published in *Proceedings of the Int. Symposium on Representations and Methodology of Future Computing Technologies*, (RM2003), Trier (Germany) March 10-11, 2003, pp. 135–144

Variable Reordering for ⊕-OBDDs

Christoph Meinel
FB IV Informatik
Universität Trier
meinel@uni.trier.de

Harald Sack
Institut für Informatik
Universität Jena
sack@minet.uni-jena.de

Abstract

Ordered Binary Decision Diagrams (OBDDs) have already proved useful in the process of electronic design automation. Due to limitations of the descriptive power of OBDDs more general models of Binary Decision Diagrams have been studied. In this paper, ⊕-OBDDs as a true extension of the OBDD data structure are addressed. As for OBDDs, the most important factor influencing the representation size of ⊕-OBDDs is the chosen order of the input variables. Adaptive variable reordering heuristics as sifting for OBDDs can be adapted to ⊕-OBDDs, while some important restrictions arise from the property of being a non canonical data structure. The viability and the efficiency of the adapted sifting algorithm for ⊕-OBDDs is shown by symbolic simulation of standard benchmarks.

1. Introduction

A major problem in the computer aided design of digital circuits is the choice of a suitable representation of the circuit functionality for the computer's internal use. A concise representation, which simultaneously provides the possibility of fast manipulation is very important for all problems given in terms of switching functions. During the last decade, Ordered Binary Decision Diagrams (OBDDs) have proved to be well qualified for this purpose (for an overview see [14]).

But, the descriptive power of OBDDs is limited, due to their property of being a canonical representation for Boolean functions. On the one hand, this important quality is responsible for the nice algorithmic behavior of OBDDs. But, on the other hand, the OBDD representation for most Boolean functions must be of exponential size w.r.t. the number of input variables, and not every Boolean function of practical importance can be represented efficiently. E.g., the OBDD-representations of the *multiplication* or the *hidden weighted bit function* are always of exponential OBDD-size [4] independent of the chosen order of the input variables. For this reason, generalizations of the OBDD data structure have been studied.

In this paper we address ⊕-OBDDs (also known as Mod2-OBDDs), a true extension of OBDDs [6]. ⊕-OBDDs are more, sometimes even exponentially more, space-efficient than OBDDs are. They preserve the algorithmic properties of OBDDs: important operations as *apply, quantification*, and *composition* have the same complexity as in the case of OBDDs. Even better, the Boolean functions *exclusive or* (XOR) and *logical equivalence* (EQU) can be performed in constant time. However, ⊕-OBDDs do not provide a canonical representation of Boolean functions and therefore, an equivalence test can only be applied in practical applications, if probabilistic techniques are deployed [6].

As for OBDDs, an important factor for the representation size of ⊕-OBDDs is the chosen order of the input variables. Besides number and placement of ⊕-nodes in ⊕-OBDDs [12], this factor is often responsible for an exponential blowup in representation size and has to be considered carefully.

In this paper we show, how to adapt the sifting-algorithm, a well known OBDD variable reordering heuristic, to ⊕-OBDDs. But, the main feature of ⊕-OBDDs that is responsible for its advantages in representation size, turns out to be the main obstacle for further improvement. With the property of being non canonical the application of powerful heuristics gets much more difficult. Because there is no unique representation for ⊕-OBDDs, search heuristics have almost no feasible way to return to a previous, advantageous representation, if important properties as the variable order are changed in the course of the heuristic. We show, how to bypass this problem by restricting the number of optimization steps to be applied. For giving proof about the efficiency of the adapted sifting heuristic, it is applied to symbolic simulation of a set of standard benchmarks [9].

The paper is structured as follows: In Section 2, we recall basic definitions concerning ⊕-OBDDs. Section 3 covers basic manipulation algorithms for

$\oplus$-OBDDs. Section 4 introduces variable exchange algorithms and shows, how to restrict optimization rules to maintain the symmetry of the applied operations for $\oplus$-OBDDs. Section 5 concludes with a discussion of achieved experimental results.

2. $\oplus$-OBDDs - an Overview

Definition of the Data Structure

A $\oplus$-OBDD P over a set $X_n = \{x_1, \ldots, x_n\}$ of Boolean variables is a directed acyclic connected graph $P = (V, E)$. V is the set of nodes, consisting of non-terminal nodes with out-degree 2, and of terminal nodes with out-degree 0. There is a distinguished non-terminal node, the *root*, which, as only node, has the in-degree 0. The two terminal nodes with no outgoing arcs are labeled with the Boolean constants 0 and 1. The remaining nodes are either labeled with Boolean variables $x_i \in X_n$ (*branching nodes*), or with the binary Boolean operator XOR (*$\oplus$-nodes*). On each path, every variable must occur at most once. In the following, let $l(v)$ denote the label of the node $v \in V$ and $|P|$ the number of non terminal nodes of P.

