similar papers at core.ac.uk
brought
provided by Elsevier

Boolean Expression Diagrams[1]

Henrik Reif Andersen and Henrik Hulgaard

Department of Innovation, IT University of Copenhagen, Glentevej 67, DK-2400 Copenhagen NV, Denmark
E-mail: hra@it.edu, henrik@it.edu

Received January 31, 1998; published online April 22, 2002

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. Two algorithms are described for transforming a BED into a reduced ordered BDD. One is a generalized version of the BDD *apply*-operator while the other can exploit the structural information of the Boolean expression. This ability is demonstrated by verifying that two different circuit implementations of a 16-bit multiplier implement the same Boolean function. Using BEDs, this verification problem is solved efficiently, while using standard BDD techniques this problem is infeasible. Generally, BEDs are useful in applications, for example tautology checking, where the end-result as a reduced ordered BDD is small. Moreover, using operators for substitution and existential quantification they allow for the verification of large hierarchical circuits. © 2002 Elsevier Science (USA)

1. INTRODUCTION

Within the past decade reduced ordered binary decision diagrams (OBDDs[2]) introduced by Bryant [4] have become a successful data structure for representing and manipulating Boolean functions. This success is due to the fact that OBDDs 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 OBDDs depends heavily on the size of the representation and unfortunately some (important) functions, e.g., the multiplication function, have no subexponential representation.

This paper presents an extension of OBDDs, called Boolean expression diagrams (BEDs). BEDs can represent any Boolean circuit (see, e.g., [3]) in linear space at the price of being noncanonical. However, since converting a circuit into a BDD via a BED can always be done at least as efficiently as constructing the BDD directly, many of the desirable properties of OBDDs are maintained. This is obtained by extending the OBDD representation with *operator vertices*:

Definition 1 (Boolean expression diagram). A Boolean expression diagram 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 $\mathit{value}(v) \in \{0, 1\}$.
- A variable vertex v has as attributes a variable $\mathit{var}(v)$ and two sons $\mathit{low}(v), \mathit{high}(v) \in V$.
- An operator vertex v has as attributes a binary Boolean operator $\mathit{op}(v)$ and two sons $\mathit{low}(v), \mathit{high}(v) \in V$.

The edge set E is defined by

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

We use **0** and **1** to denote the two terminal vertices.

[1] This work is supported by the Danish Technical Research Council. It was carried out while the authors were at Department of Information Technology, Technical University of Denmark DK-2800 Lyngby, Denmark.

[2] Throughout this paper, we will assume that all decision and expression diagrams are reduced and will omit the 'R'-prefix.

0890-5401/02 $35.00
© 2002 Elsevier Science (USA)
All rights reserved.

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

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

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

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

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

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

Before presenting the formal details of BEDs and the algorithms for manipulating them, we illustrate the use of the data structure to prove a tautology.

1.1. A Simple Example

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

$$x_1 \wedge (x_2 \vee x_3) \leftrightarrow (x_1 \wedge x_2) \vee (x_1 \wedge x_3). \tag{1}$$

The BED for this expression is shown in Fig. 1. The low-edges are drawn using dashed lines. Notice that vertices representing the same Boolean subexpressions are shared. A key operation on BEDs is the *up-step*. To explain the up-step, 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

$$(x \to f_1, f_0)\ op\ (x \to f_1', f_0') = x \to (f_1\ op\ f_1'), (f_0\ op\ f_0'). \tag{2}$$

This identity, illustrated in Fig. 2(a), is used to move the variable x above the operator *op* and is the basis for the up-step. (Equation (2) 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 [24].) 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. 2(b). In fact, this is the only way the size of the BED can increase during a transformation.

The up-step moves operators closer to the terminal vertices and 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 an OBDD.

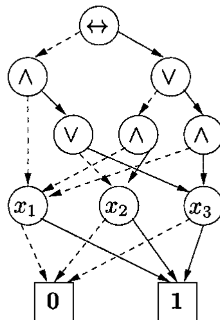

FIG. 1. The BED for Eq. (1).

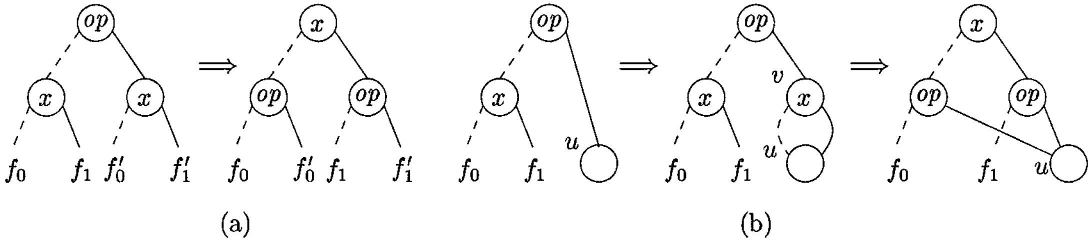

FIG. 2. Illustration of the up-step (a) for the case where variable x exists in both sons of the root and (b) for the case where x only occurs in the left son.

Consider again the example of proving the distributive law (1). Figure 3 shows how the BED from Fig. 1 is transformed into the tautology **1** by moving x_1 toward the root using repeated up-steps on x_1. As the example illustrates it may not be necessary to move *all* variable vertices to the root in order to obtain an OBDD. In fact, the variables x_2 and x_3 could have been replaced with arbitrary large BEDs, and the tautology would still have been proven with exactly the same steps.

By repeatedly pulling up variables one by one, in a way similar to x_1 in the example, a BED can gradually be converted into an OBDD. 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 two different circuit implementations of a 16-bit multiplier are identical.

An alternative way to construct an OBDD is to move all variables up simultaneously. This approach is called *up_all* and it is a generalization of the OBDD *apply*-operation. We show that the worst-case complexity of building an OBDD 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 OBDD from a BED at least as efficiently as constructing an OBDD from scratch. In fact, it seems that the reduction rules should in many cases be able to improve on the OBDD-construction by reducing the number of operators that need to be converted.

