Probabilistic Manipulation of Boolean Functions Using Free Boolean Diagrams

Amelia Shen, Srinivas Devadas
Massachusetts Institute of Technology
Cambridge, MA

Abhijit Ghosh
Mitsubishi Electric Research Laboratories
Sunnyvale, CA

Abstract

We propose a data structure for Boolean functions termed the Free Boolean Diagram. A Free Boolean Diagram allows decision vertices as in the conventional Binary Decision Diagram, but also allows function vertices corresponding to the AND and XOR functions. It has been shown previously that the equivalence of two Free Boolean Diagrams can be decided probabilistically in polynomial time.

Based on the equivalence checking method, we develop a set of algorithms for the probabilistic construction of Free Boolean Diagrams from multilevel combinational logic circuits, and for their manipulation. These algorithms are modified versions of reduced, ordered Binary Decision Diagram manipulation methods. We provide the implementation details of a Free Boolean Diagram package. We show that functions difficult to verify using reduced, ordered Binary Decision Diagrams can be verified using the Free Boolean Diagrams package using substantially less memory.

1 Introduction

The task of representing and manipulating Boolean functions efficiently is central to many different problems in logic synthesis, testing and verification. In recent years, reduced, ordered Binary Decision Diagrams (ROBDDs) [3] have gained widespread use in the areas of combinational and sequential logic verification (*e.g.* [6, 13]) due to their canonicity and ease of manipulation. The efficiency of an ROBDD representation depends strongly on the input ordering. Finding a good ordering is a difficult problem that has received considerable attention (*e.g.* [13]). However, there are classes of combinational circuits, notably multipliers, for which ROBDDs, under any possible input ordering, have a provably exponential size [4].

There have been efforts to increase the power of the ROBDD representation, while trading off canonicity or ease of manipulation. In Section 2, we describe previous work in this area.

We propose a data structure called the Free Boolean Diagram (FBD) that allows function vertices such as AND and XOR in addition to standard decision vertices. Our main contribution is the development of probabilistic algorithms for the construction and manipulation of FBDs based upon the fundamental result of Blum, Chandra and Wegman [1], and the recent result of Gergov and Meinel [9]. The FBDs presented here are not canonical but are a more powerful representation than ROBDDs because of the augmentations mentioned above. FBDs allow for different orderings along different paths from the root to a terminal vertex. While the algorithms we describe can be extended to manipulate unordered FBDs, we have not yet exploited this feature in verification applications. Further, the complexity of the algorithms increase in the case of unordered FBDs.

Our primary contribution is the development of a method that can be used to compute the Boolean *and* of two FBDs which, in conjunction with a method for cofactoring FBDs, can be used to compute arbitrary Boolean operations on FBDs. This method is similar to Brace, Rudell and Bryant's *ite* algorithm [2]. However, the method allows for the introduction and retainment of function vertices in the resulting FBD. As with the strongly canonical form for ROBDDs [2], during the *and* operation on FBDs we can ensure a reduced result. Using the theory of Gergov and Meinel [9], the *xor* operation on two FBDs can be computed trivially. We use attributed edges to complement FBDs.

Probabilistic cofactor, *complement*, *and*, and *xor* form a complete set of algorithms for FBD manipulation that can be used to directly construct an FBD from a multilevel circuit. Two multilevel circuits can be probabilistically checked for equivalence by constructing their respective FBDs and comparing their signatures.

Related work is described in Section 2. In Section 3 we give the terminology and review the equivalence checking methods of [1] and [9] that form the basis for Boolean operations on FBDs. The weakly canonical form for FBDs is presented in Section 4. We describe the implementation of *cofactor*, *xor*, and *and* methods in Section 5. Implementation details of the manipulation package are discussed in Section 6. Results obtained from using this package are given in Section 7.

2 Previous Work in Probabilistic Manipulation

The work of Blum, Chandra and Wegman [1] and that of Gergov and Meinel [9] upon which our probabilistic manipulation algorithms are based will be described in Section 3.

Semi-numeric Binary Decision Diagrams in [11], used the theory of Blum, Chandra and Wegman [1] as a means for probabilistic verification of previously unverifiable circuits. An example of the orthogonal partitioning of a hidden weighted bit function that has an $O(1.14^n)$-sized ROBDD under any possible ordering [4] into n $O(n^2)$-sized ROBDDs that have pairwise null intersections was given in [11]. While the methods of [11] augment the capabilities of ROBDD manipulation algorithms, methods are not presented that directly manipulate graphs with function vertices. We note here that Gergov and Meinel showed that polynomial-sized MOD-2-OBDDs can represent the hidden-weighted bit function [10]. The same construction can be used to show that polynomial-sized FBDs can represent the hidden-weighted bit function. The XOR vertices in FBDs are critical to achieving the exponential reduction in size relative to ROBDDs.

An example of a function that has an $O(n^2)$-sized unordered BDD and an $O(2^{\frac{n}{2}})$-sized ROBDD under any possible ordering is given in [7]. The same construction can be used to obtain an FBD of polynomial-size for this function. The AND vertices in FBDs are critical to achieving the exponential reduction in size relative to ROBDDs.

In unpublished work [5], Brace and Bryant have generalized the *apply* algorithm for ROBDDs to work with BDDs that do not have an ordering restriction. However, the variable selection heuristics used in their *apply* algorithm did not succeed in significantly outperforming ROBDD manipulation algorithms.

