

# Boolean Expression Diagrams

(Appears in LICS'97)

Henrik Reif Andersen and Henrik Hulgaard

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

## Abstract

*This paper presents a new data structure called Boolean Expression Diagrams (BEDs) for representing and manipulating Boolean functions. BEDs are a generalization of Binary Decision Diagrams (BDDs) which can represent any Boolean circuit in linear space and still maintain many of the desirable properties of BDDs. Two algorithms are described for transforming a BED into a reduced ordered BDD. One closely mimics the BDD apply-operator while the other can exploit the structural information of the Boolean expression. The efficacy of the BED representation is demonstrated by verifying that the redundant and non-redundant versions of the ISCAS 85 benchmark circuits are identical. In particular, it is verified that the two 16-bit multiplication circuits (c6288 and c6288nr) implement the same Boolean functions. Using BEDs, this verification problem is solved in less than a second, while using standard BDD techniques this problem is infeasible. BEDs are useful in applications where the end-result as a reduced ordered BDD is small, for example for tau-ology checking.*

## 1. Introduction

Within the last decade *Reduced Ordered Binary Decision Diagrams* (ROBDDs) introduced by Bryant [4] have become a successful data structure for representing and manipulating Boolean functions. This success is due to the fact that ROBDDs are canonical (making testing of functional properties such as satisfiability and equivalence straightforward) and that they are compact for many Boolean functions occurring in practice. However, the applicability of ROBDDs depends heavily on the size of the representation and unfortu-

nately some (important) functions, e.g., the multiplication function, have no sub-exponential representation.

This paper presents an extension of BDDs, called Boolean Expression Diagrams (BEDs). BEDs can represent any Boolean circuit [2] in linear space and still maintain many of the desirable properties of ROBDDs. This is obtained by extending the BDD representation with operator vertices:

### Definition 1 (Boolean Expression Diagram)

*A Boolean Expression Diagram (BED) is a directed acyclic graph with vertex set  $V$  and edge set  $E$ . The vertex set  $V$  contains 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  $low(v), high(v) \in V$ .
- 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  $E$  is defined by

$$E = \{(v, low(v)), (v, high(v)) \mid v \in V \text{ and } v \text{ is not a terminal vertex}\}.$$

We use  $\mathbf{0}$  and  $\mathbf{1}$  to denote the two terminal vertices. 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  $f_v = \text{value}(v)$ .
- If  $v$  is a variable vertex, then  $f_v$  is the function

$$f_v = \text{var}(v) \rightarrow f_{\text{high}(v)}, f_{\text{low}(v)}.$$

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

$$f_v = f_{\text{low}(v)} \text{ op}(v) f_{\text{high}(v)}.$$

### 1.1. A simple example

Consider verifying that conjunction distributes over disjunction, i.e., that the following is a tautology:

$$x_1 \wedge (x_2 \vee x_3) \Leftrightarrow (x_1 \wedge x_2) \vee (x_1 \wedge x_3). \quad (1)$$

The BED for this expression is shown in figure 1. 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. A key operation on BEDs is the *up* operation which

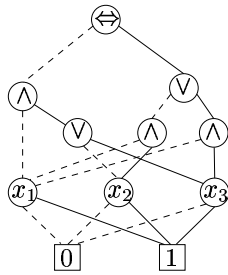


Figure 1. The BED for equation 1.

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

$$(x \rightarrow f_1, f_0) \text{ op} (x \rightarrow f'_1, f'_0) = x \rightarrow (f_1 \text{ op} f'_1), (f_0 \text{ op} f'_0). \quad (2)$$

This identity, illustrated in figure 2, is used to move the variable  $x$  above the operator  $op$  and is the basis for the *up* operation<sup>1</sup>. In cases where one of the children  $u$  does not contain the variable  $x$ , a new variable vertex  $v$ , with  $\text{var}(v) = x$  and  $\text{low}(v) = \text{high}(v) = u$ , is inserted

<sup>1</sup>Equation (2) also holds if the operator vertex  $op$  is a variable vertex. In that case, the *up* operation is identical to the level exchange operation typically used in ROBDDs to dynamically change the variable ordering [20].

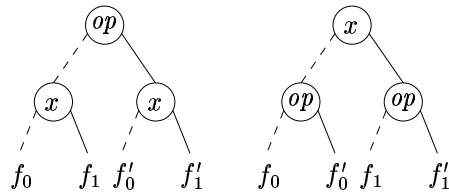


Figure 2. The left and right BED are equivalent.

below the operator vertex before performing the *up*-step. In fact, this is the only way the size of the BED can increase.

