

# Equivalence Checking of Combinational Circuits using Boolean Expression Diagrams

Henrik Hulgaard, Poul Frederick Williams, and Henrik Reif Andersen

*Abstract*—The combinational logic-level equivalence problem is to determine whether two given combinational circuits implement the same Boolean function. This problem arises in a number of CAD applications, for example when checking the correctness of incremental design changes (performed either manually or by a design automation tool).

This paper introduces a data structure called Boolean Expression Diagrams (BEDs) and two algorithms for transforming a BED into a Reduced Ordered Binary Decision Diagram (OBDD). BEDs are capable of representing any Boolean circuit in linear space and can exploit structural similarities between the two circuits that are compared. These properties make BEDs suitable for verifying the equivalence of combinational circuits. BEDs can be seen as an intermediate representation between circuits (which are compact) and OBDDs (which are canonical).

Based on a large number of combinational circuits, we demonstrate that BEDs either outperform or achieve results comparable to both standard OBDD approaches and the techniques specifically developed to exploit structural similarities for efficiently solving the equivalence problem.

Due to the simplicity and generality of BEDs, it is to be expected that combining them with other approaches to equivalence checking will be both straightforward and beneficial.

*Keywords*—Tautology checking, combinational logic-level verification, equivalence checking, Boolean circuits.

## I. INTRODUCTION

THIS paper presents a technique for formally proving that two *combinational* circuits implement the same Boolean function. This verification problem, referred to as *the combinational logic-level equivalence problem*, arises in a number of CAD applications related to validating the correctness of a circuit design:

- Due to the increase in the complexity of design automation tools and the circuits they manipulate, such tools cannot in general be assumed to be correct. Instead of attempting to formally verify the design automation tools, a more practical approach is to formally check that a circuit generated by a design automation tool functionally corresponds to the original input (the specification). Such a check is an instance of the combinational logic-level equivalence problem when the design automation tool only manipulates the combinational portion of the circuit.
- The logic-level equivalence problem also arises when a circuit is manually modified in order to accommodate special requirements which cannot be handled by the design automation tool (so-called engineering changes). The designer can ensure that no functional errors have been introduced by verifying that the original and modified designs are functionally identical.
- Finally, the combinational logic-level equivalence problem arises as a sub-problem in other (higher-level) verification problems. For example, when verifying arithmetic circuits by checking that they satisfy a given recurrence equation [1] or when verifying the equivalence of two state machines without performing a state traversal [2].

The straightforward approach to solving the combinational logic-level equivalence problem is to use Reduced Ordered Binary Decision Diagrams [3] (OBDDs). To verify that two combinational circuits with outputs  $F$  and  $G$  are equivalent, the OBDD for  $f \Leftrightarrow g$  is constructed, where  $f$  and  $g$  represent the Boolean function for  $F$  and  $G$ , respectively. Due to the canonicity of OBDDs, the two circuits implement the same Boolean function if and only if the resulting OBDD is identical to the terminal 1. This approach is simple and works well for many circuits, but it has two inherent limitations:

- The first problem is that the size of the OBDD representation for  $f$  and  $g$  may be exponential in the size of the combinational circuit, no matter what variable ordering is used. A well-known example of this problem is the multiplication function which Bryant [3] showed does not have any sub-exponential OBDD representation for any variable ordering.
- The second problem is that OBDDs cannot exploit structural similarities of the two circuits that are verified. Consider verifying that two *identical* circuits implement the same Boolean function. In this case, the full OBDD for both  $f$  and  $g$  is constructed before the identity of the circuits is verified. In typical applications, the circuits to be verified are of course not identical, but in all three application areas listed above, the two combinational circuits are structurally similar. To efficiently verify the circuits, it is essential to be able to exploit these similarities.

We suggest a newly developed data structure [4] called Boolean Expression Diagrams (BEDs) for solving the combinational logic-level equivalence problem. BEDs are an extension of OBDDs that allow any Boolean circuit to be represented in linear space. Furthermore, BEDs can recognize and share identical sub-expressions. These properties eliminate the two problems with the OBDD approach listed above and thus make BEDs a promising data structure for solving the equivalence problem.

The price one pays for the compactness of BEDs is that BEDs are not canonical. Our approach to showing that the BED for  $f \Leftrightarrow g$  is a tautology is to transform it into an equivalent OBDD. A key observation is that it is possible to construct the OBDD for  $f \Leftrightarrow g$  *without* constructing the OBDD for  $f$  and  $g$ . For example, the verification may succeed even if  $f$  and  $g$  each represent a multiplication function for which no small OBDD exists. Thus, using BEDs, one can potentially avoid an exponential blowup when computing the intermediate results.

The BED data structure is obtained by extending the OBDD representation with *operator vertices*:

*Definition 1* (Boolean Expression Diagram) A *Boolean Expression Diagram* (BED) is a directed acyclic graph  $G = (V, E)$  with vertex set  $V$  and edge set  $E$ . The vertex set  $V$  contains three types of vertices: terminal, variable, and operator vertices.

- A *terminal vertex*  $v$  has as attribute a value  $val(v) \in \{0, 1\}$ .

Financially supported by the Danish Technical Research Council.

The authors are with the Department of Information Technology, Technical University of Denmark. E-mails: {henrik,pfw,hra}@it.dtu.dk

- A *variable vertex*  $v$  has as attributes a variable  $var(v)$ , and two children  $low(v), high(v) \in V$ .
  - An *operator vertex*  $v$  has as attributes a binary Boolean operator  $op(v)$ , and two children  $low(v), high(v) \in V$ .
- The edge set  $E$  contains the edges  $(v, low(v))$  and  $(v, high(v))$  for each  $v \in V$  where  $v$  is not a terminal vertex.

We use  $\mathbf{0}$  and  $\mathbf{1}$  to denote the two terminal vertices. The relation between a BED and the Boolean function it represents is straightforward. 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, leading to the following correspondence between BEDs and Boolean functions:

*Definition 2:* A vertex  $v$  in a BED denotes a Boolean function  $f^v$  defined recursively as:

- If  $v$  is a terminal vertex, then  $f^v = val(v)$ .
- If  $v$  is a variable vertex, then  $f^v$  is the function

$$f^v = var(v) \rightarrow f^{high(v)}, f^{low(v)}.$$

- If  $v$  is an operator vertex, then  $f^v$  is the function

$$f^v = f^{low(v)} op(v) f^{high(v)}.$$

Clearly, BEDs are closely related to combinational circuits. Any Boolean circuit [5] can be transformed into a BED by replacing each input  $x$  with the BED representing  $x$  (a variable vertex  $v$  with  $var(v) = x$ ,  $low(v) = \mathbf{0}$ , and  $high(v) = \mathbf{1}$ ) and replace each  $k$ -input gate by a tree of  $k - 1$  operator vertices encoding the Boolean function of the gate. This translation is clearly linear in size. Similarly, any BED can be converted to a circuit. Each variable occurring in the BED is an input to the circuit. An operator vertex is replaced by the corresponding gate, and a variable vertex  $v$  with the sub-circuit  $(\neg x \wedge l) \vee (x \wedge h)$ , where  $x = var(v)$ ,  $l = low(v)$ ,  $h = high(v)$ . This translation is also linear. Thus, in terms of succinctness BEDs and combinational circuits are equally expressive. For instance, since there are combinational circuits implementing multiplication using only a quadratic number of gates [6], there also exists BEDs of this size representing them.

To illustrate how BEDs are used to check the equivalence of two combinational circuits, consider the circuits in Fig. 1. To verify the equivalence of the two circuits, the BED for each circuit is constructed and the corresponding outputs are connected with biimplications, see Fig. 2 (the low-edges are drawn using dashed lines). We show that both roots of this BED are tautologies without constructing OBDDs for the two circuits by performing a case-split on the variable  $i_3$ . (These steps are an approximation of how the algorithm `UP_ONE` works; the details follow in Section III-B.) When  $i_3$  is false, a simple evaluation of the BED according to Definition 2 yields that both outputs of the circuits have the value 1 and thus the biimplication reduces to  $\mathbf{1}$ . In the other case, when  $i_3$  is true, the BED is simplified but does not immediately reduce to the terminal  $\mathbf{1}$ . The BED from Fig. 2 after the case-split on variable  $i_3$  is shown in Fig. 3. Thus, by moving  $i_3$  to the top, we have shown that the outputs of the two circuits are identical for all input combinations with

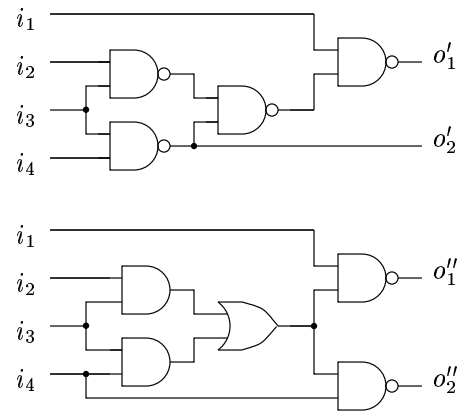


Fig. 1. Two combinational circuits implementing the same Boolean function.

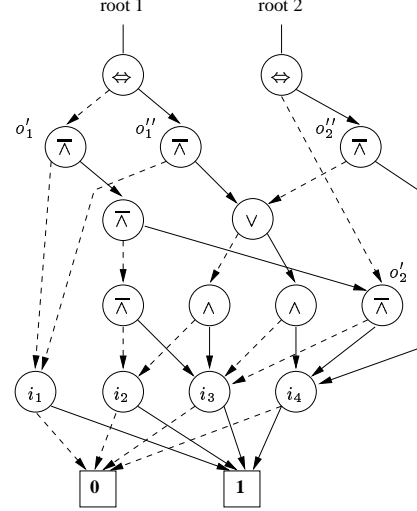


Fig. 2. The BED used in checking the equivalence of the two circuits in Fig. 1.

$i_3 = 0$  (notice that the low-edge of the variable vertex with  $i_3$  points to  $\mathbf{1}$ ). The case where  $i_3 = 1$  is proved by applying local reduction rules (these reduction rules are described in more detail in Section II-C) and identifying equivalent vertices (vertices with same operator, low- and high-child). This is shown in Fig. 4 and 5. Notice that the final OBDDs (the terminal  $\mathbf{1}$ ) for both roots are constructed without building the OBDDs for outputs  $o'_i$  and  $o''_i$  ( $i = 1, 2$ ).