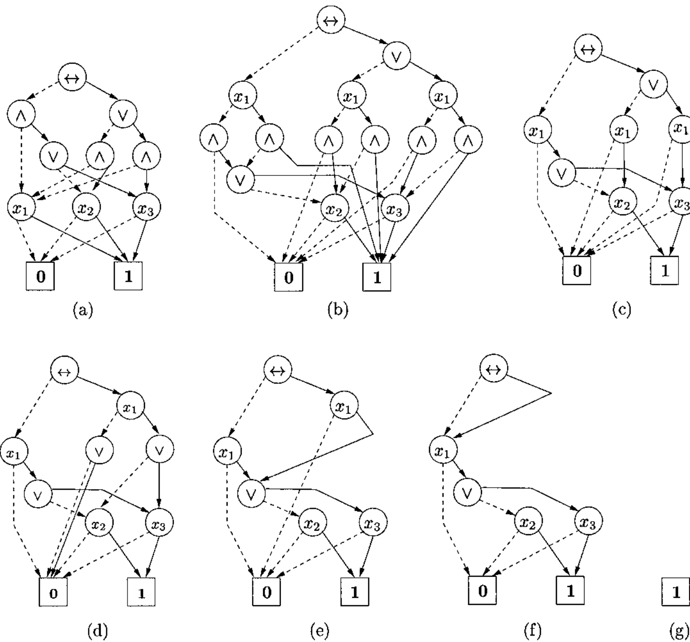

FIG. 3. Proving the distributive law. (a) BED for the distributive law. (b) x_1 is moved above the three conjunctions using three up-steps. Notice that at this point variable and operator vertices are no longer separated in two distinct layers. (c) Conjunctions with children that are constant vertices are eliminated. (d) x_1 is moved above the disjunction to the right. (e) The disjunction with both children equal to **0** is removed and the two remaining disjunctions are identified. (f) Identifying equivalent variable vertices. At this point the two children of the biimplication operator are identical and (g) the BED is reduced to **1**, proving the tautology.

1.2. Related Work

Recently, a new way of constructing OBDDs, called MORE, was proposed [13, 14]. 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 toward 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.

Prior to the work on MORE, Plessier *et al.* [17, 23] proposed a variant of BDDs 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 performing BDD 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 BDD 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 allow 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.

OBDDs 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 OBDDs can be replaced with either the positive or the negative Davio decomposition, yielding ordered functional BDDs [18]. If all three types of decomposition are allowed in one diagram, one obtains ordered Kronecker functional decision diagrams (OKFDD) [9]. They allow compact and canonical representations of different classes of Boolean functions, but none of them are powerful enough to represent all Boolean circuits in polynomial space.

Another modification of the OBDD representation is to relax the variable ordering restriction. Free BDDs (FBDDs) [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 FBDDs since BEDs are as succinct as branching programs which are exponentially more succinct than read-once branching programs [26]. Through the use of types expressed as BDD-like graphs, which impose restrictions on the occurrences of variables along paths, FBDDs can be made canonical. However, there are severe problems with the BDD-like types. The types are more general and finding good types is even more difficult than finding good variable orderings. Furthermore the Boolean operations of restriction and quantification only work for very restricted types that are similar to BDD-orderings. Graph-driven BDDs [25] are closely related to FBDDs and have similar properties.

Indexed BDDs [16] extend OBDDs with *layers* of variables. The variables in each layer are ordered but different layers may have different orderings. Thus indexed BDDs are not canonical and are as expressive as branching programs. This makes it possible to represent, e.g., the multiplication function efficiently (using sufficiently many layers). Indexed BDDs are a subclass of BEDs since BEDs make no assumptions about the ordering or freeness of variables.

Finally, OBDDs have been extended to other domains and/or codomains than Booleans. Examples include $*$BMDs [5], MTBDDs [7], and ADDs [2]. These extensions are orthogonal to the OBDD 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 OBDDs. Section 3 describes the basic representation and construction of BEDs. Section 4 describes algorithms to efficiently manipulate BEDs, including two ways to construct an OBDD from a BED, *up_one* and *up_all*. Section 5 presents an application of BEDs, demonstrating an efficient equivalence check for two multiplier circuits. Finally, Section 6 summarizes the contributions of this paper.

2. COMPLEXITY RESULTS

BEDs are closely related to Boolean circuits [3]. 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 $high(v) = \mathbf{1}$) and replacing 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 with the corresponding gate, and a variable vertex v is replaced with the subcircuit $(\neg x \wedge l) \vee (x \wedge h)$, where $x = var(v)$, $l = low(v)$, and $h = high(v)$. 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 OBDDs. 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 combinational circuits implementing this function using only a quadratic number of gates [8] (and even less), there exists a BED of this size representing it.

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 $p(n)$,*

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

Proof. A straightforward application of Theorem 2.4 in [3, p.763] using the linear transformation to circuits. ■

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.)

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. A vertex u is an OBDD vertex if all paths from u only contain variable vertices and they respect a given total order. From these definitions we get the following inclusions among subclasses of BEDs:

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

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 in this class (in this case, the OBDDs are 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. Although the different classes of BEDs are equally expressive, the algorithms on BEDs (to be presented in the following) can be optimized if the BEDs are known to belong to some of the more specific classes. In Section 4, we describe two algorithms for transforming a (general) BED into an OBDD and some optimizations of these algorithms for the classes OBEDs and FBEDs. The generality

of these algorithms makes them useful, e.g., for making a free BDD ordered (i.e., transforming it to an OBDD) or for reordering an OBDD.

3. REPRESENTATION OF BEDs

BED vertices are constructed using a single operation called *mk*. 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.)

3.1. Reductions of 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 nonterminal vertices.

DEFINITION 5. A BED is reduced if it contains at most two different terminal vertices and for all nonterminal vertices, u and v:

(i) $low(u) = low(v)$, $high(u) = high(v)$, and $\alpha(u) = \alpha(v) \;\Rightarrow\; u = v$,

(ii) $low(u) \neq high(u)$,

and for all operator vertices v:

(iii) $low(v)$ and $high(v)$ are nonterminals.

We shall assume that BEDs are always reduced. 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-sons 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-sons 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 16 Boolean connectives (shown in Table 1) reduce to one of the following six: $K0$, $K1$ (constant 0/1), π_1, π_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

TABLE 1

The 16 Binary Boolean Operators and Their Associated Truth-Table

	op	$op(x, y)$ x: 1 1 0 0 y: 1 0 1 0	Name of Boolean function
0	$K0$	0 0 0 0	Constant false
1	$\bar{\vee}$	0 0 0 1	Negated disjunction
2	$\not\leftarrow$	0 0 1 0	Negated left-implication
3	$\bar{\pi}_1$	0 0 1 1	Negation of first argument
4	$\not\rightarrow$	0 1 0 0	Negated implication
5	$\bar{\pi}_2$	0 1 0 1	Negation of second argument
6	$\not\leftrightarrow$	0 1 1 0	Exclusive or
7	$\bar{\wedge}$	0 1 1 1	Negated conjunction
8	$\wedge$	1 0 0 0	Conjunction
9	$\leftrightarrow$	1 0 0 1	Biimplication
10	π_2	1 0 1 0	Projection on second argument
11	$\rightarrow$	1 0 1 1	Implication
12	π_1	1 1 0 0	Projection on first argument
13	$\leftarrow$	1 1 0 1	Left-implication
14	$\vee$	1 1 1 0	Disjunction
15	$K1$	1 1 1 1	Constant true

used. The projections are avoided by using the proper low- or high-son instead. The negations require the 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 nonterminal vertex different from the first. 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. As shown by Bryant, reducedness ensures canonicity of OBDD vertices:

LEMMA 6 (Canonicity of OBDDs [4]). *If u and v are* OBDD *vertices and $f^u = f^v$ then $u = v$.*

3.2. 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 *mk* to look for operator vertices that differ from the one wanted only by exchanging low and high, by a negation, or by a combination of both.