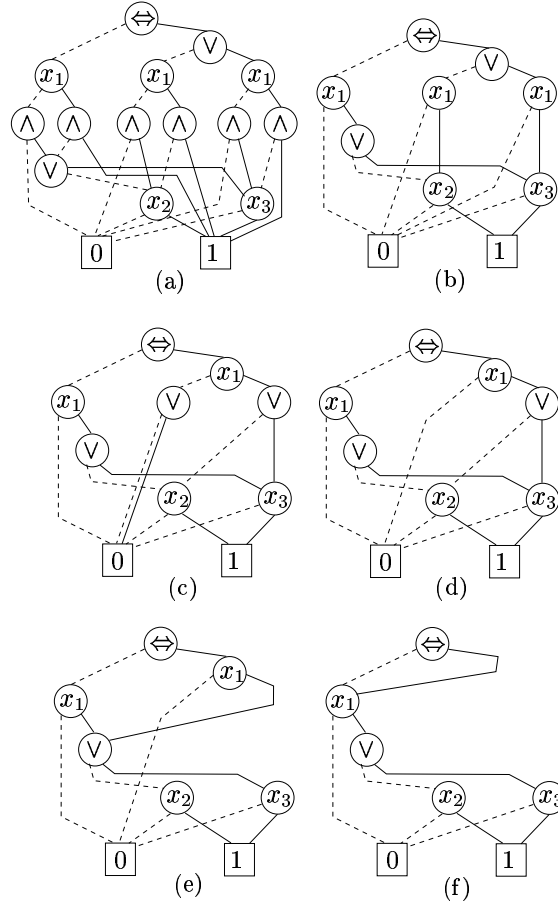


Figure 3. Proving the distributive law. (a)  $x_1$  is moved above the three conjunctions using three *up*-steps. (b) Conjunctions with children that are constant vertices are removed. (c)  $x_1$  is moved above the disjunction to the right. (d) The disjunction with both children equal to  $0$  is removed. (e-f) Identifying equivalent vertices. At this point the two children of the biconditional operator are identical and the BED is reduced to  $1$ , proving the tautology.

The *up* operation moves operators closer to the terminal vertices. If some of the expressions  $f_i$  are terminal vertices, the operators are evaluated and the BED simplified. By repeatedly moving variable vertices above operator vertices, all operator vertices are eliminated and the BED is turned into a BDD.

Consider the example of proving the distributive law (1). Figure 3 shows how the BED from figure 1 is transformed into the tautology **1** by moving  $x_1$  towards the root. This example illustrates that it may not be necessary to move *all* variable vertices to the root in order to obtain a BDD. Notice that variables  $x_2$  and  $x_3$  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 a BDD, moving the variables to the top one at a time. This approach is called *up\_one* and its main advantage is that it can exploit structural information in the expression (as was the case in the example). The efficiency of *up\_one* is demonstrated in section 5 where we verify that the redundant and non-redundant versions of the ISCAS 85 benchmark circuits implement the same functionality.

An alternative way to construct a BDD is to move all variables up simultaneously. This approach is called *up\_all* and it closely mimics the ROBDD *apply* operation. We show that the complexity of building an ROBDD bottom up using *apply* (the standard way) and building it from a BED using *up\_all* is within a constant factor. Thus, one can construct an ROBDD from a BED as efficiently as constructing an ROBDD from scratch.

## 1.2. Related work

Recently, a new way of constructing ROBDDs, called MORE, was proposed [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]. The method can be extended to any Boolean connective since disjunction and negation are functionally complete. BEDs can be seen as extending this idea to allow arbitrary operators and allowing these operators to remain in the graph. Like MORE, Extended BDDs [15] are also based on the idea of using existential quantification to represent disjunction, although the quantification is annotated on the edges of the graph. Extended BDDs are more succinct than ROBDDs, but they are not capable of representing for

example multipliers efficiently.

ROBDDs have been extended in a number of other ways, including using other types of decomposition rules, relaxing the variable ordering restrictions, and extending the domains. The Shannon decomposition used in BDDs can be replaced with either the positive or the negative Davio decomposition, yielding Ordered Functional BDDs [16]. If all three types of decomposition are allowed in one diagram, one obtains Ordered Kronecker Functional Decision Diagrams (OKFDD) [9]. However, none of them are powerful enough to represent all Boolean circuits in polynomial space.

