Published at Reed-Muller 97, Oxford

Reordering Based Synthesis *

Andreas Hett Rolf Drechsler Bernd Becker

Institute of Computer Science, Albert-Ludwigs-University
79110 Freiburg im Breisgau, Germany
email: <name>@informatik.uni-freiburg.de

Abstract

Reordering Based Synthesis (RBS) as an alternative approach to manipulate Decision Diagrams (DDs) is presented. Based on the concept of operation nodes a single "core" operation, i.e. an extended Level Exchange (LE), is sufficient to perform the usual synthesis operations on several types of DDs. RBS allows the integration of dynamic variable ordering (even) within a single synthesis operation (e.g. an AND-operation) and thus provides the possibility of avoiding huge peak sizes during the construction. The optimization potential is significantly enlarged by allowing LEs even between operation nodes. A large number of experimental results for the BDD case confirm the validity of the concept.

1 Introduction

Decision Diagrams (DDs) are often used in CAD-systems for the efficient representation and manipulation of Boolean functions. The most popular data structure in this context are *Ordered Binary Decision Diagrams* (OBDDs) [8]. In the meantime they are widely used, especially in the area of verification and synthesis. The interest in OBDDs results from the fact that the data structure is generally accepted as providing a good compromise between conciseness of representation and efficiency of manipulation [9]. However, some relevant classes of Boolean functions cannot be represented efficiently by OBDDs [27, 6]. Thus, depending on the application area several alternatives evolved, among them being OFDDs, OKFDDs, ZBDDs and MDDs. More recently, for the representation of functions from a Boolean domain to the integers so-called word-level DDs have been introduced, among them being MTBDDs, (*)BMDs, HDDs, and K(*)BMDs. (For an comparative overview see [5]).

Manipulation algorithms in almost all DD packages (see e.g. [7, 29]) realized until recently are based on exactly the same synthesis implementation methods as introduced for the origin of all mentioned DDs (the BDDs), i.e they are based on recursive function calls in combination with a *Computed Table*. The *Apply* and the *If-Then-Else* operator (ITE) are well-known representatives of this method. Although ITE gives reasonable results in several applications, recent research has shown that alternative methods might be interesting, e.g. *BFS* operations [24] and use of memory system hierarchy [26] are often more powerful for very large applications.

In [19] a new approach for the implementation of synthesis on BDDs called *Multiple OR-operation by means of Existential quantification* (MORE) was presented. This approach basically realizes an OR-operation by the introduction of coding variables and subsequent LEs, allowing the combination of an arbitrary number of BDDs in parallel using *Dynamic Variable Ordering* (DVO) [18, 27]. Since all conventional BDD packages provide the possibility to locally exchange two adjacent variables as a core operation for DVO, the MORE-approach simplifies the package implementation by discarding the need for specific synthesis algorithms.

A refinement of the MORE concept in [20] lead to efficient symbolic simulation of BDDs. Furthermore, it was demonstrated that, instead of "reducing" all synthesis operations to the OR-operation as it was done in the original MORE-approach, synthesis can be performed by *Reordering Based Synthesis* (RBS), i.e. the introduction of operator nodes for each 2 operand Boolean operation and subsequent LEs. In the meantime the MORE concept has been the starting point for several new DD-based verification techniques (e.g. see [10, 1]).

In this paper we show that the concept of RBS can be generalized to various DD types. We give insight into our methodology and improvements of RBS (e.g. by means of *Terminal Case Handling* (TCH) and *Complex Gate Calculation* (CGC) [20]). In particular, we also introduce an extension of LE to operator nodes, which furthermore increases the flexibility of RBS to synthesize the final representation efficiently

*This work was supported in part by DFG grant Be 1176/8-2

with small intermediate sizes. In the experimental part of this paper we present results to validate our approach. Currently we have a BDD package realizing RBS based on operator nodes and extended LE. Our experiments on BDDs underline the quality of this approach examplarily for various DD types. We consider symbolic simulations for the ISCAS'85, ISCAS'89 and some industrial benchmarks (known to be hard for BDDs) and make comparison against state-of-the-art packages. The results will show that our new method has low memory requirements without any overhead with respect to runtime. This is demonstrated in particular by measuring the *Peak Size*, i.e. the maximal number of nodes needed during the whole run. Additionally, we present first results showing that extended LE offers interesting possiblities for further optimizations.

2 Preliminaries

2.1 Decision Diagrams

We briefly provide essential definitions and properties of various decision diagram types. For more details see [9, 4]. Rather than giving the mathematically exact and complete formalisms we want to cover the main points in an informal way helping the reader to get a quick impression of the differences between the data structures.

