Published at IEEE European Design & Test Conference (ED&TC'97), Paris

Fast and Efficient Construction of BDDs by 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

We present a new approach to symbolic simulation with BDDs. Our method uses Reordering Based Synthesis (RBS) which allows the integration of dynamic variable ordering (even) within a single synthesis operation (e.g. an AND-operation). Thus, huge peak sizes during the construction can often be avoided, and we obtain a method that, with no penalty in runtime, is more memory efficient than traditional ITE operator based symbolic simulation.

The results are confirmed by experiments on a large set of benchmarks: We give a comparison to previously published approaches and also consider some industrial benchmarks which are known to be hard to handle.

1 Introduction

Ordered Binary Decision Diagrams (OBDDs) [5] are the state-of-the-art data structure in CAD systems for the representation and manipulation of Boolean functions. 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 [7]. Manipulation algorithms in all packages realized until recently (see e.g. [3, 14, 13, 21, 11]) 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 [15] and use of memory system hierarchy [17] are often more powerful for very large applications.

In [12] a new approach for the implementation of BDD packages has been presented that is based on *Dynamic Variable Reordering* [10, 18]. Therefore, we call the corresponding method for realizing synthesis operations *Reordering Based Synthesis* (RBS).

In this paper we present a new method for symbolic simulation based on RBS. We first demonstrate that the use of terminal cases can also be integrated in RBS. By the use of terminal cases and the extension of RBS to the computation of complex gates (as whole single-output PLAs) in one step we improved on the algorithm presented in [12] by a factor of two with respect to runtime. Furthermore, the approach is capable of interrupting the synthesis at any position without any loss of information. Thus, it is much more flexible than recursive synthesis.

Our experiments underline the quality of the approach. We consider symbolic simulations for the ISCAS'85 and some industrial benchmarks (known to be hard for BDDs). The results show that our new method has much lower memory requirements compared to ITE-based methods 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 run.

The paper is structured as follows: In Section 2 basic notations and definitions are reviewed that are important for the understanding of the paper. Section 3 presents our approach to symbolic simulation using BDDs. Terminal cases and complex gate handling are introduced. Experimental results are presented in Section 4. Finally the results are summarized.

2 Preliminaries

2.1 Binary Decision Diagrams

We briefly provide the essential definitions and properties of BDDs. For more details see [5, 6].

As well-known each Boolean function $f : \mathbf{B}^n \to \mathbf{B}$ can be represented by a *Binary Decision Diagram*

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

(BDD) [5], i.e. a directed acyclic graph where a Shannon decomposition is carried out in each non-terminal node.

A BDD is called *ordered* if each variable is encountered at most once on each path from the root to a terminal node and if the variables are encountered in the same order on all such paths. A BDD is called *reduced* if it does not contain vertices either with isomorphic sub-graphs or with both edges pointing to the same node.

For functions represented by reduced, ordered BDDs (ROBDDs) efficient manipulations are possible [5]. Furthermore, the size[1] of an ROBDD can be reduced if *Complement Edges* (CEs) are used [3]. Then a node is used to represent a function and its complement at the same time. In the following only ROBDDs with CEs are considered and for briefness these graphs are called BDDs.

2.2 Recursive Synthesis

Typically synthesis on BDDs is based on a recursive algorithm. The ternary *If-Then-Else*-operator (ITE) [3] forms the core of recursion based synthesis operations for BDDs. (ITE is normally implemented by depth-first-search, but also breadth-first-search is possible.) ITE is a Boolean function defined for three operands as follows:

$$ite(F, G, H) = F \cdot G + \overline{F} \cdot H$$

ITE can be used to implement all two-variable Boolean operations, e.g. $F + G = ite(F, 1, G)$. The recursive formulation

$$ite(F, G, H) = (v, ite(F_v, G_v, H_v), ite(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}}))$$

determines the computation of the operation. F_v and $F_{\overline{v}}$ denote F evaluated at $v = 1$ and $v = 0$, respectively (the same holds for G and H). The (time and space) complexity for the ITE-function can be given as $O(|F| \cdot |G| \cdot |H|)$ when using an ideal computed table. For binary operations only two operands are non-terminals. Thus, a binary operation has complexity $O(|F| \cdot |G|)$. (For more details see [3].)

