Edge-Valued Binary-Decision Diagrams

Yung-Te Lai, Massoud Pedram
Dept. of EE-Systems
University of Southern California
Los Angeles, CA 90089

Sarma B.K. Vrudhula
Dept. of ECE
University of Arizona
Tuscon, AZ 85721

Abstract

In this paper we present a new data structure called Edge-Valued Binary-Decision Diagrams (EVBDD). An EVBDD is a directed acyclic graph, that provides a canonical and compact representation of functions that involve both Boolean and integer quantities. In general, EVBDDs provide a more versatile and powerful representation that Ordinary Binary Decision Diagrams. We first describe the structure and properties of EVBDDs, and present a general algorithm for performing a variety of binary operations. Next, we describe an important extension of EVBDDs, called *Structural* EVBDDs, and show how they can be used for hierarchical verification.

1 Introduction

Ordered Binary-Decision Diagrams (OBDDs), proposed by Bryant [4], provide a canonical and compact representation of Boolean functions. The canonical property makes it possible to easily detect many useful properties of Boolean functions, e.g., size of the support set, unateness of variables, symmetry between variables, etc. Furthermore, OBBD's compact representation coupled with the use of data structures for caching intermediate computations, allows for the efficient implementation of many Boolean operations. For example, tautology checking and complementation take constant time while conjunction and disjunction take polynomial time in the size of OBDDs. Although the number of nodes in OBDD representations may be exponential in the input size, OBDDs have a reasonable size in many practical applications [19].

In addition to Boolean functions, OBBDS can also be used for problems that are defined in small, finite domains. For example, after encoding each element in a set of size N by a vector of $n = \lceil \log_2 N \rceil$ binary variables, a set can be represented by a Boolean function with n variables such that an element is in the set if and only if its corresponding function value is *true*. Set operations such as union and intersection then correspond to Boolean disjunction and conjunction; testing if a set is empty is equivalent to checking if its corresponding Boolean function is the constant function 0. Similar to the above 'symbolic analysis', many tasks encountered in computer aided design, combinatorial optimization, mathematical logic, and artificial intelligence can be formulated and solved by using OBDDs [6].

While OBDDS are useful for problems that require symbolic manipulation of Boolean functions, they are not very effective for problems that require arithmetic operations in the integer domain. For example, integer functions can be represented by vectors of Boolean functions. However, to perform arithmetic operations requires performing Boolean operations on each bit; a task that may be very time consuming.

In this paper, we present a new data structure called Edge-Valued Binary-Decision Diagrams (EVBDDs) [13]. An EVBDD provides a canonical and compact representation of functions that involve both Boolean and integer quantities. This makes them suitable for a variety of important applications. In [18] three such applications of EVBDDs are presented; *Integer Linear Programming*, *Spectral Transformations* of Boolean functions, and *Multiple-Output Decomposition* of Boolean functions. The focus of this paper is on demonstrating the basic properties of EVBDDs and their use in formal hierarchical verification.

EVBDDs are directed acyclic graphs that are constructed in a manner similar to OBDDs.

As in OBDDs, each node in a EVBDD either represents a constant function with no children or it is associated with a binary variable having two children. Furthermore, an ordering of the input variables is imposed on every path from the root node to the terminal node. However, in EVBDDs there is an integer value associated with each edge. Furthermore, the semantics of these two graphs are quite different. In OBDDs, a node **v** associated with variable x denotes the **Boolean function** $(x \wedge f_l) \vee (\overline{x} \wedge f_r)$, where f_l and f_r are functions represented by the two children of **v**. On the other hand, a node **v** in an EVBDD denotes the **arithmetic function** $x(v_l + f_l) + (1-x)(v_r + f_r)$, where v_l and v_r are values associated with edges going from **v** to its children, and f_l and f_r are functions represented by the two children of **v**. To achieve canonical property, we enforce v_r to be 0.

EVBDDs constructed in the above manner are more related to pseudo Boolean functions [12] which have the function type $\{0,1\}^n \rightarrow integer$. For example, $f(x,y,z) = 3x + 4y - 5xz$ with $x,y,z \in \{0,1\}$ is a pseudo Boolean function, and $f(1,1,0) = 7$ and $f(1,1,1) = 2$. However, for functions with integer variables, we must convert the integer variables to vectors of Boolean variables before using EVBDDs. In the above example, if $x \in \{0,\ldots,5\}$, then $f(x,y,z) = 3(4x_2+2x_1+x_0)+4y-5(4x_2+2x_1+x_0)z$ and $f(4,1,1) = -4$.

By treating Boolean values as integers 0 and 1, EVBDDs can also be used to represent Boolean functions and perform Boolean operations. Furthermore, when Boolean functions are represented by OBDDs and EVBDDs, they have the same size and require the same time complexity for performing operations. Thus, EVBDDs are particularly useful in applications which require both Boolean and integer operations.

We present an application of EVBDDs to logic verification, where the objective is to show the equivalence between an abstract functional specification and its logic implementation. EVBDDs are particularly useful when the specification is expressed over the domain of integers. In contrast, the use of OBDDs requires the specification to be expressed at the logic level. For example, if the behavior of a 64-bit adder is specified through 65 Boolean functions (64 bits plus carry), then the behavior of arithmetic addition can never be proved. On the other hand, if the specification language allows the operator '+' directly (e.g., '$x + y$'), then the correctness is demonstrated up to the arithmetic level. One cannot directly verify the correctness of arithmetic functions using OBDDs. Since EVBDDs can represent both Boolean and arithmetic functions, the equivalence between these two functions can be proved directly up to the arithmetic level.

This paper is organized as follows. In Section 2, we define the syntax and semantics of

EVBDDs, and present a *general* algorithm for applying any binary operator (closed over integers) on EVBDDs. Next, we present an analysis of its complexity, and show the relationship between EVBDDs and Boolean functions. In Section 3 we describe an important extension of an EVBDD, called *Structural* EVBDD (SEVBDD) for use in hierarchical verification. Conclusions are given in Section 4. **Note:** Due to limitations on space, proofs of the theorems are not included. Detailed proofs are available in [17].

2 Edge-Valued Binary-Decision Diagrams

In this section, we first define the EVBDD data structure and prove its canonical property. We then present a general paradigm for operating on EVBDDs and elaborate on some of their important properties that permit speeding up of the operations. Finally, we show that representing Boolean functions by OBDDs and EVBDDs have the same space (in terms of the number of nodes in the data structure) and time (in terms of the number of operations) complexities.

2.1 Definitions

The following definitions describe the syntax and semantics of EVBDDs.

Definition 2.1 An EVBDD is a tuple $\langle c, \mathbf{f} \rangle$ where c is a constant value and $\mathbf{f}$ is a directed acyclic graph consisting of two types of nodes:

1. There is a single *terminal node* with value 0 (denoted by $\mathbf{0}$).
2. A *nonterminal node* $\mathbf{v}$ is a 4-tuple $\langle variable(\mathbf{v}), child_l(\mathbf{v}), child_r(\mathbf{v}), value \rangle$, where $variable(\mathbf{v})$ is a binary variable $x \in \{x_0, \ldots, x_{n-1}\}$.

An EVBDD is ordered if there exists an index function $index(x) \in \{0, \ldots, n-1\}$ such that for every nonterminal node $\mathbf{v}$, either $child_l(\mathbf{v})$ is a terminal node or $index(variable(\mathbf{v})) < index(variable(child_l(\mathbf{v})))$, and either $child_r(\mathbf{v})$ is a terminal node or $index(variable(\mathbf{v})) < index(variable(child_r(\mathbf{v})))$. If $\mathbf{v}$ is the terminal node $\mathbf{0}$, then $index(\mathbf{v}) = n$. An EVBDD is reduced if there is no nonterminal node $\mathbf{v}$ with $child_l(\mathbf{v}) = child_r(\mathbf{v})$ and $value = 0$, and there are no two nonterminal nodes $\mathbf{u}$ and $\mathbf{v}$ such that $\mathbf{u} = \mathbf{v}$. ■

Definition 2.2 An EVBDD $\langle c, \mathbf{f} \rangle$ denotes the arithmetic function $c+f$ where f is the function denoted by $\mathbf{f}$. $\mathbf{0}$ denotes the constant function 0, and $\langle x, \mathbf{l}, \mathbf{r}, v \rangle$ denotes the arithmetic function $x(v+l)+(1-x)r$. ■

In this paper, we consider only reduced, ordered EVBDD. In the graphical representation of an EVBDD $\langle c, \mathbf{f} \rangle$, $\mathbf{f}$ is represented by a rooted, directed, acyclic graph and c by a dangling incoming edge to the root node of $\mathbf{f}$. The terminal node is depicted by a rectangular node labeled 0. A nonterminal node is a quadruple $\langle x, \mathbf{l}, \mathbf{r}, v \rangle$, where x is the node label, $\mathbf{l}$ and $\mathbf{r}$ are the two subgraphs rooted at x, and v is the label assigned to the left edge of x.