$E \subseteq V \times V$ denotes the set of edges. The two edges starting in a branching node v are labeled with 0 and 1. The *0(1)-successor* of node v is denoted by $v_0(v_1)$. There is a permutation π, which defines an order $x_{\pi(1)} < x_{\pi(2)} < \ldots < x_{\pi(n)}$ on the set of input variables. If w is a successor of v in P with $l(v), l(w) \in X_n$, then $l(v) < l(w)$ according to π must hold.

Note that since the $\oplus$-operation is symmetric, the outgoing edges of $\oplus$-nodes do not have to be labeled separately. The function f_P associated with the $\oplus$-OBDD P is determined in the following way: For a given input assignment $a = (a_1, \ldots, a_n) \in \{0,1\}^n$, the Boolean values assigned to the leaf nodes are extended to all other nodes of P as follows:

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

The function $f_P(a)$ computes to the value associated with the source of P.

To achieve a more compact representation, we may furthermore consider the use of complemented edges [1, 10]. $\oplus$-OBDDs are also a generalization of Kronecker Functional Decision Diagrams (KFDDs) or pseudo Kronecker Functional Decision Diagrams (pKFDDs), but, they provide a more compact representation than KFDDs or pKFDDs do [6].

$\oplus$-OBDDs do not provide a canonical representation of Boolean functions, i.e. there might be several different representations $P_f^1, \ldots P_f^k$, $k \in \mathbb{N}$ for the same Boolean function f. Thus, testing the equivalence of two $\oplus$-OBDDs becomes a rather difficult and important task.

Probabilistic Equivalence Test

Since a deterministic equivalence test for $\oplus$-OBDDs requires runtime $O(|P|^3)$ [17], for practical applications a probabilistic equivalence test as proposed in [5] is chosen, which requires only linearly many arithmetic operations in the number of variables. The equivalence test is based on a probabilistic equivalence test for *read-once branching programs* (BP1), originally introduced in [2] and further refined in [8]. Equivalence of two $\oplus$-OBDDs is determined probabilistically, after an algebraic transformation of the $\oplus$-OBDDs in terms of polynomials over a finite field [12].

Reduction Rules for $\oplus$-OBDDs

The reduction rules that are already known for OBDDs and, if exhaustively applied, guarantee a canonical representation for OBDDs, have to be extended for $\oplus$-OBDDs. Here, these reduction rules serve only for a reduction in size, but they are not able to provide canonicity for $\oplus$-OBDDs. In addition to the standard reduction rules for OBDDs that can also be applied to the branching nodes of a $\oplus$-OBDD, reductions for $\oplus$-nodes have to be considered according to the node's functionality[12].

Synthesis of $\oplus$-OBDDs

For the synthesis of $\oplus$-OBDDs, the already known *ITE*-algorithm [3] for OBDDs can easily be extended. To connect two Boolean functions f and g given in terms of OBDDs with an arbitrary Boolean operation $\otimes$ the *Boole-/Shannon decomposition (BS)* w.r.t. variable x_i is applied:

$$f \otimes g = x_i(f|_{x_i} \otimes g|_{x_i}) + \overline{x_i}(f|_{\overline{x_i}} \otimes g|_{\overline{x_i}}).$$

f_{x_i} denotes the positive cofactor of the Boolean function f, where the input variable x_i is substituted with the constant $x_i = 1$. $f_{\overline{x_i}}$ denotes the negative cofactor of f, respectively. The composition of the cofactors can be computed recursively. For efficiency reasons all Boolean operations are mapped to a single operation, the so called *If-Then-Else operator (ITE)* [3].

$$ITE(x, y, z) = x \cdot y + \overline{x} \cdot z$$

For computing the synthesis of functions f, g, h represented as OBDDs, ITE is called recursively w.r.t.

the top variable x_i of the concerned OBDDs.

$$ITE(f,g,h) = (x_i, \quad ITE(f|_{x_i}, g|_{x_i}, h|_{x_i}), \\ ITE(f|_{\overline{x_i}}, g|_{\overline{x_i}}, h|_{\overline{x_i}}))$$

Introducing new $\oplus$-nodes

For $\oplus$-OBDDs, as an extension for computing $f \oplus g$ or $f \equiv g$, a new $\oplus$-node will be created and connected to f and g, where $f \equiv g = \overline{f \oplus g}$. In all other cases, the regular *ITE*-algorithm is applied for $\oplus$-OBDDs with an adapted cofactor creation algorithm [12].

For symbolic simulation, if the circuit to be represented does not contain any XOR(EQU) gate, a function decomposition has to be chosen that is able to introduce new $\oplus$-nodes into the $\oplus$-OBDD, e.g. the positive or negative Davio expansion (pDE/nDE), also referred to as Reed-Muller expansion [15, 16]. The synthesis algorithm for $\oplus$-OBDDs based on pDE-/nDE is denoted as APPLY-$\oplus$-algorithm.

Improving $\oplus$-OBDD representation size

The most important factors influencing the size of the $\oplus$-OBDD representation are:

- the number of applied $\oplus$-nodes,
- the position of $\oplus$-nodes in the $\oplus$-OBDD, and
- the chosen order of the input variables.