Let $f : \mathbf{B}^n \to \mathbf{B}$ be a Boolean function over the variable set $X_n = \{x_1, \ldots, x_n\}$. All DDs considered here are graph-based representations, where at each (non-terminal) node labeled with a variable x_i a decomposition of the function represented by this node into subfunctions is performed. Up to one exception discussed at the end of this subsection we consider decompositions into two subfunctions, the *low*-function and the *high*-function. Furthermore, the underlying graph is *ordered*, i.e. the variables occur in the same order on all paths of the DD. For DDs the following three decompositions are considered:

$$\begin{array}{rcll} f & = & \overline{x} \cdot f_{low} \oplus x \cdot f_{high} & \textit{Shannon } (S) \\ f & = & f_{low} \oplus x \cdot f_{high} & \textit{positive Davio } (pD) \\ f & = & f_{low} \oplus \overline{x} \cdot f_{high} & \textit{negative Davio } (nD) \end{array}$$

(f is the function represented at node v labeled with variable x, f_{low} (f_{high}) denotes the function represented by the *low*-edge (*high*-edge) of v. $\oplus$ is the Boolean Exclusive OR-operation.) The recursion stops at terminal nodes labeled with 0 or 1. Based on a precise definition of the notion "decomposition type", it can be proven for the Boolean case that the three decompositions above are the only ones of interest for DDs [3]. Decomposition types are associated to the Boolean variables $x_1, x_2, \ldots, x_n$ by means of a *Decomposition Type List* (DTL) $d := (d_1, \ldots, d_n)$ where $d_i \in \{S, pD, nD\}$, i.e. d_i provides the decomposition type for variable x_i $(i = 1, \ldots, n)$. Thus, there exists only one fixed decomposition type per variable. Since small representations are a main focus when dealing with DDs reduction rules are defined (see [8, 23, 17, 16] for details).

Furthermore, the size[1] of the DDs above can be reduced if *Complement Edges* (CEs) are used. In the following (if not stated otherwise) all DDs are assumed to be ordered and reduced. In this reduced and ordered form all DDs are a canonical representation.

Recently, DDs for representing word-level functions (from the Boolean domain to the integers) have been introduced. Obviously, BDDs can be extended to represent functions $f : \mathbf{B}^n \to \{0, .., k-1\}$ and the resulting graphs are denoted as *Multi-Terminal BDDs* (MTBDDs). The operations on MTBDDs can be carried out as efficiently as in the case of two terminals [11].

For word-level DDs (analogously to the DDs presented above) different decompositions can be considered, e.g. functions of the form $f : \mathbf{B}^n \to \mathbf{Z}$:

$$\begin{array}{rcll} f & = & (1-x) \cdot f_{low} + x \cdot f_{high} & \textit{Shannon } (S) \\ f & = & f_{low} + x \cdot f_{high} & \textit{positive Davio } (pD) \\ f & = & f_{low} + (1-x) \cdot f_{high} & \textit{negative Davio } (nD) \end{array}$$

Similar to OKFDDs the idea of using different decomposition types can also be used for word-level DDs [12, 15]. (Notice that for attributed edges and edge-values as they are used for *Kronecker Multiplicative Binary Moment Diagrams* (K*BMDs) [15] additional reduction and normalization rules have to be used.)

[1] The *size* of a DD F denoted by $|F|$ equals the number of its non-terminal nodes.

Finally, we also want to review the idea of extending DDs to have more than two outgoing edges per node. It is straightforward to extend MTBDDs to *Multi-Valued Decision Diagrams* (MDDs) [30] representing functions $f : \{0, .., k-1\}^n \rightarrow \{0, .., k-1\}$. For this each internal node has k outgoing edges[2]. In [30] it has been shown that the efficient operations known for BDDs can also be carried out on MDDs using a *case*-operator instead of the *ite*-operator [7].

2.2 Recursion Based Synthesis

Typically synthesis on DDs relies on recursive algorithms. The ternary *If-Then-Else*-operator (ITE) [7] forms the core of recursion based synthesis operations for BDDs. (For simplicity we will restrict ourselves to the detailed description for BDDs if not mentioned otherwise, but similar observations hold for all other DD types as will be outlined below.) ITE is normally implemented by depth-first-search, but also breadth-first-search is possible. The recursive formulation $ite(F, G, H) = (x, ite(F_{high}, G_{high}, H_{high}), ite(F_{low}, G_{low}, H_{low}))$ determines the computation of the operation. F_{low} and F_{high} denote F evaluated at $x = 1$ and $x = 0$, respectively (the same holds for G and H). ITE can be used to implement all two-variable Boolean operations.

Binary, recursive synthesis operations on BDDs assisted by an ideal *Computed Table* to prevent recalculations of previous results are time bounded by $O(|F| \cdot |G|)$. This is *not* true for FDDs: The worst case time complexity for a binary, recursive synthesis on FDDs is exponential [6].

2.3 Level Exchange

In order to modify the variable ordering of DDs we can exchange variables in adjacent levels. This method is called *Level Exchange* (LE) [18] in the following. Since an exchange of neighbouring variables is a local operation for all DD types mentioned above (if edge values are not used) consisting only of the relinking of nodes in these two levels, this can be done very efficiently (see Figure 1). LE was first proposed for the minimization (i.e. the node reduction) of DDs due to an optimized variable ordering. This can be achieved by subsequently applying LEs in a way such that reduction cases occur (see Figure 1)[3]. In [27] it was shown that it is even possible to perform LEs as local operations when using DDs with CEs. LE for FDDs, OKFDDs, ZBDDs and MTBDDs works likewise. For MDDs the exchange of neighbouring variables is applied by a rearrangement of multiple grand-sons [22, 14].

