

# Combinational Logic-Level Verification using Boolean Expression Diagrams

Henrik Hulgaard

Poul Frederick Williams

Henrik Reif Andersen

Department of Information Technology  
Building 344  
Technical University of Denmark  
DK-2800 Lyngby, Denmark  
e-mail: {henrik,pfw,hra}@it.dtu.dk

## Abstract

*Boolean Expression Diagrams (BEDs) is a new data structure for representing and manipulating Boolean functions. BEDs are a generalization of Binary Decision Diagrams (BDDs) that are capable of representing any Boolean circuit in linear space and still maintain many of the desirable properties of BDDs.*

*This paper demonstrates that BEDs are well suited for solving the combinational logic-level verification problem which is, given two combinational circuits, to determine whether they implement the same Boolean functions. Based on all combinational circuits in the ISCAS 85 and LGSynth 91 benchmarks, we demonstrate that BEDs outperform both standard BDD approaches and the techniques specifically developed to exploit structural similarities for efficiently solving the problem.*

## 1 Introduction

The combinational logic-level verification problem is to determine whether two given combinational circuits implement the same Boolean function. Let  $f$  and  $g$  represent the Boolean functions of the two combinational circuits. Clearly, verifying that  $f = g$  is an instance of TAUTOLOGY which is a well-known co-NP-complete problem [11]. The traditional approach to proving such a tautology is to build the Reduced Ordered Binary Decision Diagram [4] (ROBDD) for  $f$  and  $g$ , and then check that the ROBDDs are identical. Due to the canonicity of ROBDDs, it is efficient to check whether the ROBDD for  $f$  is equal to that of  $g$ . However, there are two main problems with the ROBDD approach. One problem is that the size of the ROBDDs for  $f$  and  $g$  may become very large. For example, it can be shown that the multiplication function has no sub-exponential ROBDD representation. The other problem is that ROBDDs cannot exploit structural similarities between the two Boolean circuits. As a worst case scenario, consider the case where the two circuits are identical. In this case, the full ROBDD for both  $f$  and  $g$  is constructed before the identity of the circuits is verified. In typical applications, one circuit is obtained from the other, so it is essential to be able to exploit structural similarities in order to perform an efficient tautology check.

This paper suggests an alternative data structure called Boolean Expression Diagrams (BEDs). BEDs are an extension of BDDs that allow any Boolean circuit to be represented in linear space, i.e., BEDs are exponentially more succinct than BDDs, and furthermore BEDs can recognize and share identical sub-expressions. These properties make BEDs a promising data structure for solving the combinational logic-level verification problem. In this paper, we will support this claim by verifying all combinational circuits in the ISCAS 85 and LGSynth 91 benchmarks. In most cases, the BED approach outperforms all existing techniques.

## 1.1 Related Work

BEDs are closely related to a new way of constructing ROBDDs, called MORE [13, 14]. MORE is based on the observation that the BDD for  $f \vee g$  can be constructed by introducing a new variable  $x$  and implicitly existentially quantify  $x$  since  $\exists x. x \rightarrow f, g = f \vee g$ . MORE constructs the BDD by moving  $x$  towards the terminal vertices using the level exchange operation [10]. BEDs can be seen as extending this idea to allow arbitrary operators and allowing these operators to remain in the graph. This makes it possible to include operator reduction rules and develop new ROBDD synthesis algorithms (e.g., *up\_one*) which are essential for obtaining the runtimes presented in this paper.

Other extensions of ROBDDs include using other types of decomposition rules [9, 15], relaxing the variable ordering restriction [12, 22], and extending the domains and/or codomains to integers instead of Booleans [2, 5, 7]. However, none of these extensions are powerful enough to represent all Boolean circuits in polynomial space, and thus, they are all exponentially less compact than BEDs.