Example 2.1 Fig. 1 shows an EVBDD representation of an arithmetic function $f = -2+5y+yz+3xy+4xyz-2xz+z$. The ordering of the variables used to represent f is $\langle y, x, z \rangle$. Thus, choosing y as the first variable, f is represented as $-2+y(5+3x+2xz+2z)+(1-y)(z-2xz)$. This has the form $c+y(v+\mathbf{l})+(1-y)\mathbf{r}$, where $\mathbf{l} = 3x+2xz+2z$ and $\mathbf{r} = z-2xz$. This process is now repeated on the functions denoted by $\mathbf{l}$ and $\mathbf{r}$, using x, followed by z, as the decoding variables. ■

Each path in a EVBDD corresponds to an assignment of values to the variables in the path. Evaluation of a function represented by a EVBDD for a given assignment of values to its arguments, is carried out by simply summing the values along the edges (right edge values are always set to 0) in the path. For example, in Fig. 1, the value of the function with $x = 1, y = 0$ and $z = 1$ is $-2+0+0+-1 = -3$. The definition of the function *eval*, which evaluates an EVBDD, given the values of the variables, is presented below.

Definition 2.3 Given an EVBDD $\langle c, \mathbf{f} \rangle$ with variable ordering $x_0 < \ldots < x_{n-1}$, the evaluation of $\langle c, \mathbf{f} \rangle$ with respect to an input pattern $\langle b_0, \ldots, b_{i-1} \rangle, 0 \leq i < n$ is defined as follows:

$$
\begin{aligned}
eval(\langle c, \mathbf{0} \rangle, \langle b_0, \ldots, b_{i-1} \rangle) &= c, \\
eval(\langle c, \langle x_j, \mathbf{l}, \mathbf{r}, v \rangle \rangle, \langle b_0, \ldots, b_{i-1} \rangle) &= \begin{cases} eval(\langle c+v, \mathbf{l} \rangle, \langle b_0, \ldots, b_{i-1} \rangle) & \text{if } j < i \text{ and } b_j = 1, \\ eval(\langle c, \mathbf{r} \rangle, \langle b_0, \ldots, b_{i-1} \rangle) & \text{if } j < i \text{ and } b_j = 0, \\ \langle c, \langle x_j, \mathbf{l}, \mathbf{r}, v \rangle \rangle & \text{if } j \geq i. \end{cases}
\end{aligned}
$$

Lemma 2.1 *(EVBDDs are canonical)* Two EVBDDs $\langle c_f, \mathbf{f} \rangle$ and $\langle c_g, \mathbf{g} \rangle$ denote the same function (i.e., $\forall b \in B^n$,
$eval(\langle c_f, \mathbf{f} \rangle, b) = eval(\langle c_g, \mathbf{g} \rangle, b)$), if and only if $c_f = c_g$ and $\mathbf{f}$ and $\mathbf{g}$ are isomorphic. ■

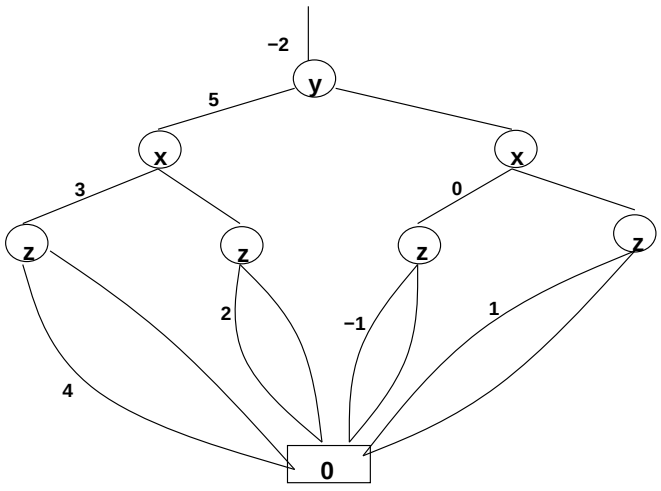

Figure 1: EVBDD representation of $f = -2 + 5y + yz + 3xy + 4xyz - 2xz + z$.

2.2 Operations

In this section we describe a general algorithm, called *apply*, which applies any binary operator *op* that is closed over the integers to two EVBDDs. That is, *apply* takes $\langle c_f, \mathbf{f}\rangle, \langle c_g, \mathbf{g}\rangle$ and *op* as arguments and returns $\langle c_h, \mathbf{h}\rangle$ such that $c_h + h \equiv (c_f + f)\ op\ (c_g + g)$.

In algorithm *apply*, a terminal case (line 1) occurs when the result can be computed directly. For example, if $op = \times$ then $\langle c_f, \mathbf{f}\rangle = \langle 1, \mathbf{0}\rangle$ is a terminal case since $\langle 1, \mathbf{0}\rangle = 1{+}0 = 1$, $\langle c_g, \mathbf{g}\rangle = c_g + g$, and $1 \times (c_g + g) = (c_g + g) = \langle c_g, \mathbf{g}\rangle$. Thus the result can be returned immediately without traversing the graph.

To speed up the computation, previously computed results are stored in a table called *comp_table*. An entry in *comp_table* has the form $\langle f, g, op, h\rangle$ which stands for $f\ op\ g = h$. To compute $f\ op\ g$, *comp_table* is first searched with the key $\langle f, g, op\rangle$. If an entry is found then the last element of the entry h is retrieved as the result; otherwise, we perform the operation *op* on the subgraphs of f and g and store the result in *comp_table*. The entries of *comp_table* are used in line 2 and stored in line 21.

After the left and right children have been computed, the result will be two EVBDDs, $\langle c_{h_l}, \mathbf{h_l}\rangle$ and $\langle c_{h_r}, \mathbf{h_r}\rangle$ (lines 17 and 18), representing the functions $c_{h_l} + h_l$, and $c_{h_r} + h_r$. If $\langle c_{h_l}, \mathbf{h_l}\rangle = \langle c_{h_r}, \mathbf{h_r}\rangle$, then the algorithm returns $\langle c_{h_l}, \mathbf{h_l}\rangle$. This is to ensure that the structure of the form $\langle x, \mathbf{k}, \mathbf{k}, 0\rangle$ will not occur.

If $\langle c_{h_l}, \mathbf{h_l}\rangle \neq \langle c_{h_r}, \mathbf{h_r}\rangle$, and the current variable is x, then the resulting structure represents the function $x(c_{h_l} + h_l) + (1{-}x)(c_{h_r} + h_r)$. This is expressed as $c_{h_r} + x(c_{h_l}{-}c_{h_r} + h_l) + (1{-}$

$x)h_r$, ensuring that the right edge value is 0. Therefore, the procedure returns $\langle c_{h_r}, \langle var, \mathbf{h_l}, \mathbf{h_r}, c_{h_l} - c_{h_r}\rangle\rangle$. Another table, called (*uniq_table*), is used to ensure the uniqueness property of EVBDD nodes. Before *apply* returns its result, it checks this table through the operation *find_or_add*, which either adds a new node to the table or returns the node found in the table.

```
apply(⟨c_f, f⟩, ⟨c_g, g⟩, op)
{
1    if (terminal_case(⟨c_f, f⟩, ⟨c_g, g⟩, op)
         return(⟨c_f, f⟩ op ⟨c_g, g⟩));
2    if (comp_table_lookup(⟨c_f, f⟩, ⟨c_g, g⟩, op, ans))
         return(ans);
3    if (index(f) ≥ index(g)) {
4        ⟨c_gl, g_l⟩ = ⟨c_g + value(g), child_l(g)⟩;
5        ⟨c_gr, g_r⟩ = ⟨c_g, child_r(g)⟩;
6        var = variable(g);
7    }
8    else {
9        ⟨c_gl, g_l⟩ = ⟨c_gr, g_r⟩ = ⟨c_g, g⟩;
10       var = variable(f);
11   }
12   if (index(f) ≤ index(g)) {
13       ⟨c_fl, f_l⟩ = ⟨c_f + value(f), child_l(f)⟩;
14       ⟨c_fr, f_r⟩ = ⟨c_f, child_r(f)⟩;
15   }
16   else { ⟨c_fl, f_l⟩ = ⟨c_fr, f_r⟩ = ⟨c_f, f⟩;}
17   ⟨c_hl, h_l⟩ = apply(⟨c_fl, f_l⟩, ⟨c_gl, g_l⟩, op);
18   ⟨c_hr, h_r⟩ = apply(⟨c_fr, f_r⟩, ⟨c_gr, g_r⟩, op);
19   if (⟨c_hl, h_l⟩ == ⟨c_hr, h_r⟩) return (⟨c_hl, h_l⟩);
20   h = find_or_add(var, h_l, h_r, c_hl − c_hr);
21   comp_table_insert(⟨c_f, f⟩, ⟨c_g, g⟩, op, ⟨c_hr, h⟩);
22   return (⟨c_hr, h⟩);
}
```

Example 2.2 An example of $apply(\langle 3, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, +)$ is shown in Fig. 2. The variable ordering $x_0 < x_1$. Fig. 2 (a) shows the initial arguments of *apply*. Since $index(\mathbf{f}) < index(\mathbf{g})$, lines 9 and 10 are executed resulting in $\langle c_{g_l}, \mathbf{g_l}\rangle = \langle c_{g_r}, \mathbf{g_r}\rangle = \langle c_g, \mathbf{g}\rangle = \langle 0, \mathbf{g}\rangle$ and $var = x_0$. Next, lines 13 and 14 are executed, resulting in $\langle c_{f_l}, \mathbf{f_l}\rangle = \langle 5, \mathbf{0}\rangle$, and $\langle c_{f_r}, \mathbf{f_r}\rangle = \langle 3, \mathbf{0}\rangle$. Figure 2(b) shows the recursive call of *apply* on line 17, and the result of this call is shown in Figure 2(c). Similarly, another call to *apply* on line 18 and its results are shown in (d) and (e). The final result is shown in (f). ■