2.3 Reordering Based Synthesis

In order to modify the variable ordering of BDDs we can exchange variables in adjacent levels. This method is called *Level Exchange* (LE) [10] in the following. Since an exchange of neighbouring variables is a local operation consisting only of the relinking of nodes in these two levels, this can be done very efficiently. In [18] it was shown that it is even possible to perform LEs as local operations when using BDDs with CEs. LE was first proposed for optimizing the variable ordering of a BDD.

Recently, in [12] a new method for performing synthesis based on LEs has been proposed. The basic idea of this approach can be stated as follows:

Assume that two BDDs F and G of size $|F|$ and $|G|$ are given. They are to be combined to result in the BDD $R = F + G$. Therefore a new variable c that is called *coding variable* is introduced and the BDD $R' = \overline{c} \cdot F + c \cdot G$ is constructed[2]. The construction of R' can be done in constant time if the BDDs F and G are given and c is introduced as the top variable.

Then the coding variable c is shifted to the bottom of the BDD based on LEs and an existential quantification of c is carried out. (Existential quantification of a variable in the bottom level is trivial.) Since during that run a non-reduced BDD can occur a final reduction run has to be carried out. In [12] this algorithm has been denoted as MORE. (For more details see [12].)

As described above for recursive synthesis operations based on ITE a polynomial upper bound of $O(|F| \cdot |G|)$ is known. For RBS only the method has been presented [12], but no upper bounds (neither for runtime nor for space complexity) have been given. In the following we prove that this method has also polynomial *worst case* behavior.

Theorem 1 Let F and G be two BDDs defined over n Boolean variables and $\odot$ any binary operation. Then $F \odot G$ can be computed in space $O(\max(|F|, |G|)^2)$ and time $O(n \cdot \max(|F|, |G|)^2)$ by RBS.

Proof: Without loss of generality we give the proof for the OR-operation between F and G. A coding variable has to be introduced and moved from the top to the bottom. Each intermediate step is done by LE. If the variable directly "jumps" from the first to the i-th position in the ordering the graph size grows at most by a quadratic number [2]. Since this holds for all i the total number of nodes is less than the bound given in the theorem. The bound on the time complexity now follows directly from the fact that LE is an operation linear in the size of the corresponding levels. □

2.3.1 Another side of the same coin

The introduction of a node with coding variable c may also be interpreted in the following way: c defines an *OR function node* representing the OR-operation of its two subfunctions. In this way also function nodes

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

[2]Note: Here we only briefly describe the OR-operation, but the approach can directly be applied for any binary synthesis operation (see also Subsection 2.3.1).

for other operations can be defined. To apply RBS we have to give the rules along which ordinary non-terminal nodes in the decision diagram can be interchanged with function nodes. Function nodes with constant subfunctions can then easily be replaced by the result of the operation. Using this interpretation the concept can also be generalized to other types of decision diagrams [1, 8]. Extensive experiments have to be made to demonstrate the validity of this approach for e.g. word-level DDs. For the purpose of this paper we restrict to BDDs and the application of RBS in symbolic simulation.

3 Reordering Based Synthesis during Symbolic Simulation

One of the main applications for BDDs is the representation of the functionality of a Boolean network. For the computation of the BDDs representing the outputs of a network the circuit is traversed in topological order and at each gate the corresponding BDD operations are carried out. This process is called *symbolic simulation*.

Recursive synthesis is known for long and several sophisticated methods have been developed how to do symbolic simulation efficiently. Also, the computation of BDD operations is often speeded up due to the early detection of terminal cases.

In this section we show that similar techniques can be used for RBS. We introduce the concept of terminal cases (that significantly differs from the ITE approach). Additionally, we study new opportunities for symbolic simulation that result from this new synthesis based on reordering.

3.1 Terminal Cases

To prevent unnecessary operations, i.e. *Level Exchanges* (LEs), it is useful to implement the detection and solving of terminal cases similar to recursion based synthesis algorithms. Whenever the creation of a node with a *coding variable* is needed that has a son that is equal to a terminal node, we can immediately return the resulting node instead of creating a new one.