The efficiency of this way of transforming a BED into an OBDD is illustrated when verifying that two combinational 16-bit multipliers (`c6288` and `c6288nr` from the ISCAS 85 benchmark) implement the same function. Using BEDs, the 32 outputs are shown to be identical in less than two seconds. Using OBDDs, this verification problem is infeasible due to the blowup of the OBDD representation whose size is exponential in the number of operand bits.

Similar to OBDDs, the efficiency of the BED approach relies on a good variable ordering. We describe two ordering heuristics which seem to work well with BEDs. Using these heuristics, we report on the verification results for a large number of circuits (more than 250 circuits) from the ISCAS 85 and LGSynth 91 benchmarks. The results show that the BED ap-

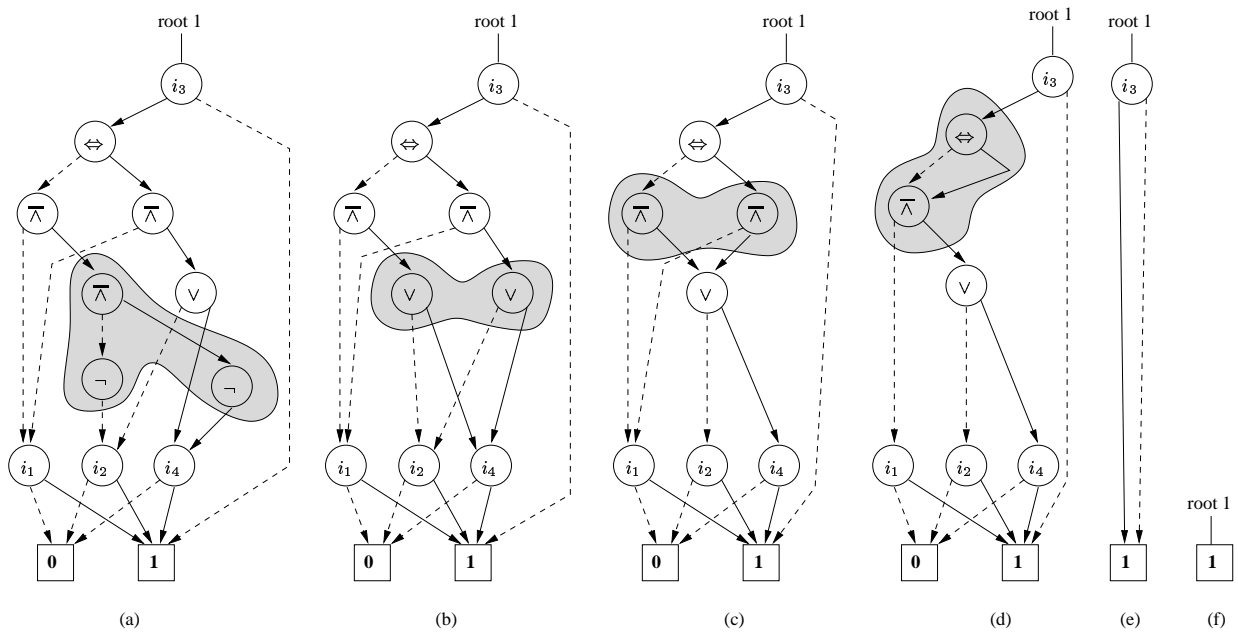


Fig. 4. Steps used to show that root 1 from Fig. 3 is a tautology. a) to b) Use the identity  $\neg(\neg x \wedge \neg y) = (x \vee y)$ . b) to c): Identify the two  $\vee$  vertices. c) to d): Identify the two  $\wedge$  vertices. d) to e): Use the identity  $(x \leftrightarrow x) = \mathbf{1}$ . e) to f): Use the identity  $(i_3 \rightarrow \mathbf{1}, \mathbf{1}) = \mathbf{1}$ .

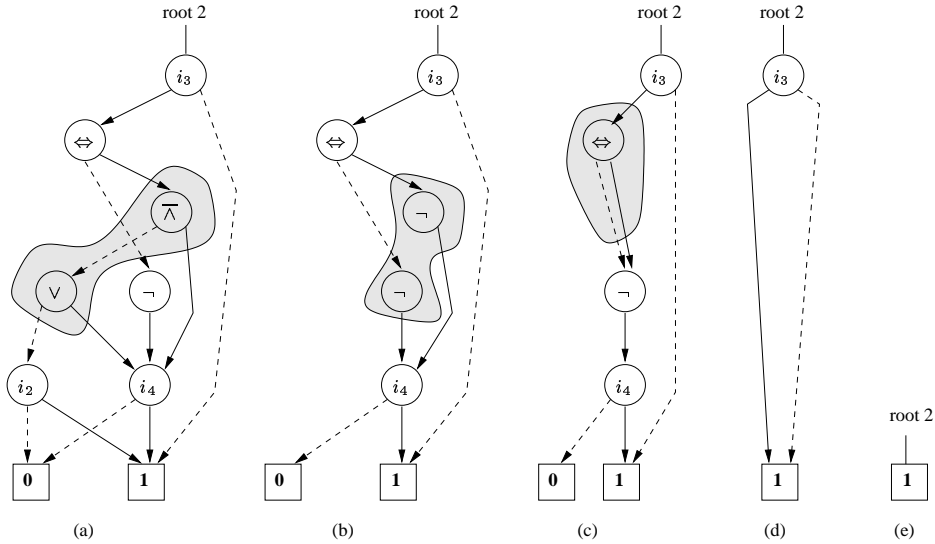


Fig. 5. Steps used to show that root 2 from Fig. 3 is a tautology. a) to b): Use the identity  $\neg(y \wedge (x \vee y)) = \neg y$ . b) to c): Identify the two  $\neg$  vertices. c) to d): Use the identity  $(x \leftrightarrow x) = \mathbf{1}$ . d) to e): Use the identity  $(i_3 \rightarrow \mathbf{1}, \mathbf{1}) = \mathbf{1}$ .

proach performs extremely well for circuits that are structurally similar (sometimes several orders of magnitude faster than existing techniques) and is capable of verifying very large circuits (up to 104,000 gates) which have been drastically modified (using SIS [7] to reduce area).

### A. Related Work

Current approaches for equivalence checking of combinational circuits can be classified into two categories: functional and structural.

The functional methods consist of representing a circuit as a canonical decision diagram. Two circuits are equivalent if and only if their decision diagrams are equal (isomorphic). To

overcome the problems with OBDDs mentioned above, a number of more expressive, yet still canonical, decision diagrams have been proposed. One can use other types of decomposition rules [8], [9], relax the variable ordering restriction [10], [11], [12], [13], or extend the domains and/or codomains to integers instead of Booleans [14], [15], [16]. These extensions are typically targeted to solving a particular class of problems, e.g., being able to represent the multiplication function. These canonical representations all have worst case exponential size, thus they are all exponentially less compact than BEDs.

The structural methods exploit similarities between the two circuits that are compared by identifying related nodes in the circuits and using this information to simplify the verification

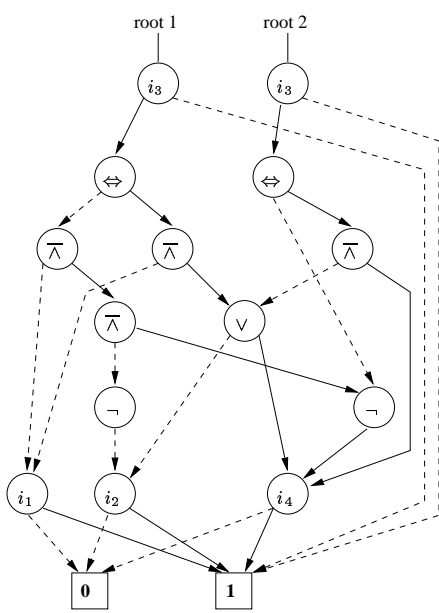


Fig. 3. The BED from Fig. 2 where variable  $i_3$  has been moved up.

problem. These techniques rely on the observation that if two circuits are structurally similar, they will have a large number of internal nodes that are functionally equivalent (typically, for more than 80% of the nodes in one circuit, there exists a node in the other circuit which is functionally equivalent [17]). This observation is used in several ways. Brand [18] uses a test generator for determining whether one node can be replaced by another in a given context (the nodes need not necessarily be functionally equivalent as long as the difference cannot be observed at the primary output). If so, the replacement is carried out. In this way, one circuit is gradually transformed into the other. The key problem is to find a sufficiently large number of pairs, yet avoid having to spend time testing all possible pairs of nodes. Several heuristics are used to select candidate pairs of nodes to check, e.g., the labeling of nodes and the results of simulation.

Test generation techniques are also the basis for the recursive learning technique for finding logical implications between nodes in the circuits by Kunz *et al.* [19], [20]. To enable the verification of larger circuits, the recursive learning techniques can be combined with OBDDs [21], [22]. The learning technique is further extended by Jain *et al.* [23] and by Matsunaga [24], introducing more general learning methods based on OBDDs and better heuristics for finding cuts in the circuits to split the verification problem into more manageable sizes.

Eijk and Janssen [25], [26] use the canonicity of OBDDs to determine whether one node is functionally equivalent to another. If two nodes are found to be identical, they are replaced with a new free variable. Heuristics are used to select candidate pairs of nodes to check for equivalence. The main problem with this technique is to manage the OBDD sizes when eliminating false negatives (when re-substituting OBDDs for the introduced free variables).

Cerny and Mauras [27] present another technique for comparing two circuits without representing their full functionality. A relation that represents the possible combinations of logic val-

ues at a given cut is propagated through the two circuits. A key problem with this and the other cut-based techniques [21], [22], [23], [24], [25], [26] is that the performance is very sensitive to how the cuts are chosen and there is no generally applicable method to choose appropriate cuts.