Another modification of the ROBDD representation is to relax the variable ordering restriction. Free BDDs [12] (also called read-once branching programs) only require that on any path from the root, a variable is tested at most once. BEDs are exponentially more succinct than Free BDDs since BEDs are as succinct as branching programs which are exponentially more succinct than read-once branching programs [23]. Graph-driven BDDs [21] are closely related to Free BDDs and have similar properties.

Finally, BDDs have been extended to other domains and/or codomains than Booleans. Examples include \*BMDs [5], MTBDDs [7] and ADDs [1]. These extensions are orthogonal to the BDD extension presented here and we believe similar extensions are possible for BEDs.

## 1.3. Overview

The paper is organized as follows. Section 2 presents some basic complexity results relating BEDs to Boolean circuits and ROBDDs. Section 3 describes the basic representation and construction of BEDs. Section 4 describes algorithms to efficiently manipulate BEDs, including two ways to construct an ROBDD from a BED, *up\_one* and *up\_all*. Section 5 presents an application of BEDs, demonstrating efficient tautology check for the circuits in the ISCAS 85 benchmarks. Finally, section 6 summarizes the contributions of this paper.

## 2. Complexity results

BEDs are closely related to combinational circuits. Any circuit can be transformed to 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  $high(v) = 1$ ) 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  $x = \text{var}(v)$ ,  $l = \text{low}(v)$ ,  $h = \text{high}(v)$ . This translation is also linear.

Using this relationship we can transfer results on circuits to BEDs. For instance, it follows immediately from the results on CIRCUIT-SAT that determining SATISFIABILITY of a BED is NP-complete and determining TAUTOLOGY is co-NP-complete [11]. As another consequence, we observe that BEDs are *exponentially more succinct* than ROBDDs. An example of this is the multiplier function. Bryant [4] showed that for all variable orderings, the multiplier function requires BDDs of exponential size. However, since there are combinatorial circuits implementing this function using only a quadratic number of gates [8] (and even less), there exists a BED of this size representing it.

Despite the exponential succinctness over BDDs, it is still the case that most functions require exponentially sized BEDs. Recall that there are  $2^{2^n}$  Boolean functions over  $n$  variables. It follows from a counting argument that a polynomially sized BED can represent almost none of these functions:

**Theorem 3 (Lower bound on size)** *Let  $\#_n(s)$  be the number of different BEDs over  $n$  variables with at most  $s$  vertices. Then for any polynomial  $p(n)$ ,*

$$\frac{\#_n(p(n))}{2^{2^n}} \longrightarrow 0 \quad \text{for } n \longrightarrow \infty.$$

**Proof:** A straightforward application of Theorem 2.4 in [2, p.763].  $\square$

Fortunately, functions with exponentially sized BEDs do not seem to be of much interest in practice. Even complicated Boolean functions, representing for instance floating-point division, have polynomially sized circuits. This is also witnessed by the fact that it is very difficult to construct explicit examples of functions that provably require exponentially many gates. (The authors have been unable to find any examples in the literature.)

### 3. Representation of BEDs

Inspired by ROBDDs we shall define certain restrictions on BEDs. These restrictions will not generally make BEDs canonical but they will entail some useful properties. First, we define restrictions on the occurrences of variable vertices:

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

Secondly, 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 5** *A BED is reduced if it contains at most two different terminal vertices and for all non-terminal vertices,  $u$  and  $v$ :*

- (1)  $low(u) = low(v), high(u) = high(v),$   
 $\alpha(u) = \alpha(v) \Rightarrow u = v,$
- (2)  $low(u) \neq high(u),$

and for all operator vertices  $v$ :

- (3)  $low(v)$  and  $high(v)$  are non-terminals.

We shall assume that BEDs are always reduced. If the BED is also ordered, we refer to it by “ROBED.” The first condition of definition 5 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-edge 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-edges 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-edge 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 (2) and (3) by taking the redundant second argument to be any non-terminal vertex. We shall assume the presence of a function

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

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  $mk$  as the only means for constructing a BED ensures that it is reduced.

### 3.1. Operator reductions

For operator vertices it is tempting to 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 eliminate all negations below binary operators since for all binary operators  $op$  there exists another operator  $op'$  with  $op'(x, y) = op(-x, y)$ .

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.