Example 1 The simplest situation of a terminal case is given in Figure 1. If the *high*-pointer of a *coding variable node* points to the terminal 1 the coding variable can be directly substituted by constant 1.

Thus, the *coding variable* is eliminated in this BDD part and needs not be shifted to the bottom. If no more nodes with the current *coding variable* are existent, the reduction run can be applied. Therefore, the terminal case handling speeds up the process of RBS.

As mentioned before, the terminal case in Figure 1 only represents the simplest case. In the following

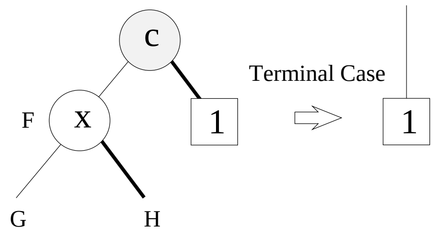

Figure 1: Simple example for Terminal Case

we describe several aspects that have to be addressed until the full power of terminal cases can be used.

Due to the usage of CEs terminal cases for RBS are only applicable with penalties. Before discussing this in more detail we give a brief example to show that CEs for coding variables must be handled in a different way than usual.

Example 2 We consider the situation of *Case A* and *Case B*, depicted in the upper left box of Figure 2. (A dot on a line denotes a CE.) In both cases the computation can be terminated, since the result is already determined although the *coding variable* has not reached the bottom level (of BDD F):

> One can observe that the complement marks on the top edges influence the determination of the terminal case since in *Case A* we get
>
> $$\exists c : (\overline{c} \cdot F + c \cdot 1) = F + 1 = 1$$
>
> and in *Case B* we get (after reversion of the complement mark normalization)
>
> $$\exists c : (\overline{c} \cdot \overline{F} + c \cdot \overline{1}) = \overline{F} + 0 = \overline{F}.$$
>
> Note that the solution of *Case A* is **not** the complement of the solution of *Case B*:
>
> $$1 \neq \overline{\overline{F}} = F$$

Remark: Obviously all terminal cases are detectable *during* LE operations although *not decidable* at this point. This is due to the fact that the information of CEs on the top edges at this point is not available without a traversal of the graph that is much too time consuming (see right box of Figure 2).

To allow terminal cases nevertheless we either have to store (at most two) possible terminal case solutions or must avoid CEs atop of *coding variable nodes*.

We implemented both approaches and in the following briefly describe their advantages and disadvantages. (All (technical) details are left out due to page limitation.)

