Probabilistic Representation and Manipulation of Boolean Functions Using Free Boolean Diagrams

by

Amelia Huimin Shen

B.S. (EE & CS), University of California, Berkeley (1987)
S.M. (EE & CS), Massachusetts Institute of Technology (1992)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

July 1994

Signature of Author_____
Department of Electrical Engineering and Computer Science
July 21, 1994

Certified by_____
Srinivas Devadas
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by_____
Frederic R. Morgenthaler, Professor of Electrical Engineering
Chairman, Department Committee on Graduate Students

Probabilistic Representation and Manipulation of Boolean Functions Using Free Boolean Diagrams

by

Amelia Huimin Shen

Submitted to the
Department of Electrical Engineering and Computer Science
on July 21, 1994 in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Abstract

The advent of increasingly dense and fast Very Large Scale Integrated (VLSI) circuits allows for the design of larger and more sophisticated digital logic circuits. Efficient logic representations are necessary for the synthesis, testing and verification of these circuits.

This thesis introduces a new logic representation, called the Free Boolean Diagram (FBD). This representation can be manipulated in time comparable to existing methods and the complexity of the representation for a number of circuit classes is provably more efficient than existing representations such as the Reduced Ordered Binary Decision Diagram (ROBDD).

Free Boolean Diagrams allow for function vertices that represent Boolean *and, or* and *exclusive-or*, in addition to the decision vertices found in conventional Binary Decision Diagrams. A previous result is extended to probabilistically determine the equivalence of Free Boolean Diagrams in polynomial time.

A strongly canonical form is maintained for Free Boolean Diagrams using "signatures". New algorithms for the probabilistic construction of Free Boolean Diagrams from multilevel combinational logic circuits and the manipulation of these graphs are developed and implemented in a software package. The results of the application of the package to combinational logic verification and Boolean satisfiability problems are presented.

Thesis supervisor: Srinivas Devadas
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

This work is the result of the efforts and contributions of several people. First and foremost, I would like to thank my thesis advisor, Professor Srinivas Devadas. His guidance, knowledge, original contributions and enthusiasm have been crucial to the progress and quality of this work. Professor Devadas ignited my interest in CAD, and has made my time at MIT a wonderful and enjoyable learning experience.

I would also like to thank Dr. Abhijit Ghosh of Mitsubishi Electic Research Laboratories for his invaluable assistance during the development of this project.

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.

I am also grateful to my thesis readers, Professor Jonathan Allen and Professor Jacob White, for all their help and encouragement during my stay at MIT.

Thanks to all the members of the 8^{th} floor VLSI CAD research group, past and present. Thanks to Robert Armstrong, Mattan Kamon, Jose Monteiro, Joel Phillips, Khalid Rahmat, Luis Miguel Silveira and Ricardo Telichevesky for their assistance and encouragement. Special thanks to Ignacio McQuirk, for all his advice and help.

Finally, I would like to thank my parents, Ann and Lawrence Shen, my siblings, David and Peggy Shen, my friends, Chavela Carr, Andrea Chen, and Songmin Kim for all their support and encouragement. Most of all, thanks to Kenneth Munson for all the understanding and support he has provided over the years.

The described work was done at the Research Laboratory of Electronics of the Massachusetts Institute of Technology. Additional work was done at Mitsubishi Electric Research Laboratories. This research was supported in part by the Defense Advanced Research Projects Agency under contract N00014-91-J-1698, in part by a NSF Young Investigator Award with matching funds from Mitsubishi and IBM Corporation, and in part by a National Science Foundation Graduate Fellowship.

Surfer Pet Peeve Number 5:
When you catch an amazing wave and realize your trunks have caught a different wave.

— David Letterman Show

Contents

List of Figures

List of Tables

1

Introduction

1.1 Boolean Representations

Boolean equations use arguments and values to represent Boolean functions. The functions implemented by digital logic circuits can be described by a set of Boolean equations. An efficient Boolean logic representation for these equations is essential in the design cycle of complex digital circuits. In recent years, a number of methods have been developed for representing and manipulating Boolean functions.

Early representations based upon truth tables or Karnaugh maps grow exponentially with the number of inputs, and consequently are practical only for Boolean functions with very few inputs. These forms, however, are canonical and easy to manipulate.

More recent representations, such as the reduced sum-of-products form and multiple-valued variables [10] are more efficient for some common functions, but still grow exponentially for other common functions. The more efficient forms, including Boolean networks, are also not canonical, because the same function can have many representations. This lack of canonicity makes questions such as equality, satisfiability, and tautology difficult to answer. Some fundamental Boolean operations, such as complementation, are expensive to compute and produce results with exponential size.

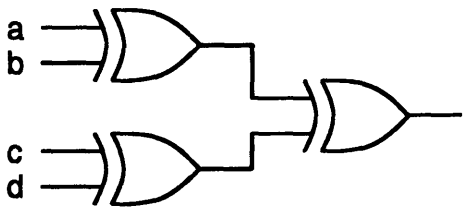
In Figure 1-1, 4 different representations of the *exclusive-or* of 4 variables are shown. The function can be expressed as a truth table, a Karnaugh map, and a multilevel circuit. The truth table and Karnaugh map are unique representations, whereas the multilevel circuit representation is not canonical. The truth table and Karnaugh map also require an exponential number of entries, while, in this case, the multilevel circuit representation is linear in the number of inputs. The multilevel circuits in Figure 1-1(c) and Figure 1-1(d) both represent the same *exclusive-or* function.

a	b	c	d	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

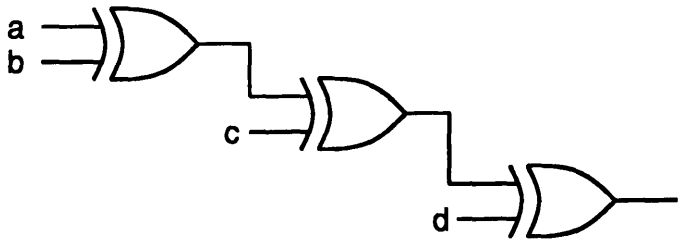
(a)

cd \ ab	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

(b)



(c)



(d)

FIGURE 1-1: $f = a \oplus b \oplus c \oplus d$ as (a) a truth table, (b) a Karnaugh map, (c) a multilevel circuit, and (d) a different multilevel circuit.

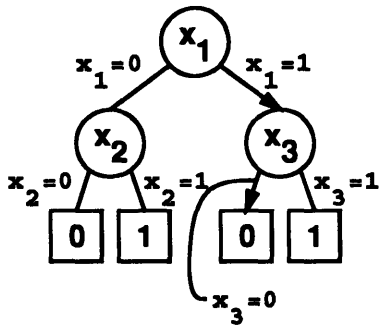


FIGURE 1-2: The ROBDD that represents $x_1 \cdot x_3 + \overline{x_1} \cdot x_2$ under input ordering x_1, x_2, x_3

1.2 Previous Work

Binary Decision Diagrams (BDDs) are directed, acyclic binary trees that represent Boolean logic functions. BDDs were first introduced by Lee [41], and further developed as a representation by Akers [2]. The polynomial-time equivalence of a class of ordered BDDs was shown in [26].

A canonical form of BDD, called the *Reduced, Ordered* Binary Decision Diagram (ROBDD), is obtained by restricting the order of appearance of the variables and reducing the graph [12]. An input variable can appear only once along any path in the ROBDD, and the input variables must appear in the same order along any path. A reduction algorithm identifies and removes isomorphic subgraphs in the graph, producing a unique and canonical ROBDD.

The ROBDD shown in Figure 1-2 represents the function $x_1 \cdot x_3 + \overline{x_1} \cdot x_2$ under the input ordering x_1, x_2, x_3. There are 3 internal nodes in the graph representation of the function, and each internal node has an associated input variable. For a given input vector, the function represented by the graph is evaluated by traversing the nodes of the graph from the root vertex to a terminal vertex with value 1 or 0. For example, given the inputs $x_1 = 1$, $x_2 = 0$ and $x_3 = 0$, we start at the root vertex with the variable x_1. Since $x_1 = 1$, we look at the edge labeled 1, and see a node with variable x_3. Since $x_3 = 0$, we look at the edge labeled 0 and terminate with the result that the function evaluates to 0 with the input vector $x_1 = 1$, $x_2 = 0$ and $x_3 = 0$.

