Edge-Valued Binary Decision Diagrams: Theory and Applications

Yung-Te Lai, Massoud Pedram
and Sarma Vrudhula

CENG Technical Report 93-31

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4458

August 10, 1993

Edge-Valued Binary-Decision Diagrams: Theory and Applications

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

August 10, 1993

Contents

List of Figures

List of Tables

Abstract

We present a new data structure called Edge-Valued Binary-Decision Diagrams (EVBDDs) for representing integer functions. EVBDDs are directed acyclic graphs constructed in the same way as Ordered Binary-Decision Diagrams (OBDDs), except that there is a dangling edge on the root node and every edge is annotated with an integer. An OBDD represents a Boolean function but an EVBDD represents an integer function. Since Boolean functions can be implemented as a special case of integer functions, EVBDD is a more powerful representation than OBDD. We present a few applications of EVBDDs: proving correctness of circuit designs, solving integer linear programming problems, computing spectral coefficients of Boolean functions, and performing multiple-output Boolean function decomposition. In each case, experimental results are provided.

1 Introduction

Ordered Binary-Decision Diagrams (OBDDs) [9] are a graphical representation of Boolean functions. With the canonical property of OBDDs, many Boolean properties such as the number of supporting variables, the unateness of variables, and the symmetry between variables can be easily detected. With the compactness property and high hit ratio for caching computational results, many Boolean operations can be effectively carried out in OBDD representation. 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.

In addition to Boolean functions, many problems defined in small, finite domains can also employ OBDD representation through binary encoding of these 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 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 through OBDD representation [10].

While OBDDs are useful for problems which can be solved through symbolic Boolean manipulation, they are not very effective for those requiring arithmetic operations in integer domain. Although we can represent integer functions by vectors of Boolean functions and perform arithmetic operations through Boolean operations on each bit, it is very time consuming. In this paper, we present a new data structure called Edge-Valued Binary-Decision Diagrams (EVBDDs) which can represent and manipulate integer functions as effectively as OBDDs do for Boolean functions.

EVBDDs are directed acyclic graphs constructed in a similar way to OBDDs. As in OBDDs, each node either represents a constant function with no children or is associated with a binary variable having two children, and there is an input variable ordering imposed in 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 [26] 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 are capable of representing 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 four applications of EVBDDs. The first application is in logic verification where the objective is to show the equivalence between a behavioral specification and an implementation. The correctness of a circuit design can only be proved up to the specification used. 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 to specify the operator '+' directly (e.g., '$x + y$'), then the correctness is up to the arithmetic addition. Since EVBDDs can represent both Boolean and arithmetic functions, the equivalence between these two functions can be proved directly up to the arithmetic behavior.

The second application is in solving integer linear programming (ILP) problems. An ILP problem is to find the maximum (or minimum) of a goal function subject to a set of linear inequality constraints. Each constraint defines a feasible subspace which can be represented as a Boolean function. The conjoining of these constraint (i.e., the conjunction of the corresponding Boolean functions) defines the overall feasible subspace. The problem is then solved by finding the maximum (or minimum) of the goal function over the feasible subspace.

The third application is in computing the spectral coefficients of a Boolean function. The main purpose of spectral methods [52] is to transform Boolean functions from Boolean domain into spectral (integer) domain so that a number of useful properties can be more easily detected. When a Boolean function is represented in Boolean domain, the function value for each minterm precisely describes the behavior of the function at that point but says nothing about the behavior of the function for any other point. In contrast, spectral representation of a Boolean function gives information which is much more global in nature. For example, for function $f(x_0, \ldots, x_{n-1}$, the spectral coefficient of $\overline{x_0} \ldots \overline{x_{n-1}}$ corresponds to the number of onset points of $f(x_0 \ldots x_{n-1})$. Since EVBDDs can represent functions in both Boolean and spectral domains, we are able to use arithmetic operations in EVBDDs to efficiently carry out the spectral transformations.

The fourth application is in the representation of multiple output Boolean functions. Clearly, we can use the OBDD representation to solve integer problems through the binary encoding of integer variables. Similarly, we can also use the EVBDD representation to perform multiple output Boolean function operations through integer interpretation of the functions (e.g., a multiple output function $f_0, \ldots, f_{m-1}$ can be represented by an integer function $2^{m-1}f_0 + \ldots + 2^0 f_{m-1}$). We will present an EVBDD-based function decomposition algorithm as an example. When this algorithm is applied to an EVBDD representing a Boolean function, it performs single-output function decomposition; when it is applied to an EVBDD representing an integer function representing a multiple-output Boolean function, it performs multiple-output function decomposition.

The remainder of this paper is organized as follows. In section 2, we define the syntax and semantics of EVBDDs, prove their canonical property, and show their relationship to Boolean functions. Four applications of EVBDDs: verifying circuit behavior, solving ILP problems, computing spectral coefficients, and performing multiple output Boolean function decomposition are presented in sections 3, 4, 5, and 6, respectively. Conclusions are given in section 7.

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 EVBDD properties which are useful in speeding up 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 labelled 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.

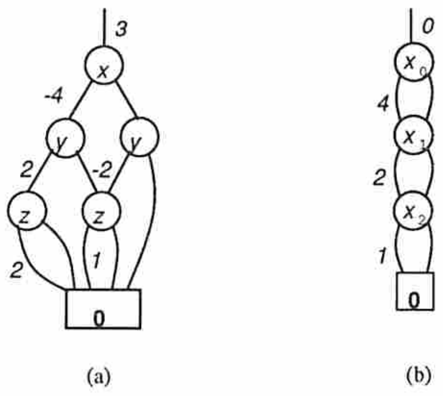


Figure 1: Two examples.

Example 2.1 Fig. 1 shows two arithmetic functions $f_0 = 3 - 4x + 4xy + xz - 2y + yz$ and $f_1 = 4x_0 + 2x_1 + x_2$ represented in EVBDDs. The second function is derived as follows:

$$\begin{aligned}
f_1 &= 0 + f_{x_0}, && \\
f_{x_0} &= x_0(4 + f_{x_1}) + (1 - x_0)f_{x_1} &&= 4x_0 + 2x_1 + x_2, \\
f_{x_1} &= x_1(2 + f_{x_2}) + (1 - x_1)f_{x_2} &&= 2x_1 + x_2, \\
f_{x_2} &= x_2(1 + 0) + (1 - x_2)0 &&= x_2.
\end{aligned}$$

□

Note that an EVBDD requires only n nonterminal nodes to represent an n variable linear function, for example, Fig. 1 (b) is a linear function which can also be interpreted as a 3-bit integer.

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:

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

From the above definition, function values in an EVBDD representation are obtained by summing edge values (right edge values are always set to 0) along the path associated with the input assignment. For example, in Fig. 1 (a), the function value of $x = 1, y = 0$ and $z = 1$ is $3 + (-4) + 0 + 1 = 0$, and the function value of $x_2 = 1, x_1 = 0$ and $x_0 = 1$ in Fig. 1 (b) is $0 + 4 + 0 + 1 = 5$.

EVBDD is a canonical representation of functions from $\{0, 1\}^n$ to the set of integers. This is stated in the following lemma.

Lemma 2.1 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.

Proof: Sufficiency: If $c_f = c_g$ and $\mathbf{f}$ and $\mathbf{g}$ are isomorphic, then $\forall b\ eval(\langle c_f, \mathbf{f}\rangle, b) = eval(\langle c_g, \mathbf{g}\rangle, b)$ directly follows from the definitions of isomorphism and *eval.*
Necessity: If $c_f \neq c_g$ then let $b = \langle 0, \ldots, 0\rangle$ be the input assignment to $\mathbf{f}$ and $\mathbf{g}$ (e.g., Fig. 2 (a)). We have the following

$$eval(\langle c_f, \mathbf{f}\rangle, b) = c_f \neq c_g = eval(\langle c_g, \mathbf{g}\rangle, b).$$

Thus, we focus on the latter condition on $\mathbf{f}$ and $\mathbf{g}$. We want to show if $\mathbf{f}$ and $\mathbf{g}$ are not isomorphic, then $\exists b \in B^n$ such that $eval(\langle 0, \mathbf{f}\rangle, b) \neq eval(\langle 0, \mathbf{g}\rangle, b)$. Without loss of generality, we assume $index(variable(\mathbf{f})) \leq index(variable(\mathbf{g}))$. Let $k = n - index(variable(\mathbf{f}))$, we prove the lemma by induction on k.
Base: When $k = 0$, $\mathbf{f}$ is a terminal node and so is $\mathbf{g}$. Furthermore, $\mathbf{f} = \mathbf{g} = \mathbf{0}$. Thus, $\mathbf{f}$ and $\mathbf{g}$ are isomorphic.
Induction hypothesis: Assume it is true for $n - index(variable(\mathbf{f})) < k$.
Induction: We show that the hypothesis holds for $n - index(variable\mathbf{f}) = k$.
Let $\mathbf{f} = \langle x_{n-k}, \mathbf{f_l}, \mathbf{f_g}, v_f\rangle$.

case 1: $n - index(variable(\mathbf{g})) = k$, that is, $\mathbf{g} = \langle x_{n-k}, \mathbf{g_l}, \mathbf{g_r}, v_g\rangle$.
If $v_f \neq v_g$, let $b = \langle \ldots, 0, 1, 0, \ldots\rangle$, that is, $b_{n-k} = 1$ and $b_i = 0, \forall i \neq n-k$, then $eval(\langle 0, \mathbf{f}\rangle, b) = v_f \neq v_g = eval(\langle 0, \mathbf{g}\rangle, b)$ (e.g., Fig. 2 (b)). If $v_f = v_g$, then either $\mathbf{f_l}$ and $\mathbf{g_l}$ are nonisomorphic, or $\mathbf{f_r}$ and $\mathbf{g_r}$ are nonisomorphic.

subcase 1: If $\mathbf{f_l}$ and $\mathbf{g_l}$ are not isomorphic, then from $n - index(variable(\mathbf{f_l})) < k$, $n - index(variable(\mathbf{g_l})) < k$, and induction hypothesis, there exists $b = \langle b_0, \ldots, b_{n-1}\rangle$ such that $eval(\langle 0, \mathbf{f_l}\rangle, b) \neq eval(\langle 0, \mathbf{g_l}\rangle, b)$. Let $b' = \langle b'_0, \ldots, b'_{n-1}\rangle$ such that $b'_{n-k} = 1$ and $b'_i = b_i$ for $i \neq n-k$, then $eval(\langle 0, \mathbf{f}\rangle, b') = eval(\langle v_f, \mathbf{f_l}\rangle, b') \neq eval(\langle v_g, \mathbf{g_l}\rangle, b') = eval(\langle 0, \mathbf{g}\rangle, b')$ (e.g., Fig. 2 (c)).

subcase 2: Otherwise, $\mathbf{f_r}$ and $\mathbf{g_r}$ are not isomorphic, then by similar arguments, letting $b'_{n-k} = 0$ and $b'_i = b_i, \forall i \neq n-k$ will result in $eval(\langle 0, \mathbf{f}\rangle, b') \neq eval(\langle 0, \mathbf{g}\rangle, b')$.

case 2: $n - index(variable(\mathbf{g})) < k$.
By definition of reduced EVBDD, we cannot have both $(v_f = 0)$ and ($\mathbf{f_l}$ and $\mathbf{f_r}$ are isomorphic). If $v_f \neq 0$, let $b_{n-k} = 1$ and $b_i = 0$ for $i \neq n-k$, then $eval(\langle 0, \mathbf{f}\rangle, b) = v_f \neq 0 = eval(\langle 0, \mathbf{g}\rangle, b)$ (e.g., Fig. 2 (d)). (Since $\mathbf{g}$ is independent of the first $n-k$ bits.) Otherwise, $\mathbf{f_l}$ and $\mathbf{f_r}$ are not isomorphic and at least one of them is not isomorphic to $\mathbf{g}$. If $\mathbf{f_l}$ ($\mathbf{f_r}$) and $\mathbf{g}$ are not isomorphic, then by induction hypothesis, there exist a b such that $eval(\langle 0, \mathbf{f_l}(\mathbf{f_r})\rangle, b) \neq eval(\langle 0, \mathbf{g}\rangle, b)$. Again, let $b'_{n-k} = 1(0)$ if $\mathbf{f_l}(\mathbf{f_r})$ is not isomorphic to $\mathbf{g}$, and $b'_i = b_i$ for $i \neq n-k$, $eval(\langle 0, \mathbf{f}\rangle, b') \neq eval(\langle 0, \mathbf{g}\rangle, b')$.

□

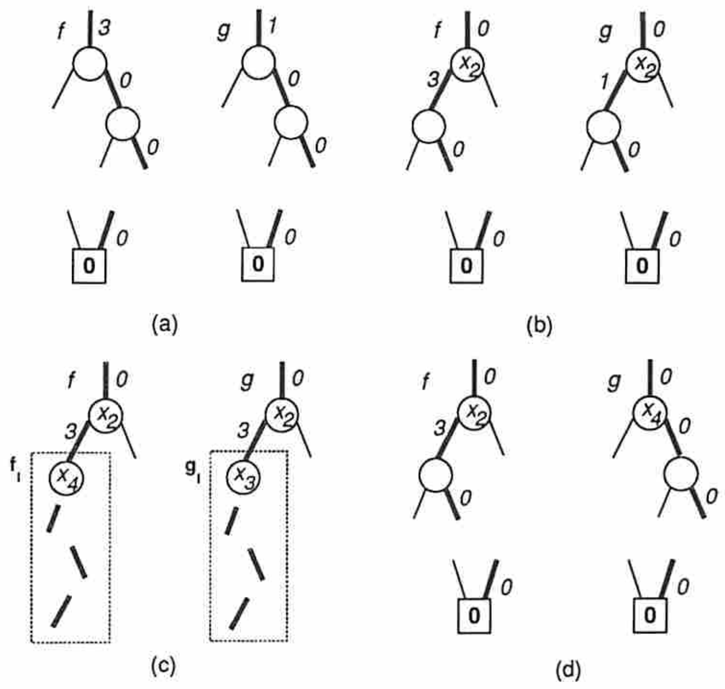


Figure 2: Examples for proving canonical property.

2.2 Operations

The following algorithm describes the function (EVBDD) *apply* which 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 $c_h + h \equiv (c_f + f)\ op\ (c_g + g)$ where *op* can be any operator which is closed over the integers.

In algorithm *apply*, a terminal case (line 1) occurs when the result can be computed directly. For example, $op = \times$ and $\langle c_f, \mathbf{f}\rangle = \langle 1, \mathbf{0}\rangle$ is a terminal case because $\langle 1, \mathbf{0}\rangle = 1 + 0 = 1$, $\langle c_g, \mathbf{g}\rangle = c_g + g$, 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.

A *comp_table* storing previously computed results is used to achieve computation efficiency. An entry of *comp_table* has the form $\langle f, g, op, h\rangle$ which stands for $f\ op\ g = h$. To compute $f\ op\ g$, we first look up the *comp_table* with 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 *op* on the subgraphs of f and g and store the result in *comp_table* after the completion of $f\ op\ g$. The entries of *comp_table* are used in line 2 and stored in line 21.