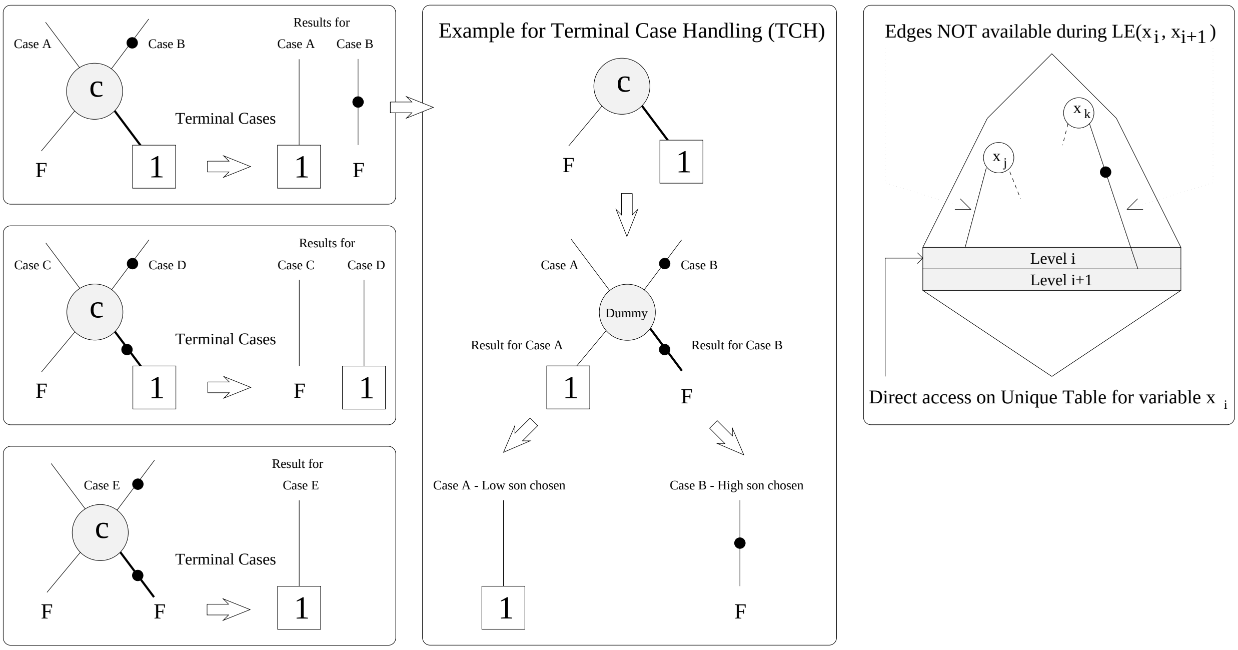

Figure 2: Terminal Cases

Store all terminal case solutions:
This strategy can be applied efficiently by manipulating the node where a terminal case was detected. We therefore exchange the *coding variable* with a *dummy variable* and store the solutions as sons of the node (see middle box in Figure 2 for the handling of *Case A* and *Case B*). Since the incoming edges are not available the decision of terminal cases can not be made during each level exchange when using CEs. The removal of dummy nodes is then performed during the reduction run.

Avoid CEs atop of coding variables:
This can be achieved by omitting the application of normalization rules whenever we need to create a *coding variable node*. By doing so, we will never produce *Case B* and therefore can detect *and* solve all terminal cases immediately (since no decision for alternatives has to be made). The drawback of this method is that all levels atop of the current *coding variable* cannot make use of complement marks and we temporarily loose canonicity. However, after the shift phase that "collects" the marks, the reduction run "spreads" the marks again and restores canonicity.

Although both strategies proved to have similar runtime behaviour, the "CE avoidance technique" was chosen as favourite method, since its memory performance is superior thanks to the immediate terminal case solving.

3.2 Complex Gates

In contrast to recursive synthesis the RBS can handle an "arbitrary" number of operands at the same time instead of only two. We will demonstrate by means of the following example that large advantages may result from this.

Example 3 Let $\mathcal{F}$ be an AND gate with three inputs f, g and h that occurs during symbolic simulation of a circuit. BDD packages based on recursive synthesis have to compute:

$$(f \cdot g) \cdot h,\ f \cdot (g \cdot h) \text{ or } (f \cdot h) \cdot g$$

The order in which the calculation is performed largely influences the number of nodes that are needed during the computation, e.g. if $f \cdot g$ is computed first, but $h = 0$. In this case the result $f \cdot g$ (which might be large) is computed first even though the results of the AND gate is 0. RBS detects this trivially.

This was only a simple case. If f, g and h are chosen as follows:

$$f = (yz + \bar{y}z)f'$$

$$g = (yz + y\bar{z})g'$$

$$h = (\bar{y}z + y\bar{z})h'$$

with the property that f', g' and h' are independent of y and z and have pairwise large BDDs when combined by AND, i.e. the result is of quadratic size (quadratic in the size of the input BDDs). Then it is straightforward to see that **each** recursive synthesis method using only two operands needs quadratic runtime, while RBS can detect the result again as constant 0 in constant time due to the terminal cases.

Thus, for more complex circuits it is more useful to describe the logical behaviour by not only using standard cells. The functionality of these gates can be expressed by a PLA description, as e.g. used in BLIF.

Obviously the BDD representing the output function of such a gate can not directly be calculated by means of classical binary synthesis operations since the number of operands is greater than two. Thus, the calculation has to be done *sequentially*.

In our approach based on RBS we compute one gate (of "arbitrary" number of inputs) in one step (using two RBS calls) and thus operate highly in *parallel*. (The details are left out due to page limitation.) The effect of this method will be demonstrated in the next section by experiments.