Several methods have been developed for solving the logic-level verification problem efficiently by exploiting structural similarities between the two circuits to be verified. These techniques all attempt to identify equivalent nodes in the two circuits. Brand [3] uses a test generator for determining whether one node can replace another while Kunz *et al.* [16, 17, 18, 20] use a technique called recursive learning to derive equivalences between nodes. BEDs only have a limited capability to find such equivalences (since only simple operator reduction rules are included) but still the performance of BEDs is significantly better. BEDs would benefit from information about equivalent nodes. This would immediately reduce the size of the BED and make possible even further identifications of nodes.

Finally, Eijk and Janssen [23] discuss how structural similarities can be exploited when using ROBDDs. The technique is similar to Brands [3] but uses ROBDDs instead of a test generator to determine whether two internal nodes are equivalent.

## 1.2 Overview

The paper is organized as follows. Section 2 presents the BED data structure and gives a short overview of the algorithms used to show that a BED is a tautology. Section 3 describes the experimental results, verifying a large number of combinational circuits from the ISCAS 85 and LGSynth 91 benchmarks. Finally, section 4 summarizes the contributions of this paper.

# 2 Boolean Expression Diagrams

This section gives a brief introduction to BEDs. For a detailed description, see [1].

## 2.1 The BED Data Structure

A BED is a BDD extended with operator vertices, that is, a BED is a directed acyclic graph with three types of vertices: A *terminal vertex*  $v$  has as attribute a value  $value(v) \in \{0, 1\}$ , a *variable vertex*  $v$  has as attributes a variable  $var(v)$ , and two children  $low(v)$  and  $high(v)$ , and an *operator vertex*  $v$  has as attributes a binary Boolean operator  $op(v)$ , and two children  $low(v)$  and  $high(v)$ .

Variable vertices correspond to the *if-then-else* operator  $x \rightarrow f_1, f_0$  defined by

$$x \rightarrow f_1, f_0 = (x \wedge f_1) \vee (\neg x \wedge f_0).$$

Operator vertices correspond to their respective Boolean connectives. Figure 1 shows the BED for the equation

$$(x \vee y \vee z) \wedge (x \vee y) \Leftrightarrow (x \vee y).$$

The low-edges are drawn using dashed lines and all edges are implicitly directed downwards. Notice that vertices representing the same Boolean sub-function are shared.

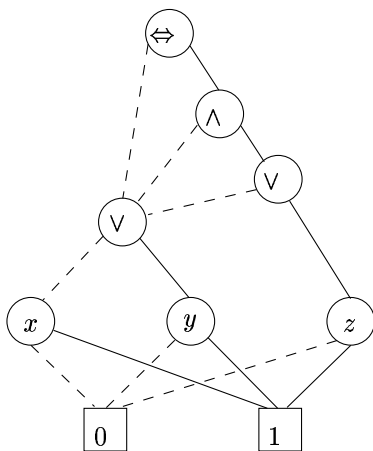


Figure 1: A BED for a simple equation.

There is a straightforward linear size transformation from combinational circuits to BEDs: Each input  $x$  is replaced by a variable vertex  $u$  with  $var(u) = x$  and each  $k$ -input gate is replaced by a tree of  $k - 1$  operator vertices encoding the Boolean function of the gate.

We observe that BEDs are *exponentially more succinct* than ROBDDs. An example of this is the multiplier function. Bryant [4] showed that for all orderings, the multiplier function requires ROBDDs of exponential size. However, since there are combinational circuits implementing this function using only a quadratic number of gates [8] (and even less), there exists a BED of this size representing it.

## 2.2 BED construction

BED nodes are created by a constant-time operation  $mk$ . A call to  $mk$ :

$$v \leftarrow mk(\alpha, l, h)$$

returns a BED vertex  $v$  with “tag”  $\alpha$  (either  $op(v)$  or  $var(v)$ ), low-child  $l$  and high-child  $h$ . This function makes sure that no *redundant* vertices are created, i.e., two vertices representing isomorphic sub-BEDs and vertices that are unnecessary for obvious reasons.