3 Reordering Based Synthesis Approach

The basic idea of MORE (based on existential quantification and restricted to the OR-operation only) and an extension to RBS was first introduced for BDDs in [19, 20]. Here we propose a general approach of *Reordering Based Synthesis* for DD types mentioned in the former section. The following subsections will provide the needed expansion of the data structures and describe the method of performing synthesis by means of expanded LEs.

3.1 Operator Nodes and Expanded Level Exchange

In addition to the variables encountered in (conventional) DDs (in the following referred to as *Boolean Variables*), we allow the coexistence of an additional type of variables called *Operator Variables*. We will see that they smoothly fit into the concept of DDs with CEs as described above. Each node labeled with an *Operator Variable*, in the following called *Operator Node*, is associated to a binary boolean operator $\odot$ and according to this interpreted as $f = f_{low} \odot f_{high}$.

It can be shown that for all DDs allowing Shannon decomposition the equations $(f \odot g)_{low} = f_{low} \odot g_{low}$ and $(f \odot g)_{high} = f_{high} \odot g_{high}$ hold for any binary operator $\odot$. Thus, we conclude

$$(\overline{b} \cdot f_{low} + b \cdot f_{high}) \odot (\overline{b} \cdot g_{low} + b \cdot g_{high}) = \overline{b} \cdot (f_{low} \odot g_{low}) + b \cdot (f_{high} \odot g_{high})$$

and the transformation from either term to the other represents a LE-operation between the *Operator Variable* associated to $\odot$ and the adjacent Boolean variable b on the corresponding DDs. We can implement this by expanding the LE algorithm, allowing *Operator Variables* change their positions. As is shown in Figure 2 this LE matches the one for Boolean variables only. However, we do apply an operator dependent terminal case handling instead of reduction rules. In Figure 2 examplary terminal cases for the OR-Operator are shown. In this case, $f + f$ is reduced to f, $g + 1$ is reduced to 1.

[2]In our application we restrict ourselves w.l.o.g. to the case that all variables are defined over the same set of values.

[3]Note that the reduction rules are dependent on the DD type.

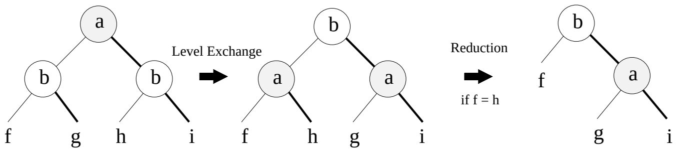

Figure 1: Level exchange for two adjacent variables

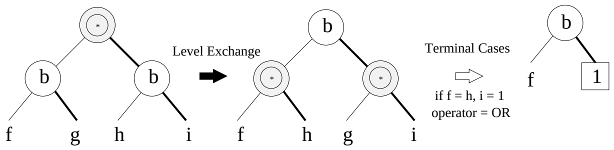

Figure 2: Level exchange for operator variables on BDDs

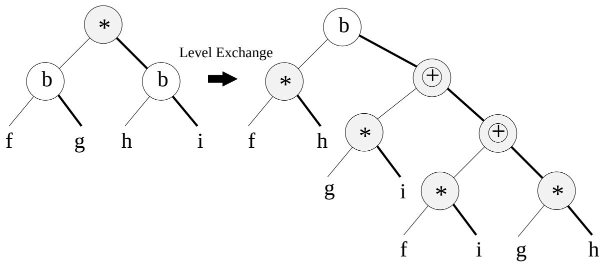

Figure 3: Examplary level exchange for operator variables on FDDs

If Davio decomposition is allowed in the DD (e.g. in FDDs) the LE algorithm still remains simple, but the impact on the data structure is usually much higher. However, the number of additional operator nodes exactly corresponds to the recursive synthesis formula. Figure 3 gives an example. Here, an AND-operation is shifted below the variable b according to

$$(f \oplus b \cdot g) \cdot (h \oplus b \cdot i) = f \cdot h \oplus b \cdot (g \cdot i \oplus f \cdot i \oplus g \cdot h).$$

Obviously a subsequent downward shift of *Operator Variables* will finally dissolve all *Operator Variables*, resulting in a operator-free DD. Since our LE (as well as the original LE) does not change the functionality of the represented function, but only transforms a representation with operator nodes to a operator-node free DD, this LE inherently performs a synthesis on the DDs. The next subsection will describe the initialization of the synthesis problem in more detail.

Note that CEs are left out in the presented DD figures solely for simplicity. At first sight these figures might suggest that terminal case handling (in the following also referred to as TCH) is trivial. Due to performance reason (see also Subsection 2.1) CEs are standard in DD packages, TCH for DDs with CEs is more complicated:

Whenever a *Complement Mark* (CM) is situated on the low edge of a node, a CM normalization is done by switching the marks on all three edges of this node[4] (i.e. a CM is eliminated, an uncomplemented edge gets a CM). Whenever a CM normalization is applied for an operator node, the operator itself has to be modified according to the rules of DeMorgan. Due to this and the fact that these operator modifications usually occur only to a subset of all operator nodes bearing the same variable, operators must not be associated to *Operator Variables* but operator nodes to allow the existence of different operators in the same level.