Heuristics for insertion and positioning of $\oplus$-nodes into $\oplus$-OBDDs have been proposed [13]. Improving the given $\oplus$-OBDD representation size can be achieved by moving existing $\oplus$-nodes through the $\oplus$-OBDD graph. Heuristics for repositioning $\oplus$-nodes are based on a simple swap-algorithm that performs an exchange of $\oplus$-nodes with adjacent branching nodes [13].

As for OBDDs, $\oplus$-OBDD optimization is rather difficult, because there is no other way to judge the quality of $\oplus$-OBDD parameters, but constructing the entire $\oplus$-OBDD according to the chosen parameters. For badly chosen parameters, $\oplus$-OBDDs might grow exponentially.

3. Adjusting $\oplus$-OBDD Variable Order

As shown in [12], algorithmic efficiency requires the introduction of meta-$\oplus$-nodes. Adjacent $\oplus$-nodes are merged to so called meta-nodes, representing an XOR-operation applied to more than just two input operands. Meta-$\oplus$-nodes can be applied to an arbitrary number of input operands. By using Meta-$\oplus$-nodes, algorithms for $\oplus$-OBDD manipulation become less complicated.

The basic operation for adjusting the variable order of a $\oplus$-OBDD is the **swap**-operation. It exchanges two adjacent variables in the variable order while restructuring the $\oplus$-OBDD accordingly.

For $\oplus$-OBDDs, $\oplus$-nodes have to be taken into account, when changing the variable order. The straightforward approach of moving all $\oplus$-nodes of the two adjacent variables to be exchanged into levels below and above the affected levels is not feasible, because the operation is rather time expensive and large memory peak sizes might prevent the algorithm from its successful application. Therefore, for exchanging two adjacent variables, a new algorithm was designed that is able to keep $\oplus$-nodes in place, if they are located between those variables, and to swap the variables around the $\oplus$-node.

Meta-$\oplus$-nodes enable an efficient implementation of this algorithm. The usage of meta-$\oplus$-nodes guarantees that between any two branching nodes there must be at most one single meta-$\oplus$-node. Because of that rule, we have achieved a standardized setting for adapting the variable exchange procedure of OBDDs for $\oplus$-OBDDs:

Let the variables to be exchanged be x_i and x_{i+1}. The $\oplus$-OBDD under consideration has the root node v labeled with x_i, and the two successors v_{x_i} and $v_{\overline{x_i}}$. If v_{x_i} and $v_{\overline{x_i}}$ are branching nodes, then the regular variable exchange procedure for OBDDs takes place.

Now, let us consider that v_{x_i} is a $\oplus$-node that is labeled with a reference to the variable x_{i+1}, i.e. the successor of v_{x_i} that is first w.r.t. the variable order is labeled with x_{i+1}. Let n_1 be the number of successors of v_{x_i}, and let $v_{x_i 1}, \ldots, v_{x_i n_1}$ be the successors of v_{x_i}.

For the exchange, v is simply relabeled with the variable x_{i+1} and connected to two successing $\oplus$-nodes $v_{x_{i+1}}$ and $v_{\overline{x_{i+1}}}$ that have to be newly created. The number of successors of $v_{x_{i+1}}$ and $v_{\overline{x_{i+1}}}$ is computed to be the maximum of the number of successors of v_{x_i} and $v_{\overline{x_i}}$ that we will denote as $n_{max} = \max(n_1, n_2)$.

The successor nodes of $v_{x_{i+1}}$ and $v_{\overline{x_{i+1}}}$ that are denoted with $v^1_{x_{i+1}}, \ldots, v^{n_{max}}_{x_{i+1}}$ and $v^1_{\overline{x_{i+1}}}, \ldots, v^{n_{max}}_{\overline{x_{i+1}}}$, will be new branching nodes that are labeled with the variable x_i.

A successor node $v^i_{x_{i+1}}$, $i \leq \min(n_1, n_2)$, if the original successor $v^i_{x_i}$ is labeled with x_{i+1}, will have $(v^i_{x_i})_{x_{i+1}}$ as a left successor and $(v^i_{\overline{x_i}})_{x_{i+1}}$ as a right successor. Otherwise, if the original successor $v^i_{x_i}$ is not labeled with x_{i+1}, then the left successor will be $v^i_{x_i}$ itself. The same holds for $v^i_{\overline{x_i}}$, respectively.