### 3.2. Updating vertices

As described in the introduction, the key operation on BEDs, called  $up$ , exchanges variables with other variables or operators, see figure 4. This transformation pulls a variable up one level. The vertices in the

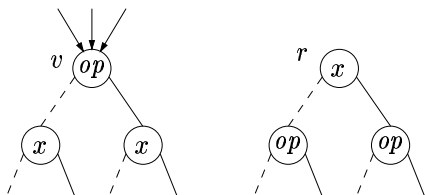


Figure 4. Performing an  $up$ -step on the vertex  $v$ .

new sub-tree (to the right in figure 4) are created by calling  $mk$  with the appropriate parameters. In order for all references to  $v$  to benefit from the  $up$ -step, and thus avoid redoing it for each reference, the referring vertices need to refer to the new vertex  $r$  instead of  $v$ . One way of doing this is to make them point directly to  $r$ . However, in order to do this efficiently we would need to store in each vertex all vertices referring to it, and this is impractical due to the high memory overhead.

Another way to make the references to  $v$  benefit from the  $up$ -step is to simply overwrite  $v$  with the contents of  $r$ . In order to maintain the reducedness property (1), we would need to eliminate the vertex  $r$  returned by  $mk$ . However, this does not work if there are other vertices referring to  $r$ , which is the case if  $r$  was not a newly created vertex but an existing vertex found by  $mk$ .

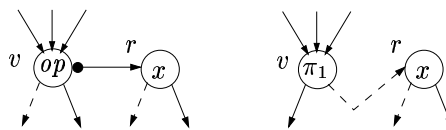


Figure 5. Updating by *linking* (to the left) or by *indirecting* (to the right). In both cases the vertex  $r$  is the “result” of the update.

Instead of copying  $r$  to  $v$ , we need to make  $v$  refer to  $r$ . There are two obvious choices for doing this, as illustrated in figure 5.

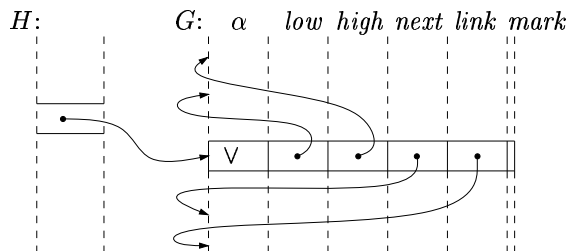
*Linking* adds a reference from vertex  $v$  to the (equivalent) vertex  $r$ . The referenced vertex  $r$  is assumed to be somehow simpler, so the link is directed. Any access to the linked vertex  $v$  should follow the link in order to obtain a simpler representation of the vertex. Section 4.2 shows an application of this where the vertex  $r$  is a fully canonical ROBDD. In such an application, links are never nested.

*Indirecting* simply overwrites the vertex  $v$  with a projection operator with the simpler vertex  $r$  as argument. At first sight it seems plausible that both solutions are equally good. Both share the property that any existing references into  $v$  will benefit from the update: when these references are followed  $r$  will be found. However, linking allows for even more reuse of updates. If later in the BED transformation a new vertex, equivalent to  $v$ , is to be created,  $mk$  will ensure (due to reducedness condition (1)) that the identity  $v$  is returned, and the link will allow new references to benefit from the previous update. This is not the case for indirectings which loses the information of the original contents of the vertex. Linking offers a very direct way of *memorizing* results of earlier transformations which, for some transformations, ensure polynomial rather than exponential running time. Section 4.2 gives an example of such a transformation.

### 3.3. Implementational aspects

Surprisingly, although BEDs are an extension of BDDs, the data structures for representing BEDs, shown in figure 6, are simpler than those for BDDs. The underlying graph of the BED is stored in a table  $G$  which to each vertex  $v$  associates a tag  $\alpha(v)$  (special tags are used for the terminal vertices),  $low(v)$ , and  $high(v)$ .  $G$  contains two additional fields,  $link(v)$  and  $next(v)$ . The field  $link(v)$  is of course used for linking, explained above. The field  $next(v)$  is used to implement chaining for resolving collisions in a hash table  $H$ . This hash table maps triples of  $(\alpha(v), low(v), high(v))$

to  $v$  and thus implements an inverse of  $G$  used by  $mk$ . Finally, the  $mark$  field is used to mark visited vertices when traversing the graph. This is used in the  $up\_one$  and  $up\_all$  algorithms presented in the next section and can also be used in mark-scan garbage collection algorithms. The total memory requirements are six words per vertex when using linking and five words when using indirection. Using these data structures, it is not difficult to implement  $mk$  as a constant time operation.