3.3 Combining Synthesis and Dynamic Variable Ordering

Before we describe the underlying algorithm of our symbolic simulation procedure we want to underline another very important aspect of RBS: The synthesis process can be interrupted at any point **without** loss of intermediate results. Thus, there arise new possibilities for the dynamic minimization during the construction of BDDs; the dynamic reordering can even support the synthesis operation, if it is guaranteed that the coding variables are never shifted towards the top.

Now 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 process as well as a minimization without restrictions or loss of intermediate results. *Supervising heuristics,* having a *finer granularity* than it is possible when the construction process is relying on recursion based synthesis algorithms, are used to control the construction process and initiate minimization procedures in appropriate situations:

Recursive synthesis based packages stop the computation after a synthesis operation, having a worst case complexity that is quadratic in the size of the graph. Otherwise, already computed results have to be thrown away or the reordering process can only be performed in a restricted region of the BDD. In our approach the synthesis can be stopped after each reordering step. Therefore we get a worst case complexity linear in the size of the current level (which is considerably lower than being quadratic in the graph size). Additionally, *none* of the results get lost.

```
construction(circuit, boundary, step, windows) {
  Determine topological order for symbolic simulation;

  for each (gate of circuit in topological order) do {
    Construct BDD for gate output in parallel using build;
  }
  return resulting BDDs for primary outputs;
}

build(operand_list consisting of op BDDs) {
  Build coding tree by use of m := ⌈log2(op)⌉ coding variables;

  for each coding variable do {
    while (coding variable not in bottom level) do {
      if (BDD size after levelexchange < boundary) {
        Shift coding variable one level downward
          by means of levelexchange;
      } else {
        Perform minimization on level windows;
        if (no improvement made)
          boundary := boundary + step;
      }
    }
    Perform existential quantification and reduce result tree;
  }
  return result tree;
}
```

Figure 3: Sketch of construction heuristic

To demonstrate the resource sparing we applied an expandable *supervising heuristic* that controls the BDD construction during symbolic simulation according to three external parameters:

- **Boundary** determines the number of BDD nodes that must be reached in order to perform dynamic minimizations.
- **Step** denotes the value by which *Boundary* will be increased if the previous minimization was not capable of reducing the number of BDD nodes.
- **Windows** determines the number of half-overlapping level windows (covering the regions above and below the currently shifted coding variable) on which the minimization algorithms operate.

Notice once more that due to the finer granularity of our approach we can immediately react to an impending upper size boundary crossing. Additionally no results have to be recomputed after performing intermediate minimizations.

A sketch of the algorithm is given in Figure 3. The parameters are dynamically applied dependent on the "hardness" of the benchmark.

Circuit	BDD Size	MORE	A.MORE
c0432	31 177	21.5	6.8
c0499	40 657	12.6	5.2
c0880	8 654	1.4	0.6
c1355	57 945	27.1	10.9
c1908	14 073	12.9	2.8
c2670	9 771	5.1	1.7
c3540	150 672	114.0	72.2
c5315	84 267	31.5	8.9
c7552	8 481	8.5	2.6
Total		234.6	111.7
Ratio		1	0.47

Table 1: Comparison to MORE

4 Experimental Results

In this section we present a set of experimental results. All methods presented above have been implemented and integrated in the program *A.MORE*. The experiments were carried out on a *SUN SPARCstation 20* with 256 MByte of main-memory and all runtimes are reported in CPU seconds. For all experiments we applied *interleaving* [9] to determine an initial variable ordering.

In the following three subsections we report three different types of experiments. We first study the effect of terminal cases and complex gate handling by a comparison to the approach from [12].

We then compare our results to the CUDD package [21]. The comparison is done for a large number of benchmarks that are known to be hard for BDDs.

Finally, we demonstrate the efficiency of our approach by considering the peak size that occurs during symbolic simulation, i.e. the maximal number of nodes needed during a run.

4.1 Terminal Cases and Complex Gates

Table 1 shows the BDD construction times needed for the ISCAS'85 benchmarks [4] for the MORE approach from [12]. The results of A.MORE are given in the last column. For this experiment no dynamic variable reordering for optimization of the BDD size has been used. As can easily be seen the methods presented in this paper speed up the synthesis based on reordering by more than a factor of two on average.

4.2 Efficient Construction

In a second series of experiments we consider symbolic simulation for the ISCAS'85 benchmarks and some industrial circuits.