For $n_1 < i \leq n_{max}$ the left successor will be the 0-sink. Otherwise, if $n_1 > n_2$, then, for $n_2 < i \leq$

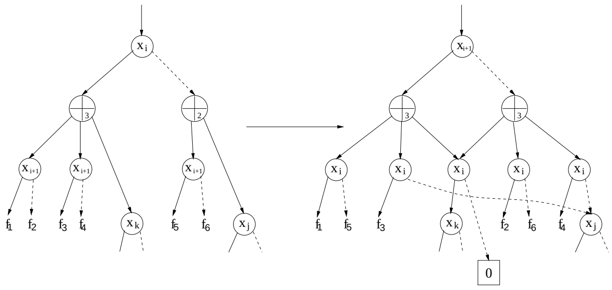

Figure 1: Example for Swap-In-Place Operation.

n_{max} the right successor will be the 0-sink.
If one of the two nodes v_{x_i} and $v_{\overline{x_i}}$ is a branching node, while the other is a meta-$\oplus$-node, we proceed in the same way, while considering the number of successors of the branching node as $n_i = 2$. See Fig. 1 for an illustrating examples and Fig. 2 for a simplified outline of the swap-in-place algorithm in pseudo code.
The algorithm given in Fig. 2 only gives a short outline of the idea behind the swap algorithm. In fact, the *swap-in-place* algorithm for $\oplus$-OBDDs differs from the variable exchange algorithm for OBDDs in the sense that several special cases have to be considered for the implementation.
One of the cases to be considered is the merging of adjacent $\oplus$-nodes that might ocur after a swap-in-place operation. In Fig. 4 an example for that situation is given: After the successfull swap-in-place operation the node $\mathrm{v}^1_{x_i}$ is retrieved from the cache memory instead of creating a new node with the same functionality, but having a branching node labeled with x_i at the top. the two $\oplus$-nodes have to be merged, because otherwise the swap-in-place algorithm could not be applied and possible reductions could be missed.
The reductions that can be applied to the $\oplus$-OBDD after a variable exchange are responsible for the decrease in representation size. Thus, it is mandatory to apply all possible reductions to achieve the smallest size. But, on the other hand, this might lead to situations that are not reversible anymore and thus, not symmetric.
Consider the situation given in Fig. 5. In the $\oplus$-OBDD P the two variables x_i and x_{i+1} are to be exchanged, while the top node v has only one successor being a $\oplus$-node labeled with x_{i+1} (1). After the first variable exchange one of the two $\oplus$-OBDDs P'_1 (2a) or P'_2 (2b) is computed. Note that both $\oplus$-OBDDs represent the same Boolean function, but have a different structure. This is due to the fact that in the two cases, the order of the

Input: $\oplus$-OBDD P, with root node v, $l(v) = x_i$.
Output: $\oplus$-OBDD P' with variables x_i and x_{i+1} exchanged.

```
swap_in_place(v) {
  v_{x_i} = v.1-succ; v_{~x_i} = v.0-succ;
  if ( v_{x_i} and v_{~x_i} are both branching nodes) {
    v = obdd_swap_regular(v);
  } else {
    n_max = max(v_{x_i}.nsucc, v_{~x_i}.nsucc);
    prepare empty ⊕-nodes v_{x_{i+1}}, v_{~x_{i+1}}
    for k = 1 to n_max do {
      if ( k < v_{x_i}.nsucc ) {
        if ( l(v^k_{x_i}) == x_{i+1} ) {
          v^k_{x_{i+1}}.1-succ = v^k_{x_i}.1-succ;
          v^k_{~x_{i+1}}.1-succ = v^k_{x_i}.0-succ;
        } else {
          v^k_{x_{i+1}}.1-succ = v^k_{x_i};
          v^k_{~x_{i+1}}.1-succ = v^k_{x_i};
        }
      } else {
        v^k_{x_{i+1}}.1-succ = 0-sink;
        v^k_{~x_{i+1}}.1-succ = 0-sink;
      }
      if ( k < v_{~x_i}.nsucc ) {
        if ( l(v^k_{~x_i}) == x_{i+1} ) {
          v^k_{x_{i+1}}.0-succ = v^k_{~x_i}.1-succ;
          v^k_{~x_{i+1}}.0-succ = v^k_{~x_i}.0-succ;
        } else {
          v^k_{x_{i+1}}.0-succ = v^k_{~x_i};
          v^k_{~x_{i+1}}.0-succ = v^k_{~x_i};
        }
      } else {
        v^k_{x_{i+1}}.0-succ = 0-sink;
        v^k_{~x_{i+1}}.0-succ = 0-sink;
      }
      v^k_{x_{i+1}} = create_new_or_find ( v^k_{x_{i+1}} );
      v^k_{~x_{i+1}} = create_new_or_find ( v^k_{~x_{i+1}} );
      v_{x_{i+1}} = insert_in_successor_list ( v^k_{x_{i+1}} );
      v_{~x_{i+1}} = insert_in_successor_list ( v^k_{~x_{i+1}} );
    }
    v.1-succ = v_{x_{i+1}};
    v.0-succ = v_{~x_{i+1}};
    l(v) = x_{i+1};
  }
  return(v);
}
```

Figure 2: Sketch of the Swap-in-Place Algorithm.