3.2 Handling of Multiple Operands

Assume that an arbitrary number of DDs F_i, $i = 1, \ldots, n$ are to be combined by means of a binary, associative operator $\odot$. Therefore, a logarithmical number $m := \lceil log_2(n) \rceil$ of *Operator Variables* are introduced and a tree of operator nodes is constructed above the operands. Since the DDs $F_i, i = 1, \ldots, n$ are already given this can be done in linear time with respect to the number of operands. Afterwards the coding variables are shifted towards the bottom of the DDs by means of our expanded LEs. During that shift phase the operator variables will be dissolved due to the TCH and vanish completely. Since during that run a non-reduced DD can occur a reduction run should be carried out at latest whenever the package user forces canonicity of the data structure.

In contrast to conventional recursive synthesis, RBS can handle an arbitrary number of operands at the same time instead of only two and operate highly in *parallel.* In [20] we have given an example to demonstrate that large advantages may result from this property. In Section 6 experiments are presented to underline this effect.

4 Improving RBS

In this section we study add-on techniques to improve RBS on various DD types. At first, a simple method to perform DVO even during synthesis processes in order to minimize on-the-fly is introduced.

[4]For Davio decompositions no CM-switching is done on the high edge of a node.

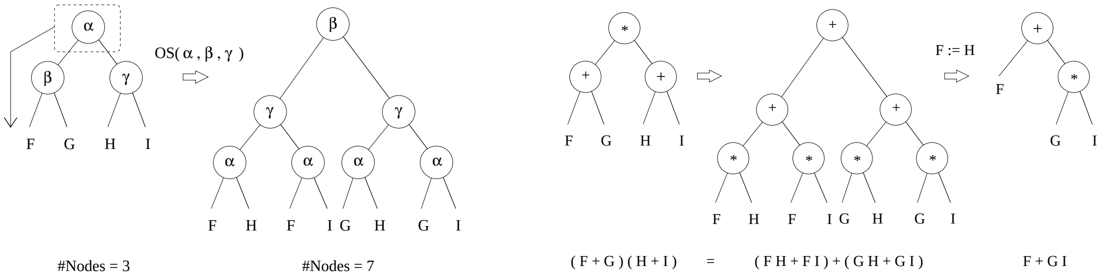

Figure 4: Operator exchange on BDDs

The second subsection gives insight into a further expansion that allows transformations on the *Operator Nodes* in order to minimize the calculation complexity.

4.1 Combining Synthesis and Dynamic Variable Ordering

In this subsection we want to outline another very important advantage of RBS: The synthesis process can be interrupted at any point **without** loss of intermediate results and additional costs. Thus, there arise new possibilities for the dynamic minimization during the construction of DDs; the dynamic reordering can even support the synthesis operation, if it is guaranteed that the *Operator Variables* are not shifted back towards the top.

With our method, we can guarantee the initialization of dynamic minimizations at any time of need, i.e. before crossing an upper size boundary, even during a single synthesis step as well as a minimization without restrictions and loss of intermediate results. *Supervising heuristics* [20], having a *finer granularity* than it is possible when the construction process is relying on recursion based synthesis algorithms, can be applied to control the construction process and initiate minimization procedures in appropriate situations.

In packages based on recursive synthesis it is not possible in a straightforward manner to interrupt a synthesis operation and perform “unrestricted” DVO. In our approach the synthesis can be stopped after each LE without loss of data and additional costs. Therefore we get a worst case complexity linear in the size of the current level.

4.2 Operator Exchanges and Synthesis

Finally, we discuss an extension of the method presented in [19, 20]. Instead of only considering the exchange of an *Operator Variable* with a *Real Variable* we now also consider the situation of exchanging two *Operator Variables*. The idea of this method will be outlined by the following (informal) example:

Example 1 If during the symbolic simulation $(F + G) \cdot (H + I)$ has to be computed, but it turns out that is is impossible to build the corresponding DDs for $F + G$. Then an exchange of the $+$ and $\cdot$ operator might be helpful. This would result in the expression $(F \cdot H + F \cdot I) + (G \cdot H + G \cdot I)$. By this exchange the “expensive” computation of $F + G$ would be avoided. (In the following we will see that the exchange might even result in a simplification of the expression with respect to the number of operations to be carried out.)

We now study this problem in more detail and give an algorithmic solution: For this, let us assume that an *Operator Variable* is situated directly above another *Operator Variable*. Neither the original nor our expanded LE-operation can handle this situation to exchange their positions. (Notice that this does not influence synthesis in general, since it is always possible to choose an *Operator Variable* atop of a Boolean variable in order to proceed). However, it might be interesting to add the exchange of two adjacent *Operator Variables* (see example above), in the following called *Operator Exchange* (OE). Exploiting the structural information kept in the DD by means of the operator nodes allows us to simplify synthesis operations before and during their processing.

In the following we restrict our discussion on BDDs only, since for other DD types further “difficulties” occur and we want to keep the presentation as clear as possible. Assume the most general case[5] as shown in Figure 4. We now focus on the following problem:

We have operators α, β, γ ($\alpha, \beta, \gamma \in \{\cdot, +\}$) and want to shift α downwards.