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

Finally, taking the identity of vertices into account allows us to exploit equivalences such as 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 reduction table would include equivalences such as the distributive laws.

Another reason for choosing $n = 3$ is that it allows us to determine *operator 2-cuts*: Consider a BED with the structure shown in Fig. 4(a); that is, for some vertex u, all paths from u to the terminals go through either vertex w_1 or w_2. We call the set $\{w_1, w_2\}$ a 2-cut. 2-cuts can be used to reduce the size of the BED as shown in Fig. 4(b).

More formally, consider a BED represented by the graph (V, E). A path of length k from a vertex $u \in V$ to a vertex $v \in V$ is a sequence $\langle v_0, v_1, v_2, \ldots, v_k\rangle$ such that $u = v_0$, $v = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. A path p from u to v is denoted by $u \overset{p}{\rightsquigarrow} v$.

DEFINITION 7 (2-cut). A set $\{w_1, w_2\} \subseteq V$ is a 2-cut for vertex $u \in V \backslash \{w_1, w_2\}$ if any path p from u to a terminal vertex $v \in \{\mathbf{0}, \mathbf{1}\}$ can be decomposed into two parts $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ such that $w \in \{w_1, w_2\}$. A 2-cut is called an *operator* 2-cut if for all paths p, p_1 only contains operator vertices. The cut $\{low(u), high(u)\}$ is called a *trivial* 2-cut for vertex u.

If the BED rooted at u has an operator 2-cut $\{w_1, w_2\}$, then there exists a binary Boolean operator *op* such that

$$f^u = f^{w_1} \; op \; f^{w_2}.$$

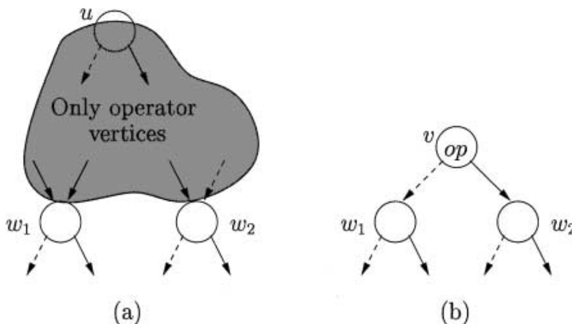

FIG. 4. A BED with an operator 2-cut $\{w_1, w_2\}$.

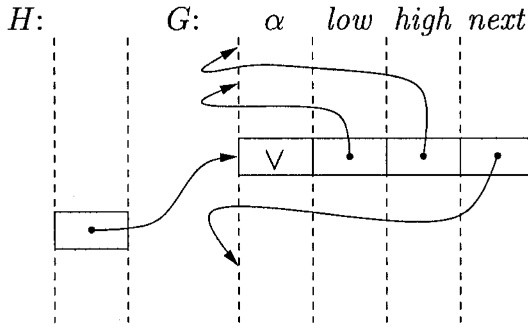

FIG. 5. The data structures used to represent BEDs.

That is, all vertices from u to the 2-cut can be replaced with a single operator vertex v with $\alpha(v) = op$ and $low(v) = w_1$ and $high(v) = w_2$. The BED can be constructed such that it contains no nontrivial operator 2-cuts. The following lemma shows that a new operator vertex only will have nontrivial 2-cuts among its children and grandchildren.

LEMMA 8. *Let $u \in V$ be an operator vertex and let $l = low(u)$ and $h = high(u)$ be the low- and high-sons of u. If l and h only have trivial operator 2-cuts, then if u has any nontrivial operator 2-cut $\{w_1, w_2\}$, it is a subset of $\{l, h, low(l), high(l), low(h), high(h)\}$.*

Proof. By contradiction; assume u has a nontrivial operator 2-cut $\{w_1, w_2\}$ which is not a subset of $\{l, h, low(l), high(l), low(h), high(h)\}$.

Observe that either l or h is not in $\{w_1, w_2\}$ (otherwise the 2-cut would be trivial). Assume without loss of generality that $l \notin \{w_1, w_2\}$. Since $\{w_1, w_2\}$ is a 2-cut for u, every path from u to a terminal vertex contains either w_1 or w_2 (or both). Any path from l to a terminal vertex also contains either w_1 or w_2 since the path is a postfix of some path from u. Thus, $\{w_1, w_2\}$ is an operator 2-cut for l which cannot be trivial since, by assumption, $\{w_1, w_2\}$ is not a subset of $\{l, low(l), high(l)\}$. This is a contradiction. ■

From this lemma it follows that nontrivial cuts only exist among the children and grandchildren of u. A reduction table constructed with $n = 3$ makes it easy to find and eliminate all nontrivial 2-cuts when constructing the BED.

3.3. Implementation Aspects

The data structures for representing BEDs, shown in Fig. 5, are very similar to 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 $high(v)$. Furthermore, G contains the field $next(v)$ which 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*. The total memory requirements are five words per vertex. Using these data structures, it is not difficult to implement *mk* as an expected constant time operation on a uniform RAM model [1].

4. OPERATIONS ON BEDs

The basic operation for constructing OBDDs, called *apply*, takes two OBDDs, l and h, and a Boolean connective *op* and constructs a new OBDD representing the Boolean expression $f^l\ op\ f^h$. For BEDs, constructing the representation for the Boolean expression $f^l\ op\ f^h$ is simply a constant time call to $mk(op, l, h)$. However, other operations, like checking for tautology or satisfiability, are difficult for BEDs but are constant time operations for OBDDs. Thus, an approach for implementing these operations on BEDs is to convert the BEDs into OBDDs. Since an (O)BDD is simply an (O)BED without operators, a strategy for converting BEDs into OBDDs is to gradually eliminate the operators, keeping all the intermediate BEDs functionally equivalent. We shall show two very different ways of elimination.

```
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
 4:      (l, h) ← (up_one(x, low(u)), up_one(x, high(u)))
         /* x can only occur in the roots of l and h */
 5:      if α(l) and α(h) are both variable x then
 6:          r ← mk′(x, mk(α(u), low(l), low(h)),
                        mk(α(u), high(l), high(h)))
 7:      else if α(l) is variable x then
 8:          r ← mk′(x, mk(α(u), low(l), h),
                        mk(α(u), high(l), h))
 9:      else if α(h) is variable x then
10:          r ← mk′(x, mk(α(u), l, low(h)),
                        mk(α(u), l, high(h)))
11:      else
12:          r ← mk(α(u), l, h)
13:      insert ((x, u), r) in M
14:      return r
```

FIG. 6. The *up_one*-operation. *Up_one* takes any BED u (which could be ordered, free, or completely arbitrary) as argument and returns an equivalent BED with x occurring at most at the root. The memorization table M must be initialized to empty prior to the first call.