⊕-node successors is changed (see dotted box in Fig. 5). Up to now, the order of ⊕-OBDD successors could be neglected for ⊕-OBDD manipulation. But, in that particular situation, there is an important difference between the two results of the back-transformation, when the *swap-in-place* algorithm is applied again to the two variables (see (3a) and (3b)). Again, both ⊕-OBDDs, P and P_2'' represent the same Boolean function. The 0-successor of the top node in P_2'' represents also the function f_4, as for P, but in our implementation, we don't have a chance to realize that equivalence in an efficient way. This is, because for P_2'', the ⊕-node depends on the variable x_{i+1}. In P, the 0-successor depends on a variable x_j, $j > i+1$, or it is simply a sink. Thus, both are residing in different hash tables and their equivalence will not be realized.
Merge operations and the application of reductions are responsible for the swap-in-place operation not being exactly reversible and thus, symmetric. But, symmetry becomes a very important issue, if the swap-in-place operation has to be reversible, as e.g. in variable reordering heuristics.

4. Adapted Sifting Heuristic for ⊕-OBDD Optimization

For adapting the well known *sifting*-heuristic from OBDDs to ⊕-OBDDs, the swap-in-place operation as the basic variable exchange operation must be symmetric. Let x_i be a distinct variable x_i, $1 \leq i \leq n$, chosen from the given variable set of the OBDD P. With successive variable exchange operations, the variable x_i is transferred to every position within the variable order and for each position, the actual node count $P`_i$ is stored. In a final processing step, $x_i$ is transferred to level $k$, where $P`_k \leq P_i$, $\forall i$. This operation is repeated for every variable and finally, a local minimum for the size of P is achieved.
If we keep all merge operations and reduction rules, only another type of greedy heuristic would be possible that never has to revert to a previous state. But, the search space of all possible variable orders would be too limited and it would be rather likely to get stuck in an unfavorable local minimum.
Instead, we tried to stick to the orignal sifting algorithm as close as possible, because it has proven to be simple and one of the best heuristics for OBDD minimization. The restraints to keep the *swap-in-place* operation symmetric are quite simple:

(1) Renounce the application of certain reduction rules for meta-⊕-nodes, and

```
Input: ⊕-OBDD P_f, with variable order Π, and growth factor γ.
Output: ⊕-OBDD P'_f, P'_f ≤ P_f with variable order Π'.

mod2_sifting (P, γ) {
   create ordered list of variables x_i, 1 ≤ i ≤ n;
   foreach variable x_i {
     repeat {
       move x_i through all levels j, 1 ≤ j ≤ n, while
       storing |P_j|, the size of P with x_i in level j
       using the symmetric swap_in_place procedure;
     }
     until (|P_j| > γ · |P| or
             all levels j have been accessed)
     target = level j with |P_j| = min(P_j), 1 ≤ i ≤ n;
     move x_i to level target;
     remove all duplicate nodes in P;
     do complete extended reduction of P;
   }
   return(P);
}
```

Figure 3: Outline of the Sifting Algorithm for ⊕-OBDDs in Pseudo Code.

(2) Don't use existing ⊕-nodes from the hash table, when branching nodes are required to maintain the meta-⊕-node rule (i.e. only one meta-⊕-node is allowed on any path between two branching nodes of adjacent variables).

If (1) is maintained and not all reduction rules are applied, a potentially smaller ⊕-OBDD for the represented Boolean function and for the same variable order might exist. We don't know, whether the actual position of the sifted variable is the best or not. To prevent that problem, we must keep track of potential reductions that might take place. For every hash table h_{x_i} in the ⊕-OBDD P, we maintain a counter r_{x_i} that stores the number of possible reductions taking place during a variable exchange operation. To calculate the actual node count P_{real}, the number of possibly reduced nodes is subtracted from the number of counted nodes $|P|$: $P_{real} = |P| - r_{x_i}$. Additionally, a separate data structure has to keep track of the actual reference count of each node.
Duplicate nodes that have been created in accordance to rule (2) do not have to be counted. A counter for duplicate nodes d_{x_i} has to be maintained to compute the real node count for the sifting heuristic. Thus, the real size of P is computed $P_{real} = |P| - (r_{x_i} + d_{x_i})$. Now, for each position of the variable x_i in the variable order, we keep track

of the real size of the $\oplus$-OBDD P. x_i will be sifted into the position, where P_{real} was minimal.
Once x_i is in the correct position, a complete reduction step including the deletion of all duplicate nodes has to be performed (see Fig. 3 for an outline of the $\oplus$-OBDD sifting algorithm). To limit the growth of the $\oplus$-OBDD during the sifting procedure, a growth factor $\gamma \in \mathbb{R}$ similar as in the case for OBDDs is introduced, preventing the sifting algorithm to proceed beyond a point, where no improvement seems to be much likely.
To apply the sifting heuristic efficiently in synthesis, it makes only sense to apply the variable reordering algorithm already dynamically during the synthesis process to prevent the $\oplus$-OBDD from growing too large. If the heuristic is applied after synthesis has finished, the $\oplus$-OBDD might already have become too large to be represent with the available resources. Thus, during the synthesis process, the construction of the $\oplus$-OBDD P is stopped, after the size of P has exceeded a certain threshold bound α_0. If $|P| > \alpha_0$, the sifting algorithm is performed. Afterwards, synthesis continues, until the next threshold value $\alpha_1 = \gamma \cdot \alpha_0$ is exceeded. We chose $\gamma = 2$, the threshold value α_i, $i \geq 0$ doubles after each iteration. When the synthesis of the $\oplus$-OBDD has finished, a final reorder process might take place to optimize the already found variable order. For further improvement, the Jiggle algorithm for dynamic $\oplus$-node placement [13] can be called either during or after synthesis.