[5]Note that we restrict this discussion to an operator set of $\{\ \cdot\ ,\ +\}$ in order to simplify the diagrams.

(A straightforward computation shows that the transformation given in the figure is correct independent on the choice of α, β and γ.) Obviously the OE would result in a representation with four additional nodes, making the situation worse (but also see Example 1). However, in many cases, not all four operands f, g, h and i are different. Indeed it occurs very often that e.g. one of the operands is a complement of another or equal to it. In these cases, the outcome completely changes:

Example 2 Figure 4 gives an impression that it can be profitable to perform the OE, if operands are somehow related as stated above. Apparently, the relationship of the four operands (in our example $F = H$) influences the profitability of the OE by allowing the elimination of redundant nodes due to terminal cases.

Furthermore, these terminal cases depend on the operators of the operator nodes. If we have to perform an OE for two adjacent *Operator Variables* and want to guarantee that no additional nodes are constructed, we have to assure that only profitable transformations are done. Therefore an OE looks like this: Both variables exchange their positions without modifying the operators in the nodes. This is a simple process since only the variables in the three operator nodes have to be exchanged. Now a transformation of the operators will be made if and only if the outcome is a simplified representation, i.e. if the exchange is profitable. Thus, we have coded all profitable transformations in static tables and select appropriate ones according to the evaluation process.

Finally, we have to answer the question when to apply transformations. There exists a wide field for experiments and developments; our methods only represent first steps. When performing a synthesis, we usually [19, 20] grab the bottommost *Operator Variable* and shift it towards the bottom. In this case Operator Variables never *have* to be exchanged. In our first attempt to integrate the process of OE we apply a very plain algorithm: We choose the topmost *Operator Variable*. By doing this, it will pass all other existent *Operator Variables*, hopefully allowing some profitable transformations during these OEs, and then will cover distance to them being shifted towards the bottom by means of LEs. Obviously more clever heuristics can be used to improve the runtime/space behaviour.

5 Expansion to other Types of Decision Diagrams

Up to now we only focused on bit-level DDs, like BDDs and FDDs. It turned out that the method (originally proposed for BDDs [19]) can be extended to Davio-based DDs.

In the following we consider the other DD types presented in Section 2:

ZBDD [23]: RBS can directly be transfered to ZBDDs. Analogously as in the case of recursive synthesis the polynomial worst case complexities proved for RBS [20] transfer to this data structure. The algorithms are similar to the ones presented here.

MTBDD [2, 13]: The algorithms also directly transfer except for the terminal case handling. The worst case complexities for RBS are the same as for recursive synthesis algorithms.

K*BMD [15]: In this case the algorithms can not directly be transfered, since the exchange of neighbouring levels is not a local operation (see Example 2 in [15]).

For this it is not straightforward to use the level exchange. One solution might be that the canonicity is (temporarily) given up (analogously as for BDDs in [19]). Further studies are needed to see whether the concept can also be applied to these more general data structures.

MDD [30]: The RBS principle can also be applied in the case of several outgoing edges per node. This is due to the fact that also for MDDs dynamic reordering methods can be applied [22, 14]. But it has to be considered that for each exchange at most $k \cdot (k-1)$ pointers must be redirected.

In addition (since the *Operator Variables* might have less than k outgoing nodes) the implementation has to deal with different node sizes. This problem is similar to dealing with different variables with varying ranges (see [14]).

All in all it turns out that the RBS approach also works for other DD types and for this is to be seen as an alternative not only for BDDs.

6 Experimental Results

In this section we demonstrate the efficiency of RBS giving experimental results for *Symbolic Simulations* on standard and some "hard" industrial benchmarks. We compare our results (implemented in the BDD package A.MORE) to the CUDD package [29]. Additionally, we outline the superior memory behaviour during DD construction by showing a comparison of RBS *Peak Sizes* against measurements for a newly proposed method to perform symbolic simulation less memory stressing called XTop-algorithm [21]. Finally, first results of *OEs on BDDs* are shown to give an impression of the expansion opportunities correlated to our approach.