After the left and right children have been computed resulting in $\langle c_{h_l}, \mathbf{h_l}\rangle$ and $\langle c_{h_r}, \mathbf{h_r}\rangle$ (lines 17 and 18), if $\langle c_{h_l}, \mathbf{h_l}\rangle = \langle c_{h_r}, \mathbf{h_r}\rangle$, the algorithm returns $\langle c_{h_l}, \mathbf{h_l}\rangle$ to ensure that the case of $\langle x, \mathbf{k}, \mathbf{k}, 0\rangle$ will not occur; otherwise, it returns $\langle c_{h_r}, \langle var, \mathbf{h_l}, \mathbf{h_r}, c_{h_l} - c_{h_r}\rangle\rangle$ to preserve the property of right edge value being 0. There is another table (*uniq_table*) used for the uniqueness property of EVBDD nodes. Before *apply* returns its result, it checks this table through operation *find_or_add* which either adds a new node to the table or returns the node found in the table.

```
$apply(\langle c_f, \mathrm{f}\rangle, \langle c_g, \mathrm{g}\rangle, op)$
{
1     if ($terminal\_case(\langle c_f, \mathrm{f}\rangle, \langle c_g, \mathrm{g}\rangle, op)$ return($\langle c_f, \mathrm{f}\rangle$ $op$ $\langle c_g, \mathrm{g}\rangle$));
2     if ($comp\_table\_lookup(\langle c_f, \mathrm{f}\rangle, \langle c_g, \mathrm{g}\rangle, op, ans)$) return($ans$);
3     if ($index(\mathrm{f}) \geq index(\mathrm{g})$) {
4         $\langle c_{g_l}, \mathrm{g_l}\rangle = \langle c_g + value(\mathrm{g}), child_l(\mathrm{g})\rangle$;
5         $\langle c_{g_r}, \mathrm{g_r}\rangle = \langle c_g, child_r(\mathrm{g})\rangle$;
6         $var = variable(\mathrm{g})$;
7     }
8     else {
9         $\langle c_{g_l}, \mathrm{g_l}\rangle = \langle c_{g_r}, \mathrm{g_r}\rangle = \langle c_g, \mathrm{g}\rangle$;
10        $var = variable(\mathrm{f})$;
11    }
12    if ($index(\mathrm{f}) \leq index(\mathrm{g})$) {
13        $\langle c_{f_l}, \mathrm{f_l}\rangle = \langle c_f + value(\mathrm{f}), child_l(\mathrm{f})\rangle$;
14        $\langle c_{f_r}, \mathrm{f_r}\rangle = \langle c_f, child_r(\mathrm{f})\rangle$;
15    }
16    else { $\langle c_{f_l}, \mathrm{f_l}\rangle = \langle c_{f_r}, \mathrm{f_r}\rangle = \langle c_f, \mathrm{f}\rangle$;}
17    $\langle c_{h_l}, \mathrm{h_l}\rangle = apply(\langle c_{f_l}, \mathrm{f_l}\rangle, \langle c_{g_l}, \mathrm{g_l}\rangle, op)$;
18    $\langle c_{h_r}, \mathrm{h_r}\rangle = apply(\langle c_{f_r}, \mathrm{f_r}\rangle, \langle c_{g_r}, \mathrm{g_r}\rangle, op)$;
19    if (($\langle c_{h_l}, \mathrm{h_l}\rangle == \langle c_{h_r}, \mathrm{h_r}\rangle$)) return (($\langle c_{h_l}, \mathrm{h_l}\rangle$));
20    $\mathrm{h} = find\_or\_add(var, \mathrm{h_l}, \mathrm{h_r}, c_{h_l} - c_{h_r})$;
21    $comp\_table\_insert(\langle c_f, \mathrm{f}\rangle, \langle c_g, \mathrm{g}\rangle, op, \langle c_{h_r}, \mathrm{h}\rangle)$;
22    return ($\langle c_{h_r}, \mathrm{h}\rangle$);
}
```

Example 2.2 An example of $apply(\langle 0, \mathrm{f}\rangle, \langle 0, \mathrm{g}\rangle, +)$ is shown in Fig. 3. Let the variable ordering be $x_0 < x_1$. Fig. 3 (a) shows the initial arguments of *apply*; (b) is the recursive call of *apply* on line 17 whose result is (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).

□

If operator *op* is commutative, then we do the following normalization to increase the hit ratio of computational results:

```
if ($is\_commutative(op)$) {
    if ($index(\mathrm{f}) > index(\mathrm{g}) || index(\mathrm{f}) == index(\mathrm{f})$ && $addr(\mathrm{f}) > addr(\mathrm{g})$)
        $swap(\langle c_f, \mathrm{f}\rangle, \langle c_g, \mathrm{g}\rangle)$;
}
```

where $addr(\mathrm{f})$ is the machine address of EVBDD node **f**. The above code is carried out before performing *comp_table_lookup* in line 2 of *apply*.

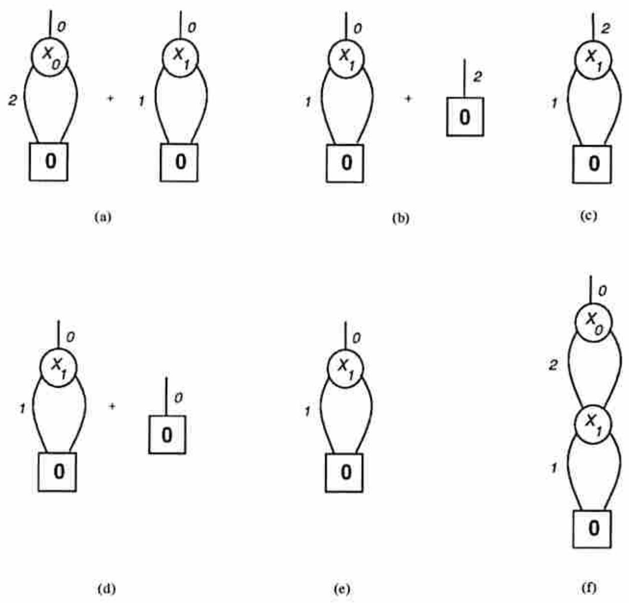


Figure 3: Example of the *apply*($\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, +$) operation.

2.2.1 Complexity Analysis and Flattened EVBDDs

The time complexity of operations in OBDD representation is $O(|\mathbf{f}| \cdot |\mathbf{g}|)$ where $|\mathbf{f}|$ and $|\mathbf{g}|$ are the number of nodes of OBDDs $\mathbf{f}$ and $\mathbf{g}$. The time complexity of operations in EVBDD representation is however **not** $O(|\langle c_f, \mathbf{f}\rangle| \cdot |\langle c_g, \mathbf{g}\rangle|)$ where $|\langle c_f, \mathbf{f}\rangle|$ and $|\langle c_g, \mathbf{g}\rangle|$ are the number of nodes of 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, we define the *domain* of $\mathbf{v}$ ($D_{\mathbf{v}}^{eval}$), and the *cardinality* of $\mathbf{v}$ ($\|\mathbf{v}\|$) as follows:

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

The *cardinality* of $\langle c_f, \mathbf{f}\rangle$, denoted as $\|\langle c_f, \mathbf{f}\rangle\|$, is then given as:

$$\|\langle c_f, \mathbf{f}\rangle\| = \sum_{\mathbf{v} \in \mathbf{f}} \|\mathbf{v}\|.$$

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

Example 2.3 Let $\langle 0, \mathbf{x_0}\rangle$ be the EVBDD in Fig. 2.1 (b), then $\|\mathbf{x_0}\| = 1$, $\|\mathbf{x_1}\| = 2$, $\|\mathbf{x_2}\| = 4$, $\|\mathbf{0}\| = 8$, and $\|\langle 0, \mathbf{x_0}\rangle\| = 15$. The $\langle c_v, \mathbf{v}\rangle$'s for node $\mathbf{x_2}$ are shown in Fig. 4.

□

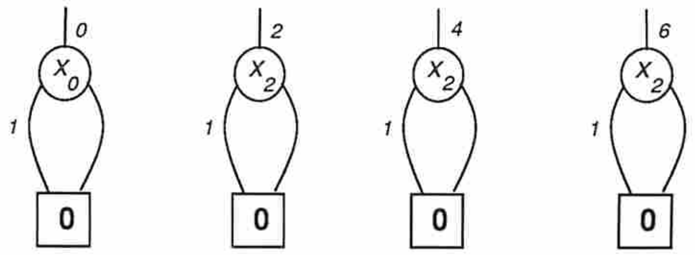


Figure 4: The $\langle c_v, \mathbf{v}\rangle$'s of $\mathbf{x_2}$.

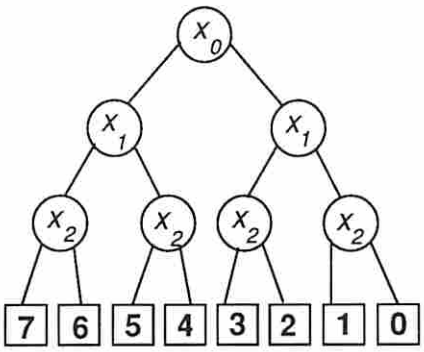


Figure 5: An example of flattened EVBDD.

To have a more precise measure of the time complexity of operations in EVBDD representation, we define *flattened* EVBDDs as follows.

Definition 2.5 A *flattened* EVBDD is a directed acyclic graph consisting of two types of nodes. A *nonterminal* node $\mathbf{v}$ is represented by a 3-tuple $\langle variable(\mathbf{v}), child_l(\mathbf{v}), child_r(\mathbf{v})\rangle$ where $variable(\mathbf{v}) \in \{x_0, \ldots, x_{n-1}\}$. A *terminal* node $\mathbf{v}$ is associated with an integer v. *Reduced, ordered,* flattened EVBDDs are defined in the same way as OBDDs.

Definition 2.6 Given a flattened EVBDD $\mathbf{f}$ with variable ordering $x_0 < \ldots < x_{n-1}$, the evaluation of $\mathbf{f}$ with respect to an input pattern $\langle b_0, \ldots, b_{i-1}\rangle, 0 \le i < n$ is defined as follows:

$$\begin{aligned}
eval_f(\mathbf{v}, \langle b_0, \ldots, b_{i-1}\rangle) &= v, \qquad \text{if } \mathbf{v} \text{ is a terminal node,} \\
eval_f(\langle x_j, \mathbf{l}, \mathbf{r}\rangle, \langle b_0, \ldots, b_{i-1}\rangle) &= \begin{cases} eval_f(\mathbf{l}, \langle b_0, \ldots, b_{i-1}\rangle) & \text{if } j < i \text{ and } b_j = 1, \\ eval_f(\mathbf{r}, \langle b_0, \ldots, b_{i-1}\rangle) & \text{if } j < i \text{ and } b_j = 0, \\ \langle x_j, \mathbf{l}, \mathbf{r}\rangle & \text{if } j \ge i. \end{cases}
\end{aligned}$$

Example 2.4 The flattened EVBDD for the function in Fig. 1 (b) is shown in Fig. 5.

□

From the above definition, flattened EVBDDs are exactly the same as Multi-Terminal OBDDs in [12]. Function values in the flattened EVBDD representation are obtained in the same way as in the OBDD representation. The flattened EVBDD representation also preserves the canonical property.

Lemma 2.2 Two flattened EVBDDs $\mathbf{f}$ and $\mathbf{g}$ denote the same function if and only if they are isomorphic.

Proof: The proof of the canonical property of OBDD representation in [9] can be used to prove the canonical property of flattened EVBDD representation by replacing terminal nodes $\mathbf{0}$ and $\mathbf{1}$ by terminal nodes $\mathbf{u}$ and $\mathbf{v}$ where $u \neq v$.

$\square$

Lemma 2.3 Given a function f represented by an EVBDD $\langle c, \mathbf{f} \rangle$ and a flattened EVBDD $\mathbf{f}'$, $\| \langle c, \mathbf{f} \rangle \| = | \mathbf{f}' |$.

Proof: For any $b \in B^i$, $eval(\langle c, \mathbf{f} \rangle, b) = \langle c_v, \mathbf{v} \rangle$ and $eval_f(\mathbf{f}', b) = \mathbf{v}'$ denote the same function. Since both EVBDD and flattened EVBDD are canonical representations, the mapping between $\langle c, \mathbf{v} \rangle$ and $\mathbf{v}'$ is one-to-one. Thus, for any $b \neq b'$, $eval_f(\mathbf{f}', b) = eval_f(\mathbf{f}', b')$ if and only if $eval(\langle c, \mathbf{f} \rangle, b) = eval(\langle c, \mathbf{f} \rangle, b')$.

$\square$

Since EVBDDs are acyclic directed graphs and there is no backtracking in *apply*, the time complexity of *apply* is $O(\| \langle c_f, \mathbf{f} \rangle \| \cdot \| \langle c_g, \mathbf{g} \rangle \|)$. In many practical applications, the number of nodes in an EVBDD may be small, but its cardinality can be very large. For example, an n-bit integer represented by an EVBDD requires only n nonterminal nodes, but its flattened form requires exponential number of nodes.

apply is a general procedure for performing operations in EVBDDs without incorporating any property to reduce complexity. In the following, we present a number of properties that can be used to reduce the computational complexity of *apply* in many situations.

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 2.7 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 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 [32] (i.e., $(c_f + f) \times 2^c$).

We use $(c_f + f) - (c_g + g)$ as an example:

$$(c_f + f) - (c_g + g) = (c_f - c_g) + (f - g).$$

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 $\langle \langle 0, \mathbf{f} \rangle, \langle 0, \mathbf{g} \rangle, - \rangle$ resulting in $\langle c_h, \mathbf{h} \rangle$, we then add $c_f - c_g$ to c_h to have 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 share the computation result of $\langle \langle 0, \mathbf{f} \rangle, \langle 0, \mathbf{g} \rangle, - \rangle$. This will then increase the hit ratio for caching the computational results. For operators satisfying the additive property, the time complexity of *apply* is $O(| \langle c_f, \mathbf{f} \rangle | \cdot | \langle c_g, \mathbf{g} |)$ (as opposed to $O(\| \langle c_f, \mathbf{f} \rangle \| \cdot \| \langle c_g, \mathbf{g} \rangle \|)$).

To implement this class of operators, we insert the following lines between lines 1 and 2 of *apply*:

```
1.1    $c_{fg} = c_f \ op\ c_g$;
1.2    $c_f = c_g = 0$;
```

We also replace lines 2, 19, and 22 of *apply* by the following lines:

```
2     if ($comp\_table\_lookup(\langle c_f, \mathbf{f}\rangle, \langle c_g, \mathbf{g}\rangle, op, \langle c_h, \mathbf{h}\rangle)$)
          return ($\langle c_h + c_{fg}, \mathbf{h}\rangle$);
19    if ($\langle c_{h_l}, \mathbf{h_l}\rangle == \langle c_{h_r}, \mathbf{h_r}\rangle$) return ($\langle c_{h_l} + c_{fg}, \mathbf{h_l}\rangle$);
22    return ($\langle c_{h_r} + c_{fg}, \mathbf{h}\rangle$);
```

For cases of $(c_f + f) \times c$ and $(c_f + f) << c$, we can further separate the processing of edge values. 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 passing down to the next level in line 3 (cf. line 4 or 13 of *apply*).

```
$times\_c(\langle c_f, \mathbf{f}\rangle, c)$
{
1     if ($\mathbf{f} == \mathbf{0}$) return $\langle c_f \times c, \mathbf{0}\rangle$;
2     if ($comp\_table\_lookup(\langle 0, \mathbf{f}\rangle, c, times\_c, \langle 0, \mathbf{h}\rangle)$) return $\langle c_f \times c, \mathbf{h}\rangle$;
3     $\langle c_{h_l}, \mathbf{h_l}\rangle = times\_c(\langle 0, child_l(\mathbf{f})\rangle, c)$;      /* $c_{h_l} = 0$ */
4     $\langle c_{h_r}, \mathbf{h_r}\rangle = times\_c(\langle 0, child_r(\mathbf{f})\rangle, c)$;      /* $c_{h_r} = 0$ */
5     $\mathbf{h} = find\_or\_add(variable(\mathbf{f}), \mathbf{h_l}, \mathbf{h_r}, value(\mathbf{f}) \times c)$;
6     $comp\_table\_insert(\langle 0, \mathbf{f}\rangle, c, times\_c, \langle 0, \mathbf{h}\rangle)$;
7     return $\langle c_f \times c, \mathbf{h}\rangle$;
}
```

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 new type of computation sharing occurring for relational operations. We use operator $\leq$ as an example. Let $\langle c_f, \mathbf{f}\rangle \leq_\forall \langle c_g, \mathbf{g}\rangle$ denote that $\langle c_f, \mathbf{f}\rangle \leq \langle c_g, \mathbf{g}\rangle$ holds for all input patterns. It follows that

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

which can be seen to be true based on the following derivation:

$$\begin{array}{rcl}
\langle 0, \mathbf{f}\rangle \leq_\forall \langle 0, \mathbf{g}\rangle & \Rightarrow & 0 + f \leq_\forall 0 + g, \\
& \Rightarrow & 0 \leq_\forall -f + g, \\
(c_f - c_g) \leq 0 & \Rightarrow & c_f - c_g \leq_\forall -f + g, \\
& \Rightarrow & c_f + f \leq_\forall c_g + g, \\
& \Rightarrow & \langle c_f, \mathbf{f}\rangle \leq_\forall \langle c_g, \mathbf{g}\rangle.
\end{array}$$

To achieve the above computation sharing, we can have a *comp_table* entry $\langle\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, \leq_\forall, \langle 1, \mathbf{0}\rangle\rangle$. However, we can do better as follows. Consider the following transformation of the above implication:

$$\begin{array}{rcl}
\langle 0, \mathbf{f}\rangle \leq_\forall \langle 0, \mathbf{g}\rangle & \Rightarrow & \langle 0, \mathbf{f}\rangle - \langle 0, \mathbf{g}\rangle \leq_\forall \langle 0, \mathbf{0}\rangle, \\
& \Rightarrow & max(\langle 0, \mathbf{f}\rangle - \langle 0, \mathbf{g}\rangle) \leq 0, \\
& \Rightarrow & -m = max(\langle 0, \mathbf{f}\rangle - \langle 0, \mathbf{g}\rangle) \leq 0, \\
(c_f - c_g) \leq m & \Rightarrow & (c_f - c_g) - m \leq 0, \\
& \Rightarrow & (c_f - c_g) + max(\langle 0, \mathbf{f}\rangle - \langle 0, \mathbf{g}\rangle) \leq 0, \\
& \Rightarrow & max(\langle c_f, \mathbf{f}\rangle - \langle c_g, \mathbf{g}\rangle) \leq 0, \\
& \Rightarrow & \langle c_f, \mathbf{f}\rangle - \langle c_g, \mathbf{g}\rangle \leq_\forall \langle 0, \mathbf{0}\rangle, \\
& \Rightarrow & \langle c_f, \mathbf{f}\rangle \leq_\forall \langle c_g, \mathbf{g}\rangle.
\end{array}$$

Based on the above implication, we replace $\langle c_f, \mathbf{f}\rangle \leq \langle c_g, \mathbf{g}\rangle$ 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 have the following terminal cases:

$$\begin{array}{l}
\text{if } (c_f + max(\mathbf{f})) \leq 0 \text{ return } \langle 1, \mathbf{0}\rangle, \text{ and} \\
\text{if } (c_f + min(\mathbf{f})) > 0 \text{ return } \langle 0, \mathbf{0}\rangle.
\end{array}$$

Another important reason for the inclusion of the maximum and minimum values in each node is that we can easily incorporate branch and bound algorithms into EVBDD representation and thus can solve optimization problems more effectively.