Operations on Boolean functions are implemented as graph algorithms on ROBDDs. Once ROBDDs have been created, the complement of a function can be computed in constant time, and the equivalence of two functions under the same input ordering can be determined in constant time [9]. Boolean operations such as the 2-input *and* can be computed in $O(size(G1) \cdot size(G2))$ time where $size(G1)$ and $size(G2)$ are the sizes of the graphs.

The efficiency of the ROBDD representation of a function depends significantly on the enforced global input ordering. For example, the Achilles Heel function grows exponentially for a poor input ordering, whereas with a good input ordering, the ROBDD size is $O(n)$, where n is the number of input variables. Consequently, the ROBDD input ordering problem has been the subject of intensive research (e.g. [5, 27, 43]). However, for some functions, including the hidden weighted bit (HWB) function and integer multipliers, ROBDDs are provably exponential in size, independent of the input ordering [13].

The ROBDD representation has gained widespread acceptance due to its canonicity and ease of manipulation. ROBDDs can be applied to a variety of problems in logic synthesis, testing, combinational verification and sequential verification. Variations of ROBDDs have also been used to represent special types of matrices and graphs [4, 18], to solve graph problems and integer programming problems [40]. The usefulness of ROBDDs has led to the exploration of other graph-based Boolean representations that are less restrictive, and more efficient in terms of memory use.

A slightly less restrictive BDD form, the Free Binary Decision Diagram (free BDD), also known as the read-only-once Branching Program (BP1), has been studied extensively by complexity theorists. Free BDDs have the property that all vertices are decision vertices, and each variable can appear at most once along any path from source to sink.

Free BDDs are computationally more powerful than ROBDDs. However, free BDD manipulation is currently impractical due to the complexity of manipulation and the difficulty in determining a good representation. The input variable ordering problem is further complicated in free BDDs because different paths may require different orderings. Note that ROBDDs are actually free BDDs restricted to the same global input ordering for every path.

The 2 free BDDs shown in Figure 1-3 represent the same function, $x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot (\overline{x_2} + \overline{x_3})$. The variables can appear in different orders along the different paths, as long as each variable appears no more than once along any path. Free BDDs are not canonical in the same sense as ROBDDs because different graphs can represent the same function, and equivalence is hard to determine.

A restricted form of free BDD, called the type-restricted free BDD was introduced in [6, 29]. Type-restricted free BDDs can be built and manipulated in a manner similar to ROBDDs. The ordering of the variables along the paths of the free BDDs are restricted by "types." The type-restricted free BDD retains the canonicity and manipulability features of ROBDDs, in addition to retaining some of the computational power of unrestricted free BDDs. However, the determination of a good type is crucial to finding a good free

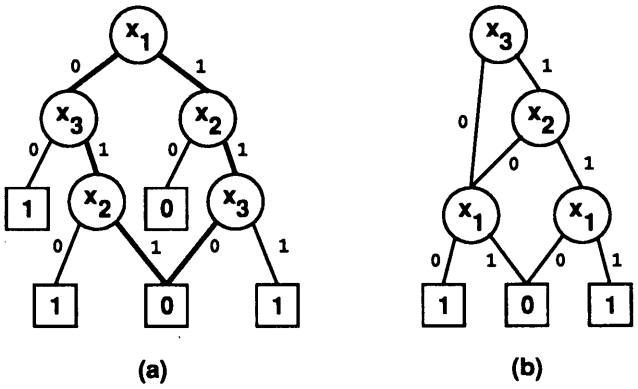


FIGURE 1-3: Two different free BDD representations for the function $x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot (\overline{x_2} + \overline{x_3})$. (a) The paths in the free BDD have different orderings. One path sees the ordering x_1, x_2, x_3 while another path sees the ordering x_1, x_3, x_2. (b) Another free BDD representation of the same function, with a different order of variables along the paths

BDD representation, and this problem is more difficult than the input ordering problem for ROBDDs.

Another powerful, ordered BDD representation was introduced as the Mod-2-OBDD in [30, 31]. The Mod-2-OBDD allows the *exclusive-or* function vertex, in addition to the decision vertices found in free BDDs and ROBDDs. Equivalence of Mod-2-OBDDs is determined probabilistically in polynomial time. Mod-2-OBDDs are more powerful than ROBDDs and exclusive-or sum-of-products representations.

Extended BDDs (XBDDs) introduced the *and* and *or* attributes on ROBDD edges [38]. These XBDDs were applied to the combinational verification problem by cofactoring the graphs until satisfiability could be determined. XBDDs sacrifice canonicity for a smaller representation. Boolean operations on XBDDs consist of adding additional edge attributes. The lack of a canonical form for XBDDs means that satisfiability, tautology and equivalence checks are computationally expensive, and can frequently result in exponential memory use.

If-then-else directed, acyclic graphs (DAGs) are graphs that represent Boolean functions using ternary nodes with *if*, *then*, and *else* children [39]. To maintain canonicity in If-then-else DAGs, the same restrictions imposed on ROBDDs are required for the DAGs, and consequently, these DAGs will exhibit the same behavior as ROBDDs. Without the restrictions, the DAGs are no longer canonical and difficult to manipulate.

Shared BDDs (SBDDs) are ROBDDs with the additional attribute edges: input inverter, output inverter and variable shifter [46]. Canonicity is maintained with the output inverter attribute, but the input inverter and variable shifter attributes result in a loss of canonicity. The proposed attributes may improve attributed ROBDD size by a con-

stant factor, but the same exponential behavior seen in ROBDDs will be displayed in attributed ROBDDs.

Several other BDD forms have been introduced for the formal verification of combinational circuits. These forms are used to compute the satisfiability of the *exclusive-or* of two circuits. In general, they trade canonicity and manipulability for a smaller representation, so the BDDs are useful for determining satisfiability, but are not viable Boolean logic representations.

Semi-Numeric Binary Decision Diagrams (SNBDDs) use the probabilistic equivalence theory of Blum, Chandra and Wegman [8] to verify the equivalence of two functions by computing integer signatures for the SNBDDs [35, 36, 37]. The method takes advantage of orthogonal partitions in circuits to replace subgraphs with integer signatures. The manipulation algorithms are ROBDD-based manipulation algorithms. Although it is possible to use SNBDDs to verify otherwise unverifiable circuits, the results are highly dependent on the circuit structure. Once the augmented ROBDDs have been built, further manipulation of the graphs as a Boolean logic representation is difficult.

General BDDs were used to verify combinational logic circuits, including 16-bit multipliers, by checking the satisfiability of the *exclusive-or* of the functions being verified. In [3], general BDDs were built by replicating inputs and using ROBDD manipulation algorithms to order and build ROBDDs for the difference function. The satisfiability of the difference is determined by smoothing away the replicated inputs.

Indexed BDDs (IBDDs) consist of k "layers" of OBDDs, where each OBDD layer in the graph is allowed a different global ordering. Satisfiability of the difference function is determined by removing inconsistent paths [7].

These generalized BDDs are typically difficult to order and construct. Once constructed, the BDDs are not easy to manipulate as function representations, and none of the forms is canonical. General BDDs and SNBDDs are useful for determining the satisfiability of the difference function of two circuits, but lack many of the characteristics that make ROBDDs efficient Boolean representations.