In recent work, the authors of this paper presented a variation of Free Boolean Diagrams with function vertices corresponding to AND, XOR and OR functions [16]. The AND vertex is identical to the AND vertex described here. However, the XOR vertex in [16] is restricted to have as children functions with disjoint support, and the OR vertex is restricted to have as children functions which have a null intersection. It can be shown that the XOR vertex used here, which does not place any restriction on the children, is strictly more powerful than the XOR/OR combination of [16]. Further, the algorithms described in [16] did not include the methods described in Section 5 for the retainment of function vertices. Automatic methods of function vertex introduction and retainment are critical to obtaining compact-sized FBDs from arbitrary logic-level circuits.

3 Preliminaries

The reduced, ordered Binary Decision Diagram is defined in [3] and an efficient implementation of a manipulation package is described in [2]. For the rest of this paper, it is assumed that the reader has a working knowledge of the above mentioned papers.

3.1 Free Boolean Diagrams

Define the **support** of an FBD f, $support(f)$, as the set of decision variables which appear along some path from the root to a terminal vertex in the FBD.

Definition 1: An FBD G with root vertex v denotes a function f_v defined recursively as:

1. If v is a terminal vertex:

   (a) If $value(v) = 1$, then $f_v = 1$.

   (b) If $value(v) = 0$, then $f_v = 0$.

2. If v is a nonterminal vertex with $index(v) = i$, where $1 \leq i \leq n$, then f_v is the function:

$$f_v \;=\; \overline{x_i} \cdot f_{low(v)} \;+\; x_i \cdot f_{high(v)}$$

   x_i is called the *decision variable* for vertex v.

3. If v is a nonterminal vertex with $index(v) = \oplus$ then f_v is the function:

$$f_v \;=\; f_{low(v)} \;\oplus\; f_{high(v)}$$

   v is called an XOR vertex.

4. If v is a nonterminal vertex with $index(v) = \bullet$ then v is an AND vertex and the function f_v is the Boolean *and* of its children:

$$f_v \;=\; f_{low(v)} \;\cdot\; f_{high(v)}$$

where $support(f_{low(v)}) \cap support(f_{high(v)}) = \phi$.

In an FBD, we require that in any path from the root to a terminal vertex a variable can appear at most once. However, the order in which these variables appear along different paths can be different. Further, we have three types of nonterminal vertices, a decision vertex as in a BDD, an AND vertex, and an XOR vertex. The function vertices XOR and AND do not have associated decision variables. Therefore an FBD is not a decision diagram per se. The function vertices can appear multiple times along a path from the root to a terminal vertex.

An example FBD is shown in Figure 1. Each vertex in the graph is either a terminal vertex (0 or 1), a decision vertex marked with an associated decision variable x_i, or a function vertex marked with $\oplus$ or $\bullet$. Subfunctions rooted at a vertex can be shared across two or more different functions. Thus, while a nonterminal vertex has exactly two children, a vertex may have many parents. Note that the two subfunctions rooted at the $\bullet$ vertex do not have any common inputs. In the figure, attributed edges are indicated by dots on them.

3.2 Probabilistic Equivalence of FBDs

It has been shown that the noncontainment problem for FBDs without function vertices, is NP-complete [8]. However Blum, Chandra and Wegman [1] have given a polynomial time algorithm to probabilistically decide the equivalence of two FBDs without AND and XOR vertices. A brief description of the bottom-up version of the algorithm follows.

Let F be an algebraic field with at least $2n$ elements. Let S be any subset of F, $|S| = 2n$. Assign each variable x_i, $1 \leq i \leq n$, a value chosen at random from S, denoted $|x_i|$, also called the signature of x_i. The value of the complemented variable is denoted as $1 - |x_i|$, also called the complement signature. (Actually, the complemented variable has a signature $|\overline{x_i}|$ such that $|x_i| + |\overline{x_i}| = 1$ using the $+$ operator of F).

Assign the graph G a value as follows:

- Assign the 1 terminal vertex the value 1.
- Assign the two arcs of each vertex the value of its complemented or uncomplemented variable associated with the vertex depending upon whether the arc is the low child or the high child of the vertex.
- Assign each vertex the sum (over the two arcs leaving it) of the product of the value of each arc times the value of the vertex at that arc's head.

- Assign G the value computed for its root vertex, call it $|G|$. ($|G|$ is called the signature of G).

It was shown in [1] that for two graphs G_1 and G_2, if $|G_1|$ is not equal to $|G_2|$, then G_1 and G_2 are definitely not equivalent. If G_1 and G_2 are not equivalent, then $|G_1|$ is not equal to $|G_2|$ with a probability of $> \frac{1}{2}$.

It was further shown in [11] that the probability of error associated with the probabilistic equivalence check is less than $(\frac{n}{|S|})^k$, where k is the number of "runs". The error probability can be made arbitrarily small by making multiple runs with different random assignments of the $|x_i|$'s to the x_i variables. Further, by increasing the cardinality of S the error probability for a single run can be reduced significantly.

3.3 XOR Vertices

Gergov and Meinel [9] have extended the the equivalence algorithm of [1] to include function vertices that correspond to parity functions, *i.e.*, XOR vertices. We describe their method below.

A primitive polynomial of degree m and characteristic 2 can be used to generate a Galois Field with 2^m elements $GF(2^m)$ [12]. Arithmetic in a finite field of characteristic 2 is essentially *mod* 2 arithmetic. Therefore, for any element $a \in GF(2^m)$, $a + a \equiv 0$, where 0 is the identity element. This leads immediately to the result that given two Boolean functions f and g the signature of $f \oplus g$ is the sum of the signature of f and the signature of g (where the sum corresponds to addition carried out in $GF(2^m)$). The common $f \cdot g$ terms in f and g cancel each other out resulting in $f \cdot \overline{g} + \overline{f} \cdot g$.