Definition 2.8 An operator *op* applied to $\langle c_f, \mathbf{f}\rangle$ and $\langle c_g, \mathbf{0}\rangle$ is said to satisfy the *bounding property* if

$$((c_f + m(f)) \ op \ c_g) = 0, \ 1, \text{ or } (c_f + f),$$

where $m(f)$ is the maximum or minimum of f.

As a result, when the maximum or minimum of a function exceeds a boundary value (e.g., c_g in Def. 2.8) in an operation, then the result can be determined without further computation.

As an example, the following pseudo code $leq0(\langle c_f, \mathbf{f}\rangle)$ performs operation $(c_f + f) \leq 0$:

```
$leq0(\langle c_f, \mathbf{f}\rangle)$
{
1    if $((c_f + max(\mathbf{f})) \leq 0)$ return($\langle 1, \mathbf{0}\rangle$);
2    if $((c_f + min(\mathbf{f})) > 0)$ return($\langle 0, \mathbf{0}\rangle$);
3    if $(comp\_table\_lookup(\langle c_f, \mathbf{f}\rangle, leq0, ans))$ return($ans$);
4    $\langle c_{h_l}, \mathbf{h_l}\rangle = leq0(\langle c_f + value(\mathbf{f}), child_l(\mathbf{f})\rangle)$;
5    $\langle c_{h_r}, \mathbf{h_r}\rangle = leq0(\langle c_f, child_r(\mathbf{f})\rangle)$;
6    if $(\langle c_{h_l}, \mathbf{h_l}\rangle == \langle c_{h_r}, \mathbf{h_r}\rangle)$ return $(\langle c_{h_l}, \mathbf{h_l}\rangle)$;
7    $\mathbf{h} = find\_or\_add(variable(\mathbf{f}), \mathbf{h_l}, \mathbf{h_r}, c_{h_l} - c_{h_r})$;
8    $comp\_table\_insert(\langle c_f, \mathbf{f}\rangle, leq0, \langle c_{h_r}, \mathbf{h}\rangle)$;
9    return $(\langle c_{h_r}, \mathbf{h}\rangle)$;
}
```

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 2.9 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:

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

Definition 2.10 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 $\mathcal{D}_{\mathbf{v}}^{op} \subset D_{\mathbf{v}}^{eval}$.

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 $D_{\mathbf{v}}^{mod} = D_{\mathbf{v}}^{eval} \cap \{0, \ldots, c-1\}$. 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, computation sharing is large; when c is large, then the following check (using the boundary property) can be used to increase the computation saving:

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

Another example is integer division operator with constant divisor:

$$(c_f + f)/c = (c_f/c) + ((c_f\ mod\ c) + f)/c,$$

assuming both c_f and c are positive integers for this example. In fact, integer division operator with constant division 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.

2.3 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:

$$\begin{aligned} x \wedge y &= xy, & (1)\\ x \vee y &= x + y - xy, & (2)\\ x \oplus y &= x + y - 2xy, & (3)\\ \overline{x} &= 1 - x. & (4) \end{aligned}$$

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

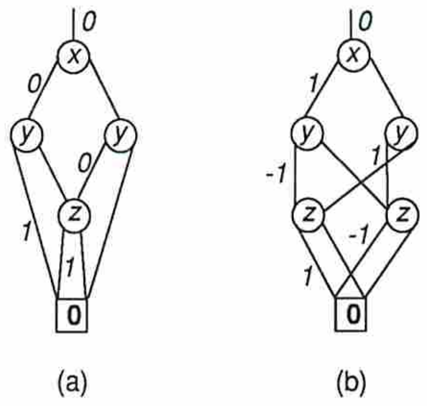


Figure 6: A full-adder represented in EVBDDs: (a) *carry* (b) *sum*.

Example 2.5 The *sum* and *carry* of a full adder in EVBDD are shown in Fig. 6. By using the above equations, we have the following arithmetic functions for *sum* and *carry*:

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

□

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. Pseudo code *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 ($(\langle c_f, \mathbf{f}\rangle == \langle 1, \mathbf{0}\rangle \,\|\, \langle c_g, \mathbf{g}\rangle == \langle 1, \mathbf{0}\rangle)$) return($(\langle 1, \mathbf{0}\rangle)$);
1.2    if ($(\langle c_f, \mathbf{f}\rangle == \langle 0, \mathbf{0}\rangle \,\|\, \langle c_f, \mathbf{f}\rangle == \langle c_g, \mathbf{g}\rangle)$) 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. This is shown by the following algorithm and lemmas.

Algorithm A: Converting a Boolean function from OBDD representation to 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) $\langle x_i, \langle 0, \mathbf{l}'\rangle, \langle 1, \mathbf{r}'\rangle\rangle \Rightarrow \langle 1, \langle x_i, \mathbf{l}', \mathbf{r}', -1\rangle\rangle$,

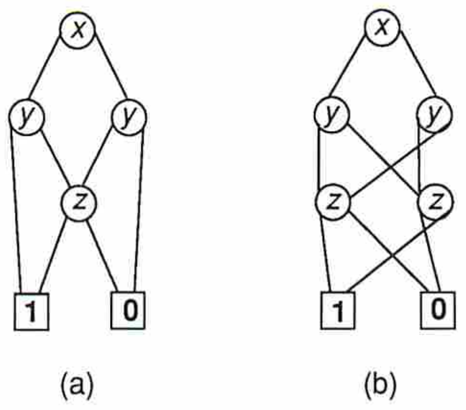


Figure 7: A full-adder represented in OBDDs: (a) *carry* (b) *sum*.

(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. 7 shows the OBDD representation of *carry* and *sum*. After Algorithm A, they will be converted to the EVBDDs in Fig. 6

□

Lemma 2.4 Algorithm A converts an OBDD $\mathbf{v}$ to either $\langle 0, \mathbf{v}' \rangle$ EVBDD or $\langle 1, \mathbf{v}' \rangle$ EVBDD.

Proof: In step 1, only $\langle 0, \mathbf{0} \rangle$ or $\langle 1, \mathbf{0} \rangle$ can be generated, and in step 2, only $\langle 0, \mathbf{v} \rangle$ and $\langle 1, \mathbf{v} \rangle$ can be generated for some $\mathbf{v}$.

□

Lemma 2.5 Algorithm A will neither add nor delete any nonterminal node or edge.

Proof: Directly follows from Algorithm A.

Lemma 2.6 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.

Proof: Let $\mathbf{v}$ represents a Boolean function with n variables, we prove the lemma by induction on n.
Base: $n = 0$. $\mathbf{v}$ is $\mathbf{0}$ or $\mathbf{1}$ and denotes constant function 0 or 1. From step 1 of Algorithm A, $\langle c, \mathbf{v}' \rangle = \langle 0, \mathbf{0} \rangle$ or $\langle 1, \mathbf{0} \rangle$ and denotes the function $0 + 0 = 0$ or $1 + 0 = 1$.
Induction hypothesis: Assume it is true for $n = 0, \ldots, k-1$.
Induction: Let $\mathbf{v}$ be an OBDD node $\langle x, \mathbf{l}, \mathbf{r} \rangle$ representing the Boolean function $(x \wedge l) \vee (\overline{x} \wedge r)$ where l and r are the functions represented by $\mathbf{l}$ and $\mathbf{r}$ respectively, and each has less than k variables. By induction hypothesis, $\mathbf{l}$ and $\mathbf{r}$ can be converted into $\langle c_l, \mathbf{l}' \rangle$ and $\langle c_r, \mathbf{r}' \rangle$ such that $l = c_l + l'$ and $r = c_r + r'$. From step 2 of Algorithm A, $\mathbf{v}$ will be converted into $\langle c, \mathbf{v}' \rangle$

and we need to show $(x \wedge l) \vee (\overline{x} \wedge r) = c + v'$. In the following, we only prove the correctness of case (b) above, the other three cases can be proved similarly. In case (b), $\langle c_l, \mathbf{l}' \rangle = \langle 0, \mathbf{l}' \rangle$ and $\langle c_r, \mathbf{r}' \rangle = \langle 1, \mathbf{r}' \rangle$. From induction hypothesis, $l = 0 + l' = l'$ and $r = 1 + r'$.

$$\begin{aligned}
\text{LHS} &= (x \wedge l) \vee (\overline{x} \wedge r) \\
&= xl + (1-x)r - xl(1-x)r \\
&= xl + r - xr - xlr + xlr \\
&= xl + r - xr \\
&= xl' + (1+r') - x(1+r') \\
&= xl' + 1 + r' - x - xr' \\
\text{RHS} &= 1 + v' \\
&= 1 + x(-1+l') + (1-x)r' \\
&= 1 - x + xl' + r' - xr' \\
&= \text{LHS}
\end{aligned}$$

□

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**.

Proof: It directly follows from lemmas 2.4, 2.5, and 2.6 and the canonical property of EVBDD representation.

□

Lemma 2.7 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.

Proof: If $\langle c, \mathbf{v} \rangle$ is generated where $c \neq 0$ and $c \neq 1$, then $eval(\langle c, \mathbf{v} \rangle, \langle 0, \ldots, 0 \rangle) = c$ which implies that it is not a Boolean function. If both $\langle 0, \mathbf{v} \rangle$ and $\langle 1, \mathbf{v} \rangle$ are generated, then there exist b_0 and b_1 such that $eval(\langle 0, \mathbf{v} \rangle, b_0) = 0$ and $eval(\langle 0, \mathbf{v} \rangle, b_1) = 1$ because $\mathbf{v}$ is a nonterminal node (i.e., $\mathbf{v}$ denotes nonconstant function). Then, $eval(\langle 1, \mathbf{v} \rangle, b_0) = 1$ and $eval(\langle 1, \mathbf{v} \rangle, b_1) = 2$ which again leads to a non-Boolean function.

□

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 $O(|\mathbf{f}| \cdot |\mathbf{g}|)$.

Proof: From lemma 2.7, since only one $\langle c, \mathbf{v} \rangle$ can exist for every nonterminal node $\mathbf{v}$ during the process of *apply*, we have $| \langle c_f, \mathbf{f} \rangle | = \| \langle c_f, \mathbf{f} \rangle \|$ and $| \langle c_g, \mathbf{g} \rangle | = \| \langle c_g, \mathbf{g} \rangle \|$. It is well known that the time complexity of Boolean operations in OBDD representation is $O(| \mathbf{f} | \cdot | \mathbf{g} |)$ where $|\mathbf{f}|$ and $|\mathbf{g}|$ are the number of nodes in OBDD representation. From theorem 2.1, both representations have the same number of nonterminal nodes, thus the complexities are also the same.

□

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).

In the following sections, we present applications of EVBDDs. For the sake of readability, we also use the flattened form of EVBDDs in graphical representation. In flattened form, edge values are pushed down to the bottom such that a terminal node is some integer representing the function value.

3 Logic Verification

The process of logic verification is to show the equivalence between the specification of intended behavior and the description of implemented designs. If both specification and implementation are Boolean expressions, then the correctness can only be proved up to the Boolean behavior. On the other hand, if the specification is an arithmetic function while the implementation is a set of Boolean expressions, then the correctness is up to the arithmetic behavior.

When used for logic verification, 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 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 $2carry+sum = x+y+z$. 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. 6. Carrying out the expression $2carry + sum$ results in the rightmost EVBDD shown in Fig. 8. On the other hand, the specification of the arithmetic behavior of the full adder, $x + y + z$, represented in EVBDDs is shown in Fig. 9. The equivalence between $2carry + sum$ and $x + y + z$ can then be checked by comparing the two rightmost EVBDDs in Figures 8 and 9.

□

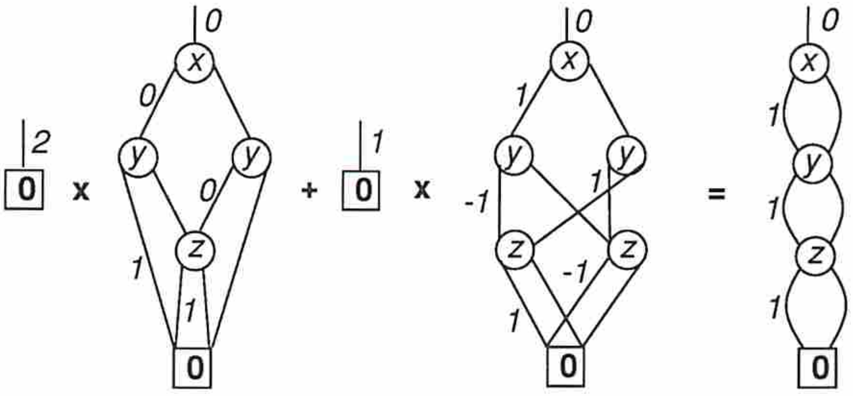


Figure 8: EVBDD expression: $2carry + sum$.

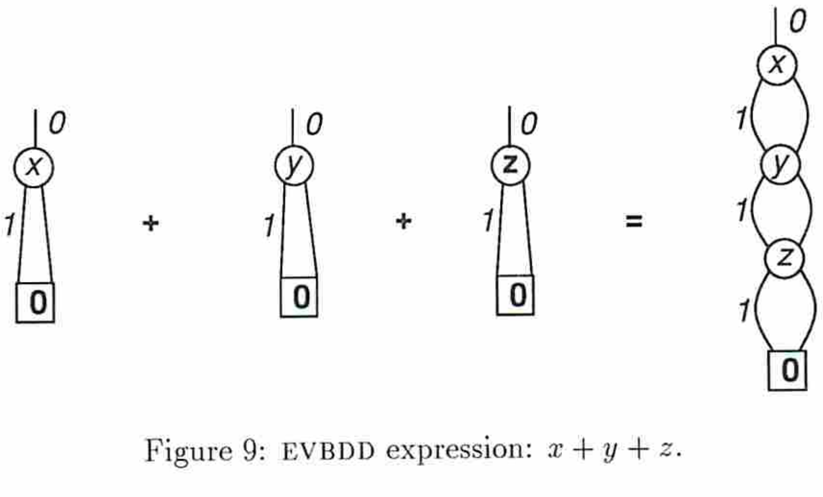


Figure 9: EVBDD expression: $x + y + z$.

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

3.1 The Verification and Synthesis Paradigm

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

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

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

Using the example in the previous section, we have:

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

And, 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 become 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

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

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

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

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 without variable binding. 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 [33] 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

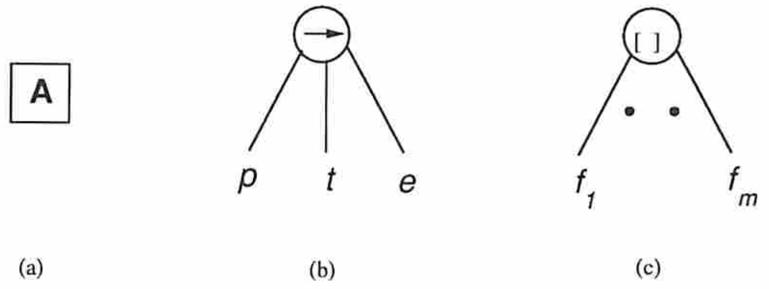


Figure 10: Graphical representation of SEVBDDs.

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 graphical representation of SEVBDDs, terminal nodes are atomic type SEVBDDs (Fig. 10 (a)) and there are two types of nonterminal nodes: a *conditional node* which has three children (Fig. 10 (b)) and a *vector node* which has an indefinite number of nodes (Fig. 10 (c)).