For operator vertices one can add more checks in order to reuse vertices, thereby reducing the size of the BED. An immediate optimization is to extend *mk* to look for operator vertices that differ from the one wanted only by exchanging low and high, by a negation, or by a combination of both.

Going a step further, considering two vertices at a time, we can eliminate all negations below binary operators since for all binary operators *op* there exists another operator *op'* with  $op'(x, y) = op(\neg x, y)$ .

Finally, taking the identity of vertices into account allows us to exploit equivalences like the *absorption laws*, e.g.,  $x \vee (x \wedge y) = x$ . There are  $16^n$  combinations of  $n$  binary Boolean operators, thus it is feasible to tabulate them all for  $n$  up to three or four. Choosing  $n = 3$  seems like a natural choice since such a table would include equivalences such as the distributive laws. These reductions are essential for the performance of the BED verification since they allow non-isomorphic sub-graphs to be identified.

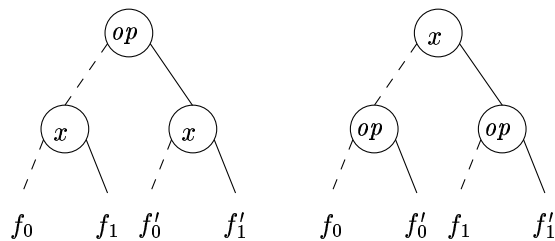
### 2.3 BED operations

It is easily seen that a BDD is simply a BED without operators. This suggests a strategy for converting BEDs into ROBDDs: gradually eliminate the operators, keeping all the intermediate BEDs functionally equivalent. We have developed two very different ways of operator elimination.

The first elimination algorithm, *up\_one*( $x, u$ ), pulls a single variable  $x$  up to the root  $u$  using a sequence of *up*-steps. Let *op* be an arbitrary binary Boolean operator, let  $x$  be a Boolean variable, and let  $f_i$  and  $f'_i$  ( $i = 0, 1$ ) be arbitrary Boolean expressions. It is simple to verify that

$$(x \rightarrow f_1, f_0) \text{ op } (x \rightarrow f'_1, f'_0) = x \rightarrow (f_1 \text{ op } f'_1), (f_0 \text{ op } f'_0). \quad (1)$$

This identity,<sup>1</sup> illustrated in figure 2, is used to move the variable  $x$  above the operator *op*. Repeatedly using *up\_one*( $x, u$ ), we can move all variables up past all the operators, which



**Figure 2:** The left and right BED are equivalent.

makes the operators disappear (they get evaluated at the terminal vertices). The run-time of *up\_one*( $u, x$ ) is linear in the size of the BED rooted at  $u$ . *Up\_one* has some distinct properties. In fortunate cases, a BED is converted into an ROBDD after moving just a few variables up. In this process, equivalent sub-BEDs, potentially containing operator vertices, are identified. This is quite unlike traditional ROBDD construction where all operators are converted in depth-first order into ROBDDs. In particular, an ROBDD is

<sup>1</sup>Equation (1) also holds if the operator vertex *op* is a variable vertex. In that case, the *up* operation is identical to the level exchange operation [10] typically used in ROBDDs to dynamically change the variable ordering [21].

constructed for *each sub-expression*. If the result is small and the intermediate ROBDDs are large, *up\_one* could be an attractive alternative.

The other elimination algorithm, *up\_all(u)*, is a generalization of Bryant's *apply*-operator. Construction of ROBDDs from a Boolean expression using recursive calls of *apply* suggests a bottom up conversion of BEDs into ROBDDs. *Up\_all(u)* mimics recursively applying *apply*: given two BDDs  $l$  and  $h$ ,

$$up\_all(mk(op, l, h)) = apply(op, l, h).$$

The running time of *up\_all* is bounded by a constant factor of *apply*, and recent research [14] suggests that *up\_all(u)* in practice is more efficient than *apply*.