1.3 Thesis Contributions

This research considers a form of a function graph that requires each variable to appear no more than once in each path, although the order of appearance is not necessarily restricted. A data structure called the Free Boolean Diagram (FBD) is introduced. The FBD does not impose a global input ordering, and allows special function vertices, in addition to the conventional multiplexor vertices. This implies that FBDs are no longer

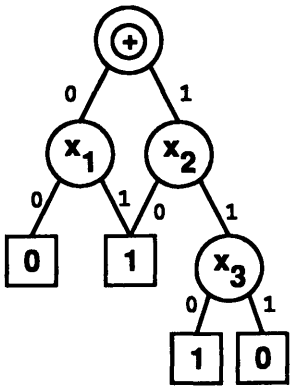


FIGURE 1-4: An FBD that represents $x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot (\overline{x_2} + \overline{x_3})$

decision diagrams, per se.

An FBD that represents the function $x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot (\overline{x_2} + \overline{x_3})$ is depicted in Figure 1-4. The root vertex of the graph represents a function, in this case the Boolean *exclusive-or* of x_1 and $\overline{x_2} + \overline{x_3}$. Since there are many different FBD representations for the same function, checking the equivalence of FBDs is also a difficult problem, like the equivalence of free BDDs.

However, the equivalence of FBDs can be determined efficiently using probabilistic methods. The equivalence check has an associated error probability, however, the upper bound on the error can be easily controlled. The accuracy of the probabilistic equivalence method is extremely high, and the error associated with the probabilistic method is orders of magnitude smaller than a comparable random simulation. The error also decreases exponentially with each additional probabilistic simulation, whereas the error of a random simulation decreases linearly with each additional vector applied.

Equivalence of FBDs is determined probabilistically in this thesis, and canonicity is maintained using "signatures". Probabilistic algorithms for the manipulation and construction of FBDs from multilevel circuits are presented and used to build two FBD manipulation packages. The first package is based upon the probabilistic equivalence theory presented by Blum, Chandra and Wegman [8]. The FBDs in this package allow function vertices such as AND, OR and XOR with restrictions on the children of these vertices. The second package is based upon the polynomial field theory of Gergov and Meinel [30]. The FBDs in this package allow for an AND function vertex as in the previous package and an unrestricted $\oplus$ function vertex. The implementation of the FBD package is analogous to the ROBDD manipulation package introduced in [9].

We also review some lower bounds for ROBDDs, free BDDs and FBDs on certain classes of circuits. The lower bound calculations show that for certain types of circuits, the

ROBDD complexity is inherently exponential, i.e., the ROBDD size will be exponential regardless of the input ordering. However, for some of these circuits, the free BDD and FBD representation is of polynomial size. Also, there exist classes of circuits that require exponentially-sized free BDD representations, but these circuits can still be represented by FBDs of polynomial size.

In this thesis, we apply FBDs to combinational logic verification and the Boolean satisfiability problem in channel routing. Two circuits can be checked for equivalence by constructing their associated FBDs and comparing the respective signatures. The satisfiability of a Boolean function is determined by building the FBD for the function and comparing the FBD signature with the 0 signature.

The organization of the thesis is as follows. Chapter 2 briefly reviews logic synthesis terminology. In Chapter 3, the ROBDD representation, manipulation algorithms, and the input ordering problem are described. Chapter 4 reviews the probabilistic equivalence check and the error bounds on the check. A comparison of the probabilistic method and random vector simulation is included in the chapter. FBDs based upon integer signatures are described in Chapter 5, and FBDs based upon polynomial signatures are the subject of Chapter 6. The implementation of an FBD package is discussed in Chapter 7. Application of FBDs to Boolean satisfiability is the topic of Chapter 8. Chapter 9 extends complexity results to FBDs, and we draw some conclusions in Chapter 10.

2

Notation and Definitions

2.1 Introduction

This section briefly reviews the notation and definitions of Boolean functions and graphs. For further information on Boolean functions, the reader is referred to [47]. For more detailed explanations of computer-aided design terminology, see [11, 23]. Graph definitions can be found in [19]. Section 2.4 defines some of the notation used with the FBD graph representations.

2.2 Graphs

A graph $G(V, E)$ is a set of *vertices* or *nodes*, V and a set of *edges*, E, consisting of pairs of vertices. If the edges are unordered pairs of vertices, the graph is called *undirected.* Ordered pairs of vertices result in *directed* graphs. A graph is *acyclic* if there are no cycles in the graph. The *out-degree* of a vertex v is the number of edges leaving v and the *in-degree* is the number of entering edges.

Two vertices u and v are *connected* if there exists an edge (u, v). A path is a sequence of vertices $(v_0, v_1, ..., v_k)$ such that the edges (v_{i-1}, v_i) exist for $1 \leq i \leq k$. Two graphs $G(V, E)$ and $G'(V', E')$ are *isomorphic* if there exists a one-to-one correspondence or bijection f such that $(u, v) \in E$ if and only if $(u' = f(u), v' = f(v')) \in E'$.

A *leaf* or *terminal vertex* is a vertex that has no outgoing edges. The *root* vertex is a node that has no incoming edges, and a graph is often identified by its root vertex. All other nodes are called *internal* nodes. For the edge (u, v), u is called the *parent* of v and v is called the *child* of u. A node u on the path from the root to node v is called an *ancestor* of v and v is a *descendant* of u.

2.3 Boolean Notation

A *Boolean function*, F, is a mapping from $B^n \rightarrow Y^m$ in Boolean space. A *Boolean equation*, f, represents a Boolean function in terms of a set of *variables* $\{x_1, x_2, ..., x_n\}$. A *literal* is a variable or its complement (e.g. $x_i, \overline{x_i}$). A *cube* is a set of literals and it is written as a conjunction of literals (e.g. $a \cdot b$). A *minterm* is a cube where every variable in the space must appear once. A variable that does not appear in a cube is called a *don't care*. The *ON-set* of a function $f(x_1, ..., x_n)$ is the set of cubes such that $f(x_1, ..., x_n) = 1$. The *OFF-set* is the set of cubes such that $f(x_1, ..., x_n) = 0$. Two cubes c_1 and c_2 are *disjoint* if $c_1 \bigcap c_2 = \phi$.

The *cofactor* or *restriction* of a function $f(x_1, ..., x_n)$ with respect to the variable x_i is denoted by $f_{x_i=1}$ or f_{x_i} where

$$f_{x_i}(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n) = f(x_1, ..., x_{i-1}, 1, x_{i+1}, ..., x_n).$$

Similarly, the cofactor of f with respect to the complement variable $\overline{x_i}$ is written as $f_{x_i=0}$ or $f_{\overline{x_i}}$ and

$$f_{\overline{x_i}}(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n) = f(x_1, ..., x_{i-1}, 0, x_{i+1}, ..., x_n).$$

The *Shannon expansion* of a function f around the variable x_i is

$$f = x_i \cdot f_{x_i} \; + \; \overline{x_i} \cdot f_{\overline{x_i}}.$$

A function can also be cofactored against cubes and other functions.

The *support* of a Boolean function f is defined as the set of variables upon which f depends. The support of $f(x_1, ..., x_n)$ is the set of variables $\{x_1, ..., x_n\}$. The supports of two functions f_1 and f_2 are disjoint if $support(f_1) \bigcap support(f_2) = \phi$. The two functions are *orthogonal* if $f_1 \bigcap f_2 = \phi$.

The symbol $\oplus$ refers to the *exclusive-or* (*xor*) function,

$$a \; \oplus \; b \; = \; a \cdot \overline{b} \; + \; \overline{a} \cdot b.$$

The symbol $\otimes$ refers to the *exclusive-nor* (*xnor*) function,

$$a \; \otimes \; b \; = \; \overline{a \; \oplus \; b} \; = \; a \cdot b \; + \; \overline{a} \cdot \overline{b}.$$

A *Boolean network* or *circuit* is a directed, acyclic graph where each node or gate, g_i has an associated Boolean function f_i. A *primary input* refers to a node with fanout, but no fanin, and it represents an external input to the network. A *primary output* is a node with fanin, but no fanout. It represents an external output of the network. All other nodes are called *internal* nodes.

2.4 FBD Notation

The *cardinality* of a field is the number of values in the field. For the algebraic field S, the cardinality is denoted by $\|S\|$. The *signature*, $|v|$, of the vertex v represents the set of values associated with that vertex. The vertex has one value for each run of the probabilistic equivalence check.