5. Experimental Results and Conclusion

For showing the efficiency of the heuristic, we have chosen the symbolic simulation of a subset of the LGSynth'93 [9] benchmarks. All experiments are computed on an Intel Pentium III 500 MHz based Linux system. Memory size is limited to 200 MB and computation time to 2 CPU hours. Circuits that are resulting in OBDDs with less than 100 nodes or that are exceeding the given resource limitations are excluded. Thus, 65 circuits remained in our benchmark set. For the probabilistic equivalence test it would have been sufficient to limit the number of signatures, i.e. the number of independent probabilistic equivalence tests, used for identifying $\oplus$-OBDDs to $n = 2$, but for reasons of security $n = 3$ was chosen.
In the first round of experiments (1), we compared only memory requirement for OBDDs and $\oplus$-OBDDs achieved by the sifting heuristic, while in the second round (2), additionally the jiggle heuristic for $\oplus$-OBDDs was applied. Additionally, to demonstrate the optimization potential of the adapted sifting algorithm in general, we also compared the size of $\oplus$-OBDDs with the variable order given by the circuit and the size of the $\oplus$-OBDD for the same circuit with the variable order achieved with the sifting algorithm (3). To get a suitable number of $\oplus$-nodes into the $\oplus$-OBDD representation, the greedy heuristic [13] for deciding, when to use *standard ITE* or *pDE*, was applied.
See Table 1 for an overview of selected results for (1), (2), and (3). In the first column the circuit name is given, while in the second column the memory requirement of the OBDDs achieved with the application of the sifting heuristic is listed. The third column gives the memory requirement for $\oplus$-OBDDs with the standard variable order given within the circuit description. Synthesis in all cases for $\oplus$-OBDDs has been performed with the APPLY-$\oplus$ algorithm based on pDE. Column four shows the $\oplus$-OBDD memory requirement for the application of the adapted sifting heuristic, while column five lists additionally the memory requirement for $\oplus$-OBDDs when dynamic $\oplus$-node relocation with the jiggle algorithm is performed. The last row sums up the memory requirements for all 65 benchmark circuits with the OBDD size denoted as 100%.
Discussion of the experimental results:

(1) Comparison of OBDD and $\oplus$-OBDD memory requirement for synthesis with dynamic sifting. If we compare the overall achieved sizes, the results for OBDDs and $\oplus$-OBDDs are quite similar, while for $\oplus$-OBDDs the overall achieved size with 92.6% of the OBDD-size is a little bit smaller. The important thing to mention is that now, with the sifting heuristic enabled, it is possible to perform $\oplus$-OBDD synthesis of 14 circuits of our benchmark set that were not computable within the given resource limitations previously. With a few exceptions, the results achieved for $\oplus$-OBDDs are always smaller or at least comparable to OBDDs. These exceptions include circuits that also previously have shown not to benefit too much from the introduction of $\oplus$-nodes. The worst case of these examples is *bw6x6*, where $\oplus$-OBDD size is about 233% the size of the OBDD. But, there are also examples of circuits that really do benefit from the introduction of $\oplus$-nodes, as e.g. *C499* or *s444*. There the $\oplus$-OBDD size could be reduced up to 20% (27%) of the OBDD size.

All in all, the impact of $\oplus$-OBDD nodes is not as obvious as for not optimized variable orders. The gain in size is less than 10%, if measured over all circuits of the benchmark in the average.

(2) $\oplus$-OBDD synthesis with enabled sifting heuristic and additional dynamic jiggle heuristic. The overall improvement for all circuits that could be achieved additionally to the sifting heuristic comes only up to 1%. But, there are some circuits, where significant improvements could be achieved, e.g. *s635* that could be reduced from 219% or the OBDD size to only 71%, or *i7*, where a reduction from 147% to 98% was possible. But for the major part of the circuits, only a small reduction could be achieved. If we take the additional amount of time required for the jiggle algorithm into account, its application is not efficient.

(3) Comparison of $\oplus$-OBDD memory requirement with and without application of sifting. The results show the significant impact of the variable order on the $\oplus$-OBDD size. 14 circuits of our benchmark set that could not be computed with the variable order given within the circuit file are manageable, if sifting is applied. The original $\oplus$-OBDD is about 40 times larger than the size achieved with the sifting heuristic.