### 3 Experimental Results

We demonstrate the efficiency of the BED data structure by verifying a number of combinational logic circuits. We verify circuits from the ISCAS 85 and LGSynth 91 benchmarks<sup>2</sup>. All runtimes are reported in CPU seconds for a Sun Ultra-SPARC 1 with 256 MB of memory. The total number of BED vertices  $N_{total}$  used for the verification is also reported. In a few examples (marked with †), garbage collection is used to reuse memory, and in those cases the number  $N_{total}$  represents the maximum rather than the total number of BED vertices. The C source code for the used BED package and the benchmark circuits with scripts to verify them can be obtained from <http://www.it.dtu.dk/~henrik/bed/>.

Multi-level combinational circuits are much harder to verify than two-level circuits. We ran all 40 two-level circuits in the LGSynth 91 benchmark through the *espresso* optimization program and verified the result against the original description. The verification takes less than one CPU second for each of the 40 circuits. Thus, in the following we only report results for multi-level combinational circuits.

In carrying out the experiments, we used the following strategy to prove (or disprove) equivalence of two outputs: *up\_all* was used on an arbitrary variable ordering. If this failed, we used *up\_one* for each variable in the support of the output. The sequence of these variables were determined in the order they are encountered during a depth-first search from the output.

#### 3.1 ISCAS 85 Benchmarks

Nine of the combinational circuits in the ISCAS 85 benchmark suite exist in a redundant and a non-redundant version. We consider the problem of determining whether these two versions implement the same functionality, corresponding to an equivalence check for each pair of outputs. The results are shown in table 1. It is observed that except for c3540, all circuits are verified in only a few CPU seconds and using less than half a million BED nodes, which is less than 12 MB of main memory.

Kunz *et al.* [18] have run the same verification on a comparable machine (a Sun SPARC 10) using a technique based on recursive learning combined with ROBDDs. The second to last column in table 1 shows the ratio between the running times of their technique (denoted CPU[18]) to the BED runtimes. The results obtained using BEDs are up to three orders of magnitude better and BEDs perform relatively better on the larger circuits. The last column of table 1 shows the ratio between the runtimes of the most recent approach based on recursive learning [20] (denoted CPU[20]) and the BED

---

<sup>2</sup>Available from <http://www.cbl.ncsu.edu/>

**Table 1:** Experimental results for verifying equivalence of the redundant and non-redundant circuits in the ISCAS 85 benchmark. ‘In’ and ‘Out’ denotes the number of primary inputs and outputs, respectively, and ‘Gates’ is the total number of logic gates in the two circuits.  $N_{\text{total}}$  is the total number of BED nodes used in the verification and ‘CPU’ is the runtimes in seconds for the BED approach.

Circuits	In	Out	Gates	$N_{\text{total}}$	CPU [s]	CPU[18]/CPU	CPU[20]/CPU
c432, c432nr	36	7	316	28259	0.5	2.0	4.0
c499, c499nr	41	32	403	234103	2.7	0.7	7.1
c499, c1355	41	32	748	467823	6.3	n/a	n/a
c1355, c1355nr	41	32	1091	492741	5.6	1.0	3.6
c1908, c1908nr	33	25	1757	429422	4.8	2.0	4.6
c2670, c2670nr	233	140	2153	26011	0.7	227.6	87.1
c3540, c3540nr	50	22	3288	427323 <sup>†</sup>	127.0	0.5	2.2
c5315, c5315nr	178	123	4604	45138	1.5	248.5	126.7
c6288, c6288nr	32	32	4814	8759	0.7	30.7	57.1
c7552, c7552nr	207	108	6908	329717	6.3	886.2	65.4

runtimes. Compared to this approach, BEDs perform close to two orders of magnitudes better on the larger examples.

The circuit c6288 implements a 16-bit multiplier. It is interesting to note that due to the exponential growth of the size of an ROBDD representation of multipliers [4], the straightforward approach of building and comparing the ROBDD for the two circuits c6288 and c6288nr is provably infeasible. The ROBDD representation of a 15-bit multiplier uses more than 12 million vertices [19] and this number is approximately 2.7 times larger for each additional bit in the operands. This clearly demonstrates the effectiveness of *up\_one*.