Therefore, if we have an FBD with XOR vertices, the signature of the function rooted at the XOR vertex can be computed simply by summing the signatures of its children. Computing the signature of decision vertices is unchanged from the Blum, Chandra and Wegman algorithm, except that multiplication is also carried out in $GF(2^m)$.

3.4 AND Vertices

Consider an FBD with AND vertices. We use the Blum, Chandra and Wegman algorithm as before. When we see an AND vertex, we multiply the signatures of the child vertices. This produces the signature for the function rooted at the AND vertex, because the children have disjoint supports. Note that multiplication has to be carried out in $GF(2^m)$.

The particular primitive polynomial used to generate the field, and other details regarding the arithmetic are given in Section 6.

4 Reduce and a Weakly Canonical Form

Given an unreduced FBD it is possible to the use the algorithm described in the previous section to *reduce* the FBD, *i.e.* remove redundant vertices and redundant subfunctions in the diagram. The reduce algorithm is similar to that of [3]. Further, the strongly canonical framework for ROBDDs developed in [2] can also be generalized to FBDs. We describe the reduction algorithm and the canonical form for FBDs in this section.

The reduction method for FBDs simply corresponds to a bottom-up traversal of the FBD computing the signature for every vertex in the FBD. That is, for every vertex v_i in the FBD we compute $|v_i|$. Note that $|v_i|$ is the signature corresponding to the Boolean function f_{v_i} rooted at v_i. We maintain a *unique table* corresponding to the unique signatures we have encountered. If we compute a $|v_j|$ for a vertex v_j and we find that $|v_j|$ is already in the unique table, this means that some other vertex v_k has a signature $|v_k| = |v_j|$. With a quantifiable probability of error, $f_{v_k} \equiv f_{v_j}$. Under a probabilistic methodology, we can remove v_k (or v_j) from the diagram, and make all the parents of v_k (or v_j) point to v_j (or v_k). Likewise, if we locate a complement signature in the unique table, such that $|v_k| = 1 - |v_j|$, we can replace all occurrences of v_k (or v_j) with the complement of v_j (or v_k).

It is easy to see that the reduction of a FBD produces another FBD which has no more vertices than the original FBD. Reduction corresponds to a traversal of the FBD with a lookup in the unique table at each vertex, and can be performed in $O(|G| \log(|G|))$ time where $|G|$ is the size, *i.e.* the number of vertices in the original FBD.

In general, when FBDs for several functions exist and FBDs are being manipulated to generate other FBDs, a global unique table can be maintained. Whenever we wish to create a new decision vertex v, we first compute the signature of the function rooted at the vertex given the signatures of the two children of the vertex and the value $|x_i|$ of the decision variable corresponding to the vertex. That is,

$$|f_v| \;=\; |x_i| \times |f_{high(v)}| \;+\; (1 - |x_i|) \times |f_{low(v)}|$$

We look up $|f_v|$ in the unique table. If it already exists, it means we have already created the function rooted at v previously, and we need not create v. Similarly if we are creating an XOR vertex we compute:

$$|f_v| \;=\; |f_{high(v)}| \;+\; |f_{low(v)}|$$

and check if $|f_v|$ is in the unique table. An analogous equation follows for the AND vertex.

$$|f_v| \;=\; |f_{high(v)}| \;\times\; |f_{low(v)}|$$

We maintain a weakly canonical form in the sense that no two functions with the same or complement signature ever exist simultaneously. Therefore we never have to explicitly reduce FBDs.

5 Boolean Operations

We will describe how basic Boolean operations like *cofactor*, *xor*, and *and* can be performed on FBD representations. It is important that the Boolean operations be implemented in such a way that a Boolean operation performed on any set of FBDs produce another FBD. In particular, we have to be careful about how we handle function vertices.

5.1 Indices and Function Vertices

Vertices can be terminal, decision, XOR or AND vertices. Two flags are required to differentiate between the different types of vertices.

Each decision variable has an associated index in the specified global ordering. The variable that appears first in the ordering is assigned the index 0, and the following variables are assigned indices in increasing order. Each decision vertex contains an index corresponding to the decision variable. Each function vertex also has an index which corresponds to the *minimum* index of any decision vertex that can be reached by traversing paths from the function vertex to a terminal vertex. In addition, function vertices are marked with a special flag to indicate whether they are XOR or AND vertices.

5.2 Cofactor

Cofactoring a function f with respect to a variable x_i corresponds to setting the variable to a 1 or a 0.

$$f_{x_i} \;=\; f(x_1,\; x_2,\; \cdots,\; x_{i-1},\; 1,\; x_{i+1},\; \cdots,\; x_n)$$

Given an FBD G for a function f, if x_i happens to be the decision variable corresponding to the root vertex of the given FBD, then the FBD for f_{x_i} is simply the high child of the root vertex of G. Similarly the FBD for $f_{\overline{x_i}}$ is the low child of the root vertex of G.

If x_i is not the decision variable corresponding to the root vertex of G a more general algorithm is required. We simply find all the vertices of G that have x_i as a decision variable. To compute the FBD corresponding to f_{x_i}, for each chosen vertex we make the parents of the chosen vertex point to the high child of the vertex (or the low child when computing $f_{\overline{x_i}}$).

