing factorization and decomposition of MBDs than the approach of using the Shannon expansion. This is illustrateed in Figure 8. Using the Shannon expansion approach, the factored forms extracted from this MBD would be

f=A'+A[B'[C'F'G'+CD'E'F'G']+BD'E'F'G']

However, if we preprocess the MBD collecting vertices into supernodes, we can factorize better. Using algorithms to find maximal supernodes, we end up with four of them: {A}, {B, C}, {F, G} and {D, E, F, G}. The first is a level-**0** supernode, while the others are level-**1**. The resulting factorization gives:

f = A'+A[B'C'+D'E'F'G'][B+C+F'G']

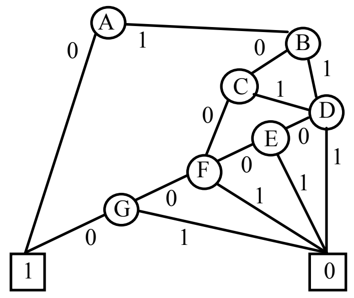

Figure 8. MBD for function f.

Here, even better results can be obtained if we restrict attention to closed supernodes only. The factorization in this case would be obtained from a single level-3 supernode with all vertices of the MBD, and the associated expression would be

f = A'+F'G'[B'C'+D'E']

The intermediate results obtained for this last case are as follows. Three closed disjoint level-1 supernodes are found in a first step: {B, C}, {D, E}, and {F, G}. After this, two closed level-2 supernodes are successively found: {B, C, D, E} and {B, C, D, E, F, G}. Finally, the level-3 supernode comprising all vertices in the ROBDD is obtained from the last, 6-vertex supernode.

VI. Experimental results

Tables 2 and 3 compare our method (column Hier) to others available in the references. Table 2 shows small to medium examples, and the times reported in it refer to a Lisp prototype running on a Macintosh II. Table 3 displays the ISCAS benchmarks results. These are the first results we obtained with our C implementation, on a DEC5000/200 workstation. Stochastic evolution algorithms are not yet available in this new version. Both tables show the time needed to build the MBD with the hierarchical ordering (column MBD t). Table 2 includes also the time needed to run the SE algorithm on the MBD, while Table3 contains the time to compute the hierarchical ordering. Sizes of MBDs are shown in number of vertices, and times in seconds. The SE column in Table 2 indicates the final size of the MBD after running the SE algorithm (absolute minimum sizes in parentheses, when available).

Some conclusions can already be drawn from these data. First, for small examples, our heuristics offers the best results in most cases, with significant gains appearing very often. For large examples, our results are superior to those presented in [10] and similar to the data in [9]. Second, SE can enhance the results of the best heuristics, at the expense of added computation. Reductions in the run time of SE can be obtained by fine-tuning the parameters of the algorithm, but this can still be quite costly. For larger examples, the prediction capacity of any heuristics is weakened. The heuristics used in [4], e.g. which is the same used in [9], is seen to be quite inferior to ours in Table 2. For larger networks, our heuristics, as well as the one in [9] are quite irregular, as seen in Table 3. The heuristics in [10] are poorer than both.

VII. Conclusions

We have presented a method to generate good orderings for BDDs. The first part of the method, hierarchical ordering, is useful in any application of BDDs, since it is very fast (C implementation, times to compute the hierarchical ordering are most often around 1 second) and gives better results when compared with available methods. The second part, stochastic evolution, is helpful when the need to minimize the BDD is more important than the time to do it. For example, the application of SE when doing verification can be too expensive, since SE may multiply the MBD building time by a factor of 10 or 100. On the other hand, if we are applying BDDs to dual cascode voltage switch synthesis (DCVS), reducing the final size of the graph is the main issue. The added computation is then directly associated with a better synthesis, and longer building times can be afforded. For intermediate cases, the faster but less performing incremental techniques described in [7] may be used.

Supernodes were introduced to identify special clusters of vertices in BDDs. Their goal is to reduce the manipulation of these graphs, but their applicability must be further investigated.

VIII. References

[1] C. Y. Lee. *"Representation of Switching Circuits by Binary Decision Programs", BSTJ,* No. 38 (July 1959), pp. 985-999.

[2] R. E. Bryant. "Graph-Based Algorithm for Boolean Function Manipulation". *IEEE Trans. on Computers,* vol. C-35, no. 8, Aug. 1986.

[3] M. Fujita, H. Fujisawa, N. Kawato. "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams", *ICCAD,* 1988.

[4] S. Malik, A. R. Wang, R. K. Brayton and A. Sangiovanni-Vincentelli. "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment". *ICCAD,* 1988.

[5] N. Calazans, R. Jacobi, Q. Zhang and C. Trullemans. "Improving Binary Decision Diagrams Through Incremental Reduction and Improved Heuristics". *CICC*, 1991.

[6] R. Jacobi, A-M Trullemans. "Generating Prime and Irredundant Covers for Binary Decision Diagrams". *EDAC*, 1992.

[7] R. Jacobi, N. Calazans and C. Trullemans. "Incremental Reduction of Binary Decision Diagrams". *ISCAS*, 1991.

[8] Y. Saab & B. Rao. "Combinational Optimization by Stochastic Evolution". *IEEE Trans. on CAD*, Vol. 10, No 4, April 1991.

[9] Karl S. Brace, Richard L. Rudell, Randal E. Bryant "Efficient Implementation of a BDD Package". 27th DAC, 1990.

[10] S. Minato, N. Ishiura, S. Yajima "Shared Binary Decision Diagrams with Attributed Edges for Efficient Boolean Function Manipulation". 27th DAC,1990.

Table 2. MBD size comparison (Running on Macintosh II - LISP).

Benchmark data			Ordering Data (vertices)				Size (vertices)	Time Data	
Name	Inp	Out	[4]	[3]	[10]	Hier	SE	MBD t(s)	SE t(s)
5xp1	7	10	96	92	75	71	70 (70)	11.23	95.0
Alu3	10	8	124	148	75	70	65	8.08	74.8
Bw	5	28	115	120	133	126	107 (107)	16.44	230.4
Duke2	22	29	873	685	495	463	370	81.15	4322.6
Misex2	25	18	133	147	150	130	91	5.60	885.8
P1	8	16	260	264	252	201	193 (193)	26.52	344.0
Signet	39	8	4496	3340	4788	2719	1690	440.40	9 hours
Sn181	14	8	1316	847	942	795	773	51.14	3330.8
X9dn	27	7	430	284	207	117	91	11.03	416.0

Table 3. MBD size for ISCAS benchmarks (Running on DEC 5000/200 - C).

Benchmark data			Ordering Data (vertices)			Time Data	
Name	Inp	Out	[9]	[10]	Hier	Ord t(s)	MBD t(s)
C432	36	7	30200	103954	30520	0.45	54.93
C499	41	32	49786	36934	44165	1.65	52.63
C880	60	26	7655	30899	7490	1.02	4.82
C1355	41	32	39858	119412	56112	1.64	63.57
C1908	33	25	12463	39517	21895	2.26	24.10
C5315	178	123	32193	41538	4433	4.76	4.62