4.1. Construction of OBDDs with up_one

The first elimination algorithm is based on the algorithm *up_one* shown in Fig. 6. *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-sons of a vertex, it makes an up-step. Repeated calls to *up_one* for each variable move all variables up past the operators, which makes the operators disappear (by requirement (iii) of reducedness). The function $mk'(x, l, h)$ is a version of mk, which further checks for duplicate variables. If any of l and h has x at the root, it is removed:

$$mk'(x, l, h) = \begin{cases} mk(x, low(l), high(h)) & \text{if } \alpha(l) = \alpha(h) = x \\ mk(x, low(l), h) & \text{if } \alpha(l) = x, \alpha(h) \neq x \\ mk(x, l, high(h)) & \text{if } \alpha(l) \neq x, \alpha(h) = x \\ mk(x, l, h) & \text{if } \alpha(l) \neq x, \alpha(h) \neq x \end{cases}$$

The table M is used to memorize previously computed results and ensures a linear expected runtime.

The introductory example was in fact a use of *up_one*. As the example shows, 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 subexpression. If the result is small and the intermediate OBDDs are large, *up_one* is an attractive alternative.

The following theorem characterizes key properties of *up_one*. Let $|u|$ denote the number of vertices in the BED reachable from u:

$$|u| = |\{v : u \rightsquigarrow v\}|\,.$$

THEOREM 9 (Up_one). *Assume u is a vertex in a* BED *and let $v = up_one(x, u)$. The following properties hold*:

(i) $f^v = f^u$.

(ii) *x does not occur below v.*

(iii) $|v| \leq 2|u| - 1$.

(iv) *If u is ordered (free) then v is also ordered (free).*

(v) *Up_one can be implemented with expected running time $O(n)$ with $n = |u|$ on a uniform* RAM *model using hashing.*

Proof. We prove (i) by induction on the number of vertices below u. In the proof we ignore the memorization table M, which speeds up the algorithm but does not influence its correctness provided only correct values are inserted into it. The base case in line 2, when u is a terminal, clearly satisfies $f^v = f^u$ since $v = u$. For the induction step assume that

$$f^l = f^{low(u)} \quad \text{and} \quad f^h = f^{high(u)}.$$

There are now four cases to consider, corresponding to the four branches. We consider only the first case since the other three are similar (or simpler). We shall use the following properties of *mk* and *mk*′:

$$\begin{aligned} f^{mk(op,l,h)} &= f^l \; op \; f^h \\ f^{mk(x,l,h)} &= f^{mk'(x,l,h)} = x \to f^h, f^l. \end{aligned}$$

From the assignment of r in line 6:

$$\begin{aligned} f^r &= x \to f^{mk(\alpha(u),high(l),high(h))}, f^{mk(\alpha(u),low(l),low(h))} \\ &= x \to \left(f^{high(l)} \; \alpha(u) \; f^{high(h)}\right), \left(f^{low(l)} \; \alpha(u) \; f^{low(h)}\right) \\ &= \left(x \to f^{high(l)}, f^{low(l)}\right) \alpha(u) \left(x \to f^{high(h)}, f^{low(h)}\right) \\ &= f^l \; \alpha(u) \; f^h \\ &= f^{low(u)} \; \alpha(u) \; f^{high(u)} \\ &= f^u. \end{aligned}$$

The first two steps use the properties of *mk*′ and *mk*. The third step is an up-step. The fourth step uses the semantics of variable vertices. The fifth step uses the induction hypothesis. The final step uses the definition of the semantics of an operator node $\alpha(u)$. The proof is similar for a variable node.

Property (ii) is also proven by induction on the number of vertices below u. The base case is trivial and in the induction step the four cases corresponding to the four branches follow directly by the induction hypothesis (which states that x does not occur below l and h) and the definition of *mk*′.

For property (iii) observe that, due to the table M, the body of *up_one* is executed at most once for each node below u. Let $|u|_x$ be the number of vertices labeled with variable x below u:

$$|u|_x = |\{v : u \rightsquigarrow v \text{ and } v \text{ is a variable vertex with } var(v) = x\}|.$$

Let $n = |u|$ and $n_x = |u|_x$. There are $n - n_x$ vertices with a label different from x. We shall first count only the new vertices with labels different from x that are being generated by the *mk*′ and *mk* calls in *up_one*. For each call to *up_one* on vertices u with $\alpha(u) \neq x$, at most two new vertices with labels different from x are generated (in lines 6, 8, 10, and 12). This gives a total of at most $2(n - n_x)$ new vertices. From property (ii) it follows that in the result v at most one x-vertex can occur; i.e., the total number of vertices reachable from v is

$$|v| \leq 2(n - n_x) + 1 = 2n + 1 - 2n_x.$$

For $n_x \geq 1$, we have that $|v| \leq 2n - 1$. When $n_x = 0$, all calls to *up_one* will fall into the last branch (line 12) and $v = u$; thus clearly $|v| \leq 2n - 1$.

Property (iv) is also proven by an induction on the number of vertices below u. We prove that if u is ordered the BED v is still ordered. The proof for freeness is similar. Assume that $x_1 < \cdots < x_n$ is an ordering for u. If $x = x_i$, the ordering for v is $x_i < x_1 < \cdots < x_{i-1} < x_{i+1} < \cdots < x_n$. The induction hypothesis for a vertex u is that *up_one*(x, u) has the new ordering and that *up_one*(x, u) does

not contain variables that where not already present in u. The base case is trivial. For the induction step we can assume from the induction hypothesis that l and h both have the new ordering. We consider only the first of the four branches. The remaining three are similar or simpler. If $\alpha(u)$ is an operator, then clearly r has the new ordering. If $\alpha(u)$ is a variable different from x, then $\alpha(u)$ must precede the variables in *low*(u) and *high*(u) according to the old ordering. Therefore $\alpha(u)$ also precedes any variables of l and h in the new ordering. If $\alpha(u)$ is the variable x, from the induction hypothesis, x also precedes the variables below l and h. From property (ii), variable x can only occur at the root of l and h and thus *mk*′ ensures that x can only occur at the root of r.

For property (v) observe again that due to the table M the body of *up_one* is executed at most once on each node below u. Insertion and searching in M can be performed in expected constant time on a uniform RAM model using hashing. Similarly, *mk* and *mk*′ can be performed in expected constant time due to the hash table H. Thus, the expected running time for each call to *up_one* is constant. This yields an overall running time of $O(n)$, where $n = |u|$. ■

The analysis of *up_one* above makes no assumptions about how the variables occur in the BED. However, if the BED is known to be at least free (see Section 2), several optimizations can be performed. Although these optimization do not improve on the bounds given in the theorem above, they reduce the actual runtime of the algorithm. The first optimization is that *mk*′ can be replaced with the slightly simpler *mk*. The second optimization is based on the observation that if u is a vertex with variable x, the recursive calls in line 4 and the following tests can be omitted. This is done by adding the following line

$2\frac{1}{2}$: **else if** $\alpha(u)$ is variable x **then return** u

after line 2.