A comparison of runtime for OBDDs and $\oplus$-OBDDs is difficult, because for OBDDs highly sophisticated and well tuned programming packages do exist for many years. We had to develop a $\oplus$-OBDD package from the scratch and it is by far not as fast as the CUDD OBDD-package [Som96] we are working with. Only to give a clue about the runtime behavior, when comparing CUDD with our $\oplus$-OBDD package, CUDD is up to 10 times faster than our package.

References

[1] S. B. Akers, Binary Decision Diagrams, *in IEEE Trans. on Computers*, vol. **c-27**, no. 6, 1978, 509-516.

[2] M. Blum, A. K. Chandra, M N. Wegman, Equivalence of Free Boolean Graphs can be decided Probabilistically in Polynomial Time, *in Information Processing letters* **10**, No. 2, 1980, 80-82.

[3] K. S. Brace, R. L. Rudell, R. E. Bryant, Efficient Implementation of a BDD Package, *in Proc. of the 27th ACM/IEEE Design Automation Conf.*, 1990, 40-45.

[4] R. E. Bryant, On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Applications to Integer Multiplication, *in IEEE Trans. on Computers* **40** Vol. 2, 1991, 205-213.

[5] J. Gergov, Ch. Meinel, Frontiers of Feasible and Probabilistic Feasible Boolean Manipulation with Branching Programs, *in Proc. 10th Annual Symp. on Theoretical Aspects of Computer Science*, **665** of LNCS, Springer, 1993, 576-585.

[6] J. Gergov, C. Meinel, Mod2-OBDDs: A Data Structure that generalizes EXOR-sum-of-products and Ordered Binary Decision Diagrams, *in Formal Methods in System Design* **8**, Kluwer, 1996, 273-282.

[7] A. Hett, R. Drechsler, B. Becker, Reordering Based Synthesis, *in Proc. of the 3rd Int. Workshop on Applications of the Reed-Muller Expansion in Circuit Design (RM'97)*, Oxford, UK, 1997, 13-22.

[8] J. Jain, M. Abadir, J. Bitner, D. S. Fussell, J. A. Abraham, IBDDs: An Efficient Functional Representation for Digital Designs, *in Proc of the European Conference on Design Automation* (1992), 440-446.

[9] LGSynth93 Benchmarks: `http://www.cbl.ncsu.edu/CBL_Docs/lgs91.html`.

[10] J.-C. Madre, J.-P. Billon, Proving Circuit Correctness using Formal Comparison between Expected and Extracted Behaviour, *in Proc. 25th ACM/IEEE Design Automation Conference (Anaheim, CA)*, 1988, 205-210.

[11] C. Meinel, H. Sack, Algorithmic Considerations of $\oplus$-OBDD Reordering, *in Proc. of the 4th International Workshop on Applications of the Reed-Muller Expansion in Circuit Design (Victoria, BC, Canada)*, 1999, 197-184.

[12] C. Meinel, H. Sack, Mod2OBDDs - a BDD Structure for Probabilistic Verification, *in Electronic Notes in Theoretical Computer Science*, vol.**22**, 2000.

[13] C. Meinel, H. Sack, A Simple Heuristic for Mod2-OBDD Minimization, *Proc. of IEEE/ACM Int. Workshop of Logic and Synthesis (IWLS2001) Lake Tahoe, CA, USA*, 2001, pp. 304-309.

[14] Ch. Meinel, T. Theobald, Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications, *Springer*, Heidelberg, 1998.

[15] D. E. Muller, Application of Boolean Algebra to Switching Circuit Design and Error Detection, *in IRE Trans. on Electronic Computing* **EC-3**, 1954, 6-12.

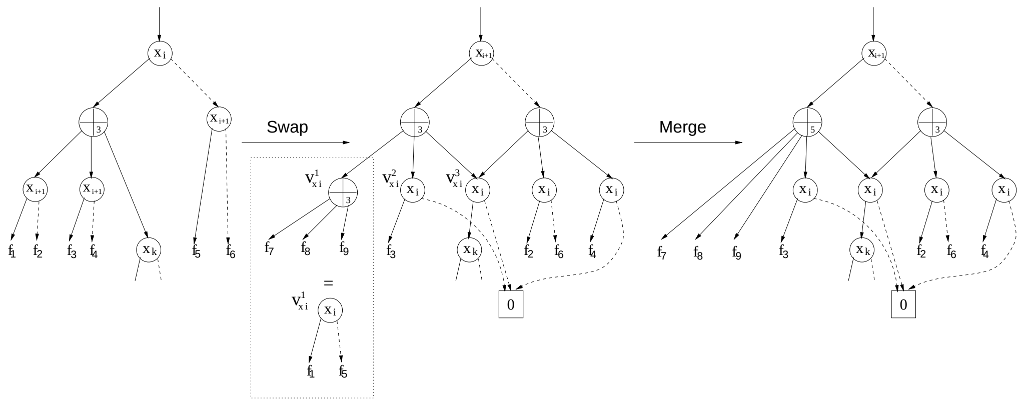

Figure 4: Merge During the Swap-in-Place Procedure.