Given an AND vertex v in the FBD we have $support(f_{low(v)}) \cap support(f_{high(v)}) = \phi$. After cofactoring, the FBD still satisfies the support condition. The XOR vertex does not place any restrictions on its children. Therefore cofactoring a FBD produces another FBD. The complexity of the cofactor algorithm is linear in the size of the given FBD.

FBDs with XOR vertices may have redundant variables after cofactoring. By redundant variables we mean vertices upon whose decision variables a function does not logically depend, but are encountered when traversing paths from the root of a FBD representation of the function to a terminal vertex. An example of cofactoring producing a redundant variable in given in Figure 2. The FBD of Figure 2(a) does not have any redundant vertices or variables. However, when the FBD of Figure 2(a) is cofactored with respect to c, we obtain the FBD of Figure 2(b), where variable b and all vertices corresponding to the variable b are redundant.

Redundant variables or vertices do not change the signature of the FBD. They can be eliminated when they are detected by setting all vertices that correspond to the redundant decision variable to either a 0 or a 1 value. (It does not matter which value the variable is set to.) During a Boolean operation assume we wish to create an XOR vertex v_j with the same signature as a vertex already in the unique table, v_k. Since the vertices are equivalent, the indices of the two vertices will be identical if no redundant vertices exist. However, if $index(v_j) > index(v_k)$ it means that the topmost vertex in v_k is redundant, and we replace v_k with v_j in the unique table. If $index(v_j) \leq index(v_k)$ we return v_k. The above strategy will guarantee that we never create FBDs which violate the property of freedom, *i.e.* that a decision variable can be encountered at most once on any path from the root to a terminal vertex.

5.3 *xor*

Computing the *xor* operation on two FBDs f and g is trivial. We create an XOR vertex with low child f and high child g. We assign the XOR vertex an index corresponding to the minimum of the indices of the root vertices in f and g.

5.4 *and*

The pseudo-code for the *and* operation is shown in Figure 3 and is explained below.

5.4.1 Special Cases

Given two FBDs f and g the first step is to check for special cases as with ROBDDs. These special cases are: If f or g equals 0 return 0. If f is equal to 1 or g return g. If g is equal to 1 return f. If f equals $\overline{g}$ return 0.

5.4.2 Computed Table

Next we want to check if the *and* was previously computed and thereby save time by not recomputing the result. Given a pair f, g we can check the computed table to see if $f \cdot g$ was previously computed and the result is still available in the computed table. If so, we return the result. If not, we can check to see if $\overline{f} \cdot g$, $f \cdot \overline{g}$ or $\overline{f} \cdot \overline{g}$ were computed. If any of these were previously computed, we can return $\textsc{xor}(g, \overline{f} \cdot g)$, $\textsc{xor}(f, f \cdot \overline{g})$, and $\textsc{xor}(\textsc{xor}(f, g), \overline{\overline{f} \cdot \overline{g}})$, respectively. We are using XOR vertices and the previously computed result to obtain the FBD for $f \cdot g$.

5.4.3 Variable Selection

If the result is not available in the computed table, we select a decision variable x_i. A vertex with an associated decision variable x_i is created and the result is formed by recursively calling **fbd-and**() with the two cofactor pairs.

We enforce the global ROBDD input ordering to select decision variables. Thus, the variable that is selected corresponds to the variable with the minimum index across f and g. Note, however, that the use of function vertices results in a representation that is more powerful than ROBDDs. The variables in the paths of the FBDs appear in order, but the function vertices "emulate" unordered paths. Namely, given the same global ordering, the FBDs can still be smaller than ROBDDs.

5.4.4 Introducing AND Vertices

Once $low(v)$ and $high(v)$ have been obtained for the result of the *and* operation on the cofactor pairs, we can compute the signature of the result as

$$s \;=\; |result| \;=\; |x_i| \times |high(v)| \;+\; (1 - |x_i|) \times |low(v)|$$

If s is equal to $|f| \times |g|$, we can assume (with a finite probability of error) that f and g have disjoint supports. We can return $\textsc{and}(f, g)$ as the result. This representation for $f \cdot g$ that uses the AND vertex is the optimal representation in terms of graph size.

The introduction of AND vertices can result in significant reduction in memory requirements. To understand this, consider the functions f and g with disjoint supports, and assume that the global ordering corresponds to the variables of f interleaved with the variables of g. In this case the ROBDD for $f \cdot g$ can have $0.5 \times size(f) \times size(g)$ vertices. However, the final result using the AND vertex only requires $size(f) + size(g) + 1$ vertices. Note that the number of recursive calls to **fbd-and** remains the same in the ROBDD and FBD cases. However, the FBD operations require much less memory, because the partially computed results can be freed once the signature check is performed.

In order to improve the computation time over the ROBDD approach, prior to computing the *and* of two FBDs, the support of each of the vertices in the FBDs can be computed and stored. If the supports are disjoint, an AND vertex is introduced. No additional error is introduced when the AND vertices are introduced after a check on the supports. This method, however, requires an additional support field for each vertex in the FBD, which can occupy a substantial amount of memory. We do not use the support field in our implementation.

The error introduced by checking the signatures to introduce an AND vertex is equivalent to the error introduced by a unique table hit, and is less than $(\frac{n}{|S|})^k$.

5.4.5 Introducing XOR Vertices

It may appear counter-intuitive but XOR vertices can be introduced within an *and* operation as well.