Up_one is a very versatile algorithm. In Section 4.3 we show how *up_one* is used to implement substitution and existential quantification in BEDs. Here we show how it is used to transform an arbitrary BED into an OBDD. Let $x_1 < \cdots < x_n$ be the variable ordering of the OBDD. The OBDD with root v is constructed by calling *up_one* for each variable in this ordering:

$$v \leftarrow up_one(x_1, up_one(x_2, \ldots up_one(x_n, u) \cdots)) .$$

These calls pull up the variables in reverse order; that is, first x_n is pulled to the root, then x_{n-1} is pulled up and so on. This is clearly inefficient since each variable has to pass through all the variables that have already been pulled up.

A more efficient approach is to pull up the variables in order and instead of pulling a variable all the way to the root each time, it is only pulled up until it reaches a variable which precedes it in the ordering. This is done by modifying *up_one* by adding the lines

$4\frac{1}{3}$: **if** $\alpha(u)$ is a variable with $var(u) < x$ **then** $r \leftarrow mk(\alpha(u), l, h)$

$4\frac{2}{3}$: **else**

after line 4. Let *up_one*′ be this modified version of *up_one*. Then we construct the OBDD by

$$v \leftarrow up_one'(x_n, up_one'(x_{n-1}, \ldots up_one'(x_1, u) \cdots)) .$$

The runtime of this computation is exponential in the worst case, even though *up_one* has linear runtime and it is called only n times. However, since BEDs are provably exponentially more succinct than OBDDs, any transformation algorithm will have exponential worst-case behavior.

4.2. Construction of OBDDs with up_all

The second elimination algorithm, *up_all*, is a generalization of Bryant's *apply*-operator, shown in Fig. 7. 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. 8. As when building an OBDD using

```
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
```

FIG. 7. 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 memorization table M must be initialized to empty prior to the first call.

apply, *up_all* requires that a total ordering $<$ of the variables is selected prior to the OBDD construction. Based on the ordering *up_all* converts any BED into an OBDD. Key properties of *up_all* are:

THEOREM 10 (Up_all). *Assume u is a vertex in a* BED *and $x_1 < \cdots < x_n$ is an ordering of the variables, and let $v = up_all(u)$. The following properties hold*:

(i) $f^v = f^u$.

(ii) *v is an* OBDD.

(iii) *If l and h are* OBDDs, *then $apply(op, l, h) = up_all(mk(op, l, h))$.*

(iv) *If l and h are* OBDDs, *the running time of $up_all(op, l, h)$ is expected $O(|l||h|)$ on a uniform* RAM *model using hashing.*

```
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) ← (up_all(low(u)), up_all(high(u)))
          /* l and h are OBDDs */
  5:      if l and h are terminal vertices then
  6:        r ← mk(α(u), l, h)
  7:      else if α(u) is a variable x with x ≤ var(l) and x ≤ var(h) then
  8:        r ← mk'(x, l, h)
  9:      else if var(l) = var(h) then
 10:        r ← mk(var(l), up_all(mk(α(u), low(l), low(h))),
                           up_all(mk(α(u), high(l), high(h))))
 11:      else if var(l) < var(h) then
 12:        r ← mk(var(l), up_all(mk(α(u), low(l), h)),
                           up_all(mk(α(u), high(l), h)))
 13:      else var(l) > var(h) :
 14:        r ← mk(var(h), up_all(mk(α(u), l, low(h))),
                           up_all(mk(α(u), l, high(h))))
 15:      insert (u, r) in M
 16:      return r
```

FIG. 8. The *up_all*-operation on a BED u. The total order $<$ is defined as for *apply* (see Fig. 7). The memorization table M must be initialized to empty prior to the first call.

Proof. Let $|u|_{\neg\,\text{OBDD}}$ be the number of vertices below u that are not OBDD vertices according to the given ordering:

$$|u|_{\neg\,\text{OBDD}} = |\{v : u \rightsquigarrow v \text{ and } v \text{ is not an OBDD vertex}\}.$$

Properties (i) and (ii) are proven simultaneously by well-founded induction on the lexicographical ordering $\prec$ of the measure

$$(|u|_{\neg\,\text{OBDD}}, |u|).$$

As in the proof of Theorem 9 we ignore the memorization table M. The induction hypothesis is that $f^w = f^{\textit{up-all}(w)}$ and $\textit{up_all}(w)$ is an OBDD for all vertices w with $w \prec u$. The base case, when u is a terminal, clearly fulfills properties (i) and (ii). For the induction step, we first argue that $\textit{low}(u) \prec u$ and $\textit{high}(u) \prec u$. If u is already an OBDD with the given variable ordering, $\textit{low}(u)$ is also an OBDD with the given ordering, and the first part of the measure is zero for both $\textit{low}(u)$ and $\textit{high}(u)$. However, the sizes of both $\textit{low}(u)$ and $\textit{high}(u)$ are less than u and thus $\textit{low}(u) \prec u$ and $\textit{high}(u) \prec u$. If u is not an OBDD with the given variable ordering, then $\textit{low}(u)$ and $\textit{high}(u)$ contain at least one non-OBDD vertex less than u, showing that $\textit{low}(u) \prec u$ and $\textit{high}(u) \prec u$.

This allows us to use the induction hypothesis to conclude that $f^l = f^{\textit{low}(u)}$ and $f^h = f^{\textit{high}(u)}$ and that l and h are OBDDs. If l and h are both terminals, the **if**-branch in line 5 is chosen and properties (i) and (ii) clearly hold for r. If $\alpha(u)$ is a variable x smaller than the variables in l and h, the **if**-branch in line 7 is chosen and r fulfills (i) and (ii) using the fact that $\textit{mk}'$ will remove duplicate occurrences of x.