**Figure 6.** The data structures used to represent a BED.

## 4. Operations on BEDs

The basic operation for constructing ROBDDs, called *apply*, takes two ROBDDs,  $l$  and  $h$ , and a Boolean connective  $op$  and constructs a new ROBDD representing the Boolean expression  $f_l op f_h$ . For BEDs, this operation is simply a constant time call to  $mk(op, l, h)$ . However, other operations, like tautology and satisfiability, are easy for ROBDDs. Thus, an approach for showing these properties for BEDs is to convert them into ROBDDs. It is easily seen that an (RO)BDD is simply an (RO)BED without operators. This suggests a strategy for converting BEDs into ROBDDs: gradually eliminate the operators, keeping all the intermediate BEDs functionally equivalent. We shall show two very different ways of elimination.

### 4.1. Construction of ROBDDs with $up\_one$

The first elimination algorithm,  $up\_one$ , pulls a single variable up to the root using a sequence of  $up$ -steps. Repeating this, we can move all variables up past the operators, which makes the operators disappear (by requirement (3) of reducedness). The algorithm is given in figure 7. Basically,  $up\_one$  performs a depth-first traversal of the BED using traditional marking of the vertices to avoid visiting the same vertex twice. Having finished the recursive calls on the low- and high-edges of a vertex, it makes an  $up$ -step and performs

```

up_one(x, u) =
if u is marked or u is terminal then
  return follow(u)
else if alpha(u) is variable x then
  mark u and return u
else
  (l, h) ← (up_one(x, low(u)), up_one(x, high(u)))
  if alpha(l) and alpha(h) are both variable x then
    r ← mk(x, mk(alpha(u), low(l), low(h)),
            mk(alpha(u), high(l), high(h)))
  else if alpha(l) is variable x then
    r ← mk(x, mk(alpha(u), l, low(h)),
            mk(alpha(u), high(l), h))
  else if alpha(h) is variable x then
    r ← mk(x, mk(alpha(u), l, low(h)),
            mk(alpha(u), l, high(h)))
  else
    r ← mk(alpha(u), l, h)
  update u to r and mark u
  return r

```

**Figure 7.** The  $up\_one$ -operation.  $Up\_one$  takes any free BED  $u$  as argument and returns an equivalent BED with  $x$  occurring at most at the root. The operation  $follow(u)$  follows any links or indirections of  $u$  and returns the result. If none is associated with  $u$ ,  $u$  is returned.  $Up\_one$  is easily extended to unrestricted BEDs by changing the terminal case where  $u$  is the variable vertex  $x$  to perform recursive calls on  $low(u)$  and  $high(u)$  and afterwards eliminate redundant  $x$ 's.

an updating of the root.  $Up\_one$  works well with both indirections and links. It has linear running time:

**Theorem 6 (Up\_one)** *If  $u$  is a vertex in an ordered (free) BED  $G$  then the sub-BED  $v = up\_one(x, u)$  is also ordered (free) and  $x$  appears at most in vertex  $v$  and nowhere else in the sub-BED rooted by  $v$ . The running time of  $up\_one(x, u)$  is  $O(n)$  where  $n$  is the number of vertices in the sub-BED rooted by  $u$ . The number of vertices in the sub-BED rooted at  $v$  is at most  $2n$ .*

**Proof:** (Sketch) Observe that due to the marking,  $up\_one$  is called at most once per vertex, and each call allocates at most one more vertex than it visits.  $\square$

The introductory example was in fact a use of  $up\_one$ .  $Up\_one$  has some distinct properties. As the example shows, in fortunate cases a BED is converted into an ROBDD after moving just a few variables up (in the example, one variable was sufficient). In this process, identical sub-BEDs, potentially containing op-

```

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

```

**Figure 8.** The *apply*-operation. Assumes *l* and *h* are ROBDDs. 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 memorization table *M* must be initialized to empty prior to the first call.

erator vertices, are identified. This is quite unlike traditional ROBDD construction where an order of the variables must be selected and all operators are converted in depth-first order into ROBDDs. In particular, an ROBDD is constructed for *each sub-expression*. If the result is small and the intermediate ROBDDs are large, *up\_one* could be an attractive alternative. As our experiments show (see section 5) this is not purely speculation.

## 4.2. Construction of ROBDDs with *up\_all*

The second elimination algorithm, *up\_all*, is a generalization of Bryant's *apply*-operator, shown in figure 8. Construction of ROBDDs from a Boolean expression using recursive calls of *apply* suggests a bottom up conversion of ROBDDs into ROBDDs. The *up\_all* algorithm does that by moving all variables up as a block past the operator vertices.