The *size* of the graph f is denoted by $size(f)$, and it represents the number of internal nodes in the graph. A graph can also be referred to by its root vertex. The notation, $AND(f_1, f_2)$, refers to an AND function vertex with low child f_1 and high child f_2. $!AND(f_1, f_2)$ indicates the AND vertex is complemented. The other function vertices will also be described in this manner. The formal definitions for FBDs and function vertices are presented in Chapters 5 and 6.

3

Reduced Ordered Binary Decision Diagrams

3.1 Introduction

A Boolean function representation that has gained widespread acceptance since its introduction by Bryant [12] is the *Reduced, Ordered* Binary Decision Diagram (ROBDD). The ROBDD representation of a Boolean function is canonical given a global input variable ordering, and operations on Boolean functions have corresponding graph algorithms on graphs. ROBDD canonicity and ease of manipulation has led to applications in logic synthesis, verification and testing.

In multilevel network optimization, local don't cares are used to optimize individual nodes in the network. The external, observability and satisfiability don't cares are expressed in ROBDD form as functions of the primary inputs. ROBDD-based image computation techniques compute the local don't care set for each node in terms of the fanin signals to the node, from the global don't cares [49].

ROBDDs have been used effectively in test pattern generation [17]. ROBDDs are used to compute the function that represents the difference between good and faulty circuits and input sequences that satisfy the function are valid test patterns.

In CMOS power dissipation estimation, the paths of the ROBDD can also be used to compute static signal probabilities because the ROBDD representation of a function is also a disjoint cover of the function [32].

Traditionally, simulation has been the predominant approach to circuit verification, but due to increasing circuit size and the presence of memory elements, exhaustive simulation is usually infeasible. Formal verification techniques can completely determine functional equivalence independently of any input sequences or patterns. ROBDDs were

used in the formal verification of combinational circuits by building the ROBDDs for the two functions being compared, and checking for equivalence [12, 43].

In sequential circuit verification, ROBDDs are used to represent sets of states and the state transition relations of finite state machines [15, 20]. ROBDD-based image computation techniques perform image and inverse-image computations on the state transition relations. The image computation method was further improved to operate on ROBDDs that represent the state transition functions [53].

The size of an ROBDD depends on the specified global input variable ordering and this ordering can affect representation size significantly. For example, the Achilles Heel function can be represented by an ROBDD with size linear in the number of input variables, yet with a poor ordering, the same function will require a representation with size exponential in the number of input variables.

For some functions, such as the integer multiplier and the hidden weighted bit function, it has been shown that the ROBDD representation size will grow exponentially, regardless of the given input ordering [13]. However, many common functions (such as adders and comparators) do have compact ROBDD representations, if a good input ordering is found. A number of heuristics have been developed for determining the input ordering of a circuit based upon the circuit characteristics [27, 43], and often these heuristics work well for common functions. An active ordering heuristic that dynamically repairs the global input ordering during the construction of ROBDDs was developed in [48]. Ultimately, for circuits such as large multipliers, ROBDDs cannot be built within reasonable computation time and memory limits, and for other circuits, hand orderings may still be required.

A good survey on the fundamentals and applications of ROBDDs can be found in [14]. This chapter reviews the definition of ROBDDs, the strongly canonical form for ROBDDs, followed by the basic manipulation algorithms, a discussion of the input variable ordering problem and its effect on ROBDD complexity. The data structures and algorithms for an efficient implementation of an ROBDD package are also reviewed [9].

3.2 ROBDD Representation

Definition 3.2.1 defines the representation of a Boolean function as a BDD graph.

Definition 3.2.1 A BDD F representing the Boolean function $f(x_1, ..., x_n)$ is a directed, acyclic graph with a root vertex v and defined recursively with two types of vertices:

1 If v is a terminal vertex and $value(v) = 1$ then $f^v = 1$. If $value(v) = 0$ then $f^v = 0$.

2 If v is a nonterminal vertex with $index(v) = i$, $1 \leq i \leq n$, and two children $high(v)$ and $low(v)$

$$f^v = x_i \cdot f^{high(v)} + \overline{x_i} \cdot f^{low(v)}$$

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

The *high* child of v or $high(v)$, represents the cofactor, f_{x_i}, of the vertex with respect to its decision variable. The *low* child or $low(v)$, represents the cofactor, $f_{\overline{x_i}}$, with respect to the complement of the decision variable. The high (low) child is also called the THEN (ELSE) child. The decision variable x_i is the input variable that appears in the i^{th} position in the global ordering.

An Ordered Binary Decision Diagram (OBDD) is a BDD subject to the restriction that the variables $\{x_1, ..., x_n\}$ appear in the same ordering along every path from root to terminal vertex. This restriction is called the *global input ordering* requirement. An OBDD is called *reduced* (ROBDD) if all vertices v and v' in the OBDD are distinct, (no isomorphic vertices) and the two children of the vertex v are distinct ($high(v) \neq low(v)$).

A non-terminal decision vertex has exactly two children, but multiple parents are possible. A vertex with multiple parents is called a *shared* node. The value of the variable associated with the vertex selects between the high child and the low child.

The non-terminal vertices in the ROBDD are also known as decision or multiplexor vertices because an ROBDD can be converted into a multiplexor-based circuit by replacing each vertex with a 2-input multiplexor. The data inputs to the multiplexor are the high and low child, and the decision variable associated with the vertex controls the select line.

The cubes of a disjoint cover of the function can be generated by traversing the paths of the ROBDD. Each cube corresponds to the conjunction of all literals encountered along a path from the root to the 1 terminal vertex, and all the cubes are disjoint from all other cubes in the cover. Likewise, a disjoint cover of the OFF-set of the function can also be generated by traversing all paths from the root to the 0 terminal vertex.

Figure 3-1 shows an example of a 6-node ROBDD for the 6-input Achilles Heel function $f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$ under the ordering $x_1, x_2, x_3, x_4, x_5, x_6$. The vertex v_0 is the root, and it is a decision vertex with decision variable x_1. The high child is vertex v_1 and the low child is v_3. In all the figures in this thesis, the left child will correspond

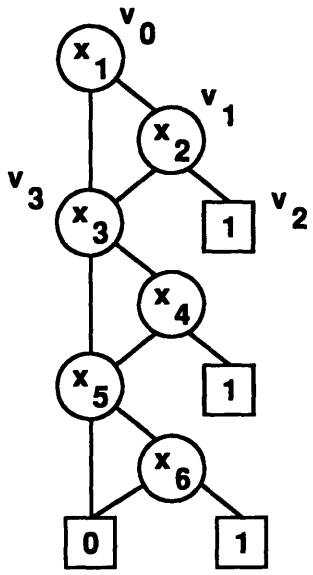


FIGURE 3-1: ROBDD for the Achilles Heel function $f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$ under the ordering $x_1, x_2, x_3, x_4, x_5, x_6$

to the low child and the right child will correspond to the high child, unless explicitly marked otherwise. Vertex v_2 is a terminal vertex with value 1.

An equivalent 14-node ROBDD representation for the same function, under a different ordering, is shown in Figure 3-2. The input variable ordering has a significant impact on the efficiency of the representation. This problem has been the subject of scrutiny and will be discussed in Section 3.5.

3.3 A Strongly Canonical Form

ROBDDs are canonical in the graph isomorphic sense; given an input ordering and a function, the ROBDD that represents the function is unique. An unreduced OBDD can be reduced by removing duplicate vertices resulting in an ROBDD where each distinct subfunction is represented by exactly one vertex. The reduction is accomplished by traversing the OBDD from the terminal vertices to the root, using the graph isomorphism equivalence test to eliminate duplicate vertices.