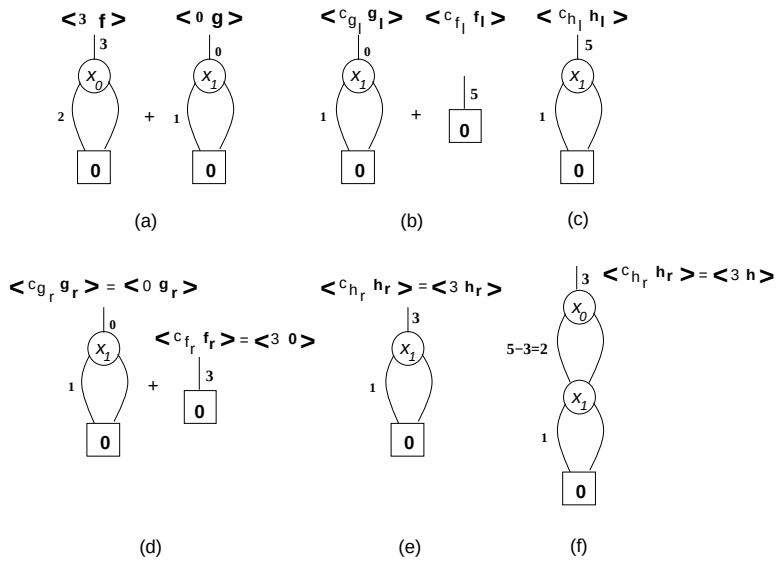

Figure 2: Example of the $apply(\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, +)$ operation.

2.2.1 Complexity Analysis

The time complexity of operations on OBDDs is $O(|\,\mathbf{f}\,| \cdot |\,\mathbf{g}\,|)$, where $|\,\mathbf{f}\,|$ and $|\,\mathbf{g}\,|$ denote the number of nodes of OBDDs $\mathbf{f}$ and $\mathbf{g}$, respectively. However, the time complexity of operations on EVBDDs is **not** $O(|\,\langle c_f, \mathbf{f}\rangle\,| \cdot |\,\langle c_g, \mathbf{g}\rangle\,|)$, where $|\,\langle c_f, \mathbf{f}\rangle\,|$ and $|\,\langle c_g, \mathbf{g}\rangle\,|$ denote the number of nodes in EVBDDs $\langle c_f, \mathbf{f}\rangle$ and $\langle c_g, \mathbf{g}\rangle$. This is because for an internal node $\mathbf{v}$ of $\langle c_f, \mathbf{f}\rangle$ or $\langle c_g, \mathbf{g}\rangle$, *apply* may generate more than one $\langle c_v, \mathbf{v}\rangle$ (lines 4, 5, 13, and 14).

Definition 2.4 Given an EVBDD $\langle c_f, \mathbf{f}\rangle$ with variable ordering $x_0 < \ldots < x_{n-1}$ and a node $\mathbf{v}$ of $\mathbf{f}$ with variable x_i, the *domain* of $\mathbf{v}$ *with respect to eval* ($D_{\mathbf{v}}^{eval}$), and the *cardinality* of $\mathbf{v}$ ($\|\,\mathbf{v}\,\|$) are defined as follows.

$$
\begin{aligned}
D_{\mathbf{v}}^{eval} &= \{c_u \mid \langle c_u, \mathbf{u}\rangle = eval(\langle c_f, \mathbf{f}\rangle, \langle b_0, \ldots, b_{i-1}\rangle) \\
&\quad \text{where} \quad \mathbf{u} = \mathbf{v}, \forall \langle b_0, \ldots, b_{i-1}\rangle \in B^i\}, \\
\|\,\mathbf{v}\,\| &= |\,D_{\mathbf{v}}^{eval}\,|.
\end{aligned}
$$

The *cardinality* of $\langle c_f, \mathbf{f}\rangle$, denoted by $\|\,\langle c_f, \mathbf{f}\rangle\,\|$, is given by $\|\,\langle c_f, \mathbf{f}\rangle\,\| = \sum_{\mathbf{v}\in\mathbf{f}} \|\,\mathbf{v}\,\|$. ■

Note that $\|\,\langle c_f, \mathbf{f}\rangle\,\|$ is the number of possible $\langle c, \mathbf{v}\rangle$'s which may be generated from $\langle c_f, \mathbf{f}\rangle$ by *apply*.

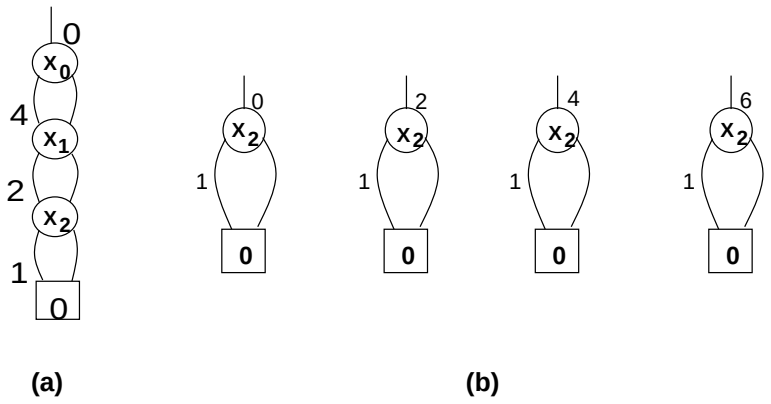

Figure 3: The $\langle c_v, \mathbf{v} \rangle$'s of node $\mathbf{v_2}$.

Example 2.3 Let $\langle 0, \mathbf{x_0} \rangle$ be the EVBDD in Fig. 3(a) and $\mathbf{v_i}$ be the node associated with variable $\mathbf{x_i}$. Then, $\| \mathbf{v_0} \| = 1$, $\| \mathbf{v_1} \| = 2$, $\| \mathbf{v_2} \| = 4$, $\| \mathbf{0} \| = 8$, and $\| \langle 0, \mathbf{x_0} \rangle \| = 15$. The $\langle c_v, \mathbf{v} \rangle$'s of node $\mathbf{v_2}$ are shown in Fig. 3(b). ■

Since EVBDDs are acyclic directed graphs and there is no backtracking in *apply*, the time complexity of *apply* is $O(\| \langle c_f, \mathbf{f} \rangle \| \cdot \| \langle c_g, \mathbf{g} \rangle \|)$ [17]. In many practical applications, the number of nodes in an EVBDD may be small, but its cardinality can be very large. However, there are some important properties of certain operators that can be exploited in the EVBDD representation that result in significant reduction in the computation complexity. These are described below.

2.2.2 The Additive Property

The EVBDD representation enjoys a distinct feature, called *additive* property, which is not seen in the OBDD representation.

Definition 2.5 An operator *op* applied to $\langle c_f, \mathbf{f} \rangle$ and $\langle c_g, \mathbf{g} \rangle$ is said to satisfy the *additive property* if $(c_f + f)\ op\ (c_g + g) = (c_f\ op\ c_g) + (f\ op\ g)$. ■

Examples of operations that satisfy the *additive* property are $(c_f + f) + (c_g + g)$, $(c_f + f) - (c_g + g)$, $(c_f + f) \times (c + 0)$, and $(c_f + f) << (c + 0)$, where $<<$ is a left shift operator as in C programming language (i.e., $(c_f + f) \times 2^c$).

To see how the *additive* property results in reduction in computation, consider the operation $(c_f + f) - (c_g + g) = (c_f - c_g) + (f - g)$. Because the values c_f and c_g can be separated

from the functions f and g, the key for this entry in *comp_table* is $\langle\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, -\rangle$. After the computation of $\langle\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, -\rangle$, which results in $\langle c_h, \mathbf{h}\rangle$, we add $c_f - c_g$ to c_h to obtain the complete result of $\langle\langle c_f, \mathbf{f}\rangle, \langle c_g, \mathbf{g}\rangle, -\rangle$. Hence, every operation $\langle\langle c'_f, \mathbf{f}\rangle, \langle c'_g, \mathbf{g}\rangle, -\rangle$ can utilize the result of computing $\langle\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, -\rangle$. For operators satisfying the additive property, the time complexity of *apply* is $O(|\langle c_f, \mathbf{f}\rangle| \cdot |\langle c_g, \mathbf{g}\rangle|)$ (as opposed to $O(\|\langle c_f, \mathbf{f}\rangle\| \cdot \|\langle c_g, \mathbf{g}\rangle\|)$).