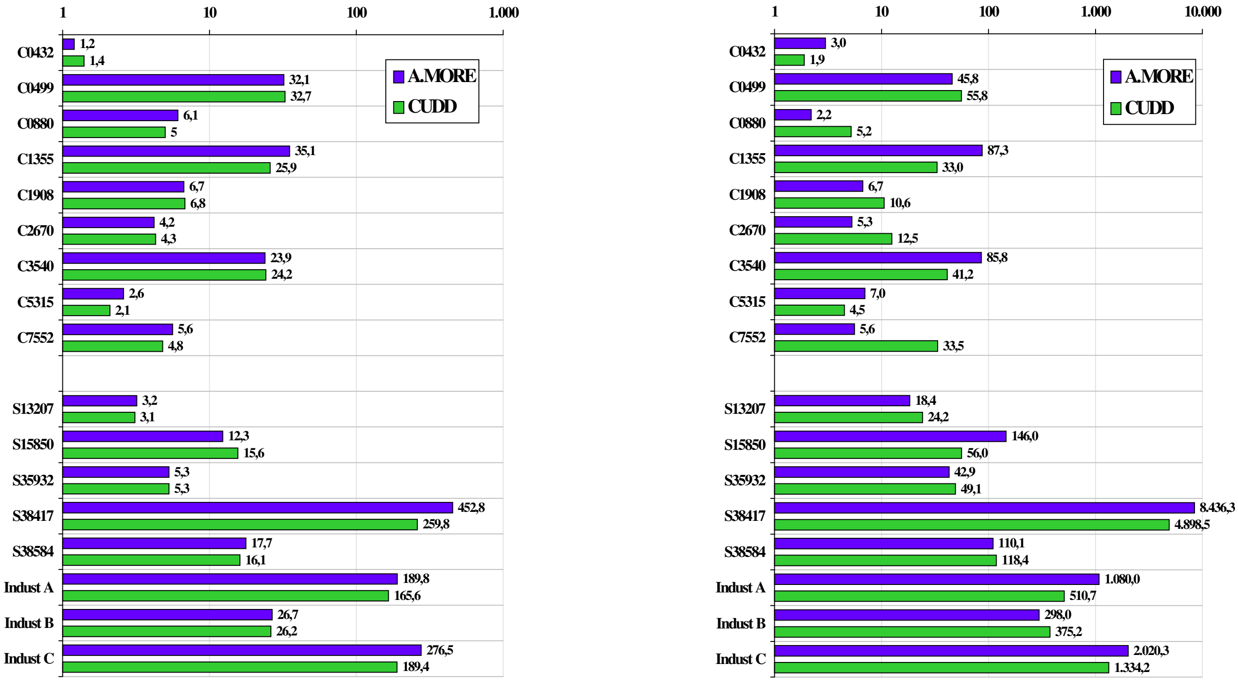

Figure 5: Final sizes [k nodes] and runtimes [CPU s] for CUDD and A.MORE

6.1 Efficient Construction

We consider symbolic simulation for ISCAS'85 and ISCAS'89 benchmarks as well as some industrial circuits. All measurements are done on a SUN workstation Ultra 1 using heuristical methods to determine a good initial variable ordering. Note that these strategies differ in both packages, influencing the calculations by means of different initial variable orders. Thus, the presented measurements allow no direct comparison of single benchmarks but give an impression of the average behaviour. We comment on the final sizes (measured in 1,000 nodes) and runtimes (measured in CPU seconds) for the whole symbolic simulation processes.

In the case of CUDD we applied *Group Sifting* [25], a minimization heuristic superior to *Symmetry Sifting* that outperforms original sifting introduced by Rudell [27] (as was shown in [25]). For our approach currently the less efficient sifting method of Rudell is the only one implemented and therefore was used.

Focussing on the top halfs of both diagrams shown in Figure 5 we can observe that the final size and runtime behaviour for all listed ISCAS'85 benchmarks equals on average. For the listed ISCAS'89 and industrial benchmarks larger discrepancies can be observed between A.MORE and CUDD. For example, the construction of benchmark s38417 is outstanding for CUDD.

However, the A.MORE results become competitive also for these benchmarks when (only) Rudell's sifting is applied for CUDD. Thus, we intend to integrate advanced minimization techniques as well as the concept of an *Interaction Matrix* [29]. The Interaction Matrix allows to perform LE operations in constant time, if no relationship of two adjacent variables is present. This feature gains importance for an increasing number of variables, which is the case for the larger benchmarks considered in our measurements. Since LE is our core algorithm for the whole package this will influence synthesis performance for RBS as well.

6.2 Peak Size

An aspect of growing interest and importance is the *Peak Size*, i.e. the maximal number of nodes needed during the whole construction. In many applications the final size is very small (see e.g. tautology checking), but due to the representation of intermediate results the number of nodes needed during the run usually exceeds this value by far. Our approach helps to avoid huge peaks by reducing the number