Consider two functions f and g such that $\overline{f} \cdot \overline{g} = \phi$. This means that $f \cdot g \equiv \overline{\overline{f} \oplus g}$. As before, we can compute the signature of the result obtained for the *and* operation, namely s, and check if:

$$s = |result| = 1 - (|f| + |g|)$$

If this is the case, and the number of vertices in f plus the number of vertices in g is less than the number of vertices in $result$, we return $\text{XOR}(\overline{f}, g)$. Since it is quite expensive to compute the number of vertices in an FBD this check is not performed at every recursion, but at every l recursions, where l is typically 4.

As before, the introduction of XOR vertices can substantially reduce memory requirements during an *and* operation.

5.4.6 Retaining XOR Vertices

Introducing function vertices in the result of an *and* operation helps reduce memory requirements. However, if another Boolean operation is to be performed on the computed result, the function vertices are "bubbled" down by recursively cofactoring with respect to decision variables. Therefore, savings in node sizes may be lost.

To see this more clearly, consider the FBDs $\text{XOR}(f_1,\, f_2)$ and g, which are to be *and*'ed. We can choose decision variables and obtain a FBD for $(f_1 \oplus f_2) \cdot g$. Alternatively, we can compute $f_1 \cdot g$ and $f_2 \cdot g$ and represent the result as $\text{XOR}(f_1 \cdot g,\, f_2 \cdot g)$. Depending on f_1, f_2, and g one or the other of the two representations will be smaller.

In order to fully exploit function vertices in the computation of $f \cdot g$, we first select decision variables and compute $f \cdot g$. If f has an XOR vertex at its root, *i.e.* is $\text{XOR}(f_1,\, f_2)$, we also compute $f_1 {\cdot} g$ and $f_2 {\cdot} g$. If $size(f {\cdot} g)$ is larger than $size(f_1 {\cdot} g) + size(f_2 {\cdot} g) + 1$, the latter representation is chosen and the former is freed. Else, if $size(f \cdot g)$ is smaller than or equal to $size(f_1 \cdot g) + size(f_2 \cdot g) + 1$, the former representation is chosen and the latter is freed.

During the computation of $f_1 \cdot g$ or $f_2 \cdot g$ the number of vertices being created is monitored. When the size of these FBDs becomes larger than the size of $f \cdot g$, the computation is terminated.

It is possible that g has an XOR vertex as its root, and f does not. This case is not included in Figure 3 for brevity, however, the same strategy applies in this case too.

5.4.7 Redundant Vertices

Once we have a representation for $f \cdot g$, we check for the existence of the signature in the unique table. If the signature is present, we use the representation that has the highest index at the root.

6 Implementation

The implementation of the FBD package mimics the structures and routines described in [2], with modifications to accommodate function signatures, and function vertices. BDD functions have also been altered to operate correctly without the imposition of a global input variable ordering, though we have not yet exploited unordered FBDs in our experiments.

6.1 Finite Field Arithmetic

We have chosen a primitive polynomial with characteristic 2 and degree 16, namely, $x^{16} + x^5 + x^3 + x^2 + 1$ with degree $m = 16$ to generate a field with 2^{16} elements. The elements are the polynomials 0, 1, x, $x+1$, x^2, x^2+1, and so on until $x^{15} + x^{14} + \cdots + x + 1$. These polynomials are represented as decimal numbers, varying from 0 through 65535. Note that the maximum degree of any of the elements is $m - 1 = 15$. For example, the polynomial $x^5 + 1$ is represented by the number 33, which can be represented as the 16-bit binary number 0000000000100001.

The addition of two polynomials is carried out under *mod* 2 arithmetic. Therefore, the addition of two polynomials becomes the bitwise exclusive-or of the corresponding binary representations. For example, the addition of $x^2 + x + 1$ and $x^3 + x + 1$ is carried out by representing them as the binary numbers 0000000000000111 and 0000000000001011, respectively, and bitwise exclusive-or'ing them to obtain the binary representation 0000000000001100. This corresponds to the polynomial $x^3 + x^2$.

Subtraction is exactly the same as addition in *mod* 2 arithmetic. Therefore, $1 - x$ equals $x + 1$.

Multiplication of two polynomials can be carried out in different ways. The most efficient way is to use *index* tables which are similar to logarithm tables. Each element in the finite field other than the 0 element has an associated index. In order to multiply two polynomials, we add their respective indices, [1] and then find the product polynomial by using an inverse index table. The index and inverse index table can be generated easily, and since each table only contains 65536 entries, they can be stored in memory.

An example of polynomial signature calculation is shown in Figure 4. The signature for the AND vertex v_3 is the *mod* 2 product of the signatures of its children. The signature for the XOR vertex v_1 is the *mod* 2 sum of the signatures $|v_2|$ and $|v_3|$. The signatures in the figure are shown as polynomials, but the polynomials are represented by binary numbers in the implementation.

For more details regarding arithmetic on finite fields the reader is referred to [12].

6.2 FBD Structures

Each vertex in an FBD is associated with an FBD vertex structure. An FBD vertex (also called a node) has forward pointers only, one to the *low* child vertex and one to the *high* child vertex. The FBD vertex has a variable index that identifies the type of vertex, *i.e.*, terminal, decision, AND, or XOR, and also identifies the decision variable associated with the vertex.