The technique by Kuehlmann and Krohm [17] represents the most recent development of the structural methods, combining several of the above techniques and developing better heuristics for determining cuts. Kuehlmann and Krohm represent the combinational circuits using a non-canonical data structure which is similar to BEDs except that only conjunction and negation operators are used. This data structure is only used to identify isomorphic sub-circuits since no operator reductions are performed. We believe that the structural technique by Kuehlmann and Krohm would benefit significantly from replacing the used circuit representation with BEDs since the continuous application of the reduction rules would reduce the circuit representation and help in identifying equivalent sub-circuits which in turn would improve the performance of their technique.

BEDs can be seen as an intermediate representation between the compact circuits and the canonical OBDDs. Compared to the functional techniques, BEDs are capable of exploiting equivalences of the two circuits and the performance is provably no worse than when using OBDDs. Compared to the structural techniques, BEDs only have a limited capability to find equivalences between pairs of nodes (since only local operator reduction rules are included). Combining BEDs with structural techniques would be beneficial since information about equivalent nodes immediately reduce the size of the BED and make even further identifications of nodes possible.

During the last three decades the AI community has worked on developing efficient satisfiability checkers. They could in principle be used to solve the equivalence problem for combinational circuits. However, comparisons between algorithms based on the prominent Davis-Putnam algorithm and OBDDs show that although efficient for typical AI problems, they are quite inferior to OBDDs on circuits [28].

Hachtel and Jacoby [29] describe an algorithm for solving the equivalence problem by searching for a counterexample using a tree formed by case splitting (co-factoring) the combined circuits on the variables of the primary inputs. If during the generation of the co-factor tree a subcircuit structurally equivalent to a previously visited subcircuit is found, the previous result is used. Equivalence is determined by matching the strings representing the formula of the two subcircuits. On smaller circuits (up to 100 gates) this approach is demonstrated to work well. To some extent one of the algorithms of synthesizing OBDDs from BEDs (UP\_ONE) can be seen as an improved version of the Hachtel-Jacoby algorithm in which the identification of equivalent subcircuits is improved both by the use of reduction rules and through the sharing of nodes.

The BED data structure is inspired by the MORE approach [30], [31] to synthesizing OBDDs. MORE is based on the observation that the OBDD 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 OBDD by moving  $x$  towards the terminal vertices using the level exchange operation [32]. The BEDs differ from the MORE ap-

proach by their use of operator reductions and the new synthesis algorithms (which work on arbitrary BEDs where operators and variables are freely mixed).

Prior to the work on MORE, Plessier *et al.* [33], [34] proposed a variant of OBDDs called Extended BDDs (XBDDs) obtained by adding *structural* variables which can be both universally and existentially quantified. Quantifications are described as annotations on pointers leading to nodes with structural variables. The quantifications allow Boolean operations to be expressed. During construction of an XBDD from a circuit, a trade-off can be made between removing a structural variable and perform OBDD synthesis or keeping the structural variable. Two algorithms for checking satisfiability of XBDDs were given (one requiring up to exponential space, the other requiring linear space but exponential time). No algorithms for converting an XBDD into a OBDD were given. Using the satisfiability algorithms the authors showed that although the growth is still exponential, the equivalence between the median bit of two structurally different multiplier circuits could be proven for two 16 bit multipliers.

BEDs extend the ideas of XBDDs and MORE to include arbitrary binary operators and allowing these operators to remain in the graph while transforming it. (In XBDDs and in the MORE approach, two nodes are needed to represent an exclusive-or or a biimplication.) This makes it possible to include operator reduction rules and develop new OBDD synthesis algorithms (e.g., UP\_ONE) which are essential for obtaining the runtimes presented in this paper.

## B. Overview

The paper is organized as follows. The construction of BEDs is presented in Section II. In Section III we described the algorithms for transforming a BED into an equivalent OBDD. Section IV describes two heuristics for choosing a variable ordering based on the topology of the circuit. Section V presents the experimental results, verifying a large number of combinational circuits from the ISCAS 85 and LGSynth 91 benchmarks. Finally, Section VI summarizes the contributions of this paper.

## II. CONSTRUCTION OF BEDS

BED vertices are constructed using a single constant-time operation called *makenode*. This operation ensures that the BED is *reduced* and also performs several optimizations of the representation. Contrary to OBDDs, reducedness will not make BEDs canonical (not even when combined with a fixed variable ordering.)

### A. Reduced BEDs

We shall forbid the existence of *redundant* vertices, i.e., two vertices representing isomorphic sub-BEDs and vertices that are unnecessary for obvious reasons. For readability, we use  $\alpha(v)$  to denote the “tag”  $op(v)$  or  $var(v)$  on non-terminal vertices.

*Definition 3:* A BED is *reduced* if it contains at most two different terminal vertices and for all non-terminal vertices,  $u$  and

$v$ :

- (i)  $low(u) = low(v)$ ,  $high(u) = high(v)$ , and  $\alpha(u) = \alpha(v)$  implies  $u = v$ ,
- (ii)  $low(u) \neq high(u)$ , and
- (iii) for all operator vertices  $v$ ,  $low(v)$  and  $high(v)$  are non-terminals.

We shall assume that BEDs are always reduced. The BEDs in Fig. 2 and 3 are reduced, but some of the intermediate BEDs in Fig. 4 and 5 violate conditions (i) and (ii) of reducedness.

The first condition of Definition 3 is fulfilled by proper reuse of vertices. This is conveniently taken care of during construction of a BED by testing, whenever a new vertex is to be created, whether another vertex with the same variable/operator, low- and high-child exists. If this is the case, that vertex is reused, otherwise a new vertex is created. Similarly, the second and third conditions are fulfilled by never constructing vertices that violate them. For variable vertices, it is clear that if the low- and high-child coincide, either one of them can be used instead of creating a new variable vertex. For operator vertices, one should observe that if the two arguments are identical, or one of them is a terminal vertex, all the sixteen Boolean connectives reduce to one of the following six:  $K0$ ,  $K1$  (constant 0/1),  $\pi_1$ ,  $\pi_2$  (projection onto first or second argument),  $\bar{\pi}_1$ ,  $\bar{\pi}_2$  (the negation of the first or second argument). In the first two cases, one of the terminal vertices is used. The projections are avoided by using the proper low- or high-child instead. The negations require creation of a negating vertex, i.e., an operator vertex with the operator  $\bar{\pi}_1$ . Such a vertex can easily be constructed so that it fulfills (ii) and (iii) by taking the redundant second argument to be any non-terminal vertex different from the first.

We shall assume the presence of a function

$$makenode(\alpha, l, h)$$

that performs all the checks above and returns the identity of the resulting vertex, equivalent to a vertex  $u$  with  $\alpha(u) = \alpha$ ,  $low(u) = l$ ,  $high(u) = h$ . Using *makenode* as the only means for constructing a BED ensures that it is reduced.

### B. Ordered and Free BEDs

Inspired by OBDDs, we define certain restrictions on the variables of BEDs:

*Definition 4:* A BED is *free* if on all paths through the graph each variable occurs at most once; it is *ordered* if on all paths the variables respect a given total order  $<$ .

We refer to a free BED as FBED and to an ordered BED as OBED. Observe that an (O)BDD is simply an (O)BED without operators. From these definitions we get the following inclusions among sub-classes of BEDs:

$$OBDD \subseteq \text{DAG of OBDDs} \subseteq OBED \subseteq FBED \subseteq BED.$$

The class “DAG of OBDDs” represents BEDs that consist of a layer of operators on top of a layer of OBDDs. Boolean circuits that are transformed into BEDs belong to this class (in this case, the OBDDs are initially very simple, each consisting of a single variable). Furthermore, this class occurs in the traditional synthesis of OBDDs, where the operators represent APPLY-calls.

Since Boolean circuits can be transformed into a “DAG of OBDDs” in linear time (and space) and a (general) BED can be transformed into a Boolean circuit in linear time (and space), the last four classes are equally expressive. This is quite unlike for OBDDs where there is an exponential gap between OBDDs and free BDDs, and between free BDDs and BDDs.

### C. Operator reductions

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 *makenode* 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  $n = 3$  allows us to determine *operator 2-cuts*. Consider a BED with the structure shown in Fig. 6 (a), that is, for some vertex  $u$ , all paths from  $u$  to the terminals go through either vertex  $w_1$  or  $w_2$ . The set  $\{w_1, w_2\}$  is a 2-cut and such a cut can be used to reduce the size of the BED as shown in Fig. 6 (b). That is, all vertices from  $u$  to the operator 2-cut can be replaced

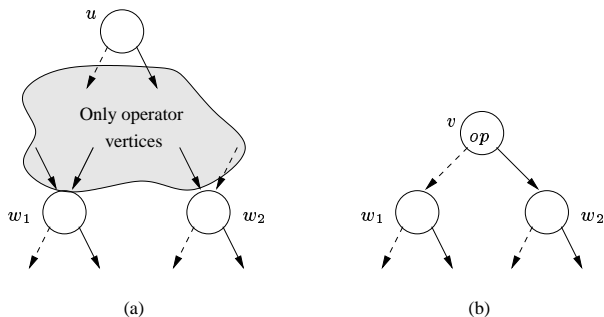


Fig. 6. A BED with an operator 2-cut  $\{w_1, w_2\}$ .

with a single operator vertex  $v$  with  $\alpha(v) = op$ ,  $low(v) = w_1$ , and  $high(v) = w_2$ . Notice that  $\{w_1, w_2\}$  with  $var(w_1) = i_2$  and  $var(w_2) = i_4$  is the operator 2-cut for the reductions in Fig. 4 (a) and Fig. 5 (a).

By using a reduction table, the BED can be constructed such that the only operator 2-cut for a vertex  $u$  is the trivial cut  $\{low(u), high(u)\}$ . This can be done by *makenode* in constant time since a new operator vertex will only have non-trivial 2-cuts among its children and grand-children and these cuts are included in a reduction table.