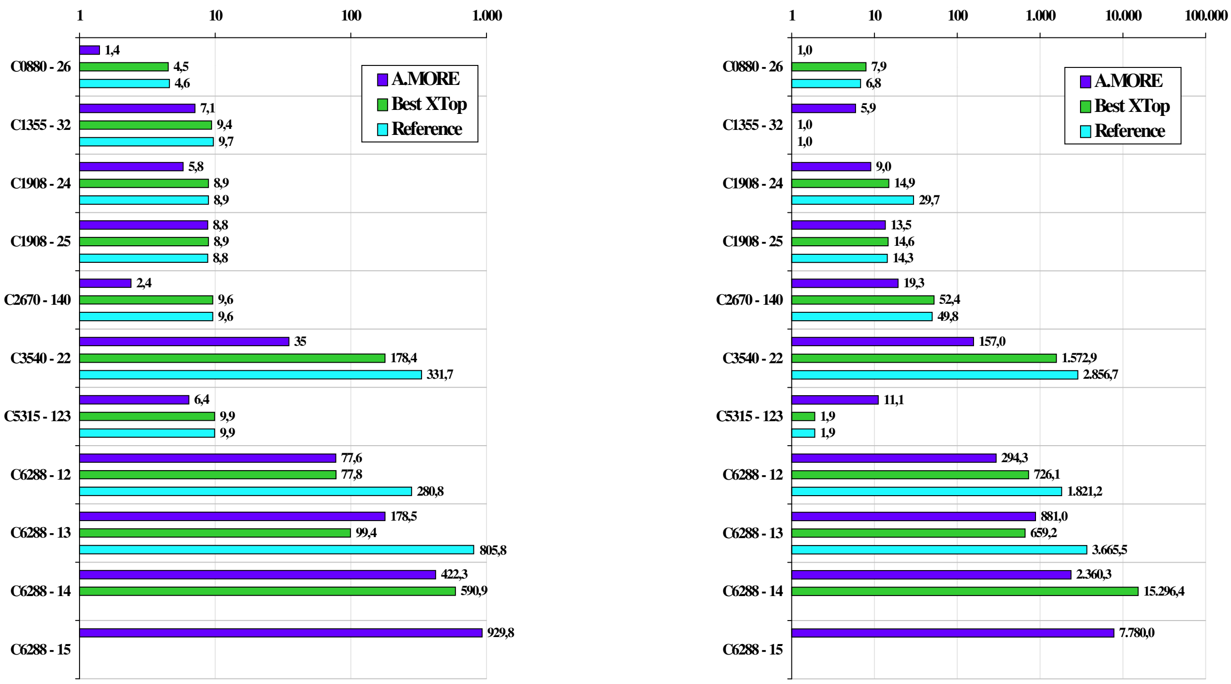

Figure 6: Peak sizes [k nodes] and runtimes [CPU s] for XTop and A.MORE

of intermediate calculations due to parallel synthesis (e.g. calculating a gate in only a single synthesis operation). The outcome is a very flat size curve, less stressing the memory resources.

Recently a new method, called XTop [21], for performing symbolic simulation with less memory overhead was proposed. XTop combines the conventional bottom-up gate construction of symbolic simulation (also applied in our method) with a top-down method. Decomposition points for the use of temporary variables (representing the gate rather than a BDD) are determined dependent on functional and/or structural properties of the circuits and thus, allowing a flat size curve during the construction. It is important to realize that this method has no exclusive-or relationship to our approach, i.e. both methods can even be combined to yield a more powerful synthesis tool. Having this in mind, the comparison of the peak sizes and runtimes is not so much a competition but a demonstration of the efficiency of our approach.

In Figure 6 the peak sizes and runtimes for a “best of”-selection of the XTop-approach as presented in [21] as well as A.MORE and a monolithic approach (denoted as Reference) are shown. The Reference algorithm represents a conventional BDD package, applying a bottom-up gate construction during symbolic simulation. (The numbers of Reference are taken from [21].) Both diagrams of Figure 6 give experimental results for ISCAS’85 benchmarks. In comparison to a full symbolic simulation, here only a single “hard” output (denoted by the number behind the benchmark name) was calculated. The calculations were restricted to an upper size limit of 1 million nodes. The XTop and Reference runtimes are stated as measured on a DECstation 5000/240 while A.MORE performed on a SUN workstation Ultra 1.

It can be easily observed that A.MORE performs superior to the other approaches. The peak size values for almost all listed benchmarks (the only exception is the 13th output of the multiplier C6288) are lower than for the best XTop results. Also note that the 14th output of the multiplier could not be calculated by the Reference algorithm within an upper node limit of 1 million nodes. A.MORE was the only one capable of calculating the 15th output of the multiplier within that limit. The runtimes are given mainly for completeness but they are not directly comparable due to different machines the measurements have been performed on. But the same interpretation as for the peak sizes also holds for the runtimes.

Circuit	LEs only	LEs and OEs
c0432.mod	5.4	4.6
c0499.mod	13.8	13.0
c0880.mod	0.8	0.6
c1355.mod	9.5	5.5
c1908.mod	10.9	9.0
c2670.mod	9.7	8.4
c3540.mod	561.2	405.1
c5315.mod	344.1	351.2
c7552.mod	5.4	4.2
add32	184.2	144.0
Total	1145.0	945.6
Ratio	1	0.82

Table 1: Operator exchange integration in RBS

6.3 Results of Operator Exchanges

In this subsection we demonstrate the influence of the integration of *Operator Exchanges* in RBS. The synthesis process was applied as described in Subsection 4.2. We comment on some modified ISCAS'85 benchmarks and an adder. The netlist of all benchmarks was modified by using the SIS "eliminate" command [28] to fuse neighboured gates resulting in more operation intense calculations to synthesize a single gate output. The runtime results are shown in CPU seconds in Table 1 as measured on a SUN workstation Ultra 1. The integration of OEs is denoted by the column "LEs and OEs".

One can observe that even the simple integration of OEs in RBS results in a performance improvement. On average the runtimes for the listed benchmarks are improved by about 18%. On a second glance it can be seen that for benchmark "c5315.mod" the runtime is slightly worse. The explanation for this behaviour is simple: In this case the number of non-profitable transformations is very high (approximately hundred percent) and the overhead of extra runtime spent for the operands/operators-constellation evaluations gets obvious.

The effect on Peak sizes is currently under investigation.