3.3.1 Equivalence

The equivalence of two OBDDs under the same ordering can be accomplished in time $O(size(G)\log(size(G)))$, where $size(G)$ is the size of the graphs. Two ROBDDs under the same ordering are equivalent if the graphs are isomorphic. If both graphs are reduced

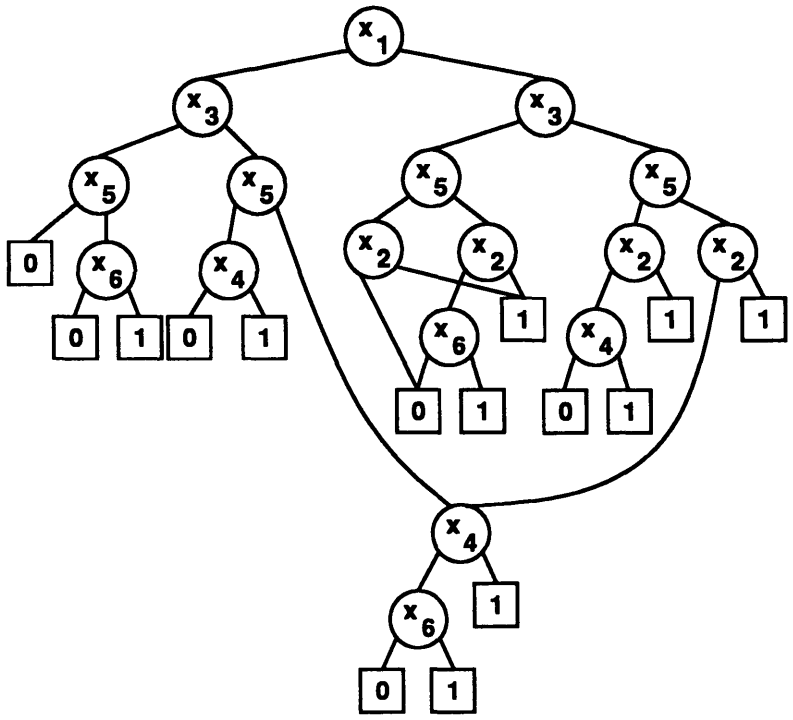


FIGURE 3-2: ROBDD for the Achilles Heel function $f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$ under the ordering $x_1, x_3, x_5, x_2, x_4, x_6$

with respect to each other, the equivalence test becomes a constant time test, namely the two ROBDDs v and v' are the same if they have the same index, $high(v) = high(v')$ and $low(v) = low(v')$.

If the two ROBDDs do not have the same ordering, then the equivalence test attains the complexity of determining equivalence for free BDDs. Under the same global input ordering, the equivalence of ROBDDs can be determined in polynomial time.

3.3.2 Reduce

Typically, ROBDDs are reduced as they are built, by identifying equivalent vertices and only building a new vertex if an equivalent vertex has not been located. However, if this is not the case, a reduction algorithm can identify graph isomorphisms and reduce the OBDD into an ROBDD in $O(size(G)\log(size(G)))$ time. An OBDD is reduced by traversing the graph from the terminal vertices to the root, labeling all unique vertices, and storing these vertices in a sorted list.

During this process, if we discover a vertex in the list with the same index and the same children, the nodes are equivalent and the existing vertex should be used.

In general, all the unique vertices encountered while constructing ROBDDs are stored

in a hash table, the *unique table.* Before creating a new ROBDD vertex, v, a hash table lookup is performed, using the children and the index of the new vertex. If the vertex already exists in the table, v is equivalent to the unique table node, and the existing node can be used.

3.4 Boolean Operations

Every Boolean arithmetic operation on a Boolean equation has a corresponding graph manipulation algorithm on an ROBDD. The following describes the fundamental ROBDD manipulation algorithms: *complement*, *cofactor*, and *apply* [12]. These algorithms are sufficient for constructing ROBDDs from multilevel combinational circuits.

3.4.1 Complement

The complement of an ROBDD G can be obtained in $O(size(G))$ time by traversing the graph, and changing the 1(0) terminal vertex into the 0(1) terminal vertex. Alternatively, Brace et al. introduced an attribute called the complement attribute [9]. The complement of an ROBDD can be obtained in constant time, simply by placing a complement attribute on G.

The attribute allows the same graph to represent both a function and its complement. An ROBDD without complement attributes may require almost twice as many nodes to represent the same function. The $n-$input parity function,

$$f = x_1 \oplus x_2 \oplus ... \oplus x_n,$$

requires $2n - 1$ nodes to represent f without complement attributes. An attributed ROBDD represents the same function with n nodes.

3.4.2 Cofactor

The cofactor (also known as restriction) algorithm simply parses a given graph G representing the function f and produces the function $f_{x_i=b}$ where x_i is the variable we are cofactoring against, and $b = 0$ or $b = 1$. The cofactor is obtained from the given graph by locating all nodes v with index i and setting all pointers to v to $high(v)$ if $b = 1$. All pointers are set to $low(v)$ if $b = 0$.

The cofactor function will create no more than $size(G)$ nodes, and in the case that $i \leq top\_index(G)$, the cofactor can be obtained in constant time without building any new nodes. The graph representing the cofactor is also guaranteed to have size $\leq size(G)$.

```
ROBDD_cofactor (F, i, phase)
{
    if (index(F) > i)
        return (F) ;
    if (index(F) == i) {
        /* The root vertex has the variable to cofactor against */
        if (phase == 1)
            v = high(F) ;
        else if (phase == 0)
            v = low(F) ;
    } else {
        /* recursively cofactor the children */
        create vertex v with decision variable x_i ;
        high(v) = ROBDD_cofactor (high(F), i, phase) ;
        low(v) = ROBDD_cofactor (low(F), i, phase) ;
        if (result = unique_table_lookup (v))) {
            Free low(v) and high(v) ;
            v = result ;
        }
    }
    return (v) ;
}
```

FIGURE 3-3: Pseudocode for the ROBDD cofactor algorithm

General cofactor is an $O(size(G)\log(size(G))$ operation. The basis for the efficient apply operation described in Section 3.4.3 is an efficient cofactor routine.

The pseudocode for the ROBDD general cofactor algorithm is shown in Figure 3-3. The first step in computing the cofactor is to check the root vertex for the variable we are cofactoring against. If the index of the variable is less than the root index, the variable is not present and the cofactor is simply the ROBDD F itself. This pruning measure is essential for an efficient cofactor algorithm.

If the variable is present, and it is the top variable in the ROBDD, the cofactor is the high child if we are cofactoring with respect to x_i and the low child if cofactoring against $\overline{x_i}$. If x_i is located deeper in the ROBDD, the cofactor function simply calls itself recursively. Once the new high and low children have been created, the cofactor routine checks for the existence of a node with index i and the same children. If the node does not already exist, a new node representing the cofactor is created.

A well-known property of cofactor and complement attributes is Observation 3.4.1.

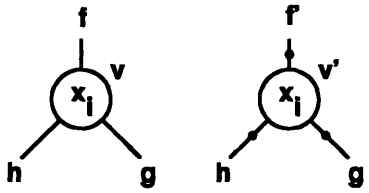


FIGURE 3-4: Different forms of the same function due to complemented edges

The implication for decision vertices is that there exist equivalent but different forms for the same function. For example, in Figure 3-4, the function f associated with vertex v is equivalent to the function f' associated with vertex v'. Canonicity can be enforced by using a standard that requires an uncomplemented high child for every vertex.

Observation 3.4.1 : $\overline{f} = x_i \cdot \overline{f_{x_i}} + \overline{x_i} \cdot \overline{f_{\overline{x_i}}}$.