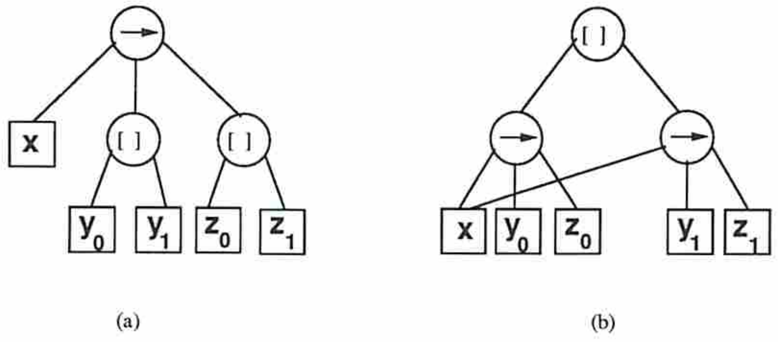


Figure 11: Examples of SEVBDDs.

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

1. x, $x \wedge y$, $\bar{x} \wedge z$, and
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 atomic type SEVBDDs; Those in groups 3 and 4 are conditional type SEVBDDs; Those in groups 5 and 6 are vector type SEVBDDs. 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. 11 (a) and (b), respectively.

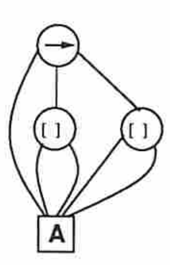
$\square$

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**.

Example 3.4 The type graphs of the SEVBDDs in Fig. 11 are shown in Fig. 12.

$\square$

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.



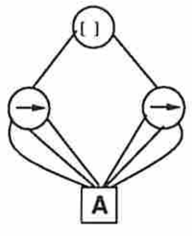


Figure 12: Examples of type graph of SEVBDDs.

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. $f = (p_f \rightarrow t_f; e_f)$, $g = (p_g \rightarrow t_g; e_g)$, p_f and p_g are isotypic, 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 $p \rightarrow t; e$. Operator $cofactor_1(p, t)$ is carried out in a similar way to the *restrict* operator in [14] 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 have the canonical property is to be *reduced* as defined in the following:

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.

To show the canonical property of the restricted form of SEVBDDs, we define a function *level* on SEVBDDs as follows:

Definition 3.5 The function $level$: SEVBDD $\rightarrow$ *integer* is defined recursively as:

1. $level(ev) = 0$, if ev is an EVBDD,
2. $level(p \rightarrow t; e) = 1 + max\{level(t), level(e)\}$,
3. $level([f_0, \ldots, f_{m-1}]) = 1 + max\{level(f_0), \ldots, level(f_{m-1})\}$.

In Ex. 3.3, the SEVBDDs in groups 1 and 2 have level 0; the SEVBDDs in group 3 have level 1, and the SEVBDDs in groups 4, 5, and 6 have level 2.

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.

Proof: Necessity: It is trivial to show that if f and g are isomorphic, then they denote the same function.
Sufficiency: If f and g are not isomorphic, then they denote different functions, that is, $\exists b \in B^n$ such that $eval(f,b) \neq eval(g,b)$. We show this by induction on $level(f)$.
Base: $level(f) = 0$, then both f and g are EVBDDs. This is true from lemma 2.1.
Induction hypothesis: Assume it is true for $level(f) < k$.
Induction:
Case 1: $f = (p \rightarrow t_f; e_f)$ and $g = (p \rightarrow t_g; e_g)$.
Since f and g are not isomorphic, then t_f and t_g are not isomorphic and/or e_f and e_g are not isomorphic. If t_f and t_g are not isomorphic then by induction hypothesis there exists b such that $eval(t_f,b) \neq eval(t_g,b)$. Because t_f and t_g are reduced, that is, if $eval(p,b) = 0$ then $eval(t_f,b) = eval(t_g,b) = 0$ by operator $cofactor_1$, we have $eval(p,b) = 1$. Thus, $eval(f,b) = eval(t_f,b) \neq eval(t_g,b) = eval(g,b)$. By a similar reasoning, we can show that the induction step holds true when e_f and e_g are not isomorphic.
Case 2: $f = [f_1, \ldots, f_m]$ and $g = [g_1, \ldots, g_m]$.
There exist $1 \leq i \leq m$ such that f_i and g_i are not isomorphic and from induction hypothesis there exists b such that $eval(f_i,b) \neq eval(g_i,b)$. Then, $eval(f,b) = \langle \ldots, eval(f_i,b), \ldots \rangle \neq \langle \ldots, eval(g_i,b), \ldots \rangle = eval(g,b)$.

□

After proving that isotypic and reduced SEVBDDs enjoy the canonical property, we need procedures for converting an SEVBDD from one form to another and/or reducing an SEVBDD. Operator $cofactor_1$ 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 [8] except that our *ite* operator also applies to arithmetic functions. Operator *ite* takes a conditional SEVBDD such as $(p \rightarrow t; e)$ (t and e are EVBDDs) as argument and returns an EVBDD f such that $(p \rightarrow t; e)$ and f denote the same function. The following pseudo codes are for converting the forms of SEVBDDs and reducing SEVBDDs, where $cofactor\_s_1$, $cofactor\_s_0$, and ite_s are SEVBDD versions of $cofactor_1$, $cofactor_0$, and *ite*, respectively.

```
$convert(f, g)$ /* converting $g$ to the same form as that of $f$ */
/* assuming $f$ and $g$ have the same number of outputs,*/
/* e.g., both $f$ and $g$ are atomic form or vector form with the same number of elements */
{
    if ($f$ is an EVBDD)
        if ($g$ is an EVBDD) return($g$);
        if ($g == (p \rightarrow t; e)$) return($ite(p, convert(f,t), convert(f,e))$);
    else if ($f == (p \rightarrow t; e)$)
        return($p \rightarrow convert(t, cofactor\_s_1(p,g)); convert(e, cofactor\_s_0(p,g))$);
    else        /* $f = [f_1, \ldots, f_m]$ */
        if ($g == (p \rightarrow t; e)$)
            return($ite\_s(p, convert(f,t), convert(f,e))$);
        else return($[convert(f_1, g_1), ..., convert(f_m, g_m)]$);
}
```

```
$reduce(f)$
{
    if ($f$ is an EVBDD) return($f$);
    else if ($f == (p \rightarrow t; e)$)
        return($reduce(p) \rightarrow reduce(cofactor\_s_1(p,t)); reduce(cofactor\_s_0(p,e))$);
    else return($[reduce(f_1), \ldots, reduce(f_m)]$);
}
```

```
$cofactor\_s_1(p,t)$
{
    if ($t$ is an EVBDD) return($cofactor_1(p,t)$);
    else if ($t == p' \rightarrow t'; e'$)
        return($cofactor\_s_1(p,p') \rightarrow cofactor\_s_1(p,t'); cofactor\_s_1(p,e')$);
    else /* $t = [t_1, \ldots, t_m]$ */
        return($[cofactor\_s_1(p,t_1), \ldots, cofactor\_s_1(p,t_m)]$);
}
```

```
$ite\_s(p,t,e)$      /* assuming $t$ and $e$ are isotypic */
{
    if ($p$ is a conditional SEVBDD) return($ite\_s(ite\_s(p) \to t; e)$);
    if ($t$ is an EVBDD) return($ite(p,t,e)$);
    if ($t == (p_t \to t_t; e_t)$ && $e == (p_e \to t_e; e_e)$)
        return($ite\_s(p,p_t,p_e) \to ite\_s(p,t_t,t_e); ite\_s(p,e_t;e_e)$);
    if ($t == [t_1,\ldots,t_m]$ && $e == [e_1,\ldots,e_m]$)
        return($[ite\_s(p,t_1,e_1),\ldots,ite\_s(p,t_m,e_m)]$);
}
```

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 specifications usually have more compact representations than that of implementations have. For example, a specification of '$x \leq y \to x+y; x-y$' 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 [49], 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 [49] 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* and then the comparison take 0.02 seconds.

□

Example 3.7 The implementation is the SN74181 chip which is a 4-bit ALU [49]. 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 = 6) return((un_comp(5)) A−B−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 to different number systems as well as 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 to complete.

□

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. A net list description of this design we use is:

```
out1  SN74L85 a0 a1 a2 a3 b0 b1 b2 b3 gt lt eq
out2  SN74L85 a4 a5 a6 a7 b4 b5 b6 b7 out1
   ⋮
out16 SN74L85 a60 a61 ... b62 b63 out15
Output : out16
```

where a net list has the form of : *output_name module_name input_name_list.* 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. A net list description of this design we use is:

```
sc1  SN74181 G P G G P a0 a1 a2 a3 b0 b1 b2 b3 c0
s1   tail    sc1
c1   head    sc1
sc2  SN74181 G P G G P a4 a5 a6 a7 b4 b5 b6 b7 c1
       ⋮
s15  tail    sc15
c15  head    sc15
sc16 SN74181 G P G G P a60 a61 ... b62 b63 c15
Output : s1 s2 ... s14 s15 sc16
```

The first 5 parameters of each SN74181 are connected to the ground or power to select the addition operation. *tail* groups all the inputs except the first one (the most significant bit) while *head* selects the first bit.

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 second. 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 [9]. 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 [9], and is easier to identify from arithmetic expressions than from Boolean expressions.

4 Integer Linear Programming

Integer Linear Programming (ILP) is an NP-hard problem [22] that appears in many applications. Most of existing techniques for solving ILP such as branch and bound [35, 16, 40] and cutting plane methods [23, 24] are based on the linear programming (LP) method. While they may sometimes solve hundreds of variables, they cannot guarantee to find an optimal solution for problems with more than , say, 50 variables. It is believed that an effective

ILP solver should incorporate integer or combinatorial programming theory into the linear programming method [4].

Jeong et al. [30] describe an OBDD-based approach for solving the 0-1 programming problems. This approach does not, however, use OBDDs for integer related operations such as conversion from linear inequality form of constraints into Boolean functions and optimization of nonbinary goal functions. Consequently, the caching of computation results is limited to only Boolean operations (i.e., for constraint conjunction).

Our approach for solving the ILP is to combine benefits of the EVBDD data structure (in terms of subgraph sharing and caching of computation results) with the state-of-the-art ILP solving techniques. We have developed a minimization operator in EVBDD which computes the optimal solution to a given goal function subject to a constraint function. In addition, the construction and conjunction of constraints in terms of EVBDDs are carried out in a divide and conquer manner in order to manage the space complexity.

4.1 Background

An ILP problem can be formulated as follows:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n} c_i x_i, \\ \text{subject to} & \sum_{i=1}^{n} a_{i,j} x_i \leq b_j,\ 1 \leq j \leq m, \\ & x_i \ \text{integer.} \end{array}$$

The first equation is referred as the *goal function* and the second equation is referred as *constraint functions*. Throughout this section we will assume the problem to be solved is a *minimization* problem. A *maximization* problem can be converted to a minimization problem by changing the sign of coefficients in the goal function.

There are three classes of algorithms for solving ILP problems [48]. The first class is known as the branch-and-bound method [35, 16, 40]. This method usually starts with an optimum continuous LP solution which forms the first *node* of a search tree. If the initial solution satisfies the integer constraints, it is the optimum solution and the procedure is terminated. Otherwise, we split on variable x (with value x^* from the initial solution) and create two new subproblems: one with the additional constraint $x \leq \lfloor x^* \rfloor$ and the other with the additional constraint $x \geq \lfloor x^* \rfloor + 1$. Each subproblem is then solved using the LP method, e.g., the simplex method [17] or the interior point method [31]. A subproblem is pruned if there are no feasible solutions, the feasible solution is inferior to the best one found, or all variables satisfy the integer constraints. In the last case, the feasible solution becomes the new best solution. The problem is solved when all subproblems are processed. Most commercial programs use this approach [36].

The second class is known as the implicit enumeration technique which deals with 0-1 programming [2, 3, 46]. Initially, all variables are *free*. Then, a sequence of *partial solutions* is generated by successively *fixing* free variables, i.e., setting free variables to 0 or 1. A *completion* of a partial solution is a solution obtained by fixing all free variables in the partial solution. The algorithm ends when all partial solutions are completions or are discarded.

The procedure proceeds similar to the branch and bound except that it solves a subproblem using the *logical tests* instead of the LP. A logical test is carried out by inserting values corresponding to a given (partial or complete) solution in the constraints. A complete solution is feasible if it satisfies all constraints. A partial solution is pruned if it cannot reach a feasible solution or could only produce an inferior feasible solution (compared to the current best solution). One advantage of this approach is that we can use *partial order* relations among variables to prune the solution space. For example, if it is established that $x \leq y$, then portions of the solution space which correspond to $x = 1$ and $y = 0$ can be immediately pruned [7, 27].

In the early days, these two methods were considered to be sharply different. The branch and bound method is based on solving a linear program at every node in the search space and uses a breadth first strategy. The implicit enumeration method is based on logical tests requiring only additions and comparisons and employs a depth first strategy. However, successively versions of both approaches have borrowed substantially from each other [3]. The two terms branch and bound and implicit enumeration are now used interchangeably.

The third class is known as the cutting-plane method [23, 24]. Here, the integer variable constraint is initially dropped and an optimum continuous variable solution is obtained. The solution is then used to chop off the solution space while ensuring that no feasible integer solutions are deleted. A new continuous solution is computed in the reduced solution space and the process is repeated until the continuous solution indeed becomes an integer solution. Due to the machine round-off error, only the first few cuts are effective in reducing the solution space [48].

4.2 The Main Algorithm

In this section, we first show a straightforward method to solve the ILP problem using EVBDDs. We then describe how to improve this method in this and the following sections.

Example 4.1 We illustrate how to solve the ILP problems using EVBDDs through a simple example. For the sake of readability, we use flattened EVBDDs (see Sec. 2.2.1).

The following is a 0-1 ILP problem:

$$\begin{array}{lll} \text{minimize} & 3x + 4y, & \\ \text{subject to} & 6x + 4y \leq 8, & (1) \\ & 3x - 2y \leq 1, & (2) \\ & x, y \in \{0, 1\}. & \end{array}$$

We first construct an EVBDD for the goal as shown in Fig. 13 (a). We then construct the constraints. The left hand side of constraint (1) represented by an EVBDD is shown in Fig. 13 (b). After the relational operator $\leq$ has been applied on constraint (1), the resulting EVBDD is shown in Fig. 13 (c). Similarly, EVBDDs for constraint (2) are shown in Fig. 13 (d) and (e). The conjunction of two constraints, Fig. 13 (c) and (e), results in the EVBDD in Fig. 13 (f) which represents the solution space of this problem. A feasible solution corresponds to a path from the root to 1.

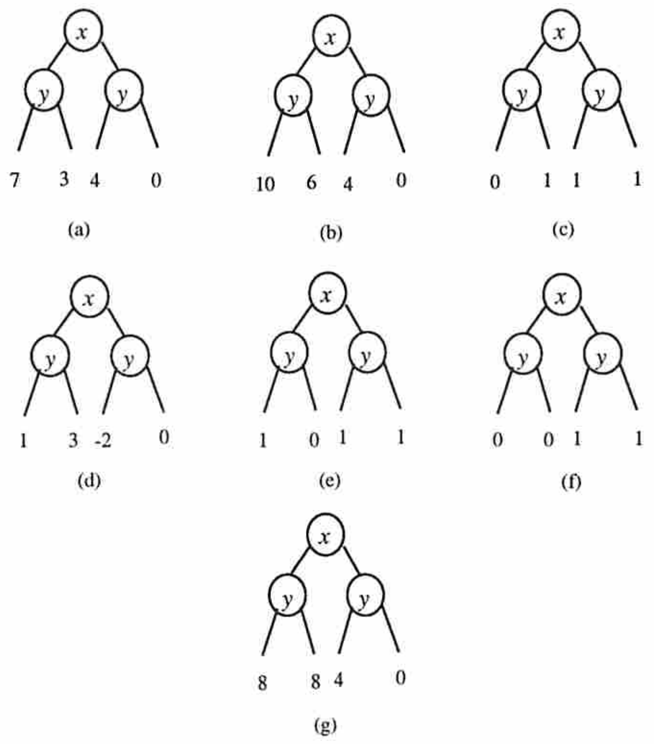


Figure 13: A simple example (using flattened EVBDDs and OBDDs).

We then *project* the constraint function c onto the goal function g such that for a given input assignment X, if $c(X) = 1$ (feasible) then $p(X) = g(X)$; otherwise $p(X) = infeasible\_value$. For minimization problems, the $infeasible\_value$ is any value which is greater than the maximum of g, and for maximization problems, the $infeasible\_value$ is any value which is smaller than the minimum of g. In our example, we use 8 as the $infeasible\_value$. Thus, in Fig. 13 (g), the two leftmost terminal values have been replaced by value 8. The last step in solving the above ILP problem is to find the minimum in Fig. 13 (g) which is 0.