To take advantage of the *additive* property, the following lines are inserted between lines 1 and 2 of *apply*

$$\begin{array}{ll} 1.1 & c_{fg} = c_f \; op \; c_g, \\ 1.2 & c_f = c_g = 0, \end{array}$$

and lines 2, 19, and 22 of *apply* are replaced by

```
2     if (comp_table_lookup(⟨c_f, f⟩, ⟨c_g, g⟩, op, ⟨c_h, h⟩))
            return (⟨c_h + c_fg, h⟩);
19    if (⟨c_hl, h_l⟩ == ⟨c_hr, h_r⟩) return (⟨c_hl + c_fg, h_l⟩);
22    return (⟨c_hr + c_fg, h⟩);
```

For multiplication by a constant, e.g., $(c_f + f) \times c$, and left-shift by a constant, e.g., $(c_f + f) << c$, further simplification is possible. The following pseudo code $times\_c(\langle c_f, \mathbf{f}\rangle, c)$ performs operation $(c_f + f) \times c$ with time complexity $O(|\mathbf{f}|)$. Note that the new edge value $value(\mathbf{f}) \times c$ is computed in line 5 instead of propagating it downward to the next level in line 3 (cf. line 4 or 13 of *apply*).

```
times_c(⟨c_f, f⟩, c)
{
1     if (f == 0) return ⟨c_f × c, 0⟩;
2     if (comp_table_lookup(⟨0, f⟩, c, times_c, ⟨0, h⟩))
            return ⟨c_f × c, h⟩;
3     ⟨c_hl, h_l⟩ = times_c(⟨0, child_l(f)⟩, c);     /* c_hl = 0 */
4     ⟨c_hr, h_r⟩ = times_c(⟨0, child_r(f)⟩, c);     /* c_hr = 0 */
5     h = find_or_add(variable(f), h_l, h_r, value(f) × c);
6     comp_table_insert(⟨0, f⟩, c, times_c, ⟨0, h⟩);
7     return ⟨c_f × c, h⟩;
}
```

An important application of this class of operators is to interpret a vector of Boolean functions as an integer function: $2^{m-1} f_0 + \ldots + 2^0 f_{m-1}$.

2.2.3 The Bounding Property

Before defining this property, we present a type of computation sharing that can be exploited in the case of relational operations. Consider $\leq$ as an example. Let $\langle c_f, \mathbf{f}\rangle \leq_\forall \langle c_g, \mathbf{g}\rangle$ indicate that $\langle c_f, \mathbf{f}\rangle \leq \langle c_g, \mathbf{g}\rangle$ holds for all input patterns. It can be easily shown that:

$$\langle 0, \mathbf{f} \rangle \leq_\forall \langle 0, \mathbf{g} \rangle \text{ and } (c_f - c_g) \leq 0 \;\Rightarrow\; \langle c_f, \mathbf{f} \rangle \leq_\forall \langle c_g, \mathbf{g} \rangle.$$

Let $m = -max(\langle 0, \mathbf{f} \rangle - \langle 0, \mathbf{g} \rangle)$. Then,

$$\langle 0, \mathbf{f} \rangle \leq_\forall \langle 0, \mathbf{g} \rangle \text{ and } (c_f - c_g) \leq m \;\Rightarrow\; \langle c_f, \mathbf{f} \rangle \leq_\forall \langle c_g, \mathbf{g} \rangle.$$

Based on the above implication, $\langle c_f, \mathbf{f} \rangle \leq \langle c_g, \mathbf{g} \rangle$ can be replaced by two operations: $\langle c_f, \mathbf{f} \rangle - \langle c_g, \mathbf{g} \rangle = \langle c_h, \mathbf{h} \rangle$ and $\langle c_h, \mathbf{h} \rangle \leq \langle 0, \mathbf{0} \rangle$. We store the maximum and minimum function values with each EVBDD node and include the following terminal cases:

if $(c_h + max(\mathbf{h})) \leq 0$ return $\langle 1, \mathbf{0} \rangle$, and
if $(c_h + min(\mathbf{h})) > 0$ return $\langle 0, \mathbf{0} \rangle$.

Storing the maximum and minimum values in each node is particularly useful when EVBDDs are used for solving combinatorial optimization problems using branch and bound techniques [16].

Definition 2.6 An operator *op* applied to $\langle c_f, \mathbf{f} \rangle$ and $\langle c_g, \mathbf{g} \rangle$ is said to satisfy the *bounding property* if $((c_f + m(f))\ op\ (c_g + m(g)) = 0,\ 1,\ (c_f + f),$ or $(c_g + g)$, where $m(f)$ and $m(g)$ denote the maximum or the minimum of f and g. ■

Definition 2.6 implies that the result of an operation that satisfies the *bounding property* can be immediately determined from $m(f)$ and $m(g)$. As an example, the following pseudo code $leq0(\langle c_f, \mathbf{f} \rangle)$, performs operation $(c_f + f) \leq 0$:

```
leq0(⟨c_f, f⟩)
{
1    if ((c_f + max(f)) ≤ 0) return(⟨1, 0⟩);
2    if ((c_f + min(f)) > 0) return(⟨0, 0⟩);
3    if (comp_table_lookup(⟨c_f, f⟩, leq0, ans))
         return(ans);
4    ⟨c_hl, h_l⟩ = leq0(⟨c_f + value(f), child_l(f)⟩);
5    ⟨c_hr, h_r⟩ = leq0(⟨c_f, child_r(f)⟩);
6    if (⟨c_hl, h_l⟩ == ⟨c_hr, h_r⟩) return (⟨c_hl, h_l⟩);
7    h = find_or_add(variable(f), h_l, h_r, c_hl − c_hr);
8    comp_table_insert(⟨c_f, f⟩, leq0, ⟨c_hr, h⟩);
9    return (⟨c_hr, h⟩);
}
```

2.2.4 The Domain-Reducing Property

In $\langle c_f, \mathbf{f}\rangle$ *op* $\langle c_g, \mathbf{g}\rangle$, where *op* satisfies the additive property, exactly one $\langle 0, \mathbf{v}\rangle$ pair is generated for each node $\mathbf{v}$ of $\mathbf{f}$ and $\mathbf{g}$. Thus, the 'effective' domain of each node becomes $\{0\}$. There are other operators which have similar effect on reducing the domain of EVBDD nodes.

Definition 2.7 Given an EVBDD $\langle c_f, \mathbf{f}\rangle$, the *domain of a node* $\mathbf{v}$ *of* $\mathbf{f}$ *with respect to an operator op* is defined as:

$D^{op}_{\mathbf{v}} = \{c_v \mid \langle c_v, \mathbf{v}\rangle$'s are the pairs that need to be generated with respect to *op*$\}$. ■

Definition 2.8 An operator *op* applied to $\langle c_f, \mathbf{f}\rangle$ and $\langle c_g, \mathbf{g}\rangle$ is said to satisfy the *domain-reducing property* if there exist some node $\mathbf{v}$ of $\mathbf{f}$ or $\mathbf{g}$ such that $\mathcal{D}^{op}_{\mathbf{v}} \subset D^{eval}_{\mathbf{v}}$. ■

An example of this is the following:

$$(c_f + f)\ mod\ c = ((c_f\ mod\ c) + f)\ mod\ c.$$

The domain of a node $\mathbf{v}$ of $\mathbf{f}$ is $D^{mod}_{\mathbf{v}} = D^{eval}_{\mathbf{v}} \cap \{0, \ldots, c-1\}$. In this case, $\langle c_f + kc, \mathbf{f}\rangle$ can share the computation result of $\langle c_f, \mathbf{f}\rangle$ for any integer k. When c is small, the savings in computation is large; when c is large, then the following check (using the boundary property) can be used to increase the sharing of computations.

$$\text{if } ((c_f + max(\mathbf{f})) < c \ \&\&\ (c_f + min(\mathbf{f})) \geq 0) \ \text{ then } \ \langle c_f, \mathbf{f}\rangle.$$

Another example is integer division by a constant. If c_f and c are positive integers, then $(c_f + f)/c = (c_f/c) + ((c_f\ mod\ c) + f)/c$. In fact, integer division operator by a constant satisfies the three properties: (c_f/c) satisfies the additive property, $(c_f\ mod\ c)$ satisfies the domain-reducing property, and

$$\text{if } ((c_f + max(f)) < c \ \&\&\ c_f + min(f) \geq 0) \ \text{ then } \ 0,$$

satisfies the bounding property.

2.2.5 Integer Multiplication

Unfortunately, the EVBDD representation of the multiplication function still requires exponential number of nodes. One way to alleviate this problem is to perform an input variable transformation as illustrated below.

Example 2.4 *To represent xy where x is a 4-bit and y is a 2-bit unsigned integers, we transform xy as follows:*

$$\begin{aligned} \text{xy} &= (8x_0 + 4x_1 + 2x_2 + x_3)(2y_1 + y_0) \\ &= 16x_0y_1 + 8x_0y_0 + 8x_1y_1 + 4x_1y_0 \\ &\quad +4x_2y_1 + 2x_2y_0 + 2x_3y_1 + x_3y_0 \\ &= 16w_0 + 8w_1 + 8w_2 + 4w_3 + 4w_4 \\ &\quad +2w_5 + 2w_6 + w_7 \end{aligned}$$

where $x_0y1, \ldots, x_3y_0$ *are replaced by new variables* $w_0, \ldots, w_7$. ■

By using the above method, an m-bit $\times$ n-bit integer multiplication can be represented in EVBDD form using $m \times n$ nodes.