We have systematically implemented reduction rules for all possible combinations of three ( $n = 3$ ) operators organized as a tree, see Fig. 7. For example, the reduction table includes the distributive law, see Fig. 8. For each combination of the possible operators ( $op$ ,  $op_u$ , and  $op_v$ ) and the possible equivalences between the nodes ( $u$ ,  $u_l$ ,  $u_h$ ,  $v$ ,  $v_l$ , and  $v_h$ ) we have determined (by exhaustive search) whether the same sub-function can be

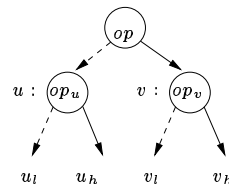


Fig. 7. A tree of three operator vertices.

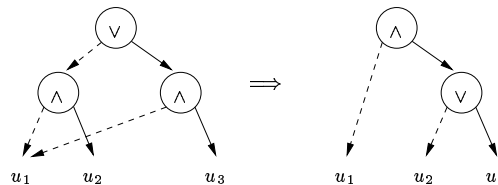


Fig. 8. A reduction table for  $n = 3$  includes information on how each combination of up to three BED operator vertices can be reduced. Here the reduction rule for one of the distributive laws is used to eliminate one of the operator vertices.

represented by fewer nodes. If this is possible, the reduction is included in the reduction table. Thus, given three operator nodes and the equivalences of the nodes and their children, it is possible by a simple table lookup (performed in constant time) to determine the optimal representation.

Fig. 9 shows the algorithms for constructing BED vertices using the reduction rules. Vertices are created using *mk* which calls the algorithm *red* to apply the reduction rules. Notice how the reduction rules are applied repeatedly until no more reductions are possible.

```

red(op, u, v) =
1: if (op, u, v) is in reduction table then
2:   return red(lookup(op, u, v))
3: else return (op, u, v)

```

```

mk(alpha, l, h) =
1: if alpha is operator then
2:   return makenode(red(alpha, l, h))
3: else return makenode(alpha, l, h)

```

Fig. 9. The *red* and *mk* algorithms.

We apply the reduction rules every time we create a new vertex; both when building the initial BED, and during the transformation of the BED into an OBDD. We have chosen to apply the same set of rules in both cases. Our rules are fast to apply because of their simple structure, yet powerful enough to capture laws such as the distributive laws and to ensure that only trivial operator 2-cuts exist. It would be possible to use different sets of rules for building and transforming the BED. For example, one set of rules could bring the BED to a form such that finding a good variable ordering was easier, while another set of rules could aim at minimizing the data structure during the transformation. Nikolskaia *et al.* [35] use a set of rules to rewrite Boolean expressions for fault trees before constructing the OBDD. The rewriting rules and strategies aim at structuring the formula such that the variable ordering heuristics work better. In the subsequent OBDD construction, they do not use any reduction rules.

It is well known that not all 16 binary Boolean connectives are needed to represent any Boolean expression. For example, the sets  $\{nand\}$  and  $\{or, not\}$  are functionally complete. It is therefore possible to limit the different operators in the BED to those of any complete set. Choosing a small set, e.g.,  $\{nand\}$ , the depth of the BED increases as several *nand* operators are need to represent other operators like exclusive-or. To effectively handle such cases, the reduction rules would need to look to a greater depth to achieve equally powerful reductions. This, however, greatly increases the number of cases to consider and thus complicates the reduction-step. Keeping a depth of two (i.e.,  $n = 3$ ) with a minimal operator set like  $\{nand\}$ , overall performance decreases since less reductions are performed. On the other hand, using a small operator set has the advantage that chances for identifying vertices increases since all operators are expressed in terms of a small set of operators. We have found the following set of operators to be a good trade-off between a small and large operator set:  $\{nand, or, right\ implication, left\ implication, biimplication\}$ . This set of operators has the property that all ten non-trivial binary Boolean operators can be expressed from it using exactly one operator or the negation of an operator which can be absorbed by an operator immediately above it. Thus, no more operator vertices are needed to represent a given Boolean expression than using the full set of all 16 operators.

### III. COMBINATIONAL VERIFICATION USING BEDS

The procedure `VERIFY`, shown in Fig. 10, determines whether two combinational circuits implement the same Boolean function. The input to `VERIFY` consists of the two combinational

```

VERIFY(circuit  $C_1$ , circuit  $C_2$ ) =
1:  $root\_list \leftarrow BUILD\_BED(C_1, C_2)$ 
2: for each root  $u$  in  $root\_list$  do
3:   Construct a variable order for the variables in  $support(u)$ .
4:   Transform  $u$  to an equivalent OBDD  $v$  using UP_ONE or UP_ALL.
5:   if  $v \neq \mathbf{1}$  then
6:     Report error. ANYNONSAT( $v$ ) is a counterexample.
7: enddo

```

Fig. 10. Algorithm for solving the combinational logic-level verification problem using BEDs.

circuits  $C_1$  and  $C_2$  to be compared. The first step is to construct the BED for  $C_1$  and  $C_2$  and connect corresponding outputs with biimplications. The resulting roots are returned in  $root\_list$  (line 1). For each pair of primary outputs (line 2), we construct a variable order (line 3) using one of the heuristics described in the next section. The  $support$  of a vertex  $u$  is the set of variables reachable from  $u$ :

$$support(u) = \{var(v) : v \text{ is a variable vertex and } u \rightsquigarrow v\},$$

where  $u \rightsquigarrow v$  denotes that there is a path from  $u$  to  $v$  in the BED. To prove that a Boolean function represented by a BED  $u$  is a tautology, we transform the BED into an equivalent OBDD (line 4) using one of the two algorithms described in the following. If the resulting (canonical) OBDD is not the terminal  $\mathbf{1}$ , we can report that the given pairs of primary outputs are not functionally equivalent and the input assignment obtained from the OBDD procedure `ANYNONSAT` is a counterexample.

In this section, we describe two algorithms for transforming a BED into an equivalent OBDD. It should be observed that any transformation algorithm from BEDs to OBDDs will necessarily have exponential worst-case runtime (assuming  $P \neq NP$ ). This follows since determining `SATISFIABILITY` of a BED is NP-complete and determining `TAUTOLOGY` is co-NP-complete [36] and these problems are easy for OBDDs.

It is easily seen that an (O)BDD is simply an (O)BED without operators. This suggests a strategy for converting BEDs into OBDDs: gradually eliminate the operators, keeping all the intermediate BEDs functionally equivalent. We have developed two very different ways of operator elimination called `UP_ONE` and `UP_ALL`. In the following, it is assumed that the BEDs are ordered (and thus also free). This assumption simplifies the algorithms and does not limit the expressiveness of BEDs (see Section II). It is straightforward to extend the algorithms to work for the general class of (non-ordered, non-free) BEDs and these algorithms have the same asymptotic runtime as the ones presented here. The more general algorithms can be used, e.g., for making a free BDD ordered (i.e., transforming it to an OBDD) or for reordering an OBDD.

#### A. The Up-Step

A key operation on BEDs is the up-step which moves a variable vertex up above an operator vertex. 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

$$(x \rightarrow f_1, f_0) op (x \rightarrow f'_1, f'_0) = x \rightarrow (f_1 op f'_1), (f_0 op f'_0).$$

This identity, illustrated in Fig. 11 (a), is used to move the variable  $x$  above the operator  $op$  and is the basis for the up-step<sup>1</sup>. In cases where one of the children  $u$  does not contain the variable  $x$ , a new variable vertex  $v$ , with  $var(v) = x$  and  $low(v) = high(v) = u$ , is inserted below the operator vertex before performing the up-step, see Fig. 11 (b). In fact, this is the only way the size of the BED can increase when moving a variable towards the root.

The up-step moves operators closer to the terminal vertices. When an operator reaches a terminal, it disappears by requirement (iii) of reducedness (or it reduces to a unary operator, i.e., a projection or a negation). By repeatedly moving variable vertices above operator vertices, all operator vertices are eliminated and the BED is turned into an OBDD.

Fig. 12 shows how  $root_2$  from Fig. 2 is shown to be a tautology by moving the variable  $i_3$  to the top using a series of up-steps. This example illustrates that it may not be necessary to move *all* variable vertices to the root in order to obtain an OBDD. The remaining three variables could have been replaced with arbitrary large BEDs, and the tautology would have been proved with exactly the same steps.

The example illustrates one way to convert a BED to an OBDD, moving the variables to the top one at a time. This approach is called `UP_ONE` and its main advantage is that it can

<sup>1</sup>The equation also holds if the operator vertex  $op$  is a variable vertex. In that case, the up-step is identical to the level exchange operation typically used in OBDDs to dynamically change the variable ordering [37].