□

The above approach has three problems:

1. Converting a constraint from inequality form to a Boolean function may require exponential number of nodes;
2. Even if all constraints can be constructed without using excessive amounts of memory, conjoining them altogether at once may create too big an EVBDD; and
3. The operator *projection* is useful when we want to find all optimal solutions. However, in many situations, we are interested in finding *any* optimal solution. Thus, full construction of the final EVBDD (e.g., Fig. 13 (g)) is unnecessary.

In the remainder of this section, we will show how to overcome the first two problems by divide and conquer methods. In the next section, we will present an operator *minimize* which combines the benefits of computation sharing and branch and bound techniques to compute any optimal solution.

In our ILP solver, called FGILP, every constraint is converted to the form $AX - b \leq 0$. Thus, we only need one operator *leq*0 (Sec. 2.2.3) to perform the conversion. $AX < b$ is converted to $AX - b + 1 \leq 0$ (since all coefficients are integer); $AX \geq b$ is converted to $-AX + b \leq 0$; and $AX = b$ is converted to two constraints $AX - b \leq 0$ and $-AX + b \leq 0$.

Initially, every constraint is an EVBDD representing the left hand side of an inequality (i.e., $AX - b$) which requires n nonterminal nodes for an n-variable function. FGILP provides users with an *n_supp* parameter such that only if a constraint has less than *n_supp* supporting (dependent) variables, then it will be converted to a Boolean function. FGILP allows users to set another parameter *c_size* to control the size of EVBDDs. Only if constraints, in Boolean function form, are smaller in size than this parameter, they will be conjoined.

Parameters *n_supp* and *c_size* provide two advantages. First, they provide FGILP with a space-time tradeoff capability. The more memory FGILP has, the faster it runs. Second, combined with the branch and bound technique, some subproblems may be pruned before the conversion to the Boolean functions or the conjunction of constraints are carried out.

When there is only one constraint and it is in Boolean form, then the problem is solved through *minimize*. Otherwise, the problem is divided into two subproblems and is solved recursively. Since both the goal and constraint functions are represented by EVBDDs. The new goal and constraint functions for the first subproblem are the left children of the root nodes of the current goal and constraints. Similarly, the new goal and constraint functions for the second subproblem are the right children of the root nodes of the current goal and constraints.

Our main algorithm, *ilp_minimize*, employs a branch and bound technique as shown in Fig. 14. In addition to goal and constraint functions, *n_supp*, and *c_size*, there are two parameters which are used as bounding condition: *Lower bound* is either given by the user or computed through linear relaxation or Lagrangian relaxation methods; *Upper bound* represents the best feasible solution found so far. The initial value of the upper bound is the maximum of the goal function plus 1.

If the maximum of goal function is less than the lower bound or the minimum of goal function is greater than or equal to the upper bound, the problem is pruned. Furthermore, if there exists a constraint whose minimum feasible solution is greater than or equal to the current best solution (upper bound), then again the problem is pruned.

Example 4.2 We want to solve the following problem:

$$\begin{array}{ll} \text{minimize} & -4x + 5y + z + 2w, \\ \text{subject to} & 3x + 2y - 4z - w \leq 0, \\ & 2x + y + 3z - 4w \leq 0, \\ & x, y, z, w \in \{0, 1\}. \end{array}$$

1. The initial goal and constraint EVBDDs are shown in Fig. 15 (a). Suppose both parameters *n_supp* and *c_size* are set to 4.

```
$ilp\_minimize(goal, constraints, lower\_bound, upper\_bound, n\_supp, c\_size)$
{
1     if ($max(goal) < lower\_bound$) return;
2     if ($min(goal) \geq upper\_bound$) return;
3     if ($\exists c \in constraints$ such that $minimize(goal, c, upper\_bound) == 0$) return;
4     $new\_constraints = conjunction\_constraint(constraints, c\_size)$;
5     if ($new\_constraints$ has only one element and is in Boolean form) {
6         $minimize(goal, new\_constraints, upper\_bound)$;
7     }
8     else {
9     $\langle\langle goal_l, new\_constraints_l\rangle, \langle goal\_r, new\_constraints_r\rangle\rangle =$
          $divide\_problem(goal, new\_constraints, n\_supp)$;
10    $ilp\_minimize(goal_l, new\_constraints_l, lower\_bound, upper\_bound, n\_supp, c\_size)$;
11    $ilp\_minimize(goal_r, new\_constraints_r, lower\_bound, upper\_bound, n\_supp, c\_size)$;
12    }
}
```

Figure 14: Pseudo code for *ilp_minimize*.

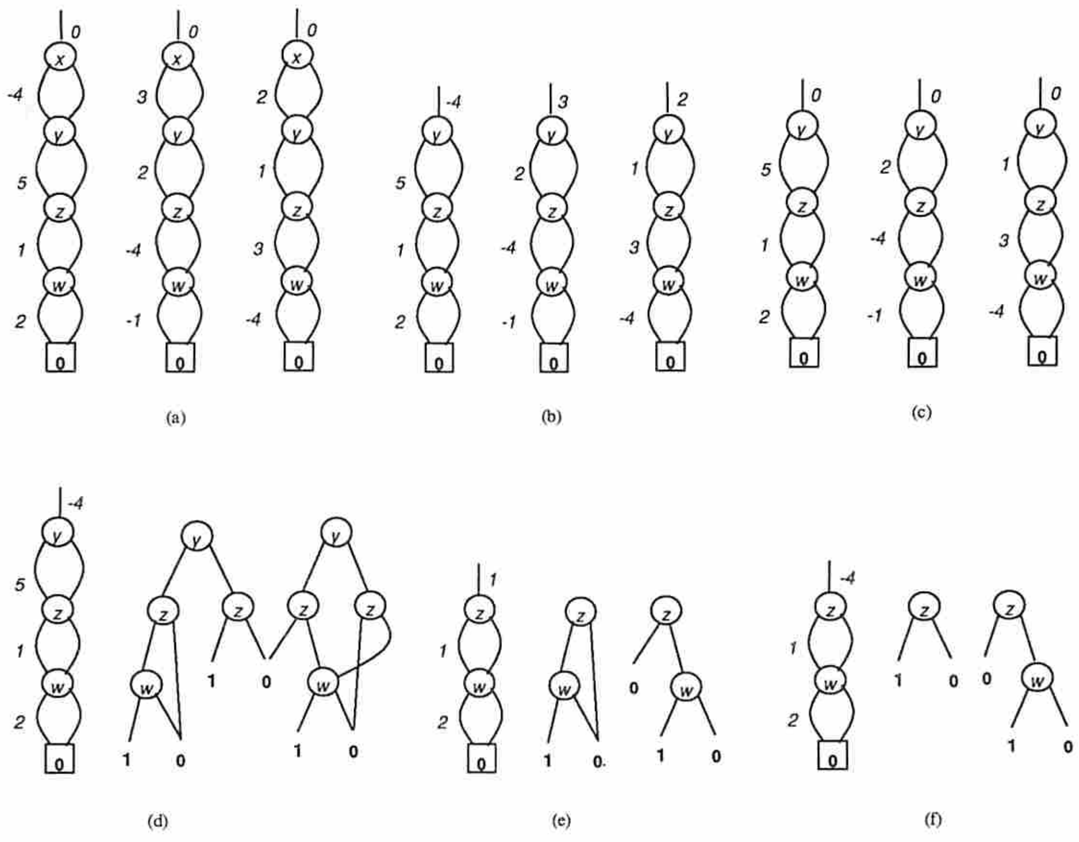


Figure 15: An example for conjoining constraints.

2. Since the number of supporting variables in the constraint EVBDDs is not less than 4, we divide the problem into two subproblems: one with $x = 1$ (Fig. 15 (b)) and the other with $x = 0$ (Fig. 15 (c)). The final solution is the minimum of solutions to these two subproblems.

3. Next, we want to solve the subproblem with $x = 1$. Since the number of supporting variables in constraint EVBDDs is smaller than *n_supp*, we convert the constraint EVBDDs into Boolean functions by carrying out operation *leq*0 (Fig. 15 (d)).

4. Since the size of constraint EVBDDs are not less than *c_size*, we divide the problem into two subproblems: one with $y = 1$ (Fig. 15 (e)) and the other with $y = 0$ (Fig. 15 (f)).

5. Now, we want to solve the subproblem with $y = 1$. Since the size of both constraint EVBDDs are less than *c_size*, we conjoin them together and then solve this subproblem using the *minimize* operator (Sec. 4.3).

6. The remaining subproblems are solved in the same way. Note that the solution found from a subproblem can be used as an upper bound for the subproblems which follow.

□

4.3 The Operator *minimize*

Operator *minimize* is similar to the *apply* operator with one additional parameter b. Given a goal function g, a constraint function c, and an upper bound b, *minimize* returns 1 if it finds a minimum feasible solution $v < b$ of g subject to c; otherwise, *minimize* returns 0. If v is found, b is replaced by v; otherwise, b is unchanged.

Note that when *minimize* returns 0, it does not imply that there are no feasible solutions with respect to g and c. This is because *minimize* only searches for feasible solutions that are smaller than b. Those feasible solutions which are greater than or equal to b are pruned because of the branch and bound procedure.

The parameter b serves two purposes: it increases the hit ratio for computation caching and is a bounding condition for pruning the problem space. To achieve the first goal, an entry of the computed table used by *minimize* has the form $\langle g, c, \langle b, v\rangle\rangle$ where v is set to the minimum of g which satisfies c and is less than b. If there is no feasible solution (with respect to g and c) which is less than b, then v is set to b.

The following pseudo code implements *minimize.* Lines 1-8 test for terminal conditions. In line 1, if the constraint function is the constant function 0, there is no feasible solution. In line 2, if the minimum of the goal function is greater than or equal to the current best solution, the whole process is pruned. If the goal function is a constant function, it must be less than *bound*; otherwise, the test in line 2 would have been true. Thus, a new minimum is found in line 3. In line 6, if the constraint function is constant 1, then the minimum of the goal function is the new optimum. Again, this must be true, otherwise, the condition tested in line 2 would have been true.

Lines 9-17 perform the table lookup operation. If the lookup succeeds, no further computation is required; otherwise, we traverse down the graph in lines 19-26 in the same way as *apply*. Since *minimize* satisfies the additive property (Sec. 2.2.2), we subtract c_g from *bound* to obtain a new local bound (*local_bound*) in line 9. c_g will be added back to *bound* in lines 13 or 32 if a new solution is found.

Suppose we want to compute the minimum of g subject to c with current local upper bound *local_bound*. We look up the computed table with key $\langle g, c\rangle$. If an entry $\langle g, c,$ $\langle entry.bound, entry.value\rangle\rangle$ is found, then there are the following possibilities:

1. $entry.value < entry.bound$, i.e., a smaller value v was previously found with respect to g, c, and *entry.bound* (i.e., the minimization of g with respect to c has been solved and the result is *entry.value*).
   (a) If $entry.value < local\_bound$, then *entry.value* is the solution we wanted.
   (b) Otherwise, the best we can find under g and c is *entry.value* which is inferior to *local_bound*, so we return with no success.
2. $entry.bound = entry.value$, i.e., there was no feasible solution with respect to g, c, and *entry.bound* (i.e., there is no stored result for the minimization of g with respect to c and *entry.bound*).
   (a) If $local\_bound \leq entry.bound$, then we cannot possibly find a solution better than *entry.bound* for g under c. Therefore, we return with no success.
   (b) Otherwise, no conclusion can be drawn and further computation is required. Although there is no better feasible solution than *entry.bound*, it does not imply that there will be no better solution than *local_bound*.

In cases 1.b and 2.a pruning takes place (also computation caching), in case 1.a, computation caching is a success, while in case 2.b both operations fail. Note that there is no need for updating an entry (of the computed table) except in case 2.b.

In lines 25-30, the branch whose minimum value is smaller is traversed first since this increases chances for pruning the other branch. Finally, we update computed table and return the computed results in lines 31-39.

```
$minimize(\langle c_g, \mathbf{g}\rangle, \langle c_c, \mathbf{c}\rangle, bound)$
{
    if ($\langle c_c, \mathbf{c}\rangle == \langle 0, \mathbf{0}\rangle$) return 0;
    if ($min(\langle c_g, \mathbf{g}\rangle) \geq bound$) return 0;
    if ($\langle c_g, \mathbf{g}\rangle == \langle c_g, \mathbf{0}\rangle$) {
      $bound = c_g$;
      return 1;   }
    if ($\langle c_c, \mathbf{c}\rangle == \langle 1, \mathbf{0}\rangle$) {
      $bound = min(\langle c_g, \mathbf{g}\rangle)$;
      return 1;   }
    $local\_bound = bound - c_g$;
    if ($comp\_table\_lookup(\langle 0, \mathbf{g}\rangle, \langle c_c, \mathbf{c}\rangle, entry)$) {
      if ($entry.value < entry.bound$) {
          if ($entry.value < local\_bound$) {
              $bound = entry.value + c_g$;
              return 1;   }
          else return 0;   }
      else {
          if ($local\_bound \leq entry.bound$) return 0;   }   }
    $entry.bound = local\_bound$;
    $\langle c_{g_l}, \mathbf{g_l}\rangle = \langle value(\mathbf{g}), child_l(\mathbf{g})\rangle$;
    $\langle c_{g_r}, \mathbf{g_r}\rangle = \langle 0, child_r(\mathbf{g})\rangle$;
    if ($index(variable(\mathbf{c})) \leq index(variable(\mathbf{g}))$) {
      $\langle c_{c_l}, \mathbf{c_l}\rangle = \langle c_c + value(\mathbf{c}), child_l(\mathbf{c})\rangle$;
      $\langle c_{c_r}, \mathbf{c_r}\rangle = \langle c_c, child_r(\mathbf{c})\rangle$;   }
    else {   $\langle c_{c_l}, \mathbf{c_l}\rangle = \langle c_{c_r}, \mathbf{c_r}\rangle = \langle c_c, \mathbf{c}\rangle$;   }
    if ($min(\mathbf{g_l}) \leq min(\mathbf{g_r})$) {
      $t\_ret = minimize(\langle c_{g_l}, \mathbf{g_l}\rangle, \langle c_{c_l}, \mathbf{c_l}\rangle, local\_bound)$;
      $e\_ret = minimize(\langle c_{g_r}, \mathbf{g_r}\rangle, \langle c_{c_r}, \mathbf{c_r}\rangle, local\_bound)$;   }
    else {
      $e\_ret = minimize(\langle c_{g_r}, \mathbf{g_r}\rangle, \langle c_{c_r}, \mathbf{c_r}\rangle, local\_bound)$;
      $t\_ret = minimize(\langle c_{g_l}, \mathbf{g_l}\rangle, \langle c_{c_l}, \mathbf{c_l}\rangle, local\_bound)$;   }
    if ($t\_ret \,||\, e\_ret$) {
      $bound = local\_bound + c_g$;
      $entry.value = local\_bound$;
      $comp\_table\_insert(\langle 0, \mathbf{g}\rangle, \langle c_c, \mathbf{c}\rangle, entry)$;
      return 1;   }
    else {
      $entry.value = entry.bound$;
      $comp\_table\_insert(\langle 0, \mathbf{g}\rangle, \langle c_c, \mathbf{c}\rangle, entry)$;
      return 0;   }
}
```

Example 4.3 We want to minimize the goal function $-4x + 5y + z + 2w$ subject to the constraint $(xz\bar{w} \vee \bar{x}yz\bar{w} \vee \bar{x}\bar{y}z \vee \bar{x}\bar{y}\bar{z}w = 1)$ shown in Fig. 16. For the sake of readability, the goal function is represented in EVBDD while the constraint function is represented in OBDD. The initial *upper bound* is $max(goal) + 1 = 0 + 5 + 1 + 2 + 1 = 9$. The reason for plus 1 is to recognize the case when there are no feasible solutions.

(a) We traverse down to nodes **a** and **b** through path $x = 1$ and $y = 1$. By subtracting the coefficients of x and y from *upper bound*, we have $9 - (-4) - 5 = 8$ which is the *local upper bound* with respect to nodes **a** and **b**. That is, we look for a minimum of **a** subject to **b** such that it is smaller than 8. It is easy to see that the best feasible solution of **a** subject to **b** is 1 which corresponds the assignments of $z = 1$ and $w = 0$. Thus, we insert $\langle \mathbf{a}, \mathbf{b}, \langle 8, 1\rangle\rangle$ as an entry into the computed table and recalculate the *upper bound* as $-4 + 5 + 1 + 0 = 2$.

(b) We traverse down to nodes **a** and **b** this time through path $x = 1$ and $y = 0$. The new local upper bound is $2 - (-4) - 0 = 6$, i.e., we look for a feasible solution which is smaller than 6. From computed table look up, we find that 1 is the best solution with respect to **a** and **b** and it is smaller than 6. Thus, the new upper bound is $-4 + 0 + 1 = -3$.