*Up\_all* is shown in figure 9. As when building an ROBDD using *apply*, *up\_all* requires that a total ordering of the variables is selected prior to the ROBDD construction. Based on the ordering *up\_all* converts any BED into an ROBDD, i.e., *up\_all*(*u*) is an ROBDD. Furthermore, *up\_all*(*u*) is not linked, i.e., *follow*(*up\_all*(*u*)) = *up\_all*(*u*). If *u* is known to be “a DAG of operator vertices on top of ROBDDs,” the lines marked with (\*) are superfluous and the statement just after them should simply return *u*. The

```

up_all(u) =
if u is marked or u is terminal then
  return follow(u)
else u is non-terminal:
  if  $\alpha$ (u) is variable x then
    (*) r ← mk(x, up_all(low(u)), up_all(high(u)))
    (*) update u to r and mark u
    return r
  else  $\alpha$ (u) is operator op :
    (+) (l, h) ← (up_all(low(u)), up_all(high(u)))
    if l and h are terminal vertices then
      r ← op(value(l), value(h))
    else if var(l) = var(h) then
      r ← mk(var(l), up_all(mk(op, low(l), low(h))),
            up_all(mk(op, high(l), high(h))))
    else if var(l) < var(h) then
      r ← mk(var(l), up_all(mk(op, low(l), h)),
            up_all(mk(op, high(l), h)))
    else var(l) > var(h) :
      r ← mk(var(h), up_all(mk(op, l, low(h))),
            up_all(mk(op, l, high(h))))
    update u to r and mark u
    return r

```

**Figure 9.** The *up\_all*-operation. The total order  $<$  is defined as for *apply* (see figure 8).

line (+) extends *up\_all* to work on not only vertices *u* where *low*(*u*) and *high*(*u*) are ROBDDs but on all BEDs. If *low*(*u*) and *high*(*u*) are in fact ROBDDs then *l* = *low*(*u*), *h* = *high*(*u*). The relationship between *apply* and *up\_all* is:

$$\mathit{apply}(op, l, h) = \mathit{up\_all}(\mathit{mk}(op, l, h))$$

for any ROBDDs *l* and *h*. This makes it clear how the four cases of *up\_all* correspond to the four cases of *apply*.

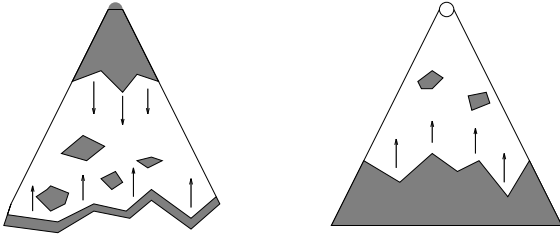
Contrary to *up\_one*, *up\_all* requires the use of linking in updates in order to efficiently reuse vertices. The linking corresponds to the memorization in *apply* (the table *M*). Although *up\_all* is more general than *apply*, the running time of *up\_all* is within a constant factor of *apply*, and due to the increased sharing of sub-expressions and memorization of results, it could even create fewer vertices.

**Theorem 7 (Up\_all)** *For any vertex u of an ROBDD, up\_all(u) is the root of an ROBDD equivalent to the ROBDD rooted at u. If l and h are ROBDDs then*

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

The number of calls to  $mk$  on variables generated by  $up\_all(mk(op,l,h))$  is exactly the same as for  $apply(op,l,h)$ , and the number of calls to  $mk$  on operators is the same as the number of calls to  $apply(op,l,h)$ .

$Up\_one$  and  $up\_all$  are significantly different strategies for building an ROBDD. Figure 10 illustrates how the operator vertices are converted into variable vertices by the two algorithms.



**Figure 10.** Converting a BED to an ROBDD. To the left, using  $up\_one$  repeatedly, to the right, during an  $up\_all$  call. Grey areas represent variables and white areas represent operators.