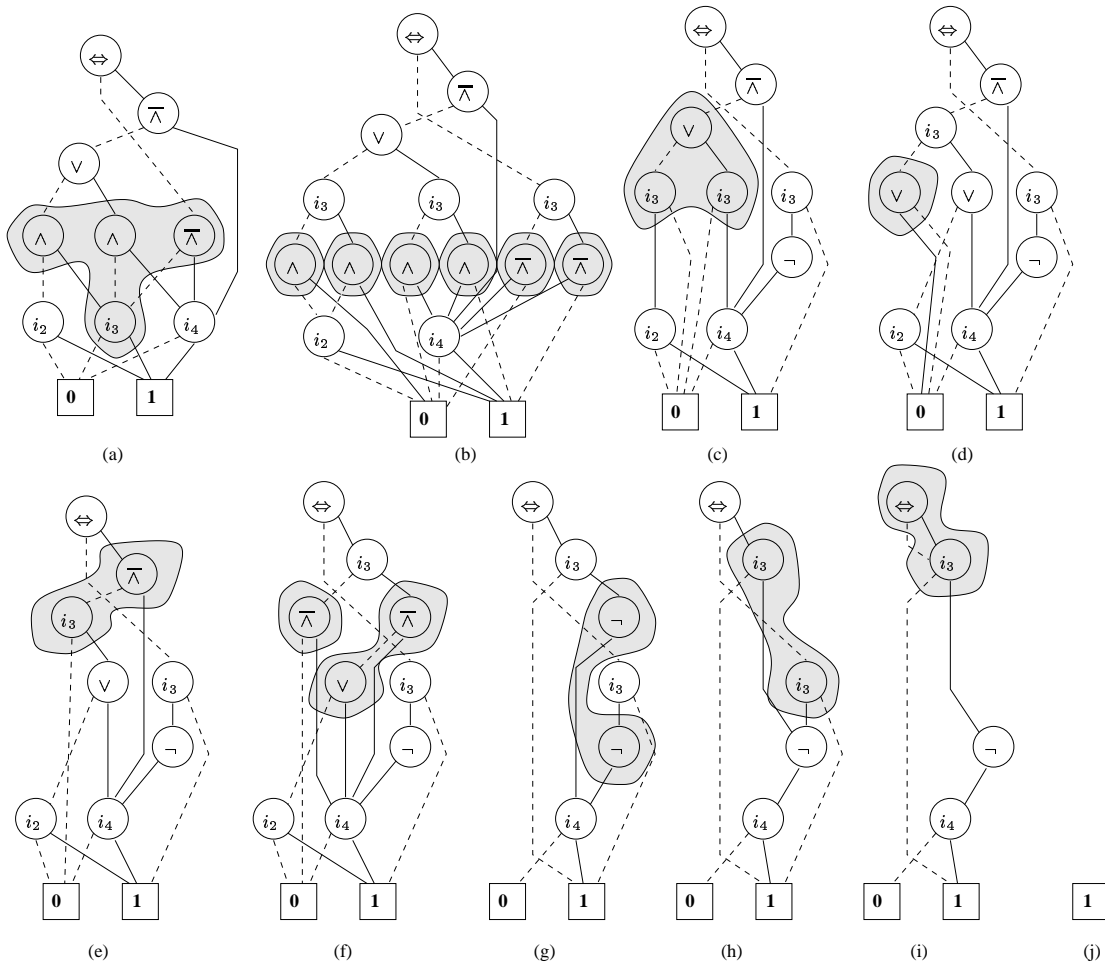


Fig. 12. Proving that  $root_2$  is a tautology. (a) The BED for  $root_2$  from Fig. 2. (b)  $i_3$  is moved above the three conjunctions (one being negated) using three up-steps. Notice that, at this point, variable and operator vertices are no longer separated in two distinct layers. (c) Conjunctions with children that are constant vertices are eliminated. (d)  $i_3$  is moved above the disjunction. (e) The disjunction with both children equal to  $\mathbf{0}$  is removed. (f)  $i_3$  is moved above the negated conjunction. (g) The conjunction with a  $\mathbf{0}$  child is eliminated. The absorption law  $y\overline{\wedge}(x \vee y) = \neg y$  is applied. (h) The negation vertices are identified. (i) Identifying the two vertices with  $i_3$ . At this point the two children of the biimplication operator are identical and (j) the BED is reduced to  $\mathbf{1}$ , proving the tautology.

exploit structural information in the expression (as was the case in the example).

### B. Construction of OBDDs with UP\_ONE

The first elimination algorithm is based on the algorithm UP\_ONE shown in Fig. 13. UP\_ONE pulls a single variable up to the root by performing a recursive depth-first traversal of the BED and after the recursive calls on the low- and high-child of a vertex, it makes an up-step. Repeated calls to UP\_ONE for each variable moves all variables up past the operators, which makes the operators disappear (by requirement (iii) of reducedness). The table  $M$  is used to memoize previously computed results and ensures a linear expected runtime.

The example in Fig. 12 shows the steps of performing  $UP\_ONE(i_3, root_2)$ . As the example illustrates, in fortunate cases a BED is converted into an OBDD after moving just a few variables up (in the example, one variable was sufficient). In this process, identical sub-BEDs, potentially containing operator vertices, are identified. This is quite unlike traditional OBDD construction where all operators are converted in depth-

first order into OBDDs. In particular, an OBDD is constructed for *each sub-expression*. If the result is small and the intermediate OBDDs are large, UP\_ONE is an attractive alternative.

The number of vertices in the BED reachable from a vertex  $u$  is denoted  $|u| = |\{v : u \rightsquigarrow v\}|$ . The following properties hold for UP\_ONE when  $v = UP\_ONE(x, u)$ :

- (i)  $f^v = f^u$ .
- (ii)  $|v| \leq 2|u| - 1$ .
- (iii) The running time of UP\_ONE is  $O(|u|)$ .

To use UP\_ONE to transform a BED  $u$  into an OBDD  $v$  with the variable ordering  $x_1 < \dots < x_n$ , UP\_ONE is called once for each variable in the ordering:

$$v \leftarrow UP\_ONE(x_n, UP\_ONE(x_{n-1}, \dots UP\_ONE(x_1, u) \dots)).$$

Even though UP\_ONE has linear runtime and it is called only  $n$  times, the runtime of this computation is exponential in the worst case due to the potential increase in size of the intermediate results.

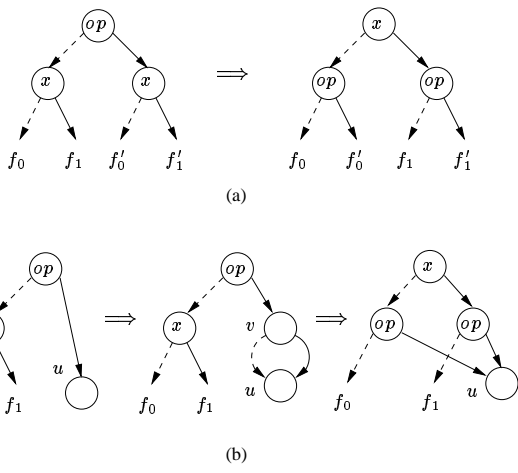


Fig. 11. Illustration of the up-step (a) for the case where variable  $x$  exists in both children of the root and (b) for the case where  $x$  only occurs in the left child.

```

UP_ONE( $x, u$ ) =
1: if ( $x, u$ ) in  $M$  then return  $M(x, u)$ 
2: else if  $u$  is a terminal then return  $u$ 
3: else if  $\alpha(u)$  is variable  $x$  then return  $u$ 
4: else
5:   ( $l, h$ )  $\leftarrow$  (UP_ONE( $x, low(u)$ ), UP_ONE( $x, high(u)$ ))
6:   if  $\alpha(u)$  is a variable with  $var(u) < x$  then
7:      $r \leftarrow mk(var(u), l, h)$ 
8:   else if  $\alpha(l)$  and  $\alpha(h)$  are both variable  $x$  then
9:      $r \leftarrow mk(x, mk(\alpha(u), low(l), low(h)),$ 
10:       $mk(\alpha(u), high(l), high(h)))$ 
11:   else if  $\alpha(l)$  is variable  $x$  then
12:      $r \leftarrow mk(x, mk(\alpha(u), low(l), h),$ 
13:       $mk(\alpha(u), high(l), h))$ 
14:   else if  $\alpha(h)$  is variable  $x$  then
15:      $r \leftarrow mk(x, mk(\alpha(u), l, low(h)),$ 
16:       $mk(\alpha(u), l, high(h)))$ 
17:   else
18:      $r \leftarrow mk(\alpha(u), l, h)$ 
19:   insert ( $(x, u), r$ ) in  $M$ 
20:   return  $r$ 

```

Fig. 13. The UP\_ONE-operation. UP\_ONE takes an ordered BED  $u$  as argument and returns an equivalent BED with  $x$  pulled up as far as possible without violating the ordering. The imposed total order on the variable vertices is denoted  $<$ . The memoization table  $M$  must be initialized to empty prior to the first call.

### C. Construction of OBDDs with UP\_ALL

An alternative way to construct an OBDD is to move all variables up simultaneously, called UP\_ALL. UP\_ALL is a generalization of Bryant's APPLY-operator, shown in Fig. 14. Construction of OBDDs from a Boolean expression using recursive calls of APPLY suggests a bottom up conversion of BEDs into OBDDs. The UP\_ALL algorithm does that by moving all variables up as a block past the operator vertices. UP\_ALL is shown in Fig. 15.

Let  $u$  be a vertex in a BED and let  $v = UP\_ALL(u)$ . Then

```

APPLY( $op, l, h$ ) =
if ( $l, h$ ) in  $M$  then return  $M(l, h)$ 
else if  $l$  and  $h$  are terminals then
   $r \leftarrow op(value(l), value(h))$ 
else if  $var(l) = var(h)$  then
   $r \leftarrow mk(var(l), APPLY(op, low(l), low(h)),$ 
   $APPLY(op, high(l), high(h)))$ 
else if  $var(l) < var(h)$  then
   $r \leftarrow mk(var(l), APPLY(op, low(l), h),$ 
   $APPLY(op, high(l), h))$ 
else  $var(l) > var(h)$  :
   $r \leftarrow mk(var(h), APPLY(op, l, low(h)),$ 
   $APPLY(op, l, high(h)))$ 
insert ( $(l, h), r$ ) in  $M$ 
return  $r$ 

```

Fig. 14. The APPLY-operation. Assumes  $l$  and  $h$  are OBDDs. The imposed total order on the variable vertices is denoted  $<$ . In the code it is assumed that terminal vertices are included at the end of this order when comparing  $var(l)$  and  $var(h)$ . The memoization table  $M$  must be initialized to empty prior to the first call.

```

UP_ALL( $u$ ) =
1: if  $u$  in  $M$  then return  $M(u)$ 
2: else if  $u$  is a terminal then return  $u$ 
3: else
4:   ( $l, h$ )  $\leftarrow$  (UP_ALL( $low(u)$ ), UP_ALL( $high(u)$ ))
  /*  $l$  and  $h$  are OBDDs */
5:   if  $l$  and  $h$  are terminal vertices then
6:      $r \leftarrow mk(\alpha(u), l, h)$ 
7:   else if  $\alpha(u)$  is a variable  $x$  then
8:      $r \leftarrow mk(x, l, h)$ 
9:   else if  $var(l) = var(h)$  then
10:     $r \leftarrow mk(var(l), UP\_ALL(mk(\alpha(u), low(l), low(h))),$ 
11:     $UP\_ALL(mk(\alpha(u), high(l), high(h))))$ 
12:   else if  $var(l) < var(h)$  then
13:     $r \leftarrow mk(var(l), UP\_ALL(mk(\alpha(u), low(l), h)),$ 
14:     $UP\_ALL(mk(\alpha(u), high(l), h))$ 
15:   else  $var(l) > var(h)$  :
16:     $r \leftarrow mk(var(h), UP\_ALL(mk(\alpha(u), l, low(h))),$ 
17:     $UP\_ALL(mk(\alpha(u), l, high(h))))$ 
18:   insert ( $(u, r)$ ) in  $M$ 
19:   return  $r$ 

```

Fig. 15. The UP\_ALL-operation on OBDDs. The total order  $<$  is defined as for APPLY (see Fig. 14). The memoization table  $M$  must be initialized to empty prior to the first call.