Remark: Since some of the runtime measurements of the mentioned publications were made on DEC Alpha stations with 256 MByte of main-memory instead of SUN workstations, we recalculated these values according to the factorial machine speed difference[3], allowing a fair comparison of all runtime results. Since the tables are, due to missing values of foreign measurements, partly incomplete, the fields for missing entries were filled with the keyword 'n/a' (not available). As a consequence, *Totals* are only summarized for lines that are complete for all competitors. However, the *Ratio* of a column was always determined for the set of all lines that were available relative to the corresponding values of the relevant columns.

The results for the ISCAS'85 benchmarks are given in Table 3. The second column denoted as *Best-Ever* gives the results from [16]. The results for the recursive synthesis based package CUDD [21] are given in column *CUDD*. (The runtimes are taken from [20].) A.MORE denotes the approach presented in this paper.

The initial parameter setting of our supervising heuristic was chosen as shown in Table 2. The parameter **Windows** turned out to be most effective when set on 3 half-overlapping windows. Variations in these parameters can take large influence on the runtimes as well as on the final sizes and peak sizes. As a general rule we can expect smaller runtimes while getting larger BDDs when increasing the boundary values. On the other hand, smaller BDDs can be acquired at higher runtime costs when choosing small initial paramater values. We made a decision for the average in this trade-off.

As can be seen our approach has about the same runtime behaviour as the ITE based package CUDD. A.MORE is on average about 9% faster. Additionally the sizes are often much better. In comparison to the *Best-Ever* final node sizes A.MORE achieved values that are only about 19% larger. For CUDD the size is 28% larger even though the runtime is higher than that of A.MORE. (Notice that *Best-Ever* is a 'best of' result selection out of several runs, while the results given in the CUDD and A.MORE columns are measured with a fixed parameter set.)

The advantage of our approach becomes even clearer if we consider larger examples, i.e. circuits with a large number of inputs and outputs (latches are handled as inputs and outputs) and industrial circuits that are known to be hard for BDDs. The results are given in Table 4. The first five columns denote the circuit name and some characteristics. For the industrial benchmarks A.MORE outperforms the CUDD package clearly with respect to the number of nodes. Even though A.MORE was not designed for BDD minimization, i.e. the sifting method in [16] is much cleverer

[3] According to the geometric mean of SPEC runtime benchmarks the specific DEC Alpha turned out to be about 2.7 times faster than our SUN workstation.

Circuit	Final BDD Size	Boundary	Step
easy	$< 10\,000$	1 000	1 000
medium	intermediate	10 000	1 000
hard	$> 100\,000$	100 000	10 000

Table 2: Parameter Setting

Circuit	Best-Ever		CUDD		A.MORE	
	Size	Time	Size	Time	Size	Time
c0432	1 210	3.2	1 259	3.8	1 210	2.2
c0499	25 866	429.5	29 686	161.4	32 137	75.1
c0880	4 083	36.6	7 065	15.8	4 889	21.7
c1355	25 866	723.2	29 622	400.9	32 857	384.0
c1908	5 708	35.0	6 742	34.2	6 027	47.7
c2670	2 008	48.5	10 375	55.8	4 039	32.1
c3540	23 828	149.8	24 023	139.8	23 843	225.0
c5315	2 104	160.7	2 565	14.6	2 528	11.6
c7552	3 730	186.7	9 582	83.5	5 194	36.0
Total	94 403	1 773.2	120 919	909.8	112 724	835.4
Ratio	1	1	1.28	0.51	1.19	0.47

Table 3: Final Node Count

than the one applied here, A.MORE was able to significantly improve on one of the larger benchmarks (see *s15850*).

4.3 Peak Size

Up to now only the BDD size of the functions representing the outputs of the circuits has been considered. 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 BDD size is very small (see e.g. tautology checking).

We report on experiments carried out with A.MORE. As a comparison we consider results from [20], [19]. The results are given in Table 5. The peak sizes and the final size needed by CUDD/SIS[4] and A.MORE are denoted in the last four columns. (Notice, that these results were obtained by a run of the algorithm with standard parameter settings and that the runtime has already been given in the preceeding tables.) As can be seen A.MORE nearly never needs much more nodes during the computation than for the representation of the final result (the 'peak size-final size' ratio is only 1.33 in comparison to more than 3 for CUDD/SIS). Thus, it clearly outperforms the recursive synthesis approach.