3.4.3 Apply

The *apply* operation is an $O(size(G1) \cdot size(G2))$ routine that computes $f_1 \langle op \rangle f_2$. This operation is used to build ROBDDs for circuit networks, and for performing Boolean operations on Boolean functions. Apply recursively computes the expansion in Equation 3.1 to build a Boolean function. $f_1 \mid_{x_i=1} (f_2 \mid_{x_i=1})$ is the cofactor of $f_1(f_2)$ with respect to x_i. $f_1 \mid_{x_i=0} (f_2 \mid_{x_i=0})$ is the cofactor of $f_1(f_2)$ with respect to $\overline{x_i}$.

$$f_1 \langle op \rangle f_2 = x_i \cdot (f_1 \mid_{x_i=1} \langle op \rangle f_2 \mid_{x_i=1}) + \overline{x_i} \cdot (f_1 \mid_{x_i=0} \langle op \rangle f_2 \mid_{x_i=0}) \qquad (3.1)$$

To maintain the global input ordering, apply looks at the indices of the roots of the two input ROBDDs, and chooses the smaller index, i. The cofactors of both functions, f_1 and f_2, are computed, and apply is called recursively until the terminal vertices are reached. The apply operation is also known as the *ITE* or *If-Then-Else* function. The ITE function takes three argument f, g and h and computes $ITE(f,g,h) = f \cdot g + \overline{f} \cdot h$. All 2-input Boolean operations such as *and*, *or*, *xor* can be expressed in terms of the apply operation:

$$f \cdot g = ITE(f,g,0)$$

$$f + g = ITE(f,1,g)$$

$$f \oplus g = ITE(f,\overline{g},g)$$

The pseudocode for the ITE algorithm is shown in Figure 3-5. ITE begins by checking for early termination cases, such as $\text{ITE}(1,g,h) = g$. If the function call does not

```
ROBDD_ITE (f, g, h)
{
    /* special cases */
    if (result = early_termination_cases (f, g, h))
        return (result) ;
    canonicalize_triple (f, g, h) ;
    if (result = computed_table_lookup (f, g, h))
        return (result) ;
    /* select the top variable to cofactor against */
    x_i = minimum of (index(f), index(g), index(h)) ;
    /* recursively call ROBDD_ITE */
    create vertex v with decision variable x_i ;
    high(v) = ROBDD_ITE (f_{x_i}, g_{x_i}, h_{x_i}) ;
    low(v) = ROBDD_ITE (f_{x_i'}, g_{x_i'}, h_{x_i'}) ;
    /* check for an equivalent node */
    if (result = unique_table_lookup(v))) {
        Free low(v) and high(v) ;
        v = result ;
    }
    insert v = (f, g, h) into computed_table ;
    return (v) ;
}
```

FIGURE 3-5: Pseudocode for the ROBDD ITE algorithm

qualify for any early termination cases, the arguments are converted into a standard form, because certain argument triples are equivalent:

$$ITE(f,g,h) = ITE(\overline{f},h,g) = !ITE(f,\overline{g},\overline{h}) = !ITE(\overline{f},\overline{h},\overline{g})$$

The standard triple form allows a check to the *computed table* for any results from previous ROBDD_ITE calls. If a previous result is not available, the algorithm selects the decision variable with the smallest index at the root of f, g or h, and proceeds to call itself recursively with the cofactors as arguments. The cofactors are calculated in constant time, because the variable x_i is either in the root vertex of the function, or the variable is not present in the function at all. A new node is created to represent the result unless the node already exists in the unique table, in which case the result is recovered from the table.

3.5 Input Variable Ordering

The Achilles Heel example of Figure 3-1 shows that a good global input variable ordering is crucial for the creation of small ROBDD representations. For some classes of functions, such as multipliers and the hidden weighted bit (HWB) function, the ROBDD representation grows exponentially. For many common circuits, such as parity trees, comparators, adders, and incrementers, the representation is quite efficient provided a good ordering is used.

Researchers have devoted significant effort to ordering heuristics [16, 27, 34, 43, 48]. Most of the heuristics are based upon a pre-processing step that determines an ordering from the topological features of the circuit network. The variables can also be re-ordered during the construction of ROBDDs, but at a significant loss of computational efficiency [48].

Malik et al. [43] present two ordering heuristics, level and fanin, based upon information obtained from the multilevel network implementation of a circuit. Nodes in the network are levelized from the primary outputs to the primary inputs. The outputs are assigned level 0, and the level assigned to any particular node is the maximum level of its fanout + 1. For the level heuristic, nodes are sorted in decreasing level order, and this captures the idea that the primary inputs that occur deep in the network capture more information about the intermediate subfunctions in the outputs. The fanin heuristic is based upon the observation that if two functions have disjoint supports, the optimum ordering for the binary *or* or *and* of the two functions is simply the concatenation of the optimal orderings for each function separately. Hence, the second heuristic computes the level-based ordering for separate fanin cones and concatenates the suborderings into a global ordering.

The ROBDD package can also periodically re-order the variables, possibly reducing the size of the ROBDDs being constructed [48]. This *dynamic variable re-ordering* is done during garbage collection. The re-ordering changes the global ordering of input variables, and requires that all ROBDDs must be adjusted to satisfy the new ordering. The impact of this calculation is reduced by maintaining multiple unique tables, where the nodes are grouped based upon levels. All nodes with the same index variable are grouped into the same level. The re-ordering algorithms proposed by Rudell [48] rely on an efficient level swap algorithm that interchanges two levels of nodes. A sifting algorithm that trys all possible positions for a variable assuming all other variables are fixed, is proposed. For a number of benchmarks, the orderings obtained were able to produce significantly smaller ROBDDs, up to 45% smaller than the ROBDDs produced by static ordering heuristics.

3.6 Efficient ROBDD Package

Several useful speedups for manipulating ROBDDs were described in [9]. These speedups include a hash table for storing nodes, reference counters for tracking which nodes are in use, and a cache for storing previous results.

The efficient package uses a *unique table* to store all unique ROBDD nodes encountered. Before a new ROBDD node is created, the unique table is searched for an existing, equivalent node. The unique table is stored as an array of bins, where each bin contains a linked list of nodes. The bin for a node is located by hashing the address of the node. Increasing the number of bins in the unique table increases the memory required by the ROBDD package, but it reduces the size of the collision list of nodes in each bin. Smaller collision lists lead to a faster lookup time for nodes in the unique table.

The concept of a *computed table* is also introduced. The computed table is a hash cache that stores the results of recent ITE computations. The first step of the ITE function is to check the computed table for any pre-existing results. The computed table is essential for reducing the number of recursions necessary to create Boolean functions, and lowers the computation time required to create ROBDDs. Usually each hit in the computed table saves many recursive ITE calls, because a root ITE call makes many recursive calls. Since there are different forms of equivalent ITE arguments (also called triples), it is necessary to maintain a canonical triple form to maximize the computed table hit rate. The canonical form for the triple (f, g, h) is to have both f and g be uncomplemented ROBDDs.

Garbage collection during the creation of ROBDDs is accomplished by adding a reference counter field to each ROBDD node. The reference counter indicates the number of parents for the node. During garbage collection, all nodes with 0 reference counts are removed.

The amortized memory cost per ROBDD node for the ROBDD package is 22 bytes per node. This cost is obtained assuming each node requires 16 bytes of memory and the cost of the unique table bins, computed table bins and computed table entries is distributed, with 4 nodes in each bin.

The sizes of the unique table and computed table have a significant effect on the memory and CPU time required to build ROBDDs. Increasing the sizes of the tables tends to reduce computation time, at the expense of increasing the memory necessary to build the results. The frequency of garbage collection tends to have the same effect. The suggested scheme for automatic garbage collection and table sizes is to garbage collect when some memory limit is reached. After the limit is reached, the table sizes and the

limit are doubled.

3.7 Conclusion

ROBDDs have gained acceptance as an efficient graph representation for Boolean functions. Many applications in logic synthesis, testing and verification are based upon ROBDDs. ROBDDs are popular because of their canonicity, and ease of manipulation for many common functions. The efficiency of manipulation is directly related to the size of the ROBDD.

Hash tables and hash caches are used to speed up ROBDD manipulation. ROBDDs are reduced during creation, and equivalence is determined by checking for the existence of identical nodes. The global input ordering is determined heuristically.

The global input ordering requirement has a significant impact on ROBDD size. Heuristics for finding a good input ordering have been intensely researched, but for certain classes of circuits, ROBDDs are highly inefficient, independent of the chosen input ordering. This characteristic is a limitation of the graph representation, and provides the motivation for investigating alternate graph representations.