UP\_ALL has the following key properties:

- (i)  $f^v = f^u$ .
- (ii)  $v$  is an OBDD.
- (iii) If  $l$  and  $h$  are OBDDs, then  $APPLY(op, l, h) = UP\_ALL(makenode(op, l, h))$ .
- (iv) If  $l$  and  $h$  are OBDDs, the running time of  $UP\_ALL(op, l, h)$  is  $O(|l||h|)$ .

Properties (iii) and (iv) make clear the relation between UP\_ALL and APPLY. The time to build an OBDD bottom up using APPLY (the standard way) and building it from a BED using UP\_ALL is within a constant factor. Experiments have shown that the time to construct an OBDD using UP\_ALL is comparable to that of state-of-the-art OBDD packages and due to the operator reductions, it can be significantly faster. However, the worst-case runtime of UP\_ALL is exponential in  $|u|$ , but for the same reason as UP\_ONE, this is optimal.

The efficiency of UP\_ONE and UP\_ALL depends on the variable order. Although the initial and final size of the BEDs are independent on the variable order when the two circuits implement the same function and thus the result is the tautology 1, the size of the intermediate BEDs depend on the ordering.

A large number of variable ordering heuristics have been developed for OBDDs based on the topology of a circuit [33], [38], [39], [40], [41], [42], [43]. The heuristics attempt to statically determine a variable order such that the OBDD representation of the circuit is small. Typically, these heuristics consist of two steps to obtain a single global variable order: first, an order of the primary outputs is constructed, then for each of the primary outputs in this order, the variables in the support of the output are ordered. We only consider the second step (finding a variable order for a *given* primary output), since different variable orders can be used for different roots of a BED (see Fig. 10). This allows a greater flexibility to find good variable orders since the orders of the primary outputs are independent. However, the cost is that there is only limited reuse between verifying different primary outputs.

Since UP\_ALL essentially works as an improved APPLY (property (iii)), the variable orders that are good for OBDDs will also be good orders to use with UP\_ALL. Thus, when using UP\_ALL we can immediately use the variable ordering heuristics developed for OBDDs.

Since UP\_ONE works quite differently than UP\_ALL, the variable ordering heuristics developed for OBDDs may not be effective when using UP\_ONE. However, our experiments show that this is not so; a good OBDD variable order also keeps the intermediate BEDs small when constructing an OBDD with UP\_ONE. The reason for this is that a good variable order for OBDDs has dependent variables close in the order. This allows UP\_ONE to collapse sub-circuits early in the verification process. Also, a good variable order has the variables that affect the output the most early in the order. UP\_ONE will then pull these variables to the root first which allows the most reductions. An example of this was the use of  $i_3$  in the introductory example in Fig. 3.

In the following we present two variable ordering heuristics, originally developed for OBDDs, which have proven to be effective for BEDs.

A number of variable ordering heuristics are based on a depth-first traversal of the circuit [39], [41], [42]. A depth-first traversal is a simple and fast heuristic that has shown to be practical for most combinational circuits [33], [39] since inputs that are close together in the circuit are also placed together in the ordering. The depth-first based heuristics differ in how they decide in what order the inputs of a gate are visited. The FANIN heuristic by Malik *et al.* [42] uses the *depth* of the inputs to a gate to determine in what order to consider the inputs. The depth of a terminal or variable vertex is 0 and the depth of an operator vertex  $u$  is  $\max(\text{depth}(\text{low}(u)), \text{depth}(\text{high}(u))) + 1$ . The total runtime of FANIN to determine the variable order of  $m$  roots is  $O(mn)$  [42] where  $n$  is the total number of reachable vertices from the  $m$  roots.

The FANIN heuristic does not capture that variables affecting the output the most should be ordered first, something which

is particularly important for UP\_ONE. The DEPTH\_FANOUT heuristic [43] attempts to determine the variables that affect an output the most by propagating a value from the output backwards towards the primary inputs. The value is distributed evenly among the input signals to a gate: if a value of  $c$  is assigned to the output of a gate with  $n$  input signals, the value assigned to each of the  $n$  fanin signals is incremented by  $c/n$  (the signal may be input to several gates and thus obtains a contribution from each gate). After propagating the value throughout the circuit to the primary inputs, the DEPTH\_FANOUT heuristic adds the primary input with the highest value to the variable order. This input is then removed from the circuit and the process is repeated until all variables in the support have been included in the variable order. The runtime of DEPTH\_FANOUT( $u$ ) is  $O(kmn)$  where  $n$  is the total number of reachable vertices from the  $m$  roots and  $k$  is the number of variables (inputs to the circuit). Thus, this heuristic takes slightly longer to compute than FANIN.

## V. EXPERIMENTAL RESULTS

In this section, we report the results from verifying a number of multi-level combinational circuits from the ISCAS 85 and LGSynth 91 benchmarks<sup>2</sup>.

The ISCAS 85 benchmark consists of eleven multi-level combinational circuits, nine of which exist both in a redundant and a non-redundant version. Furthermore, the benchmark contains five circuits that originally were believed to be non-redundant versions but it turned out that they contained errors and weren't functionally equivalent to the original circuits [20].

The circuits in the ISCAS 85 benchmark are by some researchers considered too simple to use as benchmark circuits with today's technology. This may be true for some application areas but these circuits have several properties that make them suitable as benchmark circuits for evaluating techniques for performing a combinational logic-level verification. First, the circuits, although quite small, are not easy to verify both due to their functionality (for example, one of the circuits is a multiplier for which OBDD techniques fail) and due to a rather large logic-depth (up to 125 logic levels). Even with recent structural techniques, some of these circuits take more than an hour to verify [21]. Secondly, the circuits in the ISCAS 85 benchmark are ideally suited for testing logic-level verification techniques since they come in two functionally equivalent versions.

To evaluate the BED technique on a broader and more realistic class of circuits, we also consider the 77 multi-level combinational circuits and the 40 sequential circuits from the LGSynth 91 benchmark. These circuits do not come in two versions, so instead we map each of the circuits to a gate library using SIS [7] and then optimize the circuits with respect to area. We then verify that 1) the mapped circuit corresponds to the original description, and 2) that the mapped and the optimized circuits implement the same functionality. Due to the nature of the mapping and optimization steps, the circuits differ in structure considerably more than the ISCAS 85 circuits.

All experiments are carried out on a 300 MHz Pentium II PC running Linux. Verification approaches based on decision dia-

<sup>2</sup>These benchmarks are available from The Collaborative Benchmarking Laboratory (<http://www.cbl.ncsu.edu/>)

grams typically run out of memory before running out of time. Thus, to demonstrate the effectiveness of BEDs, in all experiments we limit the memory consumption to 32 MB divided between 28 MB of memory to the node table (that is 1.46 million nodes corresponding to 20 bytes per node) and 4 MB to caches.

The runtimes to determine the variable orders are insignificant (at most two seconds for any of the circuits) when using the FANIN heuristic. Using the DEPTH\_FANOUT heuristic it takes less than three seconds for the ISCAS 85 circuits, less than five seconds for the combinational LGSynth 91 circuits, and less than ten seconds for the sequential LGSynth 91 circuits except for the circuits `s15850.1`, `s38417`, and `s38584.1` which take 33.7, 111.8, and 94.1 seconds, respectively. The times to determine variable orders are not included in the CPU times reported in the following, making a direct comparison between the different verification approaches possible.

### A. The ISCAS 85 circuits

Table I shows the size of the ISCAS 85 circuits and Table II shows the runtimes to perform the equivalence check using BEDs. When using UP\_ONE, the DEPTH\_FANOUT heuristic

TABLE I  
SIZE OF THE ISCAS 85 BENCHMARK CIRCUITS.

Circuit	Inputs	Outputs	Gates
c432/nr	36	7	433
c499/nr	41	32	516
c499/c1355	41	32	868
c1355/nr	41	32	1204
c1908/nr	33	25	2134
c2670/nr	157	63	2603
c3540/nr	50	22	3901
c5315/nr	178	123	6018
c6288/nr	32	32	4847
c7552/nr	207	107	8067

TABLE II  
RUNTIMES (IN CPU SECONDS) FOR VERIFYING EQUIVALENCE OF THE REDUNDANT AND NON-REDUNDANT CIRCUITS IN THE ISCAS 85 BENCHMARK. FOR EACH PAIR OF CIRCUITS, THE TIME IS GIVEN WHEN USING THE TWO DIFFERENT VARIABLE ORDERING HEURISTICS FANIN AND DEPTH\_FANOUT (ABBREVIATED D.\_F.) AND USING THE TWO DIFFERENT ALGORITHMS FOR TRANSFORMING A BED INTO AN OBDD, UP\_ONE AND UP\_ALL. THE BEST RUNTIMES ARE HIGHLIGHTED USING BOLDFACE. A ‘-’ REPRESENTS THAT THE VERIFICATION FAILED DUE TO LACK OF MEMORY.

Circuit	UP_ONE		UP_ALL	
	FANIN	D._F.	FANIN	D._F.
c432/nr	2.5	2.2	<b>2.1</b>	2.2
c499/nr	5.2	2.6	<b>2.4</b>	2.6
c499/c1355	<b>1.6</b>	<b>1.6</b>	<b>1.6</b>	<b>1.6</b>
c1355/nr	5.3	<b>2.6</b>	<b>2.6</b>	<b>2.6</b>
c1908/nr	<b>1.0</b>	<b>1.0</b>	<b>1.0</b>	<b>1.0</b>
c2670/nr	1.4	1.2	<b>1.0</b>	<b>1.0</b>
c3540/nr	<b>16.9</b>	21.8	17.0	33.7
c5315/nr	17.8	<b>3.1</b>	<b>3.1</b>	<b>3.1</b>
c6288/nr	<b>2.0</b>	-	-	-
c7552/nr	4.6	3.7	<b>2.6</b>	<b>2.6</b>

generally computes a better variable order than FANIN, while

there is little difference between the two ordering heuristics when using UP\_ALL.