For the remaining three **if**-branches, we first argue that the arguments of the recursive calls are before u in the order $\prec$. We consider only the first of the three branches since the remaining two are similar. The argument of the first recursive call in line 10 is

$$w = \textit{mk}(\alpha(u), \textit{low}(l), \textit{low}(h));$$

thus we must show that $w \prec u$. Since l and h are OBDDs, the number of non-OBDDs vertices in w is at most one; i.e., $|w|_{\neg\,\text{OBDD}} \leq 1$. If we reach this **if**-branch, $\alpha(u)$ is either a variable larger than l and h or u is an operator vertex. Therefore, u cannot be an OBDD-vertex and $|u|_{\neg\,\text{OBDD}} \geq 1$. Thus, either the measure decreases in its first component, i.e., $|w|_{\neg\,\text{OBDD}} < |u|_{\neg\,\text{OBDD}}$, or it is the case that $|u|_{\neg\,\text{OBDD}} = |w|_{\neg\,\text{OBDD}} = 1$. In the first case we immediately have that $w \prec u$. In the second case, u contains only one non-OBDD vertex and this must be in the root, i.e., u. Therefore $\textit{low}(u)$ and $\textit{high}(u)$ are already OBDDs with the given variable ordering. The canonicity of OBDDs together with the fact that $f^l = f^{\textit{low}(u)}$ implies that $l = \textit{low}(u)$ and similarly $h = \textit{high}(u)$. The size of w is therefore strictly less than that of u and $w \prec u$.

A similar argument allows us to conclude that the measure is also decreasing in the second recursive call. We can then rewrite from the definition of r and use the induction hypothesis and the semantics of $\textit{mk}$ to obtain the following equivalences (assuming $\alpha(u)$ is an operator, the steps are similar for a variable):

$$\begin{aligned}
f^r &= \textit{var}(l) \to f^{\textit{mk}(\alpha(u),\textit{high}(l),\textit{high}(h))}, f^{\textit{mk}(\alpha(u),\textit{low}(l),\textit{low}(h))} \\
&= \textit{var}(l) \to \left(f^{\textit{high}(l)}\ \alpha(u)\ f^{\textit{high}(h)}\right), \left(f^{\textit{low}(l)}\ \alpha(u)\ f^{\textit{low}(h)}\right) \\
&= \left(\textit{var}(l) \to f^{\textit{high}(l)}, f^{\textit{low}(l)}\right)\ \alpha(u)\ \left(\textit{var}(l) \to f^{\textit{high}(h)}, f^{\textit{low}(h)}\right) \\
&= f^l\ \alpha(u)\ f^h \\
&= f^{\textit{low}(u)}\ \alpha(u)\ f^{\textit{high}(u)} \\
&= f^u.
\end{aligned}$$

This completes the proof of properties (i) and (ii).

Property (iii) is shown by induction on the size of $|u|$ using the induction hypothesis $\textit{apply}(\alpha(u),$ $\textit{low}(u), \textit{high}(u)) = \textit{up_all}(u)$ where $\alpha(u)$ is an operator and $\textit{low}(u)$ and $\textit{high}(u)$ are OBDDs. From (ii)

and the canonicity of OBDDs, it follows that $up_all(u) = u$ if u is an OBDD. Thus, the calls to *up_all* in line 4 just return the arguments; i.e., $l = low(u)$ and $h = high(u)$. When $\alpha(u)$ is an operator, the **if**-branch in line 7 is never taken and it is then clear how the four cases of *apply* correspond exactly to the remaining four cases of *up_all*.

Property (iv) is shown by bounding the number of elements inserted into M. By induction one can show that for all pairs (w, r) inserted in M, one of the following two cases hold:

- $r = w$ and either $low(u) \rightsquigarrow w$ or $high(u) \rightsquigarrow w$, or
- w is an operator vertex with $op(w) = \alpha(u)$, $low(u) \rightsquigarrow low(w)$, and $high(u) \rightsquigarrow high(w)$.

Thus, the number of elements in M is bound by $|low(u)| + |high(u)| + |low(u)||high(u)|$ and since the body of *up_all* takes constant time, the total runtime is $O(|low(u)||high(u)|)$. ∎

The worst-case runtime of *up_all* is exponential in $|u|$. For the same reason as *up_one*, this is optimal.

As presented above, *up_all* makes no assumptions about how the variables occur in the BED. However, *up_all* can be optimized for several special cases:

Free BEDs: The call to *mk'* can be replaced by *mk* in line 8.

Ordered BEDs: The condition of the **if**-statement in line 7 can be simplified to

7': **if** $\alpha(u)$ is a variable x **then**

since $x < var(l)$ and $x < var(h)$ in an ordered BED.

DAG of OBDDs: The condition of the **if**-statement in line 2 can be relaxed to

2': **if** u is a terminal or $\alpha(u)$ is a variable **then return** u

since if $\alpha(u)$ is a variable, u is an OBDD.

An operator on OBDDs: If u is an operator vertex such that $low(u)$ and $high(u)$ are OBDDs, line 4 can be replaced with $(l, h) \leftarrow (low(u), high(u))$ since $up_all(w) = w$ when w is an OBDD. Furthermore, the test in line 7 is always false; thus in this case there is a one-to-one correspondence between *up_all* and *apply*. 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 OBDD. Figure 9 illustrates how the operator vertices are converted into variable vertices by the two algorithms.

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 function f. The simplest way to perform

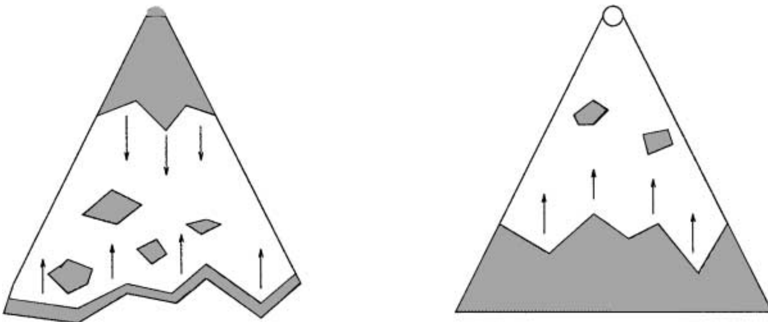

FIG. 9. Converting a BED to an OBDD. To the left, using *up_one* repeatedly, to the right, during an *up_all* call. Gray areas represent variables and white areas represent operators.

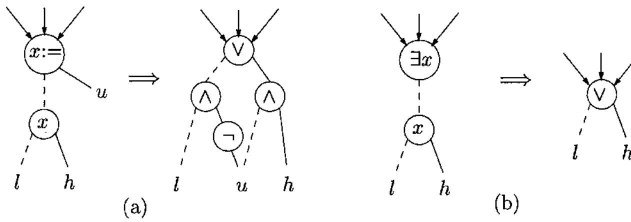

FIG. 10. (a) Elimination of substitution vertex. (b) Elimination of existential quantification vertex.

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 9 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 f. 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 $\exists x.x \to f, g = f \vee g$. For both operations, the complexity is determined by *up_one* which is linear in the size of the BED.