(c) We reach **a** and **b** through path $x = 0$ and $y = 1$. The local upper bound is $-3 - 0 - 5 = -8$. Again, from the computed table, we know 1 is the best solution which is larger than -8. Thus, no better solution can be found under **a** and **b** with respect to bound -8 and the current best solution remains -3.

(d) We reach nodes **a** and **c** through path $x = 0$ and $y = 0$. The local upper bound is $-3 - 0 - 0 = -3$. The minimum of the goal function **a** is 0 which is greater than -3.

The optimal solution is -3 with $x = 1, y = 0, z = 1$, and $w = 0$.

□

4.4 Discussion

A branch and bound/implicit enumeration based ILP solver can be characterized by the way it handles *search strategies*, *branching rules*, *bounding procedures* and *logical tests*. We will discuss these parameters in turn to analyze and explore possible improvements to FGILP.

Search Strategy

Search strategy refers to the selection of next node (subproblem) to process. There are two extreme search strategies. The first one is known as breadth first which always chooses nodes with best lower bound first. This approach tends to generate fewer nodes. The second one is depth first which chooses a best successor of the current node, if available, otherwise backtracks to the predecessor of the current node and continues the search. This strategy requires less storage space. FGILP uses the depth first strategy.

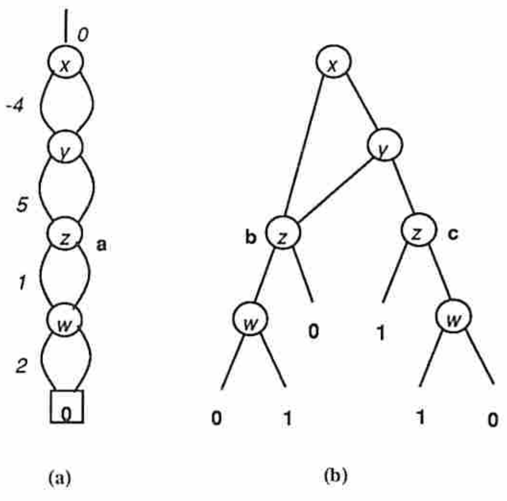


Figure 16: An example for the *minimize* operator.

Branching Rule
This parameter refers to the selection of next variable to branch. Various selection criteria which have been proposed use *priorities* [39], *penalties* [18, 50], *pseudo-cost* [5], and *integer infeasibility* [3] conditions. Currently, FGILP uses the same variable ordering as the one used to create EVBDDs because it simplifies the implementation. When the variable selected does not correspond to the variable ordering of EVBDD, operation *cofactor* (instead of $child_l$ and $child_r$) should be used.

Bounding Procedure
The most important component of a branch and bound method is the bounding procedure. The better the bound, the more pruning of the search space. The most frequently used bounding procedure is to use the linear programming method. Other procedures which can generate better bounds, but are more difficult to implement include the cutting planes, Lagrangian relaxation [21, 45], and disjunctive programming [4]. The bounding procedure used in FGILP is similar to the one proposed in [2]. In our experience, the most pruning takes place at line 3 of the code for *ilp_minimize*. This pruning rule however has two weak points. First, it is carried out on each constraint one at a time. Thus, it is only a 'local' method. Second, it can only be applied to a constraint which is in the Boolean form. The other bounding procedures described above are 'global' methods which are directly applicable to the inequality form.

Logical Tests
It is believed that logical tests may be as important as the bounding procedure [41]. In addition to partial ordering of variables, a particularly useful class of tests, when available, are those based on *dominance* [29, 30]. Currently, FGILP employs no logical tests. We believe that the inclusion of logical tests in FGILP will improve its performance.

Despite the fact that there are many improvements which can be made to FGILP, the performance of our ILP solver, as it is now, is already comparable to that of LINDO [44] which is one of the most widely used commercial tools [41] for solving ILP problems.

4.5 Experimental Results

FGILP has been implemented in C under the SIS environment. Table 1 shows our experimental results on ILP problems from MIPLIB [38]. It also shows the results of LINDO [44] (a commercial tool) on the same set of benchmarks. FGILP was run under SPARC station 2 (28.5 MIPS) with 64 MB memory while LINDO was run under SPARC station 10 (101.6 MIPS) with 384 MB memory. In Table 1, column 'Problem' lists the name of problems, columns 'Inputs' and 'Constraints' indicate the number of input variables and constraints, and columns 'FGILP' and 'LINDO' are the running time in seconds for obtaining the optimal solution shown in the last column.

FGILP provides three options for the order in which constraints are conjoined together. When all constraints are conjoined together, the order of conjunction will not affect the size of final EVBDD, but it does affect sizes of the intermediate EVBDDs. It is possible that an intermediate EVBDD has size much larger than the the final one. Our motivation for this ordering is to control the required memory space and save computation time. These three options are:

1. Based on the order of constraints in the input file. This provides users with direct control of the order.

2. EVBDDs with smallest size are conjoined first.

3. Constraints with the highest probability of not being satisfied are conjoined first.

The parameters used for the problems in Table 1 are summarized below:

1. Constraint conjunction order. Using the third option in problem 'p0201' led to much less space and computation time than the other two options. The same option led to more time in other problems due to the overhead of computing the probability of function values being 0. For consistency, results are reported for this option only.

2. EVBDD size of constraints. Without setting *c_size*, 'bm23' failed to finish and 'stein27' required 71.56 seconds. The run time reported in Table 1 for the above two problems were obtained by setting *c_size* = 8000 while others were run under no limitation of *c_size*. In general, this parameter has a significant impact on the run time. We believe that the correct value for *c_size* is dependent on the size of available memory for the machine.

3. Size of supporting variables. There was no limitation on the size of *n_supp*.

As results indicate, the performance of FGILP is comparable to that of LINDO. Since ILP is an NP-complete problem, it is quite normal that one solver outperforms the other solver in some problems while performs poorly in others.

FGILP, however, requires much more space than LINDO. As technology improves, memory is expected to become cheaper in cost and smaller in size. Increasing the available memory size will improve the speed of FGILP while will not benefit LINDO as much.

Problem	Inputs	Constraints	FGILP (sec)	LINDO (sec)	Optimal
bm23	27	20	1509.07	Error	34
lseu	89	28	Unable	186.44	1120
p0033	33	16	2.91	4.31	3089
p0040	40	23	0.98	0.37	62027
p0201	201	133	765.48	529.46	7615
stein15	15	36	1.44	1.66	9
stein27	27	118	51.24	120.03	18
stein9	9	13	0.13	0.31	5

Table 1: Experimental results of ILP problems.

5 Spectral Techniques

The main purpose of spectral methods [52] is to transform Boolean functions from Boolean domain into another domain so that the transformed functions have more compact implementations. It was conjectured that these methods would provide a unified approach to the synthesis of analog and digital circuits [51]. Although spectral techniques have solid theoretical foundation, until recently they did not receive much attention due to their expensive computation times. With new applications in fault diagnosis, spectral techniques have recently invoked interest [28]. New computational methods have been proposed. In [20], a technique based on arrays of disjoint ON- and DC-cubes is proposed. In [51], a cube-based algorithm for linear decomposition in spectral domain is proposed.

Recently, [13] proposed two OBDD-based methods for computing spectral coefficients. The first method was to treat integers as bit vectors and integer operations as the corresponding Boolean operations. The main disadvantage of this representation is that arithmetic operations must be performed bit by bit which is very time consuming. The second method employed a variation of OBDD called Multi-Terminal Binary Decision Diagrams (MTBDDs) [12] which are exactly the same as the flattened form of EVBDDs. The major problem with using MTBDDs is the space requirement when the number of distinct coefficients is large.

We propose EVBDD-based algorithms for computing Hadamard (sometimes termed Walsh-Hadamard) spectrum [52]. In our approach, the matrix representing Boolean function values used in spectral methods is represented by EVBDDs. This takes advantage of compact representation through subgraph sharing. The transformation matrix and the transformation itself are carried out through EVBDD operations. Thus, the benefit of caching computational results is achieved. The algorithms presented here include both the transformation from Boolean domain to spectral domain and the operations within the spectral domain itself.

The Hadamard transformation is carried out in the following form:

$$T^n \, Z^n \;=\; R^n, \tag{5}$$

where T^n is a $2^n \times 2^n$ matrix called *transformation matrix*,

Z^n is a $2^n \times 1$ matrix which is the truth table representation ofa Boolean function,

R^n is a $2^n \times 1$ matrix which is the spectral coefficients of a Boolean function.

Different transformation matrices generate different spectra. Here, we use the Hadamard transformation matrix [52] which has a recursive structure as follows:

$$T^n = \begin{bmatrix} T^{n-1} & T^{n-1} \\ T^{n-1} & -T^{n-1} \end{bmatrix}$$

and

$$T^0 = 1.$$

Example 5.1 The spectrum of function $f(x, y) = x \oplus y$ is computed as

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 0 \\ -2 \end{bmatrix}.$$

The *order* of each spectral coefficient r_i (i^{th} row of R^n) is the number of 1's in the binary representation of $i, 0 \leq i \leq 2^n - 1$. For example, r_{00} is the zeroth-order coefficient, r_{01} and r_{10} are the first-order coefficients, and r_{11} is the second-order coefficient. Let $R_i^n = \{$ multi-set of the absolute value of r_k's where r_k is an i^{th}-order coefficient of $R^n\}$, $0 \leq i \leq n$. In Example 5.1, $R_0^2 = \{2\}$, $R_1^2 = \{0, 0\}$, and $R_2^2 = \{2\}$. An operation on f and its R^n which does not modify the sets R_i^n is referred as an *invariance* operation. Given a function $f(x_1, \ldots, x_n)$ with spectrum R^n, three invariance operations on f and R^n are as follows (formal proofs may be found in [19]):

1. Input negation invariance: if x_i is negated, then the new spectrum $R^{n'}$ is formed by $r'_k = -r_k$ where the i^{th} bit of k is 1, and $r'_k = r_k$ otherwise.

2. Input permutation invariance: if input variables x_i and x_j are exchanged, then the new spectrum is formed by exchanging r_k's and r_l's where $k - 2^i = l - 2^j$. That is, the i^{th} and j^{th} bits of k and l are $\langle 1, 0\rangle$ and $\langle 0, 1\rangle$, respectively, while all other bits of k and l are the same.

3. Output negation invariance: if f is negated, the $R^{n'}$ is formed by replacing all r_k by $-r_k$.

Lemma 5.1 Two Boolean functions are input-negation, input-permutation, and output-negation equivalent (NPN-equivalent) only if their R_i^n's are equivalent.

Proof: For negation equivalent, since $| r_k |=| -r_k |$, replacing r_k by $-r_k$ will not change R_i^n. For permutation equivalent, since r_k's and r_l's have the same order, exchanging their values is equivalent to permuting values in multi-set R_i^n.

□

With this property, R_i^n can be used as a filter to improve performance in a Boolean matching algorithm as presented in [33].

5.1 Spectral EVBDD (SPBDD)

The major problem with Equation 5 is that all matrices involved are of size 2^n. Therefore, only functions with a small number of inputs can be computed. We overcome this difficulty by using EVBDDs to represent both Z^n and R^n. When an EVBDD is used to represent R^n, we refer to it as SPectral EVBDD, or SPBDD for short. The difference between EVBDDs and SPBDDs is in the semantics, not the syntax. A path in EVBDDs corresponds to a function value while a path in SPBDDs corresponds to a spectral coefficient. The matrix multiplication by T^n is implicitly carried out in the transformation from Z^n to R^n (i.e., from EVBDD to SPBDD).

We define Z^n and R^n recursively as follows:

$$Z^n = \begin{bmatrix} Z_0^{n-1} \\ Z_1^{n-1} \end{bmatrix}$$

and

$$R^n = \begin{bmatrix} R_0^{n-1} \\ R_1^{n-1} \end{bmatrix}.$$

Then, Equation 5 can be rewritten as:

$$\begin{bmatrix} T^{n-1} & T^{n-1} \\ T^{n-1} & -T^{n-1} \end{bmatrix} \begin{bmatrix} Z_0^{n-1} \\ Z_1^{n-1} \end{bmatrix} = \begin{bmatrix} T^{n-1}Z_0^{n-1} + T^{n-1}Z_1^{n-1} \\ T^{n-1}Z_0^{n-1} - T^{n-1}Z_1^{n-1} \end{bmatrix} = \begin{bmatrix} R_0^{n-1} \\ R_1^{n-1} \end{bmatrix}. \qquad (6)$$

Equation 6 then is implemented through EVBDD [2] as :

$$\tau(\langle x, Z_1^{n-1}, Z_0^{n-1}\rangle) = \langle x, \tau(Z_0^{n-1}) - \tau(Z_1^{n-1}), \tau(Z_0^{n-1}) + \tau(Z_1^{n-1})\rangle, \qquad (7)$$

$$\tau(1) = 1, \qquad (8)$$

$$\tau(0) = 0. \qquad (9)$$

where τ is the transformation function which converts an EVBDD representing a Boolean function to an SPBDD. To show the above equations correctly implement Equation 6, we prove the following lemma.

Lemma 5.2 Let τ : EVBDD $\rightarrow$ SPBDD as defined in Equations 7-9, then τ implements T, that is, $\tau(f^n) = T^n f^n$, where f^n is an n-input function. Or, equivalently, $\tau(\langle x_n, Z_1^{n-1}, Z_0^{n-1}\rangle) = \langle x_n, R_1^{n-1}, R_0^{n-1}\rangle$.

Proof: By induction on input size n.
Base: $n = 0$, $f^0 = c$ is a constant function, $c \in \{0, 1\}$,

$\tau(c) = c$ by Equations 8 and 9,

$T^0[c] = [1][c] = [c]$.

Induction hypothesis: assume it is true for $0 \le n \le k-1$, $\tau(f^n) = T^n f^n$.
Induction:

$$\begin{aligned} & \tau(\langle x^k, Z_1^{k-1}, Z_0^{k-1}\rangle) \\ = & \langle x^k, \tau(Z_0^{k-1}) - \tau(Z_1^{k-1}), \tau(Z_0^{k-1}) + \tau(Z_1^{k-1})\rangle && \text{by Equation 7} \\ = & \langle x^k, T^{k-1}Z_0^{k-1} - T^{k-1}Z_1^{k-1}, T^{k-1}Z_0^{k-1} + T^{k-1}Z_1^{k-1}\rangle && \text{by induction hypothesis} \\ = & \langle x^k, R_1^{k-1}, R_0^{k-1}\rangle && \text{by Equation 6} \end{aligned}$$

[2]For the sake of readability, we use flattened EVBDD in this section.

Example 5.2 The exclusive-or function in Example 5.1 is redone in terms of EVBDD representation.

$$\begin{aligned}
& \tau(\langle x, \langle y,0,1\rangle, \langle y,1,0\rangle\rangle) \\
= \; & \langle x, \tau(\langle y,1,0\rangle) - \tau(\langle y,0,1\rangle), \tau(\langle y,1,0\rangle) + \tau(\langle y,0,1\rangle)\rangle \\
= \; & \langle x, \langle y, \tau(0)-\tau(1), \tau(0)+\tau(1)\rangle - \langle y, \tau(1)-\tau(0), \tau(1)+\tau(0)\rangle, \\
& \quad \langle y, \tau(0)-\tau(1), \tau(0)+\tau(1)\rangle + \langle y, \tau(1)-\tau(0), \tau(1)+\tau(0)\rangle\rangle \\
= \; & \langle x, \langle y,-1,1\rangle - \langle y,1,1\rangle, \langle y,-1,1\rangle + \langle y,1,1\rangle\rangle \\
= \; & \langle x, \langle y,-2,0\rangle, \langle y,0,2\rangle\rangle
\end{aligned}$$

Pseudo code *evbdd_to_spbdd(ev, level, n)* is the implementation of Equation 6. Because of the following situation, this procedure requires *level* and n as parameters:

$$\begin{aligned}
\tau(\langle x,z,z\rangle) &= \langle x, \tau(z)-\tau(z), \tau(z)+\tau(z)\rangle, \\
&= \langle x, 0, 2\times\tau(z)\rangle.
\end{aligned}$$

In reduced EVBDD, $\langle x,z,z\rangle$ will be reduced to z while $\langle x,0,2\times\tau(z)\rangle$ cannot be reduced in SPBDD. We need to keep track of the current *level* so that when the index of the root node ev is greater than *level*, we generate $\langle level, 0, 2\times\tau(ev)\rangle$ (lines 3-8).

```
$evbdd\_to\_spbdd(ev, level, n)$
{
1       if ($level == n$) return $ev$;
2       if ($ev == 0$) return 0;
3       if ($index(ev) > level$) {
4           $sp = evbdd\_to\_spbdd(ev, level+1, n)$;
5           $left = 0$;
6           $right = evbdd\_add(sp, sp)$
7           return $new\_evbdd(level, left, right)$;
8       }
9       $sp_l = evbdd\_to\_spbdd(child_l(ev), level+1, n)$;
10      $sp_r = evbdd\_to\_spbdd(child_r(ev), level+1, n)$;
11      $left = evbdd\_sub(sp_r, sp_l)$;
12      $right = evbdd\_add(sp_r, sp_l)$;
13      return $new\_evbdd(level, left, right)$;
}
```

5.2 Boolean Operations in Spectral Domain

In this section, we show how to perform Boolean operations in SPBDDs. We first present the algorithm for performing Boolean conjunction in SPBDDs by the following definition.