[4]Note that the peak size measurements for CUDD were done within a SIS-environment with a different garbage-collection-scheme in comparison to CUDD-stand-alone-runs.

5 Conclusions

In this paper a new method for BDD construction based on *Reordering Based Synthesis* has been presented. The use of terminal cases and complex gate handling has improved the performance by a factor of two in comparison to the originally presented method.

Experimental results for large circuits have demonstrated the advantages of our approach in comparison to previously published methods.

In particular, the novel reordering based technique is much more memory efficient than previously known techniques what has been demonstrated by measuring the peak size during symbolic simulation.

Acknowledgement

The authors like to thank Ingo Wegener for providing Example 3 and Ellen Sentovich for the industrial benchmarks and helpful support.

References

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

[2] B. Bollig, M. Löbbing, and I. Wegener. Simulated annealing to improve variable orderings for OBDDs. In *Int'l Workshop on Logic Synth.*, pages 5b:5.1–5.10, 1995.

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

Circuit	PI	PO	Gates	Latches	Best-Ever		CUDD		A.MORE	
					Size	Time	Size	Time	Size	Time
s15850	77	150	9 786	534	10 111	785.1	n/a	n/a	2 253	244.0
s38417	38	304	19 407	1 426	266 772	22 752.7	n/a	n/a	584 467	39 900.1
s38584	28	106	22 397	1 636	13 303	1 037.1	n/a	n/a	16 397	502.0
indust	97	58	2 638	231	n/a	n/a	n/a	n/a	22 572	2 336.3
snecma	23	5	727	70	n/a	n/a	646 327	1 823.4	115 377	1 300.3
sequenceur	55	98	633	121	n/a	n/a	347 521	957.8	334 318	15 100.1
tcintnc	19	20	449	90	n/a	n/a	1 431	0.8	686	1.4

Table 4: Large circuits

Circuit	CUDD/SIS		A.MORE	
	Peak	Final	Peak	Final
c0432	5 273	1 259	2 000	1 210
c0499	55 769	29 686	41 000	32 137
c0880	16 093	7 065	8 000	4 889
c1355	121 406	29 622	40 000	32 857
c1908	21 163	6 742	11 000	6 027
c2670	21 239	10 375	8 000	4 039
c3540	129 000	24 023	49 000	23 843
c5315	11 001	2 565	7 000	2 528
c7552	49 012	9 582	7 000	5 194
s15850	n/a	n/a	21 000	2 253
s38417	n/a	n/a	590 000	584 467
s38584	n/a	n/a	20 000	16 397
indust	n/a	n/a	40 000	22 572
snecma	n/a	646 327	374 000	115 377
sequenceur	n/a	347 521	370 000	334 318
tcintnc	n/a	1 431	1 000	686
Total	429 956	122 350	173 000	112 724
Ratio	3.51	1	1.33	1

Table 5: Peak sizes and final sizes

[4] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational circuits and a target translator in fortran. In *Int'l Symp. Circ. and Systems, Special Sess. on ATPG and Fault Simulation*, pages 663–698, 1985.

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

[6] R.E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM, Comp. Surveys*, 24:293–318, 1992.

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

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

[9] H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 38–41, 1993.

[10] 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.

[11] A. Hett, R. Drechsler, and B. Becker. *The DD Package PUMA - An Online Documentation.* http://www.informatik.uni-freiburg.de/FREAK/puma/puma.html, 1996.

[12] 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.

[13] D.E. Long. *Long-Package Sun Release 4.1 Overview of C Library Functions.* 1993.

[14] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagrams with attributed edges for efficient boolean function manipulation. In *Design Automation Conf.*, pages 52–57, 1990.

[15] 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.

[16] 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.

[17] 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.

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

[19] E. Sentovich. Personal communication, 1996.

[20] E. Sentovich. A brief study of BDD package performance. In *FMCAD*, 1996.

[21] F. Somenzi. *CUDD Release 1.1.1.* 1996.