4

Probabilistic Equivalence

4.1 Introduction

The deterministic noncontainment problem for free BDDs is NP-complete [26]. The function f contains the function g if $g \subseteq f$. This means that for all input vectors $\boldsymbol{x}$ such that $g(\boldsymbol{x}) = 1$, $f(\boldsymbol{x}) = 1$ for the same input vectors. In other words, all minterms in the ON-set of g are also in the ON-set of f. Deterministic noncontainment means that we can give a *yes* or *no* answer to the question "does f contain g" without any possibility of error in the answer. Although researchers have been able to prove the NP-completeness of the noncontainment problem, no results have been proven for the deterministic equivalence of free BDDs. However, it was shown that the equivalence problem is likely to be equally difficult.

In contrast, the equivalence of free BDDs can be determined probabilistically by computing signatures for the graphs. The method is computationally efficient, and the error bounds are very low.

Blum, Chandra and Wegman show that the equivalence of free BDDs can be decided in random polynomial time by assigning signatures from a field to the inputs of the free BDD, and calculating the signatures for the vertices in the graphs [8]. The given algorithm applies to free BDDs, but it also applies to FBDs with function vertices, because the function vertices maintain the validity of the signature calculation.

Graphs with different signatures are definitely not equivalent. If two graphs have the same signature, there is a finite probability of error in assuming the graphs are equivalent. The probabilistic equivalence check is very powerful because the probability of error is bounded and decreases exponentially. Also, the algorithm is defined on an algebraic field, so different types of fields can be used in the actual implementation of the check. The

information in this chapter is valid for both an FBD package using integer values, and an FBD package using the polynomial values from a polynomial field of characteristic 2.

This chapter will review the probabilistic equivalence result of [8] in Section 4.2. More detailed bounds that are also applicable to FBDs were shown in [37] and will be rederived in Section 4.3. In Section 4.4, we compare of the probabilistic equivalence method and random vector simulation.

4.2 Probabilistic Equivalence Check

A free BDD is a decision diagram that has a root node, termination nodes, and internal nodes. The nodes are assigned variables, and each node, except for a terminal vertex, has two descendants, one corresponding to the literal x_i and one corresponding to $\overline{x_i}$, where x_i is associated with node i. Along any path from root to terminal vertex, each variable can be encountered at most once, in either complemented or uncomplemented form. Two graphs, $G_1(x_1, x_2, ..., x_n)$ and $G_2(x_1, x_2, .., x_n)$ are equivalent if the Boolean functions they represent are equivalent.

The algorithm for computing the equivalence of two free BDDs, G_1 and G_2, is shown in Figure 4-1. This algorithm can be repeated with different random assignments to the input variables x_i to reduce the probability of error.

The *signature* of a graph G, $|G|$, is the set of values assigned to the graph from the algebraic field. Multiple values can be assigned to each graph, one value for each random assignment to the input variables. Each assignment is called a *run* or *pass*.

The cardinality, or number of values, in the algebraic field S is denoted by $\|S\|$. The algorithm randomly assigns values from the algebraic field with cardinality of at least $\|S\| = 2n$ to the input variables. The complement signature $|\overline{x_i}|$ is defined as $1 - |x_i|$ and n is the number of input variables. The graphs are parsed from the bottom to the top, computing a signature for each vertex in the graph.

If two graphs have different values, the graphs must be different, because the value of a graph is simply the sum of the values of its minterms, and if the values are different, so are the minterms. By the same line of reasoning, if two graphs are equivalent, they must have the same value. If two graphs are not equivalent, there is a chance that they will still have the same value. However, for n inputs, there are at least $(\|S\| - 1)^n$ assignments to the variables out of $\|S\|^n$ possible assignments so that the values of the graphs will be different. The probability that the different graphs will be distinguished, denoted by $Pr\{|G_1| \neq |G_2| \text{ given } (G_1 \not\equiv G_2)\}$ is given by Equation 4.2. The terminology, $Pr\{A \text{ given } B\}$, refers to conditional probability [24], defined by:

Equivalence ($G_1(x_1, x_2, ..., x_n)$, $G_2(x_1, x_2, ..., x_n)$)
{
    Given an algebraic field S with at least $2n$ elements ;
    Assign values from S to the variables x_i ;
    For (all vertices v from terminal vertices to the root in G_1 and G_2)
        $|v| = |x_i| \cdot |high(v)| + |1 - x_i| \cdot |low(v)|$;
    if($|G_1| == |G_2|$)
        return (G_1 and G_2 are probably equivalent) ;
    else **if** ($|G_1| \neq |G_2|$)
        return (G_1 and G_2 are definitely not equivalent) ;
}

FIGURE 4-1: Probabilistic equivalence test

$$Pr\{A \text{ given } B\} = \frac{Pr\{A \cdot B\}}{Pr\{B\}} \tag{4.1}$$

$$Pr\{(|G_1| \neq |G_2|) \text{ given } (G_1 \not\equiv G_2)\} > \frac{(\|S\| - 1)^n}{\|S\|^n} \tag{4.2}$$

The probability that the non-equivalent graphs are identified equivalent is:

$$Pr\{(|G_1| = |G_2|) \text{ given } (G_1 \not\equiv G_2)\} = 1 - Pr\{(|G_1| \neq |G_2|) \text{ given } (G_1 \not\equiv G_2)\} \tag{4.3}$$

If we substitute Equation 4.2 into Equation 4.3, we get the following:

$$Pr\{(|G_1| = |G_2|) \text{ given } (G_1 \not\equiv G_2)\} < 1 - \frac{(\|S\| - 1)^n}{\|S\|^n} \tag{4.4}$$

The binomial approximation states that if $n << \|S\|$:

$$(1 - \frac{1}{\|S\|})^n > 1 - \frac{n}{\|S\|}; \quad n << \|S\| \tag{4.5}$$

The error probability bound becomes:

$$Pr\{(|G_1| = |G_2|) \text{ given } (G_1 \not\equiv G_2)\} < \frac{n}{\|S\|} \tag{4.6}$$

Equation 4.6 gives us the upper bound on the error probability associated with using the probabilistic equivalence check because the error associated with using the check is simply the probability that two different graphs have the same signature, denoted by $Pr\{(|G_1| = |G_2|) \cdot (G_1 \not\equiv G_2)\}$. The other error case never occurs because two equivalent graphs can never have unequal signatures so, $Pr\{(|G_1| \neq |G_2|) \cdot (G_1 \equiv G_2)\} = 0$. If we

rewrite Equation 4.6 using the definition of conditional probability in Equation 4.1, we get:

$$Pr\{(|G_1| = |G_2|) \cdot (G_1 \not\equiv G_2)\} < \frac{n}{\|S\|} \cdot Pr\{G_1 \not\equiv G_2\} \tag{4.7}$$

In the worst case, $Pr\{G_1 \not\equiv G_2\} = 1$ and the upper bound is still:

$$Pr\{(|G_1| = |G_2|) \cdot (G_1 \not\equiv G_2)\} < \frac{n}{\|S\|} \tag{4.8}$$

The probability of error can be decreased by making successive independent "runs" of equivalence checks using different values assigned to the input variables. In this case, let k be the number of runs. After k runs, we made an error in the probabilistic equivalence check of two graphs only if all runs returned an erroneous answer:

$$Pr\{\text{error in check}\} = Pr_k\{(|G_1| = |G_2|) \cdot (G_1 \not\equiv G_2)\} < \left(\frac{n}{\|S\|}\right)^k \tag{4.9}$$

When $\|S\| = 2n$ the probability of identifying two non-equivalent graphs as equivalent is less than $(\frac{1}{2})^k$. Once all the values for the vertices in a free BDD have been calculated, the equivalence test is simply a constant time check for equivalent values. Therefore, if signatures are always maintained with the nodes, this equivalence test is comparable to the ROBDD equivalence test utilizing pointer comparisons.

4.3 Error Probability Bounds

The probability that the probabilistic equivalence check made an error during the construction of graphs for a circuit is related to the number of nodes assumed equivalent based upon their signatures. The unique signatures and the associated nodes are stored in the *unique table*. The number of unique table hits, denoted by *ut_hits*, is the number of nodes assumed equivalent.