2.2.6 Some Remarks

Srinivasan et al. [21] proposed an extension of OBDDs to Multi-valued Decision Diagrams (MDDs). In MDDs, a nonterminal node can have more than two children and a terminal node assumes integer values. All operations are carried out through the CASE operator, which although works for arbitrary discrete functions, cannot directly perform arithmetic operations.

Clarke et al. [8, 9] recently proposed another extension of OBDDs, called *Multi-Terminal Binary Decision Diagram* (MTBDD). This extension is the same as flattened EVBDDs [18]. In general, for functions where the number of distinct terminal values is large, an MTBDD (or flattened EVBDD) will require larger number of nodes than an EVBDD. However, for functions where the number of distinct terminal values is small, an MTBDD may require less storage space depending on the number of nodes in the corresponding graphs.

An EVBDD requires $n + 1$ nodes to represent $2^{n-1}x_0 + \ldots + 2^0x_{n-1}$, while an MTBDD requires $2^{n+1} - 1$ nodes to represent the same function. When there are only two different terminal nodes (e.g., 0 and 1), EVBDDs, MTBDDs, and OBDDs are equivalent in terms of the number of nodes and the topology of the graph [17]. In this case, an EVBDD will require more space to represent the the edge-values.

The worst case time complexity for performing operations on EVBDDs is the same as that for MTBDDs. However, due to the properties stated above, many operations on EVBDDs are much more efficient then corresponding operations on MTBDDs.

2.3 Representing Boolean Functions

By using integers 0 and 1 to represent Boolean values *false* and *true*, Boolean operations can be implemented through arithmetic operations as shown below:

$$\begin{aligned} x \wedge y &= xy, \\ x \vee y &= x + y - xy, \\ x \oplus y &= x + y - 2xy, \\ \overline{x} &= 1 - x. \end{aligned}$$

Thus, Boolean functions are a special case of integer functions, and OBDDs are a special case of EVBDDs.

Example 2.5 The EVBDDs for the *sum* and *carry* functions of a full adder are shown in Fig. 4. By using the above equations, *sum* and *carry* can be expressed as follows.

$$\begin{aligned} sum &= x + y + z - 2xy - 2yz - 2zx + 4xyz, \\ carry &= xy + yz + zx - 2xyz. \end{aligned}$$ ■

A full adder represented by arithmetic functions may seem more complicated than when it is represented by Boolean functions. However, the above equations are only for converting from Boolean functions to arithmetic functions. Procedure *apply* is capable of directly performing Boolean operations. For example, Boolean disjunction is carried out through $apply(\langle c_f, \mathbf{f}\rangle, \langle c_g, \mathbf{g}\rangle, \vee)$ with the following terminal cases:

1.1 if $(\langle c_f, \mathbf{f}\rangle == \langle 1, \mathbf{0}\rangle \parallel \langle c_g, \mathbf{g}\rangle == \langle 1, \mathbf{0}\rangle)$
return$(\langle 1, \mathbf{0}\rangle)$;
1.2 if $(\langle c_f, \mathbf{f}\rangle == \langle 0, \mathbf{0}\rangle \parallel \langle c_f, \mathbf{f}\rangle == \langle c_g, \mathbf{g}\rangle)$
return$(\langle c_g, \mathbf{g}\rangle)$;
1.3 if $(\langle c_g, \mathbf{g}\rangle == \langle 0, \mathbf{0}\rangle)$ return$(\langle c_f, \mathbf{f}\rangle)$;

Furthermore, when a Boolean function is represented by an EVBDD, it requires the same number of nonterminal nodes and nearly the same topology as when it is represented by an OBDD. These properties are summarized in the following algorithm and lemmas.

Algorithm A: *To convert a Boolean function from on* OBDD *to an* EVBDD *representation.*

1. Convert terminal node **0** to $\langle 0, \mathbf{0}\rangle$ and **1** to $\langle 1, \mathbf{0}\rangle$.

2. For each nonterminal node $\langle x_i, \mathbf{l}, \mathbf{r}\rangle$ in OBDD such that **l** and **r** have been converted to EVBDDs as $\langle c_l, \mathbf{l}'\rangle$ and $\langle c_r, \mathbf{r}'\rangle$, apply the following conversion rules:

   (a) $\langle x_i, \langle 0, \mathbf{l}'\rangle, \langle 0, \mathbf{r}'\rangle\rangle \Rightarrow \langle 0, \langle x_i, \mathbf{l}', \mathbf{r}', 0\rangle\rangle$,

   (b) $\langle x_i, \langle 0, \mathbf{l}'\rangle, \langle 1, \mathbf{r}'\rangle\rangle \Rightarrow \langle 1, \langle x_i, \mathbf{l}', \mathbf{r}', -1\rangle\rangle$,

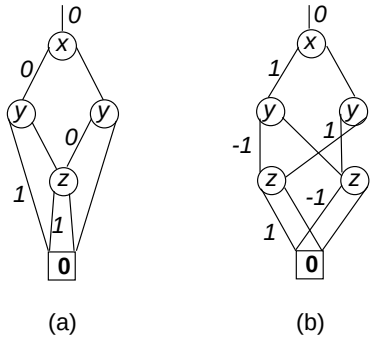

Figure 4: A full-adder represented in EVBDDs: (a) *carry* (b) *sum*.

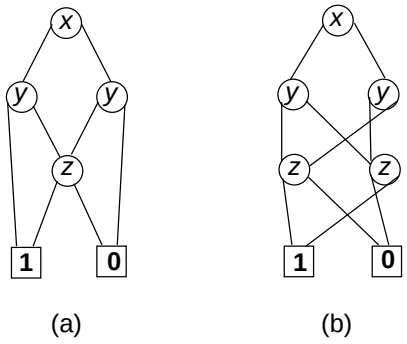

Figure 5: A full-adder represented in OBDDs: (a) *carry* (b) *sum*.

(c) $\langle x_i, \langle 1, \mathbf{l}' \rangle, \langle 0, \mathbf{r}' \rangle \rangle \Rightarrow \langle 0, \langle x_i, \mathbf{l}', \mathbf{r}', 1 \rangle \rangle$,

(d) $\langle x_i, \langle 1, \mathbf{l}' \rangle, \langle 1, \mathbf{r}' \rangle \rangle \Rightarrow \langle 1, \langle x_i, \mathbf{l}', \mathbf{r}', 0 \rangle \rangle$.

Example 2.6 Fig. 5 shows the OBDD representation of *carry* and *sum*. After Algorithm A, they will be converted to the EVBDDs in Fig. 4 ■

Lemma 2.2 Algorithm A has the following properties.

Algorithm A converts an OBDD $\mathbf{v}$ to either $\langle 0, \mathbf{v}' \rangle$ EVBDD or $\langle 1, \mathbf{v}' \rangle$ EVBDD.

1. **Algorithm A** will neither add nor delete any nonterminal node or edge.
2. **Algorithm A** preserves functionality. That is, given an OBDD $\mathbf{v}$, if the application of Algorithm A on $\mathbf{v}$ results in an EVBDD $\langle c, \mathbf{v}' \rangle$, then $\mathbf{v}$ and $\langle c, \mathbf{v}' \rangle$ denote the same function. ■

Theorem 2.1 Given a Boolean function represented by an OBDD $\mathbf{v}$ and an EVBDD $\langle c, \mathbf{v}' \rangle$, then $\mathbf{v}$ and $\mathbf{v}'$ have the same topology except that the terminal node **1** is absent from the EVBDD **v'** and the edges connected to it are redirected to the terminal node **0**. ■

Lemma 2.3 When EVBDDs are used to represent Boolean functions, exactly one of $\langle 0, \mathbf{v} \rangle$ or $\langle 1, \mathbf{v} \rangle$ can be generated during the process of *apply* (lines 4, 5, 9, 13, 14, and 16), where $\mathbf{v}$ is a nonterminal node. ■

Theorem 2.2 Given two OBDDs $\mathbf{f}$ and $\mathbf{g}$ and the corresponding EVBDDs $\langle c_f, \mathbf{f}' \rangle$ and $\langle c_g, \mathbf{g}' \rangle$, the time complexity of Boolean operations on EVBDDs (using apply) is $O(|\mathbf{f}| \cdot |\mathbf{g}|)$. ■

Based on the above theorem, we can use EVBDDs to replace OBDDs for representing Boolean functions with the following overhead:

1. An integer representing the dangling edge for each function (graph),
2. An integer representing the left edge value for each nonterminal node, and
3. One addition and one subtraction for each call of *apply* operation (lines 4 and 20).

3 Formal Verification

Formal verification requires showing the equivalence between a specification of the intended behavior and a description of the implemented design. Based on how circuit behavior is modeled, many approaches have been proposed: symbolic-simulation based such as [5], state-machine based such as [7], function based such as [2], calculus based such as [20], and logic based such as [1, 11].

When using OBDDs or EVBDDs for logic verification, if both specification and implementation are Boolean expressions, then the correctness can only be proved up to the logic level. On the other hand, if the specification is an arithmetic function while the implementation is a set of Boolean expressions, then the equivalence can be demonstrated up to the arithmetic level. Thus, EVBDDs provide two advantages over OBDDs. First, they allow equivalence checking between Boolean functions and arithmetic functions. Second, they handle hierarchical designs, that is, the implementation of a design can be described using previously verified components rather than having to flatten the design down to the gate level.