Definition 5.1 Given two SPBDDs f and g, the operator $\wedge$ is carried out in the following way:

$$\begin{aligned}
f \wedge g &= f \times g, \quad \text{if } f \text{ and } g \text{ are terminal nodes,} \\
\langle x, f_l, f_r\rangle \wedge \langle x, g_l, g_r\rangle &= \langle x, (f_l \wedge g_r + f_r \wedge g_l)/2, (f_l \wedge g_l + f_r \wedge g_r)/2\rangle, \quad \text{otherwise.}
\end{aligned}$$

The following lemma and theorem prove that the above definition carries out the Boolean conjunction in SPBDDs.

Lemma 5.3 $(f+g) \wedge (i+j) = f \wedge i + f \wedge j + g \wedge i + g \wedge j$, where $f, g, i, j \in$ SPBDD. (Note that $+$'s may be replaced by $-$'s.)

Proof: by induction on the size of inputs.
Base: $n = 0$, f, g, i, and j are constant functions.

$$\begin{array}{rll} & (f+g) \wedge (i+j) & \text{from LHS} \\ = & (f+g) \times (i+j) & \text{by Def. 5.1} \\ = & f \times i + f \times j + g \times i + g \times j & \text{by distributive laws of } \times \text{ and } + \end{array}$$

$$\begin{array}{rll} & f \wedge i + f \wedge j + g \wedge i + g \wedge j & \text{from RHS} \\ = & f \times i + f \times j + g \times i + g \times j & \text{by Def. 5.1} \end{array}$$

Induction hypothesis (IH): assume it is true for $0 \le n < k$.
Induction: let $f = \langle x_k, f_l, f_r \rangle$, $g = \langle x_k, g_l, g_r \rangle$, $i = \langle x_k, i_l, i_r \rangle$, and $j = \langle x_k, j_l, j_r \rangle$, where f_l, f_r, g_l, g_r, i_l, i_r, j_l, and j_r have at most $k-1$ inputs.

$$\begin{array}{rll} & (\langle x_k, f_l, f_r \rangle + \langle x_k, g_l, g_r \rangle) \wedge (\langle x_k, i_l, i_r \rangle + \langle x_k, j_l, j_r \rangle) & \text{from LHS} \\ = & \langle x_k, f_l + g_l, f_r + g_r \rangle \wedge \langle x_k, i_l + j_l, i_r + j_r \rangle & \text{by + in SPBDD} \\ = & \langle x_k, ((f_l + g_l) \wedge (i_r + j_r) + (f_r + g_r) \wedge (i_l + j_l))/2, & \\ & \quad \langle ((f_l + g_l) \wedge (i_l + j_l) + (f_r + g_r) \wedge (i_r + j_r))/2 \rangle & \text{by Def 5.1} \\ = & \langle x_k, (f_l \wedge i_r + f_l \wedge j_r + g_l \wedge i_r + g_l \wedge j_r + f_r \wedge i_l + f_r \wedge j_l + g_r \wedge i_l + g_r \wedge j_l)/2, & \\ & \quad (f_l \wedge i_l + f_l \wedge j_l + g_l \wedge i_l + g_l \wedge j_l + f_r \wedge i_r + f_r \wedge j_r + g_r \wedge i_r + g_r \wedge j_r)/2 \rangle & \text{by IH} \end{array}$$

$$\begin{array}{rll} & \langle x_k, f_l, f_r \rangle \wedge \langle x_k, i_l, i_r \rangle + \langle x_k, f_l, f_r \rangle \wedge \langle x_k, j_l, j_r \rangle + & \\ & \langle x_k, g_l, g_r \rangle \wedge \langle x_k, i_l, i_r \rangle + \langle x_k, g_l, g_r \rangle \wedge \langle x_k, j_l, j_r \rangle & \text{from RHS} \\ = & \langle x_k, (f_l \wedge i_r + f_r \wedge i_l)/2, (f_l \wedge i_l + f_r \wedge i_r)/2 \rangle + & \\ & \langle x_k, (f_l \wedge j_r + f_r \wedge j_l)/2, (f_l \wedge j_l + f_r \wedge j_r)/2 \rangle + & \\ & \langle x_k, (g_l \wedge i_r + g_r \wedge i_l)/2, (g_l \wedge i_l + g_r \wedge i_r)/2 \rangle + & \\ & \langle x_k, (g_l \wedge j_r + g_r \wedge j_l)/2, (g_l \wedge j_l + g_r \wedge j_r)/2 \rangle & \text{by Def 5.1} \\ = & \langle x_k, (f_l \wedge i_r + f_r \wedge i_l)/2 + (f_l \wedge j_r + f_r \wedge j_l)/2 + & \\ & \quad (g_l \wedge i_r + g_r \wedge i_l)/2 + (g_l \wedge j_r + g_r \wedge j_l)/2, & \\ & \quad (f_l \wedge i_l + f_r \wedge i_r)/2 + (f_l \wedge j_l + f_r \wedge j_r)/2 + & \\ & \quad (g_l \wedge i_l + g_r \wedge i_r)/2 + (g_l \wedge j_l + g_r \wedge j_r)/2 \rangle & \text{by + in SPBDD} \\ = & \langle x_k, (f_l \wedge i_r + f_r \wedge i_l + f_l \wedge j_r + f_r \wedge j_l + g_l \wedge i_r + g_r \wedge i_l + g_l \wedge i_r + g_r \wedge j_l)/2, & \\ & \quad (f_l \wedge i_l + f_r \wedge i_r + f_l \wedge j_l + f_r \wedge j_r + g_l \wedge i_l + g_r \wedge i_r + g_l \wedge j_l + g_r \wedge j_r)/2 \rangle & \text{by + and /} \end{array}$$

□

Theorem 5.1 Given two Boolean functions f and g represented in EVBDDs, $\tau(f \cdot g) = \tau(f) \wedge \tau(g)$, where $\cdot$ is the conjunction operator in Boolean domain.

Proof: by induction on the size of inputs.
Base: $n = 0$, f and g are terminal nodes or constant functions.

$\tau(f \cdot g) = \tau(f \times g) = f \times g.$

$\tau(f) \wedge \tau(g) = f \wedge g = f \times g.$

Induction hypothesis (IH): assume it is true for $0 \leq n \leq k-1$.
Induction: let $f = \langle x_k, f_l, f_r\rangle$ and $g = \langle x_k, g_l, g_r\rangle$ where f_l , f_r, g_l and g_r have at most $k-1$ inputs.

$$\begin{array}{lll}
 & \tau(\langle x_k, f_l, f_r\rangle \cdot \langle x_k, g_l, g_r\rangle) & \text{from LHS} \\
= & \tau(\langle x_k, f_l \cdot g_l, f_r \cdot g_r\rangle) & \text{by } \cdot \text{ in EVBDD} \\
= & \langle x_k, \tau(f_r \cdot g_r) - \tau(f_l \cdot g_l), \tau(f_r \cdot g_r) + \tau(f_l \cdot g_l)\rangle & \text{by } \tau \text{ operation} \\
= & \langle x_k, \tau(f_r) \wedge \tau(g_r) - \tau(f_l) \wedge \tau(g_l), \tau(f_r) \wedge \tau(g_r) + \tau(f_l) \wedge \tau(g_l)\rangle & \text{by IH}
\end{array}$$

$$\begin{array}{lll}
 & \tau(x_k, f_l, f_r\rangle) \wedge \tau(\langle x_k, g_l, g_r\rangle) & \text{(RHS)} \\
= & \langle x_k, \tau(f_r) - \tau(f_l), \tau(f_r) + \tau(f_l)\rangle \wedge \langle x_k, \tau(g_r) - \tau(g_l), \tau(g_r) + \tau(g_l)\rangle & (\tau) \\
= & \langle x_k, ((\tau(f_r) - \tau(f_l)) \wedge (\tau(g_r) + \tau(g_l)) + (\tau(f_r) + \tau(f_l)) \wedge (\tau(g_r) - \tau(g_l)))/2, & (\wedge) \\
 & \quad ((\tau(f_r) - \tau(f_l)) \wedge (\tau(g_r) - \tau(g_l)) + (\tau(f_r) + \tau(f_l)) \wedge (\tau(g_r) + \tau(g_l)))/2\rangle & \\
= & \langle x_k, (\tau(f_r) \wedge \tau(g_r) + \tau(f_r) \wedge \tau(g_l) - \tau(f_l) \wedge \tau(g_r) - \tau(f_l) \wedge \tau(g_l) + & \text{(LEMMA 5.3)} \\
 & \quad \tau(f_r) \wedge \tau(g_r) - \tau(f_r) \wedge \tau(g_l) + \tau(f_l) \wedge \tau(g_r) - \tau(f_l) \wedge \tau(g_l))/2, & \\
 & \quad (\tau(f_r) \wedge \tau(g_r) - \tau(f_r) \wedge \tau(g_l) - \tau(f_l) \wedge \tau(g_r) + \tau(f_l) \wedge \tau(g_l) + & \\
 & \quad \tau(f_r) \wedge \tau(g_r) + \tau(f_r) \wedge \tau(g_l) + \tau(f_l) \wedge \tau(g_r) + \tau(f_l) \wedge \tau(g_l))/2\rangle & \\
= & \langle x_k, \tau(f_r) \wedge \tau(g_r) - \tau(f_l) \wedge \tau(g_l), \tau(f_r) \wedge \tau(g_r) + \tau(f_l) \wedge \tau(g_l)\rangle & (+, -, /)
\end{array}$$

$\square$

Other Boolean operations in SPBDDs are carried out by the following equations:

$$\begin{array}{rcll}
f \vee g & = & f + g - f \wedge g, & (10) \\
f \oplus g & = & f + g - 2 \times (f \wedge g), & (11) \\
\bar{f} & = & J^n - f, & (12) \\
 & & & (13)
\end{array}$$

where $\vee$, $\oplus$, and $\bar{}$, are *or*, *xor*, and *not* in spectral domain (SPBDD),

$J^n = [2^n, 0, \ldots, 0]^t$.

These operations $\vee, \oplus$, and $\bar{}$ are from [28] with minor modification to match the τ operation. Operations $+$, $-$, and $\times$ are carried out in the same way as in EVBDDs (Sec. 2.2).

5.3 Experimental Results

Table 2 shows the results of some benchmarks represented in both EVBDD and SPBDD forms. Column 'EVBDD' depicts the size and time required for representing and constructing a circuit using EVBDDs while column 'SPBDD' depicts the size of SPBDDs and the time required for converting from EVBDDs to SPBDDs. In average, the ratio of the number of nodes required for representing SPBDDs over that of EVBDDs is 6.8, and the ratio of the conversion time for SPBDDs over the construction time of EVBDDs is 41.

One application of spectral coefficients is that they can be used as a filter for pruning search space in the process of Boolean matching [13, 33]. The performance of a filter depends on its capability of pruning (effectiveness) and its computation time (cost). Experimental results of [13] show that this filter is quite good because this filter rejected all unmatchable functions that were encountered. However, according to results of Table 2, this filter is relatively expensive to compute when comparing with other filters [33]. We believe that this filter should be used only after other filters have failed to prune.

	In	Out	EVBDD		SPBDD	
			Size	Time	Size	Time
9symml	9	1	24	0.17	39	0.15
c8	28	18	142	0.09	1310	4.79
cc	21	20	76	0.02	951	1.59
cmb	16	4	35	0.04	88	0.17
comp	32	3	145	0.10	809	35.31
cordic	23	2	84	0.09	208	1.35
count	35	16	233	0.14	1197	8.39
cu	14	11	66	0.04	266	0.67
f51m	8	8	65	0.08	295	0.34
lal	26	19	99	0.07	1111	2.99
mux	21	1	86	0.11	144	0.99
my-adder	33	17	456	0.62	5043	14.1
parity	16	1	16	0.02	30	0.14
pcle	19	9	94	0.03	640	1.18
pcler8	27	17	139	0.05	1617	4.53
pm1	16	13	57	0.01	465	0.74
sct	19	15	101	0.07	1295	2.42
ttt2	24	21	173	0.23	2046	5.35
unreg	36	16	134	0.04	816	5.78
x2	10	7	41	0.01	305	0.42
z4ml	7	4	36	0.09	83	0.13
b9	41	21	212	0.10	1809	12.08
alu2	10	6	248	0.69	889	2.69
alu4	14	8	1166	3.78	5913	66.12
term1	34	10	614	1.39	5643	104.35
apex7	49	37	665	0.35	3017	52.51
cht	47	36	133	0.08	452	5.19
example2	85	66	752	0.22	5163	23.14

Table 2: Experimental results of SPBDDs.

6 Multiple-Output Boolean Functions

While a problem with finite domain can be solved by conversion to Boolean functions, a problem related to multiple-output Boolean functions can also be solved by interpreting them as the bit representation of an integer function. For example, a multiple-output Boolean function $\langle f_0, \ldots, f_{m-1} \rangle$ can be transformed to an integer function F by $F = 2^{m-1} f_0 + \ldots + 2^0 f_{m-1}$. Based on this formulation, we present the application of EVBDDs to performing function decomposition of multiple-output Boolean functions.

The motivation for using function decomposition in logic synthesis is to reduce the complexity of the problem by a divide-and-conquer paradigm: A function is decomposed into a set of smaller functions such that each of them is easier to synthesize.

The function decomposition theory was studied by Ashenhurst [1], Curtis [15], and Roth and Karp [42]. In Ashenhurst-Curtis method, functions are represented by Karnaugh maps and the decomposability of functions are determined from the number of distinct columns in the map. In Roth-Karp method, functions are represented by cubes and the decomposability of functions are determined from the cardinality of compatible classes. Recently, researchers [6, 11, 34, 43] have used OBDDs to determine decomposability of functions. However, most of these works only consider single-output Boolean functions.

In this section, we start with definitions of function decomposition and cut_sets in EVBDD representation. Based on the concept of cut_sets, we develop an EVBDD-based disjunctive decomposition algorithm.

6.1 Definitions

Definition 6.1 A pseudo Boolean function $f(x_0, \ldots, x_{n-1})$ is said to be *decomposable* under *bound set* $\{x_0, \ldots, x_{i-1}\}$ and *free set* $\{x_i, \ldots x_{n-1}\}$, $0 < i < n$, if f can be transformed to $f'(g_0(x_0, \ldots, x_{i-1}), \ldots, g_j(x_0, \ldots, x_{i-1}), x_i, \ldots, x_{n-1})$ such that the number of inputs to f' is smaller than that of f. If j equals 1, then it is *simple decomposable.*

Note that since inputs to a pseudo Boolean function are Boolean variables, function g_k's are Boolean functions. Here, we consider only disjunctive decomposition (the intersection of bound set and free set is empty).

Definition 6.2 Given an EVBDD $\langle c, \mathbf{v} \rangle$ representing $f(x_0, \ldots, x_{n-1})$ with variable ordering $x_0 < \ldots < x_{n-1}$ and bound set $B = \{x_0, \ldots, x_i\}$, we define

$$cut\_set(\langle c, \mathbf{v} \rangle, B) = \{\langle c', \mathbf{v}' \rangle \mid \langle c', \mathbf{v}' \rangle = eval(\langle c, \mathbf{v} \rangle, j), 0 \le j < 2^i\}.$$

For the sake of readability, we use the flattened form of EVBDDs in this section.

Example 6.1 Given a function f as shown in Fig. 17 with bound set $B = \{x_0, x_1, x_2\}$, $cut\_set(f, B) = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$.

□

If an EVBDD is used to represent a Boolean function, then each node in the cut_set corresponds to a distinct column in the Ashenhurst-Curtis method [1, 15] and a compatible class in the Roth-Karp decomposition algorithm [42].

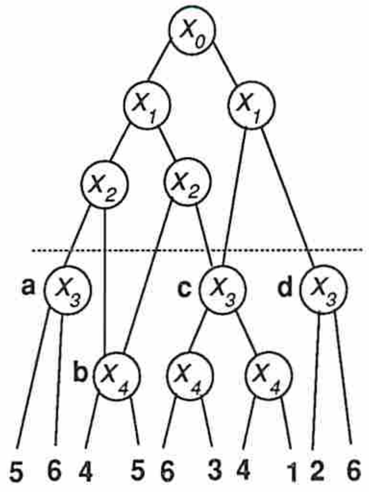


Figure 17: An example of cut_set in EVBDD.