The performance of UP\_ONE and UP\_ALL is comparable except for the circuit `c6288` where only UP\_ONE succeeds. This circuit implements a 16-bit multiplier for which it is known that the OBDD representation grows exponentially [3]. The OBDD representation of a 16-bit multiplier uses more than 40 million vertices [44] and this number is approximately 2.7 times larger for each additional bit in the operands. Thus, UP\_ALL will fail on this circuit no matter what variable ordering is used. In contrast, UP\_ONE never builds the OBDD representation of the multiplication function and thus the circuits can be verified in just a few seconds.

### A.1 Proving Non-Equivalence

Table III shows the runtimes to determine non-equivalence of the erroneous ISCAS 85 circuits. The reported CPU times are for finding *all* errors. Although it does take longer to prove non-equivalence, as expected since less equivalences exist, the increase in the runtimes is insignificant.

TABLE III  
RUNTIMES (IN CPU SECONDS) FOR SHOWING NON-EQUIVALENCE OF THE REDUNDANT AND NON-REDUNDANT CIRCUITS IN THE ISCAS 85 BENCHMARK.

Circuit	# errs.	UP_ONE		UP_ALL	
		FANIN	D._F.	FANIN	D._F.
c1908/nr_old	1	<b>1.0</b>	<b>1.0</b>	<b>1.0</b>	<b>1.0</b>
c2670/nr_old	6	6.5	-	<b>1.3</b>	-
c3540/nr_old	5	26.6	23.3	<b>17.3</b>	27.9
c5315/nr_old	33	29.1	4.3	3.6	<b>3.5</b>
c7552/nr_old	28	7.0	8.7	<b>2.9</b>	3.6

### A.2 Effect of Operator Reductions

To illustrate the effect of operator reductions, we repeat the experiments in Table II and III but without performing any of the operator reductions described in Section II-C. The only reductions performed are those required to maintain reducedness, see Definition 1. The operation of UP\_ALL then reduces to that of APPLY, that is, the performance of UP\_ALL corresponds very closely to that of APPLY in a reasonable implementation of an OBDD package. The results are shown in Table IV. Clearly, the efficiency of UP\_ONE relies heavily on the operator reductions to identify identical nodes in the BED and thus avoiding to transforming them into OBDDs. Without reductions, a large number of the circuits cannot be verified (with 32 MB of memory) and the runtimes for those that do succeed are up to several orders of magnitude longer.

When using UP\_ALL the situation is quite different. In building an OBDD using UP\_ALL, any vertex that is constructed during the transformation will have non-operator vertices as the children. I.e., whenever  $\text{makenode}(\alpha, l, h)$  is called in the body of UP\_ALL, both  $l$  and  $h$  are variable or terminal vertices. Thus, the operator reductions only affect the performance of UP\_ALL by reducing the initial size of the BED. For some circuits (e.g., `c3540`) this initial reduction has a large impact on the runtime

TABLE IV

RUNTIMES (IN CPU SECONDS) FOR VERIFYING EQUIVALENCE OF THE REDUNDANT AND NON-REDUNDANT CIRCUITS IN THE ISCAS 85 BENCHMARK WITHOUT PERFORMING OPERATOR REDUCTIONS.

Circuit	UP_ONE		UP_ALL	
	FANIN	D._F.	FANIN	D._F.
c432/nr	2.9	2.6	2.5	2.3
c499/nr	166.1	—	2.5	4.1
c499/c1355	532.5	—	3.9	4.8
c1355/nr	743.5	—	4.1	4.8
c1908/nr	—	15.3	1.0	1.0
c2670/nr	—	29.0	1.4	1.9
c3540/nr	111.7	—	56.9	145.4
c5315/nr	—	4.8	3.5	3.3
c6288/nr	—	—	—	—
c7552/nr	7.4	7.2	3.1	3.7
c1908/nr_old	—	14.9	1.0	1.0
c2670/nr_old	—	—	1.5	—
c3540/nr_old	116.4	—	60.4	126.4
c5315/nr_old	—	6.0	4.0	3.7
c7552/nr_old	10.0	15.2	3.1	4.1

of UP\_ALL. This experiment indicates that performing an initial operator reduction step, as discussed in the introduction, can improve the time to construct an OBDD.

### B. The LGSynth 91 Benchmarks

By construction, the redundant and non-redundant versions of the ISCAS 85 benchmark circuits have many structural similarities and are thus ideally suited for the BED approach. To test the verification strategy on a broader range of circuits with fewer structural similarities, we consider the circuits from the LGSynth 91 benchmark. This benchmark includes 77 multi-level combinational circuits and 40 sequential circuits. These circuits are mapped to a gate library (`msu.genlib`) using SIS and then optimized for area using the SIS script `script.algebraic`. As mentioned above, there are two verification problems: one is to verify that the original circuits correspond to the mapped versions and one is to verify the mapped versions against the optimized versions. Due to the nature of the mapping and optimization steps, the circuits differ in structure considerably more than the ISCAS 85 circuits. The results for the 77 combinational circuits are shown graphically in Fig. 16. The eleven ISCAS 85 circuits are included in the LGSynth 91 benchmark and although some of the LGSynth 91 circuits are considerably larger than the ISCAS 85 circuits, the ISCAS circuits are the most difficult ones to verify using the BED approach. The mapping of each circuit is verified in less than three minutes. The results for the verification of the optimization step are similar, except that the verification of the optimization step of C6288 failed using both UP\_ONE and UP\_ALL.

The results for the verification of (the combinational portion of) the 40 sequential circuits in the LGSynth 91 benchmark are shown in Fig. 17. The mapping of the traditionally difficult circuit `s38417` takes 20 minutes to verify and the verification of `mm9b` fails for both variable ordering heuristics. The mapping of the remaining 38 circuits is verified in less than one minute. The optimization of each circuit, except for `mm9b` and `s38417`, is also verified in less than one minute. The verification of `mm9b`

and `s38417` both fail when using 32 MB of memory. Using 64 MB of memory, `s38417` is verified in an hour using the FANIN ordering heuristic. The circuit `mm9b` is an instance where the two ordering heuristics fail to construct a good variable order, thus the verification of both the mapping and the optimization steps fail, even when using 64 MB of memory. Using the order in which the variables appear in the specification, the mapping of `mm9b` is verified using 64 MB of memory in 207 seconds and 270 seconds using UP\_ONE and UP\_ALL, respectively. Similarly, the optimization of `mm9b` is verified in 69 seconds and 115 seconds using UP\_ONE and UP\_ALL, respectively.

### C. Comparisons of Results

The ISCAS 85 benchmark has been used extensively by researchers to test techniques for solving the equivalence problem. Since all researchers have solved the exact same verification problems, there is a good basis for comparing the different approaches. Table V shows the runtimes to verify the ISCAS 85 circuits using recent methods.

The experiments are carried out on different machines and are therefore not directly comparable. However, the efficiency of the machines only differ by a small constant and not by orders of magnitude and therefore the comparisons still give a good indication of the relative virtues of the different approaches. The experiments of Brand [18] is an exception since he does not report runtimes for comparing the redundant and the non-redundant versions. Instead the circuits are synthesized and optimized, much in the same way as we have done for the LGSynth 91 circuits. This might well be a more difficult verification problem. From Table V it is clear that the learning-based approaches [21],

TABLE V

RUNTIMES (IN CPU SECONDS) OF OTHER APPROACHES FOR VERIFYING THE ISCAS 85 BENCHMARKS. NOTICE THAT THE RESULTS OF BRAND [18] ARE NOT DIRECTLY COMPARABLE SINCE A DIFFERENT VERIFICATION PROBLEM IS SOLVED. "N/A" DENOTES THAT THE RUNTIME HAS NOT BEEN REPORTED.

Circuit	BED	[18]	[21]	[22]	[24]	[26]	[45]
c432/nr	2.1	4.0	1.0	2.0	0.8	0.2	0.4
c499/nr	2.4	38.0	1.9	5.0	1.2	0.2	0.4
c1355/nr	2.5	9.0	6.6	20.0	3.4	0.5	1.0
c1908/nr	1.0	22.0	11.2	22.0	6.2	1.6	2.1
c2670/nr	1.0	58.0	159.3	61.0	3.9	0.8	3.4
c3540/nr	16.9	39.0	67.6	281.0	17.4	3.0	12.7
c5315/nr	3.1	29.0	372.8	190.0	14.0	2.7	8.3
c6288/nr	2.0	193.0	21.5	40.0	9.1	4.3	7.2
c7552/nr	2.6	136.0	5583.3	412.0	20.6	34.6	20.8
c1908/old	1.0	n/a	n/a	n/a	n/a	2.5	n/a
c2670/old	1.3	n/a	n/a	n/a	n/a	54.6	n/a
c3540/old	17.3	n/a	n/a	n/a	n/a	2.9	n/a
c5315/old	3.5	n/a	n/a	n/a	n/a	8.3	n/a
c7552/old	2.9	n/a	n/a	n/a	n/a	26.2	n/a

[22] are inefficient for larger circuits. The runtimes of the BED approach is generally comparable to (and sometimes better than) the other three approaches [24], [26], [45]. Moreover, these runtimes should be seen in the light of the fact that the BED experiments only use 32 MB of memory.

Only van Eijk [26] has reported runtimes for the erroneous ISCAS 85 circuits. For these circuits it is observed that the BED



PC using only 32 MB of memory. Even known difficult circuits (like  $\epsilon 38417$ ) are verified using modest resources (in less than an hour with 64 MB of memory). This demonstrates that BEDs combined with very simple algorithms are effective for solving the combinational logic-level equivalence problem.

BEDs can be seen as an intermediate form between the compact circuits and the canonical OBDDs, and thus combines the functional and the structural verification techniques. All standard OBDD operations can be performed on BEDs as well. Some operations, like existential quantification and substitution, can be performed directly on the data structure by using UP\_ONE [4], making the runtime of these operations linear in the size of the BED. Other operations, like satisfiability and tautology, can be performed by transforming the BED into an equivalent OBDD.