An alternative way to implement substitution and existential quantification is to consider them special operator vertices in the BED; see Fig. 10. The up-step from Fig. 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 Fig. 10. 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 subexpressions. Consider the BED in Fig. 11. If the vertices v and v' are identified 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 f equal to either true or false. Notice that all the operations described in this section have linear running times which is better than the corresponding OBDD operations. Other operations, like finding a satisfying truth-assignment, can be performed by constructing an OBDD using *up_all* and performing the operation on the OBDD. Since *up_all* never has

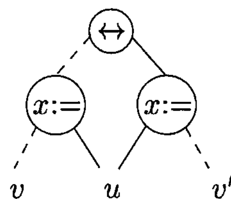

FIG. 11. A BED containing substitution operators. The low-edge points to the expression in which x is to be substituted with f^u, pointed to by the high-edge.

TABLE 2

Experimental Results for Verifying Equivalence of the Redundant and Nonredundant Circuits in the ISCAS 85 Benchmark with and without Performing Operator Reductions

	Up_one		*Up_all*	
Circuit	Reductions [*s*]	No reductions [*s*]	Reductions [*s*]	No reductions [*s*]
c432, c432nr	**2.5**	2.9	**2.1**	2.5
c499, c499nr	**5.2**	166.1	**2.4**	2.5
c499, c1355	**1.6**	532.5	**1.6**	3.9
c1355, c1355nr	**5.3**	743.5	**2.6**	4.1
c1908, c1908nr	**1.0**	—	**1.0**	**1.0**
c2670, c2670nr	**1.4**	—	**1.0**	1.4
c3540, c3540nr	**16.9**	111.7	**17.0**	56.9
c5315, c5315nr	**17.8**	—	**3.1**	3.5
c6288, c6288nr	**2.0**	—	—	—
c7552, c7552nr	**4.6**	7.4	**2.6**	3.1

Note. All runtimes are in seconds and measured on a 300 MHz Pentium II PC running Linux with a memory limit of 32 MB.

a worse running time than *apply*, the total running time for these operations is no worse than if they are performed directly on an OBDD.

5. AN APPLICATION OF BEDs

Tautology checking of a BED is an application where the end-result as an OBDD is known to be small (the terminal vertex **1**) if the BED indeed is a tautology. An example of this is the combinational logic-level equivalence problem which is to determine whether two given combinational circuits implement the same Boolean function. By combining the two circuits into one using biimplications, it becomes an instance of the tautology problem. We have carried out a detailed investigation of the application of BEDs to the verification of combinational circuits, including experiments on more than 250 circuits with up to 100k gates [15]. In summary, the BEDs either outperform or achieve results comparable to any other published technique including those specifically developed for the equivalence problem.

When using *up_one* or *up_all* an ordering of the variables must be selected. 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*. Among the many published ordering heuristics, the FANIN heuristic [19] and the DEPTH_FANOUT heuristic [20] have turned out to perform particularly well on BEDs.

To demonstrate the efficiency of BEDs, we consider two implementations of a 16-bit multiplier (c6288 and c6288nr from the ISCAS-85 benchmarks). Checking whether these two versions implement the same functionality corresponds to a tautology check for each pair of outputs. A BED is built directly from the circuit netlist as explained in Section 2. The circuits c6288 and c6288nr contain 2416 and 2399 gates, respectively, and the resulting reduced BED contains 3478 vertices. All 32 outputs are verified using *up_one* in 2.0 CPU seconds on a standard PC.

This example was chosen since multipliers are notoriously difficult to verify using OBDDs [4]. Due to the exponential growth of the size of the OBDD representation (in the number of operand bits), the straightforward approach of building and comparing the OBDD for the two circuits c6288 and c6288nr is infeasible. The OBDD representation of a 16-bit multiplier uses more than 40 million vertices [27] and this number is approximately 2.7 times larger for each additional bit in the operands [22]. With *up_one* the verification takes 2 s, which clearly demonstrates the effectiveness of *up_one*.

To investigate the effect of operator reductions, we have carried out a series of experiments on the ISCAS 85 benchmark.[3] The equivalence of the circuits was verified using *up_one* and *up_all* with and

[3] Available from The Collaborative Benchmarking Laboratory (`http://www.cbl.ncsu.edu/`).

TABLE 3

Verifying Equivalence of Hierarchically Specified n-Bits Multipliers

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

Note. 'Ops' is the number of ordinary operator vertices in the BED specification, 'Subst' is the number of substitution vertices, and N_{total} is the total number of vertices used.

without performing any of the operator reductions described in Section 3.2. (The reductions needed to maintain reducedness, see Definition 5, are applied in all experiments.) 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 2. Clearly, the efficiency of *up_one* relies heavily on the operator reductions to identify identical nodes in the BED and thus avoid 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 nonoperator vertices as the children. I.e., whenever $mk(\alpha, l, h)$ 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 of *up_all*. On the other hand, the time to perform a table lookup is saved for each BED vertex that is created when no operator reductions are used. Thus the runtimes for some of the circuits do not improve.

Substitution operators can be used to represent a circuit hierarchy. Consider a circuit with two instances of a subcircuit *cell*; see Fig. 12(a). Instead of flattening the hierarchy, the two instances of *cell* are represented in the BED data structure using substitutions; see Fig. 12(b). This information can greatly improve the performance when verifying equivalence of large hierarchical circuits.

We illustrate this by verifying equivalence between two implementations of n-bit multipliers. A combinational n-bit multiplier can be constructed using four $n/2$-bit multipliers. The four multipliers each compute partial products which are shifted and added to form the result. In this way we build two versions of hierarchically specified n-bit multipliers based on the 16-bit multipliers c6288 and c6288nr. Table 3 shows the results of our experiments. The BED representing two hierarchically specified 1024-bit multipliers contains 11.3 million vertices (requiring approximately 226 MB of memory), yet the equivalence can be established in only 7 CPU minutes.

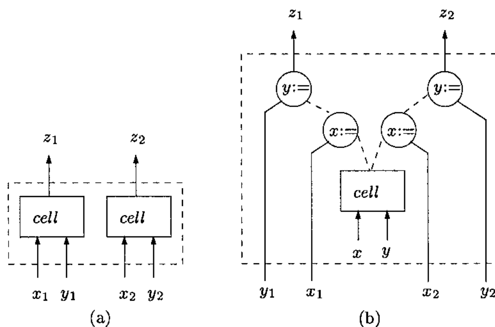

FIG. 12. (a) Two instances of a sub-circuit *cell*. (b) Representing the same circuit using substitutions.

6. CONCLUSION

We have presented a new data structure called Boolean expression diagram for representing and manipulating Boolean expressions. BEDs are as succinct as Boolean circuits. Properties such as TAUTOLOGY and SATISFIABILITY are determined by transforming the BED representation into a reduced ordered BDD. This can typically be done efficiently by using one of the two algorithms *up_one* or *up_all* although, of course, the worst-case behavior is exponential. As shown in Theorem 10, the cost of constructing an OBDD from scratch using *apply* and the cost of building a BED and transforming it into an OBDD using *up_all* are within a constant factor. In fact, recent research [14] has shown that an algorithm similar to *up_all* is a highly efficient approach to the construction of OBDDs; it uses considerably less memory and no more time than when the OBDD is constructed with *apply*.