Each vertex also stores signatures. The number of signatures (or passes) determines the error probability when testing for function equivalence in the unique table. Each signature corresponds to a particular assignment of values to the variables of the FBD. As mentioned in Section 6.1, we use a polynomial of degree 16 to generate a field with 2^{16} elements. Each signature requires 16 bits, *i.e.* 2 bytes.

[1] Addition is carried out in binary or decimal arithmetic.

We also implemented the complement edge scheme described in [2]. When a complement edge on a vertex F is encountered, the signature for the corresponding function is simply $|1 - |F||$. Note that we do not have to enforce the restrictions on the use of negative edges as in [2], as our canonicity property is based on signatures.

6.3 Hash Tables

The unique table is a hash table that records vertices with unique signatures. The unique table is essential for maintaining the canonicity of the FBD vertices. Before a FBD vertex is created, we calculate the signature of the function associated with the vertex, and use this signature as the key to determine if a unique table entry already exists for this vertex. If the vertex exists, it can be used to represent the function with a quantifiably small probability of error. Otherwise, a new vertex is created and entered into the unique table under its signature. Only uncomplemented vertices are stored in the unique table, and when we search for a signature in the unique table, we look for both the signature of the complemented function and the signature of the uncomplemented function.

Because of the use of function vertices, it is possible to create different representations of a function. Therefore, while using function vertices, if a unique table hit occurs, we always choose the representation with smaller size and the highest index at the root.

The computed table is a hash-based cache that stores the results of previous *and* results. The entries in the cache are accessed by hashing the signatures of the pair of vertices, and then comparing the current pair with the cache entry. If the pairs match, the result of the *and* is taken from the cache entry. Before we look for an entry in the computed table, we convert the pair to a standard pair, similar to the standard triple of [2].

Since all the tables use the FBD vertex signatures as keys, and the results of the computed table lookups are signatures, the table entries do not rely on actual vertex addresses. The actual vertices can be replaced, altered and updated in the unique table without corrupting other table entries.

6.4 Memory Requirement

It was shown in [2] that the memory usage per ROBDD vertex is 22 bytes per vertex under certain assumptions. The same computation shows the amortized memory cost for the FBD package is 32 bytes per vertex. This number includes the hash tables and 4 passes for the signature computation. Each pass corresponds to a signature which requires 2 bytes of storage. If 6 passes are used then the amortized memory cost would be 39 bytes per vertex.

7 Results

The results of applying the FBD package to some benchmarks are summarized in Table 1. In most cases the same global ordering for all outputs was given to the FBD package and the ROBDD package implemented in the program `sis` [15]. Examples marked with an asterisk required different orderings for different outputs to create BDDs. For these examples, any further manipulation of the output BDDs requires the FBD package. All CPU times are reported in seconds on a SPARCstation 10.

Each row in Table 1 shows the BDD sizes and CPU times for the given benchmark. The entries under the heading ROBDD give the sizes and times required by the ROBDD package in `sis`. The sizes and times used by the FBD package with only a global input ordering and no function vertices are listed under the heading FBD. The FBD package without any AND and XOR vertices and the same global ordering produces exactly the same graphs as the ROBDD package for all the examples, and requires about 1.4X more CPU time on the average. In some cases, the FBD package is faster and this is attributable to the different hashing scheme for the computed table. The purpose of this comparison is to verify the FBD package has smaller computational requirements than the ROBDD package and is within a constant factor in CPU time required.

FBDs built using a global input ordering and using AND and XOR vertices appear next in the table. When function vertices are added, the sizes of FBDs decrease by as much as a factor of 6 in certain cases.

The last column in the table shows the number of function vertices in the final FBD for the circuit. Differences in size between FBDs and ROBDDs using the same global ordering are directly attributable to the function vertices.

Most of the examples are from the ISCAS-85 and 89 benchmark set. **mm** is the min-max function. The example **hwb** is the hidden weighted bit function described in [4]. FBDs for this function can be created with $O(n^2)$ size using OR vertices, whereas any ROBDD representation has $O(1.14^n)$ vertices. **rotate32** is the function from [7]. FBDs for this function can be created with $O(n^2)$ size using AND vertices, whereas any ROBDD representation has $O(2^{\frac{n}{2}})$ vertices. **C2670** and **C5315** also benefit significantly from XOR vertices.

In examples **C7552** and **s9234** we need different orderings for different outputs to create FBDs. Multipliers are particularly difficult circuits even for FBDs. We have been able to create FBDs for multipliers of up to 12 bits. For example **C6288**, we have been able to generate FBDs up to the 14^{th} output.

The error probabilities on the circuits were small, and can be made arbitrarily small by increasing the number of passes. The upper bound on the error probability for the circuits is given by $(\frac{n}{|S|})^k \times num\_ut\_hits$, the error probability for the equivalence check times the number of unique table hits [11]. In the above examples, we used a field size of $|S| = 2^{16}$ and 4 passes. A circuit that has 100 inputs and requires 10^6 unique table hits to construct the FBDs for the circuit has an error probability of less than 5.42×10^{-6}.

8 Conclusions

This paper applies the confluence of two directions of research to the problem of Boolean function representation and manipulation, namely probabilistic equivalence checking and graph-based algorithms for Boolean function manipulation. We have used the theory of probabilistic equivalence checking for FBDs to arrive at a set of FBD-based Boolean manipulation algorithms. We have been able to exploit the large amount of research and engineering effort that has gone into making ROBDD-based Boolean manipulation highly efficient in our FBD manipulation package.