Some of the original non-redundant circuits in the ISCAS 85 benchmark were incorrect [17], i.e., the non-redundant version was not functionally equivalent to the redundant version. Table 2 reports the runtimes of the BED approach to determine non-equivalence. The reported CPU times are for finding *all* errors. It is noticed that although it does take longer to prove non-equivalence, the runtimes are still reasonably small.

**Table 2:** Experimental results for showing non-equivalence of the redundant and non-redundant circuits in the ISCAS 85 benchmark.  $N_{\text{error}}$  is the number of outputs that are non-equivalent.  $N_{\text{total}}$  is the total number of BED nodes for the verification.

Circuits	$N_{\text{error}}$	$N_{\text{total}}$	CPU [s]
c1908, c1908nr_old	1	411212	5.4
c2670, c2670nr_old	6	146473	4.8
c3540, c3540nr_old	3	3189370 <sup>†</sup>	541.8
c5315, c5315nr_old	29	189854	12.9
c7552, c7552nr_old	28	335672	17.7

### 3.2 The LGSynth 91 Benchmarks

By construction, the redundant and non-redundant versions of the ISCAS 85 benchmark circuits have many structural similarities. To test the verification strategy on a broader range of examples, we consider the 76 combinational multi-level circuits from the LGSynth 91 benchmark. These circuits are described in the technology independent format BLIF (Berkeley Logic Interchange Format). Using SIS from Berkeley, the circuits are mapped to a gate library (`msu.genlib`). The first verification problem is to verify that this mapping is correct. Then the circuits are optimized with respect to area using the SIS script `script.algebraic`. The second verification problem is to prove that the mapped and the optimized circuits are equivalent. Due to the nature of the mapping and optimization steps, we expect that the circuits differ in structure considerably more than the ISCAS 85 circuits. For most of the circuits, the two verification problems are solve in just a few seconds of CPU time when using BEDs. Out of the 76 circuits, 65 circuits can each be verified in less than 5 CPU seconds. The remaining 11 circuits are harder to verify. The results for these circuits are shown in table 3.

**Table 3:** Multi-level combinational circuits from LGSynth 91 that take more than 5 CPU seconds to verify. The columns ‘In’ and ‘Out’ show the number of inputs and outputs, respectively. The columns ‘Orig’, ‘Map’ and ‘Opt’ show the number of gates in the original, mapped and optimized circuit, respectively.  $\text{CPU}_{\text{map}}$  is the CPU time in seconds to verify the equivalence of the original circuit and the mapped circuit.  $\text{CPU}_{\text{opt}}$  is the CPU time in seconds to verify the equivalence of the mapped circuit and the optimized circuit.

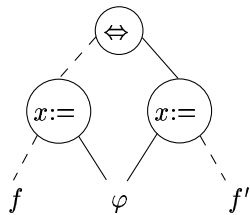
Circuit	In	Out	Orig	Map	Opt	$N_{\text{total}}$	$\text{CPU}_{\text{map}}$ [s]	$N_{\text{total}}$	$\text{CPU}_{\text{opt}}$ [s]
C499	41	32	202	225	197	509873	7.8	518142	7.7
C1355	41	32	546	493	263	696921	10.7	894122	12.8
C1908	33	25	880	327	240	696921	10.2	894122	12.7
C3540	50	22	1669	742	602	6404865 <sup>†</sup>	562.2	5729413 <sup>†</sup>	327.1*
C5315	178	123	2307	1160	838	3536016 <sup>†</sup>	42.4	4211313 <sup>†</sup>	68.6
C6288	32	32	2416	2325	2067	4754	0.9	Out of memory	
C7552	207	108	3512	1627	1191	383597	18.4	439626	46.3
des	256	245	926	2516	2082	976767	13.1	1016780	18.3
i10	257	224	2497	1504	1288	1642547 <sup>†</sup>	385.7	1623388 <sup>†</sup>	413.5
rot	135	107	243	429	414	494342	10.6	317674	6.7
too_large	38	3	43	478	228	345849	6.2	294493	4.2

\*We were unable to verify one of the outputs due to lack of memory.