7 Conclusions

In this paper we discussed *Reordering Based Synthesis* as a method for performing synthesis operations on *Decision Diagrams*. This outlines an alternative method for the implementation of DD packages. We pointed out several advantages of this new idea and discussed the occuring difficulties resulting from an extension of the concept from BDDs to more general DDs, like e.g. OKFDDs.

An extension of the approaches in [19, 20] has been presented that allows the exchange of *Operator Variables* resulting in a much finer granularity. Experimental results have been given that underline the efficiency of our approach with respect to runtime behaviour and space requirements.

It is focus of current work to also implement the presented technique for word-level DDs, like (K)*BMDs, that have shown to be well-suited for verification of arithmetic functions.

Acknowledgement

The authors like to thank Martin Keim for helpful discussions about operator exchange implementations.

References

[1] H. Andersen and H. Hulgaard. Boolean expression diagrams. In *LICS*, 1997.

[2] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Prado, and F. Somenzi. Algebraic decision diagrams and their application. In *Int'l Conf. on CAD*, pages 188–191, 1993.

[3] B. Becker and R. Drechsler. How many decomposition types do we need? In *European Design & Test Conf.*, pages 438–443, 1995.

[4] B. Becker and R. Drechsler. Decision diagrams in synthesis - algorithms, applications and extensions -. In *VLSI Design Conf.*, pages 46–50, 1997.

[5] B. Becker, R. Drechsler, and R. Enders. On the computational power of bit-level and word-level decision diagrams. In *ASP Design Automation Conf.*, pages 461–467, 1997.

[6] B. Becker, R. Drechsler, and R. Werchner. On the relation between BDDs and FDDs. *Information and Computation*, 123(2):185–197, 1995.

[7] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conf.*, pages 40–45, 1990.

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

[9] R.E. Bryant. Binary decision diagrams and beyond: Enabeling techniques for formal verification. In *Int'l Conf. on CAD*, pages 236–243, 1995.

[10] Y.-A. Chen, B. Yang, and R.E. Bryant. Breadth-first with depth-first bdd construction: A hybrid approach. Technical report, CMU-CS-97-120, 1997.

[11] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi terminal binary decision diagrams: An efficient data structure for matrix representation. In *Int'l Workshop on Logic Synth.*, pages P6a:1–15, 1993.

[12] E.M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams - overcoming the limitations of MTBDDs and BMDs. In *Int'l Conf. on CAD*, pages 159–163, 1995.

[13] E.M. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with application to technology mapping. In *Design Automation Conf.*, pages 54–60, 1993.

[14] R. Drechsler. Verification of multi-valued logic networks. In *Int'l Symp. on multi-valued Logic*, pages 10–15, 1996.

[15] R. Drechsler, B. Becker, and S. Ruppertz. K*BMDs: A new data structure for verification. In *European Design & Test Conf.*, pages 2–8, 1996.

[16] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski. Efficient representation and manipulation of switching functions based on Ordered Kronecker Functional Decision Diagrams. In *Design Automation Conf.*, pages 415–419, 1994.

[17] R. Drechsler, M. Theobald, and B. Becker. Fast OFDD based minimization of Fixed Polarity Reed-Muller expressions. In *European Design Automation Conf.*, pages 2–7, 1994.

[18] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level synthesis. In *European Conf. on Design Automation*, pages 50–54, 1991.

[19] A. Hett, R. Drechsler, and B. Becker. MORE: Alternative implementation of BDD packages by multi-operand synthesis. In *European Design Automation Conf.*, pages 164–169, 1996.

[20] A. Hett, R. Drechsler, and B. Becker. Fast and efficient construction of BDDs by reordering based synthesis. In *European Design & Test Conf.*, 1997.

[21] J. Jain, A. Narayan, C. Coelho, S.P. Khatri, A. Sangiovanni-Vincentelli, R.K. Brayton, and M. Fujita. Decomposition techniques for efficient robdd construction. In *FMCAD*, 1996.

[22] D.M. Miller. Multiple-valued logic design tools. In *Int'l Symp. on multi-valued Logic*, pages 2–11, 1993.

[23] S. Minato. Zero-suppressed BDDs for set manipulation in combinational problems. In *Design Automation Conf.*, pages 272–277, 1993.

[24] H. Ochi, N. Ishiura, and S. Yajima. Breadth-first manipulation of SBDD of boolean functions for vector processing. In *Design Automation Conf.*, pages 413–416, 1991.

[25] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Int'l Workshop on Logic Synth.*, pages 5b:5.11–5.20, 1995.

[26] R.K. Ranjan, J.V. Sanghavi, R.K. Brayton, and A. Sangiovanni-Vincentelli. High performance BDD package based on exploiting memory hierarchy. In *Design Automation Conf.*, pages 635–640, 1996.

[27] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 42–47, 1993.

[28] E. Sentovich, K. Singh, L. Lavagno, Ch. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vicentelli. SIS: A system for sequential circuit synthesis. Technical report, University of Berkeley, 1992.

[29] F. Somenzi. *CUDD: CU Decision Diagram Package Release 1.1.1.* University of Colorado at Boulder, 1996.

[30] A. Srinivasan, T. Kam, S. Malik, and R.E. Brayton. Algorithms for discrete function manipulation. In *Int'l Conf. on CAD*, pages 92–95, 1990.