The augmentation in the FBD representation (as opposed to the ROBDD representation) is that special function vertices are allowed in a FBD, provided certain conditions are satisfied. For a number of Boolean functions encountered in hardware design, these augmentations can result in significantly smaller sized representations, as demonstrated by our results. Different orderings are allowed along different paths from the root to a terminal vertex in a FBD. However, in order to retain the efficiency of ROBDD manipulation methods the FBD algorithms currently require a good global input ordering.

Ongoing work includes the development of dynamic variable selection methods during the *and* operation, the incorporation of dynamic re-ordering as in [14], and application to sequential logic verification.

9 Acknowledgements

Thanks to Randy Bryant and Richard Rudell for discussions regarding free Boolean Diagram construction. Thanks to Jordan Gergov and Christoph Meinel for bringing $\oplus$-BP1s and MOD-2-OBDDs to our attention. This research was supported in part by the Defense Advanced Research Projects Agency under contract N00014-91-J-1698 and in part by a NSF Young Investigator Award with matching funds from Mitsubishi and IBM Corporation.

References

[1] M. Blum, A. K. Chandra, and M. N. Wegman. Equivalence of Free Boolean Graphs Can Be Decided Probabilistically in Polynomial Time. *Information Processing Letters*, 10(2):80–82, March 1980.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *Proc. of* 27^{th} *Design Automation Conference*, pages 40–45, June 1990.

[3] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] R. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. In *IEEE Transactions on Computers*, pages 205–213, February 1991.

[5] R. Bryant. Free Binary Decision Diagrams. Personal Communication, April 1992.

[6] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Using Boolean Functional Vectors. In *IMEC-IFIP Int'l Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, November 1989.

[7] S. Devadas. Comparing Two-Level and Ordered Binary Decision Diagram Representations of Logic Functions. In *IEEE Transactions on Computer-Aided Design*, pages 722–723, May 1993.

[8] S. Fortune, J. Hopcroft, and E. M. Schmidt. The Complexity of Equivalence and Containment for Free Single Variable Program Schemes. In *Goos, Hartmanis, Ausiello and Böhm Eds., Lecture Notes in Computer Science*, volume 62, pages 227–240, 1978.

[9] J. Gergov and C. Meinel. Frontiers of Feasible and Probabilistic Feasible Boolean Manipulation With Branching Programs. *Lecture Notes in Computer Science*, 665:576–585, February 1993.

[10] J. Gergov and C. Meinel. Mod-2-OBDD's: A Generalization of OBDD's and EXOR-Sum-of-Products. In *Proceedings of the IFIP Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 1993.

[11] J. Jain, J. Bitner, D. Fussell, and J. Abraham. Probabilistic Verification of Boolean Functions. In *Formal Methods in System Design: An International Journal*, volume 1, pages 63–118, July 1992.

[12] R. Lidl and H. Niederreiter. *Finite Fields*. Addison-Wesley, Reading, MA, 1983.

[13] S. Malik, A. R. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 6–9, November 1988.

[14] R. Rudell. Dynamic Variable Ordering in Ordered Binary Decision Diagrams . In *Proceedings of the Int'l Conference on Computer-Aided Design*, November 1993.

[15] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proceedings of the Int'l Conference on Computer Design: VLSI in Computers and Processors*, pages 328–333, October 1992.

[16] A. Shen, S. Devadas, and A. Ghosh. Probabilistic Construction and Manipulation of Free Boolean Diagrams. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 544–549, November 1993.

EX	#i	#o	ROBDD		FBD		AND/XOR		#
			Nodes	Time	Nodes	Time	Nodes	Time	F.V.
C432	36	7	31,195	26	31,195	28	28,629	43	2,411
C499	41	32	33,214	14	33,214	11	5,354	17	60
C880	60	26	7,761	2	7,761	3	6,861	4	1,109
C1355	41	32	33,214	14	33,214	20	33,214	92	24
C1908	33	25	12,734	9	12,734	12	12,553	26	365
C2670	233	140	57,767	21	57,767	32	21,805	55	171
C3540	50	22	88,652	82	88,652	116	87,617	114	1,906
C5315	178	123	26,129	11	26,129	14	14,694	41	273
C6288(14)	32	32	115,607	535	115,607	1099	115,607	1522	0
C7552*	207	108	19,187	9	19,187	9	7,365	16	346
i10	257	224	880,254	306	880,254	602	680,100	471	50,804
s5378	199	213	6,854	2	6,854	1	4,009	1	631
s9234*	247	250	1,069,633	632	1,069,633	1786	824,958	742	262,342
s13207	700	790	13,892	9	13,892	5	5,054	6	472
s15850	611	684	53,612	26	53,612	28	31,986	36	4,982
s35932	1763	2048	6,190	36	6,190	5	5,731	3	290
s38584	1464	1730	40,817	56	40,817	20	32,484	20	2,639
hwb16	16	1	614	0.4	614	0.6	614	0.5	0
hwb32	32	1	40,126	36	40,126	36	7,596	520	271
hwb64	64	1	unable	unable	unable	unable	11,599	70	64
mm16	67	64	3308	4	3308	3	3,263	4	37
mm30	123	120	11,071	15	11,071	14	10,983	17	64
mult12	24	24	610,093	950	610,093	1632	610,112	2344	0
rotate32	69	1	unable	unable	unable	unable	2117	0.3	992