### 3.3 Using Substitutions

Substitution (also called composition) is a standard operation of Boolean functions. The substitution operation replaces all occurrences of a variable  $x$  in a formula  $f$  with an expression  $\varphi$ . This operation can be represented directly in the BED data structure by using an operator vertex  $v$  with  $\text{var}(v)$  equal to  $x$ ,  $\text{low}(v)$  pointing to  $f$  and  $\text{high}(v)$  pointing to  $\varphi$ . The substitution operator vertices can be eliminated like any other operator in the BED by pulling the variables up past the operators. An operator is eliminated either when it meets a variable vertex  $x$  or when it reaches terminal vertices. However, one need not immediately eliminate the substitution vertices. Keeping them in the BED

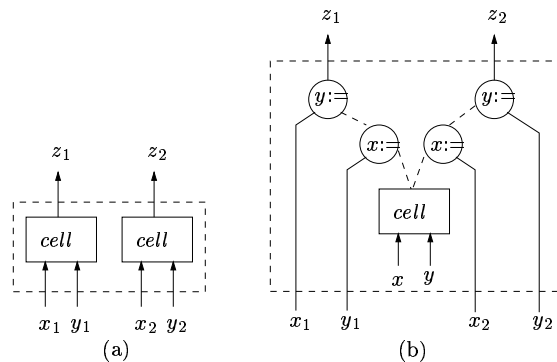
allows efficient reuse of sub-expressions. Consider the BED in figure 3. If the vertices



**Figure 3:** A BED containing substitution operators. The low-edge (dotted) points to the expression in which  $x$  is to be substituted with  $\varphi$ .

$f$  and  $f'$  are identified at some point in the manipulations, the bi-implication is proven immediately, *without* actually performing the substitution.

This can be used to take advantage of a hierarchy in the logic description. Consider a circuit with two instances of a sub-circuit ‘cell’, see figure 4 (a). Instead of flattening the hierarchy, the two instances of ‘cell’ are represented in the BED data structure using substitutions, see figure 4 (b). This information can greatly improve the performance when verifying large hierarchical circuits. We demonstrate this by verifying equivalence



**Figure 4:** (a) Two instances of a sub-circuit ‘cell’. (b) Representing the same circuit using substitutions.

between two implementations of  $n$ -bit multipliers. A combinational  $n$ -bit multiplier can be constructed using four  $n/2$ -bit multipliers. The four multipliers each compute partial products which are shifted and added to form the result. In this way we build two versions of hierarchically specified  $n$ -bit multipliers based on the 16-bit multipliers `c6288` and `c6288nr`. The results of verifying equivalence between these two versions for  $n$  up to 1024 are shown in table 4. Since verifying equivalence of `c6288` and `c6288nr` is very efficient (see table 1) verification of larger multipliers is only limited by the number of substitutions (which is quadratic in  $n$ ).

## 4 Conclusion

We have demonstrated that the BED data structure for representing and manipulating Boolean expressions is effective for solving the combination logic-level verification problem.

BEDs can be seen as an intermediate form between circuits and the highly structured ROBDDs. Although not presented here, the BED data structure is *simpler* than the ROBDD data structure and the algorithms operating on BEDs are at least as efficient as those operating on ROBDDs, making BEDs applicable in all applications using ROBDDs.

**Table 4:** Verifying equivalence of hierarchically specified  $n$ -bits multipliers. ‘Ops’ is the number of ordinary operator vertices in the BED specification and ‘Subst’ is the number of substitution vertices.