### 4.3. Further BED operations

Two commonly used Boolean operations are substitution and existential quantification. Substitution replaces all occurrences of a variable  $x$  with a Boolean formula  $\varphi$ . The simplest way to perform substitution on a BED rooted at  $v$  is the following. First perform a call to  $up\_one$ :

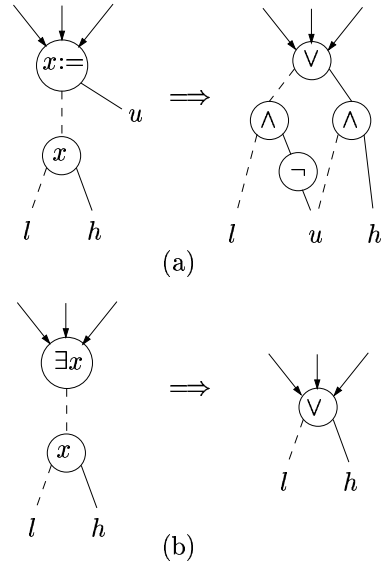
$$w \leftarrow up\_one(x,v).$$

From theorem 6 we know that  $low(w)$  and  $high(w)$  do not contain any occurrences of variable  $x$ . If  $w$  is not the variable  $x$ , then  $x$  is not in the BED rooted at  $w$  and the result is  $w$ . Otherwise,  $var(w) = x$  and the result is the BED for

$$(u \wedge high(w)) \vee (\neg u \wedge low(w))$$

where  $u$  is the root of the BED representing  $\varphi$ . This expression follows immediately from the definition of a variable vertex.

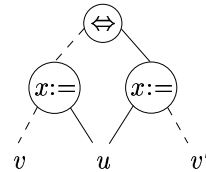
Existential quantification of the variable  $x$  can also be implemented using  $up\_one$ . Again we call  $up\_one$  obtaining  $w$  and if  $w$  does not contain  $x$  the result is  $w$ . Otherwise, the result is  $low(w) \vee high(w)$  since  $\exists x.x \rightarrow f, g = f \vee g$ . For both operations, the complexity is determined by  $up\_one$  which is linear in the size of the BED.



**Figure 11.** (a) Elimination of substitution operator vertex. (b) Elimination of existential quantification operator vertex.

An alternative way to implement substitution and existential quantification is to consider them special operator vertices in the BED, see figure 11. The  $up$ -step from figure 2 is exactly the same for these new operator vertices, except in the case where the variable below the operator is the variable  $x$ . In those cases, the special operator vertex is replaced with the sub-BED shown in figure 11. These eliminations can easily be performed by adding reduction rules to  $mk$ . The substitution and existential quantification operators 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 corresponding variable or when it reaches terminal vertices.

One need not immediately eliminate these newly added operator vertices. Keeping them in the BED allows efficient reuse of sub-expressions. Consider the BED in figure 12. If the vertices  $v$  and  $v'$  are identified



**Figure 12.** A BED containing substitution operators. The low-edge points to the expression in which  $x$  is to be substituted with  $\varphi$ , pointed to by the high-edge.

at some point in the manipulations, the biimplication is proven immediately, *without* actually performing the substitution.

Other standard BDD operations include restriction  $x := b$ , where  $b$  is a Boolean constant. Clearly, restriction is a special case of substitution with  $\varphi$  equal to either **0** or **1**. Notice that all the operations described in this section have linear running times which is better than the corresponding ROBDD operations. Other operations, like satisfy-one, can be performed by constructing an ROBDD using *up\_all* and perform the operation on the ROBDD. Since *up\_all* never has worse running time than *apply*, the total running time for these operations is no worse than if they are performed directly on an ROBDD.

## 5. An application of BEDs

Tautology checking of a BED is an application where the end-result as an ROBDD is known to be small (the terminal vertex **1**) if the BED indeed is a tautology. Thus, to demonstrate the efficiency of BEDs, we consider the combinational logic-level verification problem which is to determine whether two given combinational circuits implement the same Boolean function. The ISCAS 85 benchmark suite<sup>2</sup> contains a number of combinational circuits, nine of which exist both in a redundant and a non-redundant version. We consider the problem of determining whether these two versions implement the same functionality, corresponding to a tautology check for each pair of outputs. The results<sup>3</sup> are shown in table 1. It is observed that except for the c3540, all circuits are verified in only a few CPU second and using at most half a million BED nodes, which is less than 12 MB of main memory. These results are up to several orders of magnitude faster than those reported in [3, 17, 18]. The circuit c6288 is particularly interesting because it implements a 16-bits multiplier and multipliers are notoriously difficult to verify using ROBDDs [4]. Due to the exponential growth of the size of the ROBDD representation (in the number of operand bits), the straightforward approach of building and comparing the ROBDD for the two circuits c6288 and c6288nr is not feasible. 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 demonstrates the effectiveness of *up\_one*.

<sup>2</sup>Can be obtained from <http://www.cbl.ncsu.edu>

<sup>3</sup>The C source code for a small BED package and the benchmark circuits with scripts to verify them can be obtained from <http://www.it.dtu.dk/~henrik/bed/>.