6.2 Disjunctive Decomposition

Disjunctive decomposition algorithms for single-output Boolean functions based on OBDDs can be found in [11, 34, 43]. Here, we present an EVBDD-based disjunctive decomposition algorithm which is generalization of the previous work.

Algorithm D: Given a function f represented in an EVBDD $\mathbf{v_f}$ and a bound set B, a disjunctive decomposition with respect to B is carried out in the following steps:

1. Compute the cut_set with respect to B. Let $cut\_set(\mathbf{v}, B) = \{\mathbf{u_0}, \ldots, \mathbf{u_{k-1}}\}$.

2. Encode each node in the cut_set by $\lceil \log_2 k \rceil = j$ bits.

3. Construct $\mathbf{v_m}$'s to represent g_m's, $0 \leq m < j$:
   Replace each node $\mathbf{u}$ with encoding $b_0, \ldots, b_{j-1}$ in the cut_set by terminal node $\mathbf{b_m}$.

4. Construct $\mathbf{v_{f'}}$ to represent function f':
   Replace the top part of $\mathbf{v_f}$ by a new top on variables $g_0, \ldots, g_{j-1}$ such that $eval(\mathbf{v_{f'}}, l) = \mathbf{u_l}$ for $0 \leq l < k-1$, $eval(\mathbf{v_{f'}}, l) = \mathbf{u_{k-1}}$ for $k-1 \leq l < 2^j$.

The correctness of this algorithm can be intuitively argued as follows: For any input pattern m in the bound set, the evaluation of m in function f will result at a node in the cut_set with encoding e. The evaluation of m on the g_l functions should thus produce the function values e. The evaluation of e in function f' should also end at the same node in the cut_set. Thus, the composition of f' and g_l's becomes equivalent to f.

In step 2 of Algorithm D, we use an arbitrary input encoding which is not unique. Different encodings will result in different decompositions. Furthermore, when $k < 2^j$, not every j-bit pattern is used in the encoding of the cut_set; Function g_l's can never generate function values which correspond to the patterns absent from the encoding, thus we can assign these patterns to any node in the cut_set. In step 4, we assign them to the last node in the cut_set ($\mathbf{u_{k-1}}$). Alternatively, we could have made them into explicit don't-cares.

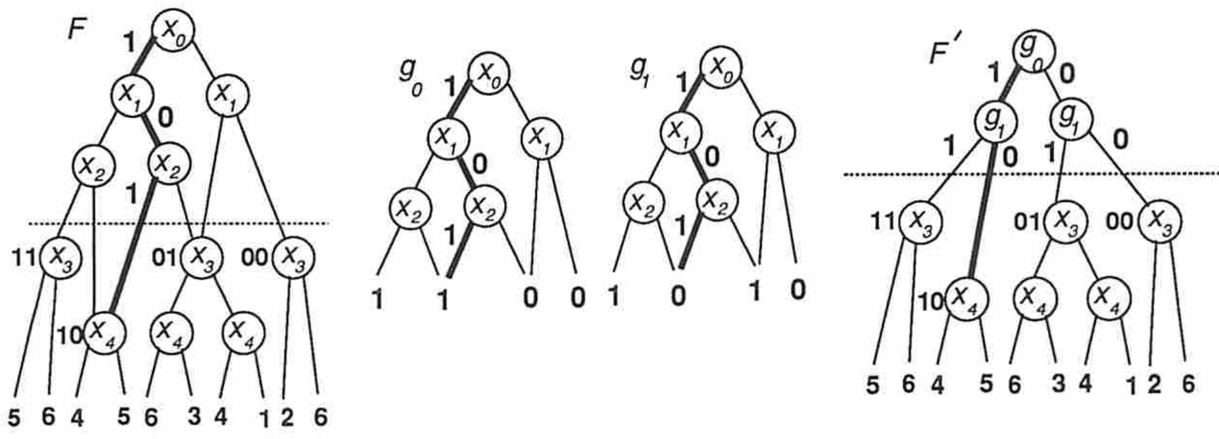


Figure 18: An example of disjunctive decomposition in EVBDD.

Lemma 6.1 Given an EVBDD $\mathbf{v_f}$ with variable ordering $x_0 < \ldots < x_{n-1}$ representing $f(x_0, \ldots, x_{n-1})$, a bound set $B = \{x_0, \ldots, x_{i-1}\}$ and $cut\_set(\mathbf{v_f}, B) = \{\mathbf{u_0}, \ldots, \mathbf{u_{k-1}}\}$, if Algorithm D returns EVBDDs $\mathbf{v_{f'}}, \mathbf{v_{g_0}}, \ldots, \mathbf{v_{g_{j-1}}}$, then

$$f(x_0, \ldots, x_{n-1}) = f'(g_0(x_0, \ldots, x_{i-1}), \ldots, g_{j-1}(x_0, \ldots, x_{i-1}), x_i, \ldots, x_{n-1})$$

where $f', g_0, \ldots, g_{j-1}$ are the functions denoted by $\mathbf{v_{f'}}, \mathbf{v_{g_0}}, \ldots, \mathbf{v_{g_{j-1}}}$, respectively.

Proof: Consider the behavior of an input pattern $\langle b_0, \ldots, b_{i-1}\rangle$ on $\mathbf{v_f}, \mathbf{v_{f'}}$ and $\mathbf{v_{g_m}}$'s. Suppose $\mathbf{u_m}$ is the node we reach in $\mathbf{v_f}$ using the input pattern, that is, $eval(\mathbf{v_f}, \langle b_0, \ldots, b_{i-1}\rangle) = \mathbf{u_m}$, where $\mathbf{u_m} = f(b_0, \ldots, b_{i-1}, x_i, \ldots, x_{n-1})$. Since $\mathbf{u_m}$ has been replaced by the l^{th} bit of m in $\mathbf{v_{g_l}}$, $eval(\mathbf{v_{g_l}}, \langle b_0, \ldots, b_{i-1}\rangle) = b_{m_l}$, that is, $g_l(b_0, \ldots, b_{i-1}) = b_{m_l}$. Because of the way we construct $\mathbf{v_{f'}}$, $eval(\mathbf{v_{f'}}, \langle b_{m_0}, \ldots, b_{m_{j-1}}\rangle) = \mathbf{u_m}$, that is,
$f'(b_{m_0}, \ldots, b_{m_{j-1}}, x_i, \ldots, x_{n-1}) = \mathbf{u_m} = f(b_0, \ldots, b_{i-1}, x_i, \ldots, x_{n-1})$.
Thus, $f'(g_0(b_0, \ldots, b_{i-1}), \ldots, g_{j-1}(b_0, \ldots, b_{i-1}), x_i, \ldots, x_{n-1}) = f(b_0, \ldots, b_{i-1}, x_i, \ldots, x_{n-1})$, for any input pattern in the bound set.

□

Example 6.2 Fig. 18 shows an example of disjunctive decomposition in EVBDDs. The evaluation of the input pattern $x_0 = 1$, $x_1 = 0$, and $x_2 = 1$ in function F will end at the leftmost x_4-node which has encoding 10. The evaluation of the same input pattern in functions g_0 and g_1 would produce function values 1 and 0. Then, with g_0 being 1 and g_1 being 0 in function F', it would also end at the leftmost x_4-node.

□

When an EVBDD is used to represent a Boolean function, Algorithm D corresponds to a disjunctive decomposition algorithm for Boolean functions; when an EVBDD represents an integer function, then Algorithm D can be used as a disjunctive decomposition algorithm for multiple-output Boolean functions as shown in the following example.

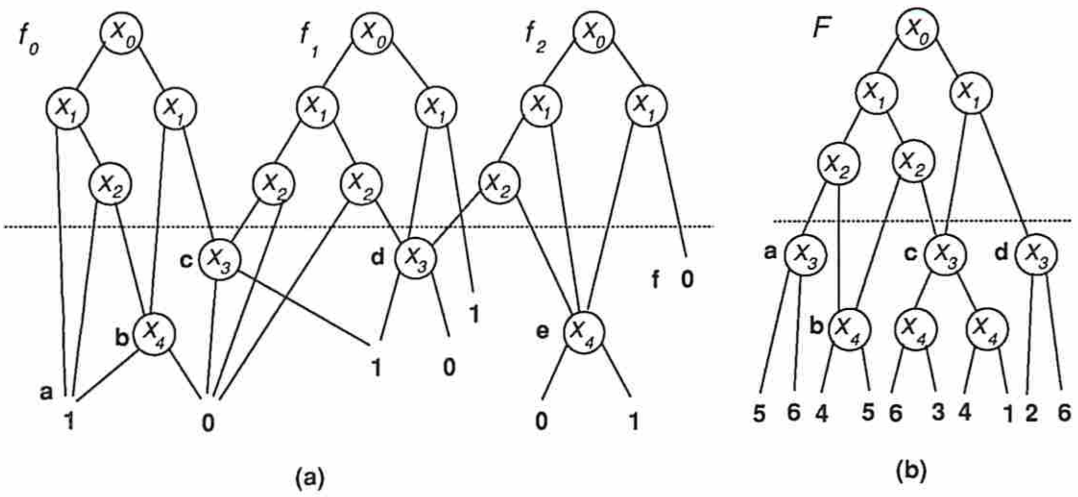


Figure 19: Representation of multiple-output functions.

Example 6.3 A 3-output Boolean function as shown in Fig 19 (a) can be converted into an integer function as shown in Fig. 19 (b) through $f = 4f_0 + 2f_1 + f_2$. The application of Algorithm D on F is the one shown in the previous example. After applying the synthesis paradigm described in Sect. 3.1 on F', we can convert f back to a 3-output Boolean function f'_0, f'_1, and f'_2.

□

7 Conclusions

We 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 compactness and canonical properties, OBDDs and EVBDDs have been shown effective for handling verification problems (e.g., section 3); because of additive property, EVBDDs are also useful for solving optimization problems (e.g., section 4). Boolean values are a subdomain of integer values and Boolean operations are special cases of arithmetic operation. With this interpretation, EVBDDs are particular useful for applications which require both Boolean and integer operations. Examples are shown in performing spectral transformations (e.g., section 5) and representing multiple output Boolean functions (e.g., section 6).

EVBDDs could be used for other applications. For example, [12] uses MTBDDs to represent general matrices and to perform matrix operation such as standard and Strassen matrix multiplication, and LU factorization; [25] uses OBDDs to implement a symbolic algorithm for maximum flow in 0-1 network.

References

[1] R. L. Ashenhurst, "The decomposition of switching functions," *Ann. Computation Lab.*, Harvard University, vol. 29, pp. 74-116, 1959.

[2] E. Balas, "An additive algorithm for solving linear programs with zero-one variables," *Operations Research*, 13 (4), pp. 517-546, 1965.

[3] E. Balas, "Bivalent programming by implicit enumeration," *Encyclopedia of Computer Science and Technology* Vol.2, J. Belzer, A.G. Holzman and Kent, eds., M. Dekker, New York, pp. 479-494, 1975.

[4] E. Balas, "Disjunctive programming," *Annals of Discrete Mathematics 5*, North-Holland, pp. 3-51, 1979.

[5] M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere and O. Vincent, "Experiments in mixed-integer linear programming," *Math. Programming 1*, pp. 76-94, 1971.

[6] M. Beardslee, B. Lin, and A. Sangiovanni-Vincentelli, "Communication based logic partitioning," *Proc. of the European Design Automation Conf.*, pp. 32-37, 1992.

[7] V. J. Bowman, Jr. and J. H. Starr, "Partial orderings in implicit enumeration," *Annals of Discrete Mathematics*, 1, North-Holland, pp. 99-116, 1977.

[8] 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.

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

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

[11] S-C. Chang and M. Marek-Sadowska, "Technology mapping via transformations of function graph," *Proc. International Conf. on Computer Design*, pp. 159-162, 1992.

[12] 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.

[13] 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.

[14] 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.

[15] H. A. Curtis, "A new approach to the design of switching circuits," Princeton, N.J., Van Nostrand, 1962.

[16] R. J. Dakin, "A tree-search algorithm for mixed integer programming problems," *Comput. J. 9*, pp. 250-255, 1965.

[17] G. B. Dantzig, *Linear Programming and Extensions*, Princeton, N. J.: Princeton University Press, 1963.

[18] N. J. Driebeck, "An algorithm of the solution of mixed integer programming problems," *Management Science 12*, pp. 576-587, 1966.

[19] C. R. Edwards, "The application of the Rademacher-Walsh transform to Boolean function classification and threshold-logic synthesis," *IEEE Transactions on Computers*, C-24, pp. 48-62, 1975.

[20] B. J. Falkowski, I. Schafer and M. A. Perkowski, "Effective computer methods for the calculation of Rademacher-Walsh spectrum for completely and incompletely specified Boolean functions," *IEEE Transaction on Computer-Aided Design*, Vol. 11, No. 10, pp. 1207-1226, Oct. 1992.

[21] M. L. Fisher, "The Lagrangian relaxation method for solving integer programming problems," *Management Science*, pp. 1-18, 1981.

[22] M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," Freeman, San Francisco, 1979.

[23] R. E. Gomory, "Outline of an algorithm for integer solutions to linear program," *Bulletin of the American Mathematical Society 64*, pp. 275-278, 1958.

[24] R. E. Gomory, "Solving linear programming problems in integers," in *Combinatorial Analysis*, R.E. Bellman and M. Hall, Jr., eds., American Mathematical Society, pp. 211-216, 1960.

[25] G. D. Hachtel and F. Somenzi, "A symbolic algorithm for maximum flow in 0-1 networks," *International Workshop on Logic Synthesis*, pp. 6b:1-6, May 1993.

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

[27] P. L. Hammer and B. Simeone, "Order relations of variables in 0-1 programming," *Annals of Discrete Mathematics*, 31, North-Holland, pp. 83-112, 1987.

[28] S. L. Hurst, D. M. Miller and J. C. Muzio, *Spectral Techniques in Digital Logic,* London, U.K. : Academic, 1985.

[29] T. Ibaraki, "The power of dominance relations in branch and bound algorithm," *J. Assoc. Comput. Mach. 24*, pp. 264-279, 1977.

[30] S-W. Jeong and F. Somenzi, "A new algorithm for 0-1 programming based on binary decision diagrams," Logic Synthesis Workshop, in Japan, pp. 177-184, 1992.

[31] N. Karmarkar, "A new polynomial-time algorithm for linear programming," *Combinatorica*, 4:373-395, 1984.

[32] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1978.

[33] Y-T. Lai, S. Sastry 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.

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

[35] A. H. Land and A. G. Doig, "An automatic method for solving discrete programming problems," *Econometrica* 28, pp. 497-520, 1960.

[36] A. Land and S. Powell, "Computer codes for problems of integer programming," *Annals of Discrete Mathematics* 5, North-Holland, pp. 221-269, 1979.

[37] 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.

[38] Department of Mathematical Sciences, Rice University, Houston, TX 77251.

[39] G. Mitra, "Investigation of some branch and bound strategies for the solution of mixed integer linear programs," *Math. Programming 4*, pp. 155-170, 1973.

[40] L. G. Mitten, "Branch-and-bound methods: General formulation and properties," *Operations Research*, 18, pp. 24-34, 1970.

[41] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, Wiley, New York, 1988.

[42] J. P. Roth and R. M. Karp, "Minimization over Boolean graphs," *IBM Journal*, pp. 227-238, April 1962.

[43] T. Sasao, "FPGA design by generalized functional decomposition," in *Logic Synthesis and Optimization*, Sasao ed., Kluwer Academic Publisher, pp. 233-258, 1993.

[44] L. Schrage, Linear, Integer and Quadratic Programming with LINDO, Scientific Press, 1986.

[45] J. F. Shapiro, "A survey of lagrangian techniques for discrete optimization," *Annals of Discrete Mathematics 5*, North-Holland, pp. 113-138, 1979.

[46] K. Spielberg, "Enumerative methods in integer programming," *Annals of Discrete Mathematics 5*, North-Holland, pp. 139-183, 1979.

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

[48] H. A. Taha, "Integer programming," in *Mathematical Programming for Operations Researchers and Computer Scientists*, ed. A.G. Holzman, Marcel Derrer, pp.41-69, 1981.

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

[50] J. A. Tomlin, "Branch and bound methods for integer and non-convex programming," in J. Abadie, ed., *Integer and non-linear programming*, North-Holland, Amsterdam, 1970.

[51] D. Varma and E. A. Trachtenberg, "Design automation tools for efficient implementation of logic functions by decomposition," *IEEE Transactions of Computer-Aided Design*, Vol. 8, No. 8, Aug. 1989, pp. 901-916.

[52] J. S. Wallis, "Hadamard matrices," *Lecture Notes No. 292,* Springer-Verlag, New York, 1972.