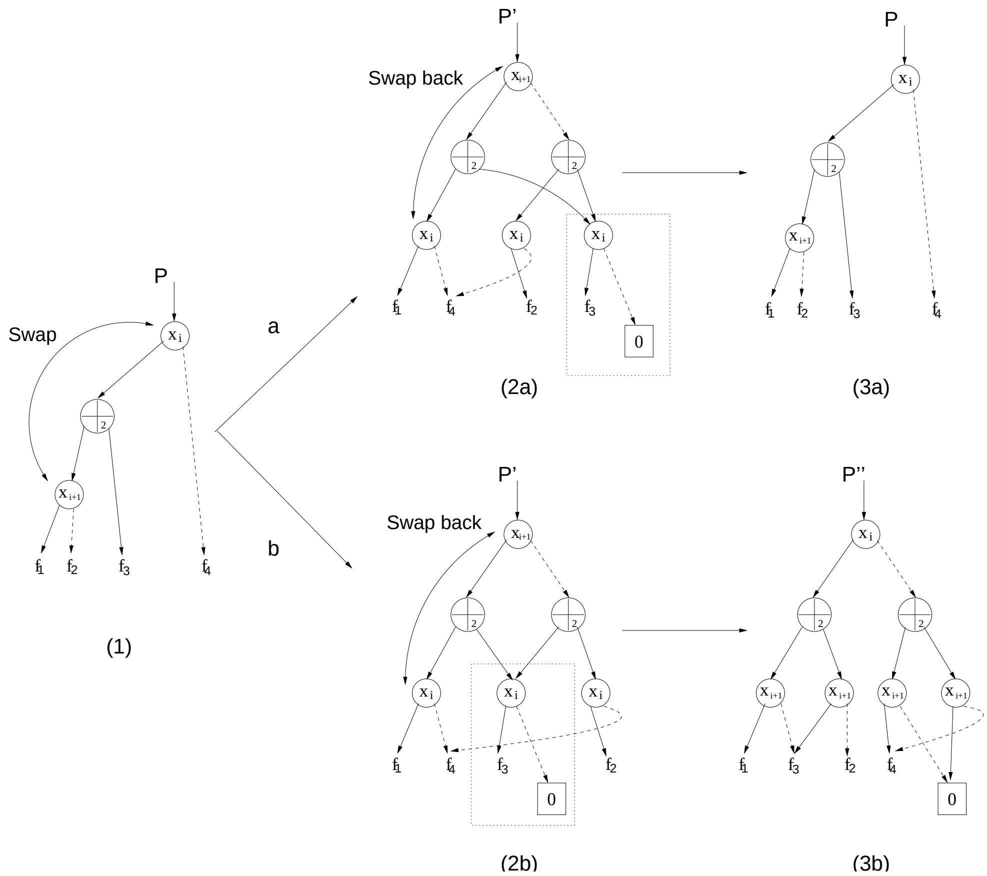

Figure 5: Non Symmetrical Situation for the Swap-in-Place Operation.

Circuit	OBDD-size	⊕-OBDD size [Bytes]		
	Sifting	pDE-meta	Sifting	+ Dynamic Jiggle
pair	128664	3409992	106776	106376
my_adder	4716	18873336	5868	5100
i7	14148	31172	20800	13928
i2	7380	10880	7344	7344
bw6x6	28980	95832	67940	67628
alu2	5832	9856	5768	5760
C499	958464	670188	196708	187892
C432	43560	184040	37160	36792
C1355	1064232	670024	1065340	1064196
comp	4608	27142104	3928	3796
i10	2446956	-	1906792	1904572
mm30a	3621276	-	3010564	3009244
C7552	296674	-	254196	254196
sbc	38052	135332	38556	35672
s820	8172	16532	7044	6264
s713	22644	115428	18408	17888
s641	22644	115360	16044	14660
s635	4716	64692	10364	3356
s444	5796	11136	1588	1340
s1269	78804	1101680	79140	76328
mm9b	90936	18157936	74108	73292
mm9a	72828	14583716	57264	56472
mult16a	7380	23852212	8032	8032
s838.1	6444	-	6516	5984
s3384	32616	-	40744	32432
Σ	**13.701.388**	**-**	**12.700.536**	**12.535.556**
	100%		**92.7%**	**91.5%**

Table 1: Comparison of OBDD and ⊕-OBDD size for the Sifting Heuristic.

[16] L. S. Reed, A Class of Multiple Error-Correcting Codes and their Decoding Scheme, *in IRE Trans. on Information Theory* **4**, 1954, 38-42.

[Som96] F. Somenzi, CUDD: Colorado University Decision Diagram Pachage, *ftp://vlsi.colorado.edu/pub/*, 1996.

[17] S. Waack, On the Descriptive and Algorithmic Power of Parity Ordered Binary Decision Diagrams, *in Proc. 14th Symp. on Theoretical Aspects of Computer Science*, **1200** of LNCS, Springer, 1997.