In this section, we first present a simple example of how to use EVBDDs to verify the functional behavior of circuit designs and then describe our verification paradigm for proving data

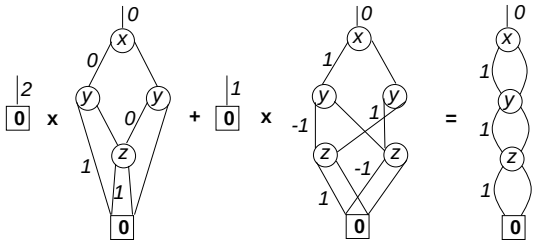

Figure 6: EVBDD expression: $2 \times carry + sum$.

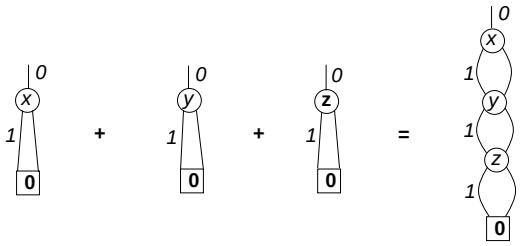

Figure 7: EVBDD expression: $x + y + z$.

paths. In order to verify control paths and do hierarchical verification, we extend EVBDDs to *structured* EVBDDs. Finally, the input variable ordering strategy for logic verification will be discussed. [1]

Example 3.1 We prove that $carry(x, y, z)$ and $sum(x, y, z)$ implement the full adder $x + y + z$. That is, with the interpretation of $\langle carry, sum \rangle$ as a 2-bit integer, we show $2 \times carry + sum = x + y + z$. Given a gate-level (Boolean) description of a full adder, it is easy to construct the EVBDD representation of the *carry* and *sum* functions as shown in Fig. 4. Carrying out the expression $2 \times carry + sum$ results in the rightmost EVBDD shown in Fig. 6. On the other hand, the specification of the arithmetic behavior of the full adder, $x + y + z$, represented in EVBDDs is shown in Fig. 7. The equivalence between $2 \times carry + sum$ and $x + y + z$ can then be checked by comparing the two rightmost EVBDDs in Figures 6 and 7.■

As shown in the above example, the implementation of a design is described by Boolean functions while its behavioral specification is described as an arithmetic function. The equiv-

[1] The experimental results in this section were generated on a Sun 3/200 with 8 MB of memory.

alence checking between two different levels of abstraction is carried out by using one representation – EVBDD.

3.1 The Verification Paradigm

In this section, we show how EVBDDs can be used to perform functional verification.

We are given the following:

1. The description of an implementation:

$$imp(x_{11}, \ldots, x_{nk}) = \langle g_1(x_{11}, \ldots, x_{nk}), \ldots, g_m(x_{11}, \ldots, x_{nk}) \rangle,$$

   where x_{ij}'s are Boolean variables and g_i's are Boolean functions.

2. The interpretation of the input variables x_{ij}'s:

$$\begin{aligned} X_1 &= f_1(x_{11}, \ldots, x_{1j}) \text{ (for a } j\text{-bit integer)}, \\ &\vdots \\ X_n &= f_n(x_{n1}, \ldots, x_{nk}) \text{ (for a } k\text{-bit integer)}, \end{aligned}$$

   where $X_i = f_i(x_{i1}, \ldots, x_{ip})$ describes how variables $\langle x_{i1}, \ldots, x_{ip} \rangle$ should be interpreted as a p-bit integer through function f_i. Thus, X_i is an integer variable and f_i specifies the number system used. A number system may be unsigned, two's complement, one's complement, sign-magnitude, or residue. For example, if X_i is an unsigned integer, then $f_i(x_{i1}, \ldots, x_{ip}) = 2^{p-1}x_{i1} + \ldots + 2^0 x_{ip}$.

3. The interpretation of the output variables g_i's: $G = g(g_1, \ldots, g_m)$. Again, g is a function representing a number system.

4. The description of a specification:

$$spec(X_1, \ldots, X_n) = f(X_1, \ldots, X_n),$$

   where function f specifies the intended behavior of the implementation.

To show imp realizes $spec$, we show the following equivalence relation:

$$\begin{aligned} f(X_1, \ldots, X_n) &= g(g_1, \ldots, g_m) \quad \text{or} \\ f(f_1(x_{11}, \ldots, x_{1j}), \ldots, f_n(x_{n1}, \ldots, x_{nk})) &= g(g_1(x_{11}, \ldots, x_{nk}), \ldots, g_m(x_{11}, \ldots, x_{nk})). \end{aligned}$$

Using the example in the previous section, we have:

$$
\begin{aligned}
imp(x,y,z) &= \langle carry(x,y,z), sum(x,y,z)\rangle, \\
X &= x, \\
Y &= y, \\
Z &= z, \\
G &= 2carry + sum, \\
spec(X,Y,Z) &= X + Y + Z.
\end{aligned}
$$

The correctness of the full adder is verified by showing $x + y + z = 2carry(x,y,z) + sum(x,y,z)$.

The above paradigm can be reversed to result in a procedure for functional synthesis. Again, we use the full adder as an example except now the goal $imp(x,y,z)$ is not given. From the description of *spec*, we have

$$
\begin{aligned}
sum(x,y,z) &= spec \ mod \ 2, \\
carry(x,y,z) &= (spec - (spec \ mod \ 2))/2,
\end{aligned}
$$

where $spec = x+y+z$. The following sequence of *apply* operations on EVBDDs then produces the *sum* and *carry* automatically:

$$
\begin{aligned}
\langle 0, \mathbf{xy}\rangle &= apply(\langle 0, \mathbf{x}\rangle, \langle 0, \mathbf{y}\rangle, +), \\
\langle 0, \mathbf{fa}\rangle &= apply(\langle 0, \mathbf{z}\rangle, \langle 0, \mathbf{xy}\rangle, +), \\
\langle 0, \mathbf{sum}\rangle &= apply(\langle 0, \mathbf{fa}\rangle, \langle 2, \mathbf{0}\rangle, mod), \\
\langle 0, \mathbf{temp}\rangle &= apply(\langle 0, \mathbf{fa}\rangle, \langle 0, \mathbf{sum}\rangle, -), \\
\langle 0, \mathbf{carry}\rangle &= apply(\langle 0, \mathbf{temp}\rangle, \langle 2, \mathbf{0}\rangle, /).
\end{aligned}
$$

As presented in Sec. 2.2.4, operations modulo and integer division can be effectively carried out in EVBDDs. An application of the above synthesis procedure is in logic verification where the mapping between the variables is not given. For example, we can specify a 64-bit adder as '$x + y$' while the variable sets in the implementation are a's and b's. In this case, we first convert the arithmetic expression into a vector of Boolean functions and then use Boolean matching [14] to perform the equivalence checking.

Example 3.2 The design (*imp*) is a 64-bit 3-level carry lookahead adder which has 129 inputs, 65 outputs, and 420 logic gates. The intended behavior (*spec*) is specified as:

```
unsigned(65) add64(x, y, c)
    unsigned(64) x, y;
    unsigned c;
{
    return(x + y + c);
}
```

where (64) and (65) declare the number of bits. In our experimental implementation, the generation of 65 EVBDDs of *imp* (575 nodes in total) takes 1.47 seconds and the generation of one EVBDD of *spec* (129 nodes) takes 0.17 seconds. The verification process which converts 65 EVBDDs into one, performing $2^{64} \times b_0 + \ldots + 2^0 \times b_{64}$, and then compares the result with the *spec* takes 4.48 seconds. That is, it takes less than 5 seconds to show 65 Boolean expressions are really carrying out an addition. ■

3.2 Structured Edge-Valued Binary Decision Diagrams

As shown in the previous section, we can use EVBDDs to show the equivalence between Boolean expressions and arithmetic expressions. In this section, we introduce Structured EVBDDs, or SEVBDDs for short, which can be used to show the equivalence between Boolean expressions and conditional expressions. For example, the implementation of a multiplexer can be described as '$(x \wedge y) \vee (\bar{x} \wedge z)$' while the specification can be described as 'if x then y else z'. In addition to the specification of conditional statements, SEVBDDs also allow the declaration of vectors.

Definition 3.1 SEVBDDs are recursively defined as follows:

1. An EVBDD is an SEVBDD. (This is the *atomic type* of SEVBDDs.)

2. $(p \rightarrow t; e)$ is an SEVBDD if p is an SEVBDD with the $\{0, 1\}$ range, and t and e are SEVBDDs. For every input assignment b, the function denoted by $(p \rightarrow t; e)$ returns the value $t(b)$, if $p(b) = 1$; otherwise it returns $e(b)$. (This is the *conditional type* of SEVBDDs.)

3. $[f_1, \ldots, f_m]$ is an SEVBDD if $f_1, \ldots, f_m$ are SEVBDDs. For some input assignment b, $[f_1, \ldots, f_m]$ returns the vector $\langle f_1(b), \ldots, f_m(b) \rangle$. (This is the *vector type* of SEVBDDs.)

■

In the graphical representation of SEVBDDs, terminal nodes are atomic type SEVBDDs (Fig. 8 (a)). There are two types of nonterminal nodes: a *conditional node* which has three children (Fig. 8 (b)) and a *vector node* which has an indefinite number of nodes (Fig. 8 (c)).