**Table 1.** Experimental results for the ISCAS 85 benchmark.  $N_{total}$  is the total number of BED nodes for the verification. The CPU times are in seconds on a Sun Ultra-SPARC 1.

Circuits	In	Out	Gates	$N_{total}$	CPU
c432, c432nr	36	7	316	28259	0.5
c499, c499nr	41	32	403	234103	2.7
c499, c1355	41	32	748	467823	6.3
c1355, c1355nr	41	32	1091	492741	5.6
c1908, c1908nr	33	25	1757	429422	4.8
c2670, c2670nr	233	140	2153	26011	0.7
c3540, c3540nr	50	22	3288	427323 <sup>†</sup>	127.0
c5315, c5315nr	178	123	4604	45138	1.5
c6288, c6288nr	32	32	4814	8759	0.7
c7552, c7552nr	207	108	6908	329717	6.3

<sup>†</sup>Garbage collection is used in this example, making this number the maximum number rather than the total number of BED vertices.

Several other methods exploit the structural similarities between the two circuits [3, 18, 22] thereby achieving an efficiency comparable to the BED-based technique. However, these techniques are specifically tailored to solve the combinational logic-level verification problem for two circuits which have similar structure, making them unsuitable for general Boolean function manipulations.

## 6. Conclusion

We have presented a new data structure called Boolean Expression Diagrams for representing and manipulating Boolean expressions. BEDs are as succinct as Boolean circuits, yet they have many of the desirable properties of BDDs. Properties like TAUTOLOGY and SATISFIABILITY are determined by transforming the BED representation into a reduced ordered BDD. This can be done efficiently by using for example one of the two algorithms *up\_one* or *up\_all*. As shown in theorem 7, the cost of constructing an ROBDD from scratch using *apply* and the cost of building a BED and transform it into an ROBDD using *up\_all* are within a constant factor. In fact, recent research [14] has shown that *up\_all* is a highly efficient approach to build an ROBDD; it uses considerably less memory and no more time than when the ROBDD is constructed with *apply*.

*Up\_one* is a new way to construct an ROBDD which can exploit structural similarities between sub-expressions. For some applications *up\_one* is highly efficient, for example as demonstrated by proving the identity of two 16-bits multipliers (c6288 and c6288nr

from the ISCAS 85 benchmarks) in less than a second. *Up\_one* is also the basis for other operations like existential quantification and substitution, making the running times of these operation linear in the size of the BED.

BEDs are particularly useful when the end-result is expected to have a small ROBDD representation, e.g., for tautology checks. Another area that may benefit from using the BED representation is in symbolic model checking. Several researchers have observed that when performing fixed-points 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.

BEDs can be seen as an intermediate form between circuits and the highly structured ROBDDs. In this paper we have focussed on ways to obtain an ROBDD from the BED description. However, there seems to be a huge unexploited potential for manipulating the BEDs directly, without necessarily converting them to ROBDDs. For example, it seems plausible that ideas from logic can be transferred to BEDs; the reduction of operator vertices described in section 3.1 is a first step in this direction.

## Acknowledgments

The authors would like to thank Bernd Becker and Rolf Drechsler from the Institute of Computer Science, Albert-Ludwigs-University, for introducing us to the MORE approach for building ROBDDs.

## References

- [1] 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.
- [2] R. B. Boppana and M. Sipser. The complexity of finite functions. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 758–804. Elsevier Science Publisher, 1990.
- [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, Aug. 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. 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. 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. 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, Oct. 1994.
- [13] A. Hett, R. Drechsler, and B. Becker. MORE: Alternative implementation of BDD-packages by multioperand 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] S.-W. Jeong, B. Plessier, G. Hachtel, and F. Somenzi. Extended BDD's: Trading off canonicity for structure in verification algorithms. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 464–467, 1991.
- [16] U. Keeschull, E. Schubert, and W. Rosenstiel. Multi-level logic synthesis based on functional decision diagrams. In *Proc. European Conference on Design Automation (EDAC)*, pages 43–47, 1992.

- [17] 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.
- [18] 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, Sept. 1994.
- [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] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 42–47, 1993.
- [21] D. Sieling and I. Wegener. Graph driven BDDs – a new data structure for Boolean functions. *Theoretical Computer Science*, 141(1-2):283–310, 1995.
- [22] C. 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.
- [23] I. Wegener. On the complexity of branching programs and decision trees for clique functions. *Journal of the ACM*, 35(2):461–471, Apr. 1988.