Table 1: Results using the FBD package

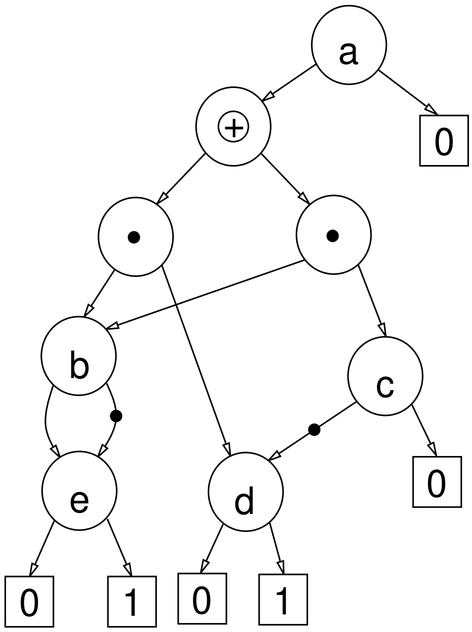

Figure 1: An Example Free Boolean Diagram

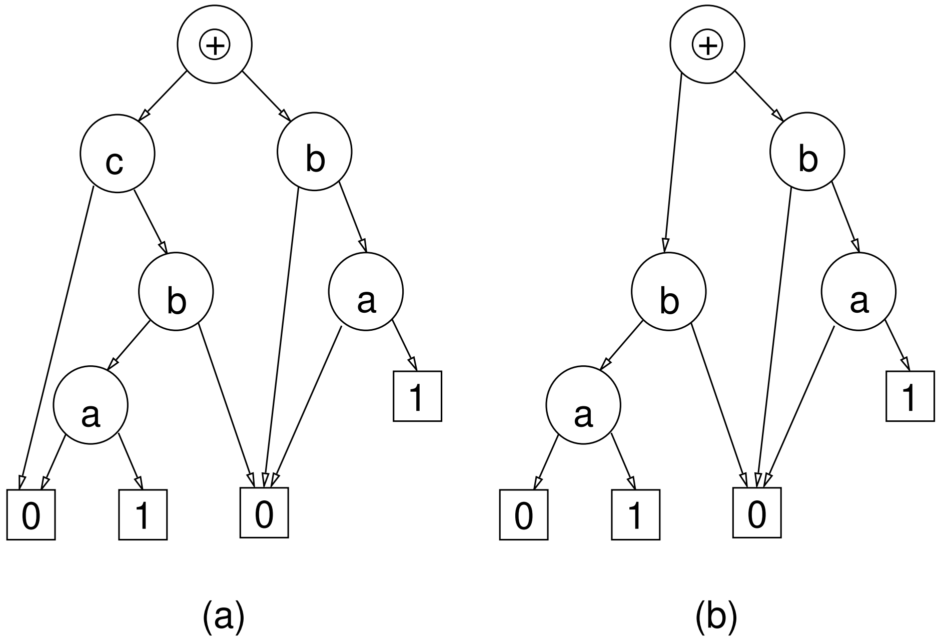

Figure 2: Redundant Variables and Vertices After Cofactoring

```
fbd-and(f, g):
{
    if (result = special_cases(f, g))
        return(result) ;
    if (result = computed_table_lookup(f, g))
        return(result) ;
    x_i = select_variable(f, g) ;
    create vertex v with decision variable x_i ;
    low(v) = fbd-and(f_{x̄_i}, g_{x̄_i}) ;
    high(v) = fbd-and(f_{x_i}, g_{x_i}) ;
    Compute signature s of result = (x_i, low(v), high(v)) ;
    if (s ≡ |f| × |g|) {
        Free low(v) and high(v) ;
        Create an AND vertex v with f and g as children ;
    }
    else if (s ≡ 1 − (|f| + |g|)) {
        if (size(result) > size(f) + size(g) + 1) {
            Free low(v) and high(v) ;
            Create an XOR vertex v with
            f̄ and g as children ;
        }
    }
    if (f ≡ XOR(f_1, f_2)) {
        Compute f_1 · g and f_2 · g ;
        if (size(result) > size(f_1 · g) + size(f_2 · g) + 1) {
            Free low(v) and high(v) ;
            Create an XOR vertex v with f_1 · g and f_2 · g as children ;
        }
        else
            Free f_1 · g and f_2 · g ;
    }
    if (result = unique_table_lookup(s))) {
        if (index(result) > index(v)) {
            Free low(v) and high(v) ;
            v = result;
        }
    }
    insert v = (f, g) into computed_table ;
    return(v) ;
}
```

Figure 3: Procedure to compute $f \cdot g$

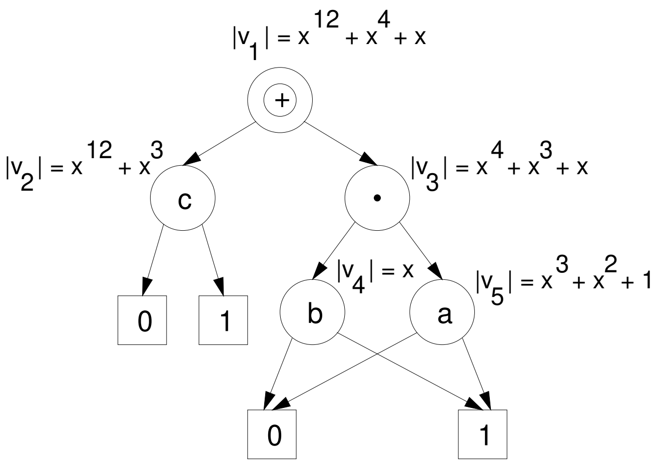

Figure 4: An example of one pass signature calculation using polynomials.