Example 3.3 Let x, y, z, y_0, y_1, z_0, and z_1 be EVBDDs. Consider the following expressions:

1. x, $x \wedge y$, $\bar{x} \wedge z$, and

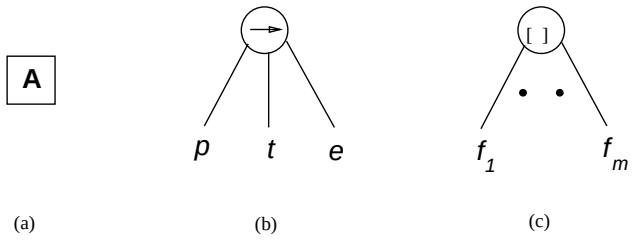

Figure 8: Graphical representation of SEVBDDs.

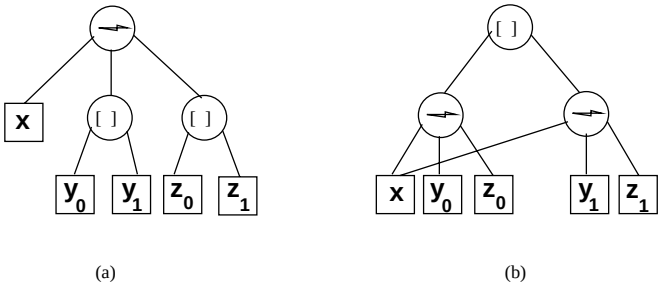

Figure 9: Examples of SEVBDDs.

2. $(x \wedge y) \vee (\bar{x} \wedge z)$;

3. $(x \rightarrow y; z)$, $(x \rightarrow x \wedge y; z)$, $(x \rightarrow y; \bar{x} \wedge z)$, $(x \rightarrow x \wedge y; \bar{x} \wedge z)$, and

4. $(x \rightarrow [y_0, y_1]; [z_0, z_1])$;

5. $[(x \rightarrow y_0; z_0), (x \rightarrow y_1; z_1)]$ and

6. $[(x \wedge y_0) \vee (\bar{x} \wedge z_0), (x \rightarrow x \wedge y_1; \bar{x} \wedge z_1)]$.

SEVBDDs in groups 1 and 2 are of a atomic type. Those in groups 3 and 4 are of a conditional type and those in groups 5 and 6 are of a vector type. Note that the SEVBDDs in groups 2 and 3 represent a 1-bit multiplexer while the SEVBDDs in groups 4, 5, and 6 represent two 1-bit multiplexers which have the same control signal x. The graphical representation of those in groups 4 and 5 are shown in Fig. 9 (a) and (b), respectively. ■

Definition 3.2 The *type graph* of an SEVBDD **f** is obtained by replacing all terminal nodes of **f** by a unique terminal node **A**. ■

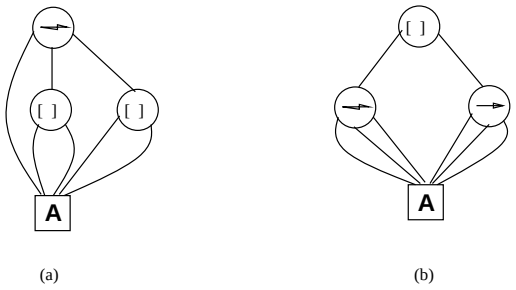

Figure 10: Examples of type graph of SEVBDDs.

Example 3.4 The type graphs of the SEVBDDs in Fig. 9 are shown in Fig. 10. ■

An SEVBDD would be a canonical representation if two SEVBDDs denote the same function if and only if they are isomorphic. This is however not true because we can have two SEVBDDs denoting the same function which have different types (e.g., Ex. 3.3). However, with proper restrictions, SEVBDDs can still have the canonical property. That is, if two SEVBDDs satisfy those conditions then they denote the same function if and only if they are isomorphic. In the following, we define two conditions such that the subset of SEVBDDs which satisfy these conditions have the canonical property.

The first condition is to be *isotypic* which is defined as follows:

Definition 3.3 Two SEVBDDs are *isotypic* if their type graphs are isomorphic. Equivalently, two SEVBDDs f and g are *isotypic* if

1. Both f and g are EVBDDs, or
2. $f = (p \rightarrow t_f; e_f)$, $g = (p \rightarrow t_g; e_g)$, t_f and t_g are isotypic, and e_f and e_g are isotypic, or
3. $f = [f_1, \ldots, f_m]$, $g = [g_1, \ldots, g_m]$ and every pair of f_i and g_i are isotypic. ■

Example 3.5 In Ex. 3.3, the SEVBDDs in groups 1 and 2 are isotypic; the SEVBDDs in group 3 are isotypic but none of them is isotypic to that of 4; SEVBDDs in groups 5 and 6 are not isotypic. ■

Note that two SEVBDDs which are isotypic but are not isomorphic, may still denote the same function. In Ex. 3.3, the SEVBDDs in group 3 are isotypic but are not isomorphic, yet they all denote the same function. Given an SEVBDD $(p \rightarrow t; e)$, for any input assignment

b such that $p(b) = 1$, the function value of $e(b)$ will not influence the result; similarly, if $p(b) = 0$, then $t(b)$ is irrelevant. Therefore, we can use operators $cofactor_1(p, t)$ and $cofactor_0(p, e)$ to transform t and e to t' and e' such that if $p(b) = 1$, then $t'(b) = t(b)$ and $e'(b) = 0$; if $p(b) = 0$, then $t'(b) = 0$ and $e'(b) = e(b)$. Consequently, we obtain a reduced form $(p \rightarrow t'; e')$ for $(p \rightarrow t; e)$. The $cofactor_1(p, t)$ operator is carried out in a similar way to the *restrict* operator in [10] except for the following differences: When $p = 0$, *restrict* returns *error* while $cofactor_1$ returns 0; *Restrict* applies to Boolean functions while $cofactor_1$ applies to arithmetic and Boolean functions.

The second condition for SEVBDDs to be canonical is for them to be *reduced*.

Definition 3.4 An SEVBDD is *reduced* if

1. It is an EVBDD, or
2. It is a conditional SEVBDD of the form $(p \rightarrow t; e)$ with $cofactor_1(p, t) = t$, $cofactor_0(p, e) = e$, and t and e are reduced, or
3. It is $[f_1, \ldots, f_m]$ and every f_i is reduced. ■

In Ex. 3.3, the SEVBDDs in groups 1 and 2 are reduced; the last SEVBDD in group 3 and the one in group 6 are also reduced.

Lemma 3.1 If two SEVBDDs f and g are isotypic and reduced, then f and g denote the same function if and only if they are isomorphic. ■

Since isotypic and reduced SEVBDDs are canonical, we need procedures for converting an SEVBDD from one form to another and/or reducing an SEVBDD. Operators $cofactor_1$ and $cofactor_0$ are used for converting from atomic (EVBDDs) to conditional form. To convert from conditional to atomic form, we use operator ite, which is nearly the same as the one described in [3] except that our ite operator is also applicable to arithmetic functions. Operator ite takes a conditional SEVBDD such as $(p \rightarrow t; e)$ (t and e are EVBDDs) as arguments and returns an EVBDD f such that $(p \rightarrow t; e)$ and f denote the same function. The following procedures conver the forms of SEVBDDs and reduce SEVBDDs. **Note:** $cofactor\_s_1$, $cofactor\_s_0$, and ite_s are SEVBDD versions of $cofactor_1$, $cofactor_0$, and ite, respectively.

```
convert(f, g) /* converting g to same form of f */
  /* assumes f and g have same number of outputs *
1   if (f is an EVBDD)
2     if (g is an EVBDD) return(g);
3     if (g == (p → t; e))
          return(ite(p, convert(f, t), convert(f, e)));
4   else if (f == (p → t; e))
5     return((p → convert(t, cofactor_s1(p, g));
                  convert(e, cofactor_s0(p, g))));
6   else    /* f = [f_1, ..., f_m] */
7     if (g == (p → t; e))
8       return(ite_s(p, convert(f, t), convert(f, e)));
9     else return([convert(f_1, g_1), ..., convert(f_m, g_m)]);
}

reduce(f)
{
1   if (f is an EVBDD) return(f);
2   else if (f == (p → t; e))
3     return(reduce(p) → reduce(cofactor_s1(p, t));
                         reduce(cofactor_s0(p, e)));
4   else return([reduce(f_1), ..., reduce(f_m)]);
}

cofactor_s1(p, t) /* cofactor_s0(p, t) is similarly defined */
{
1   if (t is an EVBDD) return(cofactor_1(p, t));
2   else if (t == (p' → t'; e'))
3     return((cofactor_s1(p, p') → cofactor_s1(p, t');
                                   cofactor_s1(p, e')));
4   else /* t = [t_1, ..., t_m] */
5     return([cofactor_s1(p, t_1), ..., cofactor_s1(p, t_m)]);
}

ite_s(p, t, e)      /* assuming t and e are isotypic */
{
1   if (p == (p_p → t_p; e_p))
      return(ite_s(ite_s(p_p, t_p, e_p), t, e));
2   if (t and e are EVBDDs) return(ite(p, t, e));
3   if (t == (p_t → t_t; e_t) && e == (p_e → t_e; e_e))
4     return((ite_s(p, p_t, p_e) → ite_s(p, t_t, t_e);
                                   ite_s(p, e_t; e_e)));
5   if (t == [t_1, ..., t_m] && e == [e_1, ..., e_m])
6     return([ite_s(p, t_1, e_1), ..., ite_s(p, t_m, e_m)]);
}
```

To show the equivalence between a specification and an implementation described in two different forms, we need to convert from one form to another. In our implementation, we use the specification as the target form and convert the implementation to the target form. This is because a specification usually has a more compact representation than an implementation. For example, a specification of '$(x \leq y \rightarrow x+y; x-y)$' where x and y are n-bit integers, requires $3n$, $2n$, and $2n$ nonterminal nodes for representing $x \leq y$, $x+y$, and $x-y$, respectively. On the other hand, a gate implementation of the above specification requires $n+1$ Boolean functions in which the i^{th} function (for generating i^{th} bit) requires at least $2i$ nonterminal nodes, and the carry function (bit) requires at least $2n$ nonterminal nodes. Thus, it requires at least $n(n+3)$ nonterminal nodes. The following two examples verify SN74L85 and SN74181 chips [22], where the first one is a 4-bit comparator and the second one is a 4-bit ALU.