Up_one is a new way to construct an OBDD which can exploit structural similarities between subexpressions. For some applications *up_one* is highly efficient, for example as demonstrated by proving the identity of two 16-bit multipliers (`c6288` and `c6288nr` from the ISCAS 85 benchmarks) in 2 s. *Up_one* is also the basis for other operations such as existential quantification and substitution, making the running times of these operations linear in the size of the BED.

BEDs are particularly useful when the end result is expected to have a small OBDD 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-point iterations using OBDDs, 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 OBDD. In fact, some of the tricks researchers have used to make OBDDs more efficient are embodied in BEDs. For example, Burch *et al.* [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 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.

BEDs can be seen as an intermediate form between circuits and the highly structured OBDDs. In this paper we have focused on ways to obtain an OBDD from the BED description. However, there seems to be an unexploited potential for manipulating the BEDs directly, without necessarily converting them to OBDDs. For example, it seems plausible that ideas from theorem proving can be transferred to BEDs; the reduction of operator vertices described in Section 3.2 is only a first step in this direction. An indication of the benefits is given by the recent work on generation of BDDs for fault trees which has shown that initial rewritings more powerful than the operator reductions allow for the generation of BDDs for fault trees of larger complexity than possible without the rewrites [21].

ACKNOWLEDGMENT

Thanks to an anonymous referee for constructive comments.

REFERENCES

1. Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974), "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, MA.
2. Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., and Somenzi, F. (1993), Algebraic decision diagrams and their applications, *in* "Proc. International Conf. Computer-Aided Design (ICCAD)," pp. 188–191.
3. Boppana, R. B., and Sipser, M. (1990), The complexity of finite functions, *in* "Handbook of Theoretical Computer Science," (J. van Leeuwen, Ed.), Vol. A: Algorithms and Complexity, pp. 758–804, Elsevier, Amsterdam/New York.
4. Bryant, R. E. (1986), Graph-based algorithms for Boolean function manipulation, *IEEE Trans. Comput.* **35**, 677–691.
5. Bryant, R. E., and Chen, Y.-A. (1995), Verification of arithmetic functions with binary moment diagrams, *in* "Proc. ACM/IEEE Design Automation Conference (DAC)," pp. 535–541.
6. Burch, J. R., Clarke, E. M., and Long, D. E. (1991), Representing circuits more efficiently in symbolic model checking, *in* "Proc. ACM/IEEE Design Automation Conference (DAC)," pp. 403–407.
7. Clarke, E. M., McMillan, K. L., Zhao, X., Fujita, M., and Yang, J. (1993), Spectral transforms for large Boolean functions with application to technology mapping, *in* "Proc. ACM/IEEE Design Automation Conference (DAC)," pp. 54–60.

8. Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990), "Introduction to Algorithms," MIT Press, Cambridge, MA.
9. Drechsler, R., Sarabi, A., Theobald, M., Becker, B., and Perkowski, M. A. (1994), Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams, *in* "Proc. ACM/IEEE Design Automation Conference (DAC)," pp. 415–419.
10. Fujita, M., Matsunga, Y., and Kakuda, T. (1991), On variable ordering of binary decision diagrams for the application of multi-level synthesis, *in* "Proc. European Conference on Design Automation (EDAC)," pp. 50–54.
11. Garey, M. R., and Johnson, D. S. (1979), "Computers and Intractability—A Guide to the Theory of NP-Completeness," Freeman, San Francisco.
12. Gergov, J., and Meinel, C. (1994), Efficient Boolean manipulation with OBDD's can be extended to FBDD's, *IEEE Trans. Comput.* **43**, 1197–1209.
13. Hett, A., Drechsler, R., and Becker, B. (1996), MORE: Alternative implementation of BDD-pakages by multi-operand synthesis, *in* "European Design Conference."
14. Hett, A., Drechsler, R., and Becker, B. (1997), Fast and efficient construction of BDDs by reordering based synthesis, *in* "IEEE European Design & Test Conference."
15. Hulgaard, H., Williams, P. F., and Andersen, H. R. (1999), Equivalence checking of combinational circuits using Boolean expression diagrams, *in* "IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems," Vol. 18, p. 7.
16. Jain, J., Bitner, J., Abadir, M. S., Abraham, J. A., and Fussell, D. S. (1997), Indexed BDDs: Algorithmic advances and techniques to represent and verify Boolean functions, *IEEE Trans. Comput.* **46**, 1230–1245.
17. Jeong, S.-W., Plessier, B., Hachtel, G., and Somenzi, F. (1991), Extended BDD's: Trading off canonicity for structure in verification algorithms, *in* "Proc. International Conf. Computer-Aided Design (ICCAD)," pp. 464–467.
18. Kebschull, U., Schubert, E., and Rosenstiel, W. (1992), Multilevel logic synthesis based on functional decision diagrams, *in* "Proc. European Conference on Design Automation (EDAC)," pp. 43–47.
19. Malik, S., Wang, A. R., Brayton, R. K., and Sangiovanni-Vincentelli, A. (1988), Logic verification using binary decision diagrams in a logic synthesis environment, *in* "Proc. International Conf. Computer-Aided Design (ICCAD)," pp. 6–9.
20. Minato, S. (1996), "Binary Decision Diagrams and Applications for VLSI CAD," Kluwer Academic, Dordrecht.
21. Nikolskaia, M., Rauzy, A., and Sherman, D. J. (1998), Almana: A BDD minimization tool integrating heuristic and rewriting methods, *in* "Formal Methods in Computer Aided Design."
22. Ochi, H., Yasuoka, K., and Yajima, S. (1993), Breadth-first manipulation of very large binary-decision diagrams, *in* "Proc. International Conf. Computer-Aided Design (ICCAD)," pp. 48–55, IEEE Comput. Soc. Press, Los Alamitos, CA.
23. Plessier, B., Hachtel, G. D., and Somenzi, F. (1994), Extended BDD's: Trading off canonicity for structure in verification algorithms, *Formal Methods in System Design* **4**, 167–185.
24. Rudell, R. (1993), Dynamic variable ordering for ordered binary decision diagrams, *in* "Proc. International Conf. Computer-Aided Design (ICCAD)," pp. 42–47.
25. Sieling, D., and Wegener, I. (1995), Graph driven BDDs—A new data structure for Boolean functions, *Theoret. Comput. Sci.* **141**, 283–310.
26. Wegener, I. (1988), On the complexity of branching programs and decision trees for clique functions, *J. Assoc. Comput. Mach.* **35**, 461–471.
27. Yang, B., Chen, Y.-A., Bryant, R. E., and O'Hallron, D. R. (1998), Space- and time-efficient BDD construction via working set control, *in* "Proceedings of ASP-DAC'98," pp. 423–432.