$n$	Ops	Subst	$N_{\text{total}}$	CPU [s]
32	6505	8192	26665	1.6
64	9827	40960	75070	3.1
128	16447	172032	270275	8.6
256	29783	696320	$1.05 \cdot 10^6$	29.5
512	56401	2793472	$4.19 \cdot 10^6$	119
1024	109643	11192080	$14.3 \cdot 10^6$	408

BEDs are particularly useful in applications where the end-result as a reduced ordered BDD is small, for example, for tautology checking. Another area that may benefit from using the BED representation is symbolic model checking. Several researchers have observed that when performing fixed-point iterations using ROBDDs, the intermediate results are often much larger than the final result. Clearly, the succinctness of BEDs compared to BDDs can alleviate this problem. This is possible because all operations used in performing the fixed-point computation can be performed directly on the BED without first expanding it to an ROBDD. In fact, many of the tricks researchers have used to make ROBDDs more efficient are embodied in BEDs. For example, Burch, Clarke, and Long [6] demonstrated that the complexity of BDD-based symbolic verification is drastically reduced by using a *partitioned transition relation* where the transition relation is represented as an implicit conjunction of ROBDDs. This corresponds to representing the transition relation as a BED with conjunction vertices at the top level and only lifting the variables up to just under these vertices.

## References

- [1] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. In *IEEE Symposium on Logic in Computer Science (LICS)*, July 1997.
- [2] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 188–191, 1993.
- [3] D. Brand. Verification of large synthesized designs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 534–537, 1993.
- [4] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.
- [5] R. E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 535–541, 1995.
- [6] J. R. Burch, E.M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 403–407, 1991.
- [7] E. M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large Boolean functions with application to technology mapping. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 54–60, 1993.

- [8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [9] 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 *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 415–419, 1994.
- [10] M. Fujita, Y. Matsunga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level synthesis. In *Proc. European Conference on Design Automation (EDAC)*, pages 50–54, 1991.
- [11] M. R. Garey and D. S. Johnson. *Computers and Intractability—A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
- [12] J. Gergov and C. Meinel. Efficient Boolean manipulation with OBDD's can be extended to FBDD's. *IEEE Transactions on Computers*, 43(10):1197–1209, October 1994.
- [13] A. Hett, R. Drechsler, and B. Becker. MORE: Alternative implementation of BDD-packages by multi-operand synthesis. In *European Design Conference*, 1996.
- [14] A. Hett, R. Drechsler, and B. Becker. Fast and efficient construction of BDDs by reordering based synthesis. In *IEEE European Design & Test Conference*, 1997.
- [15] U. Keschull, E. Schubert, and W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. In *Proc. European Conference on Design Automation (EDAC)*, pages 43–47, 1992.
- [16] W. Kunz. HANNIBAL: An efficient tool for logic verification based on recursive learning. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 538–543, 1993.
- [17] W. Kunz and D. K. Pradhan. Recursive learning: A new implication technique for efficient solutions to CAD problems – test, verification, and optimization. *IEEE Transactions on Computer Aided Design*, 13(9):1143–1158, September 1994.
- [18] W. Kunz, D. K. Pradhan, and S. M. Reddy. A novel framework for logic verification in a synthesis environment. *IEEE Transactions on Computer Aided Design*, 15(1):20–32, January 1996.
- [19] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 48–55, 1993.
- [20] D. K. Pradhan, D. Paul, and M. Chatterjee. VERILAT: Verification using logic augmentation and transformations. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, November 1996.
- [21] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 42–47, 1993.
- [22] D. Sieling and I. Wegener. Graph driven BDDs – a new data structure for Boolean functions. *Theoretical Computer Science*, 141(1-2):283–310, 1995.
- [23] C.A.J. van Eijk and G. L. J. M. Janssen. Exploiting structural similarities in a BDD-based verification method. In *Theorem Provers in Circuit Design*, number 901 in Lecture Notes in Computer Science, pages 110–125. Springer-Verlag, 1994.