Example 3.6 The implemented design (*imp*) is the SN74L85 chip [22] which is a 4-bit comparator. This chip has 11 inputs, 3 outputs and 33 gates. The specification (*spec*) of the design may be described as:

```
unsigned(3) comp4(x, y, gt, lt, eq)
    unsigned(4) x, y;
    unsigned gt, lt, eq;
{
    if (x > y) return(⟨1, 0, 0⟩);
    else if (x < y) return(⟨0, 1, 0⟩);
    else return(⟨gt, lt, eq⟩)
}
```

It takes 0.05 seconds to generate the SEVBDD of *imp* which has 39 nodes and it takes 0.02 seconds to construct the conditional SEVBDD of *spec* which has 25 nodes. The conversion from the SEVBDD of *imp* to that of *spec*, followed by the comparison, takes 0.02 seconds. ■

Example 3.7 The implementation is the SN74181 chip which is a 4-bit ALU [22]. A partial specification is given below. Note: *un_comp*, *two* and *unsigned* perform type coercion. *un_comp* results in an unsigned integer, with the most significant bit being complemented. *two* means that the result is to be a two's complement integer.

```
SN74181(M, S, A, B, Cin)
 unsigned M, Cin;
 unsigned(4) S, A, B;
{
 if (M = 0)
    if (S = 0) return((un_comp (5)) A + (− Cin));
          ⋮
    else if (S = 3) return((two(5)) − Cin);
          ⋮
 else
    if (S = 0) return((unsigned (4)) not(A));
    else if (S=1) return((unsigned(4)) not(A or B));
          ⋮
}
```

Note that we allow the interpretation of the same outputs in different number systems as well as allow different sizes in different branches of conditional statements.

The implementation SEVBDD has 765 nodes and can be generated in 0.31 seconds. The specification SEVBDD has 187 nodes and can be constructed in 0.13 seconds. And the verification process takes 0.35 seconds. ■

In addition to providing the ability to check equivalence between Boolean and arithmetic expressions and between conditional and nonconditional expressions, SEVBDDs are suitable for hierarchical verification, i.e., verification without having to flatten a component which has already been verified. In the following two examples, a 64-bit comparator and a 64-bit adder, the implementations are constructed from 4-bit comparators and 4-bit ALU's. The construction of implementation SEVBDDs are however based on the specification SEVBDDs of the 4-bit comparator and 4-bit ALU's.

Example 3.8 The design is a 64-bit comparator implemented through serial connection of 16 SN74L85s. The specification of this design is the same as the one in Example 3.6 except that the size declaration is changed from 4 to 64. Generation of implementation and specification SEVBDDs take 0.26 and 0.39 seconds respectively, and the proof takes 3.35 seconds. ■

Example 3.9 The design is a 64-bit ripple-carry adder implemented through serial connection of 16 SN74181s. The specification of this design is exactly the same as the one used in Example 3.2. Time to generate the SEVBDDs for the implementation and specification

are 2.09 and 0.16 seconds, respectively and time to verify their equivalence is 0.98 seconds. Note that generation of implementation SEVBDD takes longer time while verification takes less time than the case in Example 3.2. This is because, here, we generate 16 SEVBDDs each with the sum of 4 bits instead of 64 SEVBDDs each with the sum of 1 bit. ■

3.3 Ordering Strategy

The conditional type of SEVBDDs provides information for determining the ordering of input variables. For example, for SEVBDD $(p \rightarrow t; e)$, we assign variables occurring in p lower indices compared to those in t and e. This ordering strategy matches the suggestion (controlling variables should be put on top of OBDDs) in [4]. It is more difficult to identify controlling variables in a Boolean expression. In addition, we assign variables with larger integer coefficients lower indices compared to those with smaller integer coefficients. This ordering strategy also matches the observation in [4], and is easier to identify from arithmetic expressions than from Boolean expressions.

4 Conclusions

It was demonstrated that by associating an integer with each edge of an OBDD and giving a new meaning to each node of the OBDD, a new graphical data structure is created whose domain is that of the integer functions. The new data structure, called EVBDD, admits arithmetic operations. EVBDDs preserve the canonical property as well as the capability to cache computational results. With these two properties, we have found EVBDDs to be valuable in many applications.

Because of the compactness and canonical properties, EVBDDs have been shown to be effective for handling verification problems. Because of the additive property, EVBDDs are also useful for solving integer linear programming problems [16]. Other applications of EVBDDs include performing spectral transformation and matrix representation.

References

[1] G. V. Bochmann, "Hardware specification with temporal logic: An example," *IEEE Trans. on Computers*, 31(3):223-231, March 1982.

[2] R. T. Boute, "Representational and denotational semantics of digital systems," *IEEE Trans. on Computers*, 38(7):986-999, July 1989.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," *Proc. of the 27th Design Automation Conference*, pp. 40-45, 1990.

[4] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, C-35(8): 677-691, August 1986.

[5] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler, "COSMOS: a compiled simulator for MOS circuits," *Proc. of the 24th Design Automation Conference*, pp. 9-16, 1987.

[6] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," *Computing Surveys*, Vol. 24, No. 3, pp. 293-318, Sept. 1992.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Prog. Lang. Syst.*, 8(2), 1986.

[8] E. M. Clarke, M. Fujita, P. C. McGeer, K. L. McMillan, and J. C.-Y. Yang, "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation," *International Workshop on Logic Synthesis*, pp. 6a:1-15, May 1993.

[9] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. C.-Y. Yang, "Spectral transforms for large Boolean functions with applications to technology mapping," *Proc. of the 30th Design Automation Conference*, pp. 54-60, 1993.

[10] O. Coudert, C. Berthet, J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution," *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989.

[11] M. J. C. Gordon, "Why higher-order logic is a good formalism for specifying and verifying hardware," in G. J. Milne and P. A. Subrahmanyam, eds, *Formal Aspects of VLSI Designs*, pp. 153-177, 1986.

[12] P. L. Hammer and S. Rudeanu, *Boolean Methods in Operations Research and Related Areas*, Heidelberg, Springer Verlag, 1968.

[13] Y-T. Lai and S. Sastry (S. B. K. Vrudhula), "Edge-Valued binary decision diagrams for multi-level hierarchical verification," *Proc. of 29th Design Automation Conf.*, pp. 608-613, 1992.

[14] Y-T. Lai, S. Sastry (S. B. K. Vrudhula) and M. Pedram, "Boolean matching using binary decision diagrams with applications to logic synthesis and verification," *Proc. International Conf. on Computer Design*, pp. 452-458, 1992.

[15] Y-T. Lai, M. Pedram and S. Sastry (S. B. K. Vrudhula), "BDD based decomposition of logic functions with application to FPGA synthesis," *Proc. of 30th Design Automation Conf*, pp. 642-647, 1993.

[16] Y-T. Lai, M. Pedram and S. Sastry (S. B. K. Vrudhula), "FGILP: An Integer Linear Program Solver Based on Function Graphs," *Proc. Int. Conf. CAD*, 1993.

[17] Y-T. Lai, "Logic verification and synthesis using function graphs," Ph.D. Dissertation, Computer Engineering, Univ. of Southern Calif., December 1993.

[18] Y-T. Lai, M. Pedram and S.B.K. Vrudhula, "EVBDD-based Algorithms for Integer Linear Programming, Spectral Transformation, and Function Decomposition," *IEEE Trans. on CAD*, CAD-13(8): 959-975, August 1994.

[19] H-T. Liaw and C-S Lin, "On the OBDD-representation of general Boolean functions," *IEEE Trans. on Computers*, C-41(6): 661-664, June 1992.

[20] G. J. Milne, "CIRCAL and the representation of communication, concurrency, and time," *ACM Trans. of Programming Languages and Systems*, 7(2):270-298, April 1985.

[21] A. Srinivasan, T. Kam, S. Malik and R. Brayton, "Algorithms for Discrete Function Manipulation," *Proc. Int. Conf. CAD*, pp. 92-95, 1990.

[22] Texas Instruments, "The TTL Data Book for Design Engineers," *Texas Instruments*, 1984.

Contents

Captions of Figures

List of Tables