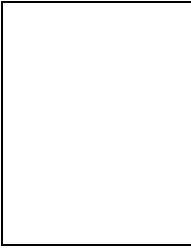
Due to the simplicity and generality of BEDs, it is to be expected that combining them with other approaches to equivalence checking will be both straightforward and beneficial. The benefits could be in two directions. Firstly, BEDs could be used either as a “filter” in the filter-based approaches [45] or as the basic data structure for representing circuits in any of the other approaches allowing, for instance, a gradual and smooth transition from circuits to BDDs. Secondly, BEDs could benefit from other equivalence checkers. Whenever two subcircuits by some means can be determined equivalent, the corresponding nodes of the BED could be merged into one, resulting in an immediate reduction in size. Moreover, this immediate reduction could result in further reductions being possible by the reduction rules and improve on the efficiency on later BDD-conversions.

BEDs are particularly useful in applications where the end-result as an OBDD 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 OBDDs, the intermediate results are often much larger than the final result. Clearly, the succinctness of BEDs compared to OBDDs can alleviate this problem. In fact, many of the tricks researchers have used to make OBDDs more efficient are embodied in BEDs. For example, Burch, Clarke, and Long [46] 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 OBDDs. 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.

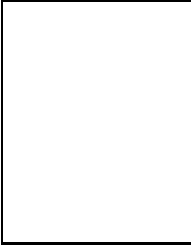
## REFERENCES

- [1] M. Fujita, “Verification of arithmetic circuits by comparing two similar circuits,” in *Computer Aided Verification (CAV)*, 1996, Lecture Notes in Computer Science, pp. 159–168, Springer-Verlag.
- [2] C.A.J. van Eijk, “Sequential equivalence checking without state space traversal,” in *Proc. International Conf. on Design Automation and Test of Electronic-based Systems (DATE)*, 1998.
- [3] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, Aug. 1986.
- [4] H. R. Andersen and H. Hulgaard, “Boolean Expression Diagrams,” in *IEEE Symposium on Logic in Computer Science (LICS)*, July 1997.
- [5] R. B. Boppana and M. Sipser, “The complexity of finite functions,” in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. A: Algorithms and Complexity, pp. 758–804. Elsevier Science Publisher, 1990.
- [6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
- [7] E. Sentovich et al., “SIS: A system for sequential circuit synthesis,” Tech. Rep. Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, Dept. of EECS, University of California, Berkeley, 1992.
- [8] 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)*, 1994, pp. 415–419.
- [9] U. Kebeschull, E. Schubert, and W. Rosenstiel, “Multilevel logic synthesis based on functional decision diagrams,” in *Proc. European Conference on Design Automation (EDAC)*, 1992, pp. 43–47.
- [10] J. Gergov and C. Meinel, “Efficient Boolean manipulation with OBDD’s can be extended to FBDD’s,” *IEEE Transactions on Computers*, vol. 43, no. 10, pp. 1197–1209, Oct. 1994.
- [11] E. I. Goldberg, Y. Kukimoto, and R. K. Brayton, “Canonical TBDD’s and their application to combinational verification,” in *Proc. International Workshop on Logic Synthesis*, 1997.
- [12] J. Jain, J. Bitner, M. S. Abadir, and J. A. Abraham and D. S. Fussell, “Indexed BDDs: Algorithmic advances in techniques to represent and verify Boolean functions,” *IEEE Transactions on Computers*, vol. 46, no. 11, pp. 1230–1245, Nov. 1997.
- [13] D. Sieling and I. Wegener, “Graph driven BDDs – a new data structure for Boolean functions,” *Theoretical Computer Science*, vol. 141, no. 1-2, pp. 283–310, 1995.
- [14] 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)*, 1993, pp. 188–191.
- [15] R. E. Bryant and Y.-A. Chen, “Verification of arithmetic functions with binary moment diagrams,” in *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1995, pp. 535–541.
- [16] 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)*, 1993, pp. 54–60.
- [17] A. Kuehlmann and F. Krohm, “Equivalence checking using cuts and heaps,” in *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1997, vol. 34, pp. 263–268.
- [18] D. Brand, “Verification of large synthesized designs,” in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1993, pp. 534–537.
- [19] W. Kunz, “HANNIBAL: An efficient tool for logic verification based on recursive learning,” in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1993, pp. 538–543.
- [20] 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*, vol. 13, no. 9, pp. 1143–1158, Sept. 1994.
- [21] 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*, vol. 15, no. 1, pp. 20–32, Jan. 1996.
- [22] D. K. Pradhan, D. Paul, and M. Chatterjee, “VERILAT: Verification using logic augmentation and transformations,” in *Proc. International Conf. Computer-Aided Design (ICCAD)*, Nov. 1996.
- [23] J. Jain, R. Mukherjee, and M. Fujita, “Advanced verification techniques based on learning,” in *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1995, pp. 629–634.
- [24] Y. Matsunaga, “An efficient equivalence checker for combinational circuits,” in *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1996, pp. 629–634.
- [25] 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*, 1994, number 901 in Lecture Notes in Computer Science, pp. 110–125, Springer-Verlag.
- [26] C.A.J. van Eijk, *Formal Methods for the Verification of Digital Circuits*, Ph.D. thesis, Technische Universitet Eindhoven, 1997.
- [27] E. Cerny and C. Murras, “Tautology checking using cross-controllability and cross-observability relations,” in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1990.
- [28] T. E. Uribe and M. E. Stickel, “Ordered binary decision diagrams and the Davis-Putnam procedure,” in *1st International Conference on Constraints in Computational Logics*, J.P. Jouannaud, Ed., Sept. 1994, vol. 845 of *Lecture Notes in Computer Science*.
- [29] G.D. Hachtel and R.M. Jacoby, “Verification algorithms for VLSI synthesis,” *IEEE Transactions on Computer Aided Design*, pp. 616–640, May 1988.

- [30] A. Hett, R. Drechsler, and B. Becker, "MORE: Alternative implementation of BDD-packages by multi-operand synthesis," in *European Design Conference*, 1996.
- [31] A. Hett, R. Drechsler, and B. Becker, "Fast and efficient construction of BDDs by reordering based synthesis," in *IEEE European Design & Test Conference*, 1997.
- [32] 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)*, 1991, pp. 50–54.
- [33] S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi, "Extended BDD's: Trading off canonicity for structure in verification algorithms," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1991, pp. 464–467.
- [34] B. Plessier, G. D. Hachtel, and F. Somenzi, "Extended BDD's: Trading off canonicity for structure in verification algorithms," *Formal Methods in System Design*, vol. 4, no. 2, pp. 167–185, Feb. 1994.
- [35] M. Nikol'skaia, A. Rauzy, and D. J. Sherman, "Almana: A BDD minimization tool integrating heuristic and rewriting methods," in *Formal Methods in Computer Aided Design*, Nov. 1998.
- [36] M. R. Garey and D. S. Johnson, *Computers and Intractability—A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.
- [37] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1993, pp. 42–47.
- [38] K. M. Butler, D. E. Ross, R. Kapur, and M. R. Mercer, "Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams," in *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1991, pp. 417–420.
- [39] P.-Y. Chung, I. N. Hajj, and J. H. Patel, "Efficient variable ordering heuristics for shared ROBDD," in *Proc. International Symposium on Circuits and Systems (ISCAS)*, 1993, pp. 1690–1693.
- [40] H. Fujii, G. Ootomo, and C. Hori, "Interleaving based variable ordering methods for ordered binary decision diagrams," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1993, pp. 38–41.
- [41] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and improvements of Boolean comparison methods based on Binary Decision Diagrams," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, Nov. 1988, pp. 2–5.
- [42] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1988, pp. 6–9.
- [43] S. Minato, *Binary Decision Diagrams and Applications for VLSI CAD*, Kluwer Academic Publishers, 1996.
- [44] B. Yang, Y.-A. Chen, R. E. Bryant, and D. R. O'Hallron, "Space- and time-efficient BDD construction via working set control," in *ASP-DAC '98*, Feb. 1998, pp. 423–432.
- [45] R. Mukherjee, K. Takayama J. Jain, M. Fujita, J. A. Abraham, and D. S. Fussell, "Flover: Filtering oriented combinational verification approach," International Workshop on Logic Synthesis, May 1997.
- [46] 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)*, 1991, pp. 403–407.

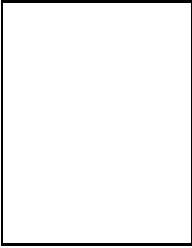


**Poul Frederick Williams** received the M.Sc.Eng. degree from the Technical University of Denmark in 1997. In 1995-96 he studied at the Rheinisch-Westfälische Technische Hochschule in Aachen, Germany. He is presently a Ph.D. candidate in the area of computer science at the Technical University of Denmark. His research interests include formal verification and computer aided design. In 1998 he received a third place in the robot competition, RoboCup, at the Technical University of Denmark.



**Henrik Reif Andersen** received the M.Sc. and Ph.D. degree in Computer Science from Aarhus University in 1990 and 1993, respectively. He is currently an Associate Professor with the Department of Information Technology at the Technical University of Denmark. His primary research interest is in automatic formal verification of concurrent and reactive systems. His secondary research interest is in modeling, programming, and testing of embedded software. He is currently holding a grant from the Danish Technical Research Council funding the research project VERIS.

This project is devoted to the study of algorithms and data structures for efficiently verifying interacting systems.



**Henrik Hulgaard** received the M.S. degree in Electrical Engineering from the Technical University of Denmark in 1990 and the M.S. and Ph.D. degree in Computer Science from the University of Washington in 1992 and 1995, respectively. He is currently an Assistant Professor with the Department of Information Technology at the Technical University of Denmark. His primary research interest is timing analysis and verification of asynchronous circuits and real-time systems. His secondary research interest is in formal verification of concurrent systems, with particular emphasis on the analysis of embedded software. He received the best CAD paper award at the 1993 IEEE International Conference on Computer Design.

emphasis on the analysis of embedded software. He received the best CAD paper award at the 1993 IEEE International Conference on Computer Design.