We assume that any error made by the probabilistic equivalence check during the construction of graphs for a circuit will be result in an error in the representation of the circuit. Error is introduced only when we assume two nodes are equivalent, so the probability that the circuit representation is correct, denoted by $Pr\{C \text{ is correct}\}$ is given by the probability that the equivalence check never made an error when using a node from the unique table.

$$Pr\{C \text{ is correct}\} > (1 - Pr_k\{(|G_1| = |G_2|) \cdot (G_1 \not\equiv G_2)\})^{ut\_hits} \tag{4.10}$$

Substituting the bounds from Equation 4.9:

$$Pr\{C \text{ is correct}\} > \left(1 - \left(\frac{n}{\|S\|}\right)^k\right)^{ut\_hits} \tag{4.11}$$

Taking the complement to get the probability that the representation C is defective, $Pr\{C \text{ is incorrect}\}$:

$$Pr\{C \text{ is incorrect}\} < 1 - \left(1 - \left(\frac{n}{\|S\|}\right)^k\right)^{ut\_hits} \tag{4.12}$$

Using the binomial approximation again, and assuming $ut\_hits << (\frac{n}{\|S\|})^{-k}$, we obtain the following upper bound on the probability that graph for the circuit is incorrect.

$$Pr\{C \text{ is incorrect}\} < \left(\frac{n}{\|S\|}\right)^k \times ut\_hits \tag{4.13}$$

The probability that we made an error during the equivalence test can be decreased by increasing the cardinality of the field, $\|S\|$, or by increasing the number of passes made. Given a desired probability of error, the appropriate number of passes and field cardinality can be selected to guarantee the error bound. The likelihood that an error occurred increases with the number of hits in the unique table.

To get an idea of the types of bounds obtained given some typical parameters, let $\|S\| = 2^{16} = 65,536$, $k = 4$, and $ut\_hits = 10^6$ for a circuit with $n = 100$ inputs. Substituting into Equation 4.13, the error probability associated with the graphs for the circuit is less than 5.42×10^{-6}. This bound is an absolute upper bound and it is not affected by the distribution of the functions in the circuit.

4.4 Comparison with Random Vector Simulation

The confidence in the probabilistic equivalence method is extraordinarily high, and a comparison with random vector simulation illustrates the differences between the two methods, and shows why the accuracy of the probabilistic method is much higher.

Given a circuit with n inputs, random pattern simulation can determine equivalence with another circuit exactly, but only if all of the possible 2^n input vectors are simulated. The worst case for random simulation occurs when comparing two functions that differ in only one minterm. The probability that two functions that differ in one minterm are identified as equivalent after the simulation of p vectors is:

$$1 - \frac{p}{2^n}$$

For each additional vector we simulate, we improve the error probability by:

$$\frac{1}{2^n}$$

To obtain an error bound similar to the upper bound on the probabilistic equivalence check, we would have to simulate a huge number of input vectors. If we set the error probabilities equal, the number of input vectors necessary is:

$$p = 2^n \cdot \left(1 - ut\_hits \times \left(\frac{n}{\|S\|}\right)^k\right)$$

We can use the values $\|S\| = 2^{16}$ and $k = 4$, for a circuit with 100 inputs and $ut\_hits = 10^6$. To determine the equivalence of two functions using random vector simulation to the same error bound as the upper bound for the probabilistic equivalence check, we would have to simulate over 99.999% of the 2^{100} possible input vectors. This gives an indication of the efficiency of the probabilistic method, and the degree of certainty involved with the answer given by the method. The probabilistic method also achieves this reliability very efficiently in all cases, in polynomial time, whereas the random simulation would require exponential computation time to attain the same result in the worst case.

In random vector simulation, the coverage is tied to the number of minterms that have been simulated. The error probability arises from the unsimulated minterms. The signature used by the probabilistic equivalence check represents all minterms, because all minterms contribute to the value of the signature. The error occurs because the number of possible signatures is smaller than the number of Boolean functions, so multiple functions have the same signature.

The number of possible n-input Boolean functions is 2^{2^n}. The number of unique signatures possible under the probabilistic equivalence check is $\|S\|^k$. For $\|S\| = 2^{16}$ and $k = 4$, the number of signatures is 2^{64}. Although this number is much smaller than the number of possible functions, it is also much larger than the number of functions we could see in a circuit. Also, since the number of signatures is controlled by adjusting the size of the field and the number of passes, the error bound on the probabilistic method can be adjusted until it is satisfactorily small.

For a more detailed discussion of the error bound, and comparisons with other verification techniques, including random vector simulation, see [37].

4.5 Conclusion

The equivalence of free BDDs and FBDs can be decided probabilistically in polynomial time by assigning signatures to the graphs. We reviewed the algorithm for computing

the signatures and the error bounds on the equivalence test. The specifics on the type of arithmetic and fields used in the probabilistic equivalence check for FBDs will be discussed in Chapters 5 and 6.

The probability that the equivalence check makes an error is less than $\left(\frac{n}{\|S\|}\right)^k$ where n is the number of inputs, $\|S\|$ is the cardinality of the field, and k is the number of runs. This bound can be reduced by choosing appropriate $\|S\|$ and k. The probability that an error occurred while constructing the free BDDs or FBDs for a circuit is bounded above by $ut\_hits \times \left(\frac{n}{\|S\|}\right)^k$.

The probabilistic equivalence check is a robust and computationally efficient method with a high degree of accuracy. The confidence in the check can be increased asymptotically to 1 by increasing the number of runs in the check, until the error probability is arbitrarily small. Increasing the number of runs also increases the runtime of the probabilistic equivalence check, but this is offset by the increasing speed and datapath widths of modern computers.

5

Free Boolean Diagrams Based Upon Integer Hashing

5.1 Introduction

A free BDD is a decision diagram made up of decision vertices labeled by an input variable, and two descendants, the high child and the low child. An input variable appears no more than once along any path from root to terminal vertex, but the order in which the variables appear is not restricted. This chapter introduces a graph-based representation for Boolean functions based upon the methodology of [8] for probabilistically determining the equivalence of two Free Binary Decision Diagrams (free BDDs) in polynomial time. This polynomial-time method is a significant improvement over deterministic methods for free BDDs because the deterministic noncontainment of two free BDDs is an NP-complete problem and deterministic equivalence is likely to be equally difficult [26].

Free Boolean Diagrams (FBDs) *differ* from free BDDs in that in addition to the decision variables, FBDs also permit the use of restricted OR, AND and XOR vertices [51]. FBDs are not decision diagrams per se, due to the presence of the function vertices. The probabilistic equivalence checking scheme remains valid under the restrictions on the function vertices. FBDs are not canonical in the graph isomorphism sense because two different FBDs with the same input variable ordering can represent the same function, but a form of canonicity is enforced by allowing the existence of only one graph for each unique hash signature.

We present FBD manipulation algorithms that can be used to build FBDs for multi-level combinational circuits. These algorithms are comparable to ROBDD algorithms, in terms of memory use and computation time. We also give a methodology for introducing function vertices into the FBDs. For a number of benchmarks, we were able to obtain

FBDs that were significantly smaller than the ROBDDs.

This chapter is organized in the following fashion. The first section describes the probabilistic equivalence check with integer modular arithmetic fields. The definition of the FBD representation is presented in Section 5.3. A strongly canonical form and graph reduction are described in Section 5.4. Section 5.5 shows how the manipulation algorithms for FBDs parallel the graph algorithms for conventional ROBDDs. Results obtained for some benchmarks using the FBD package are presented in Section 5.6, followed by the conclusion.

5.2 Probabilistic Equivalence

The probabilistic equivalence check described by Blum et al. [8] is defined for an algebraic field. One such field is a field of non-negative integers defined over modular-p integer arithmetic, where p is prime. The values are integers between 0 and $p-1$, inclusive, and all arithmetic operations in the field are modular-p operations [37].

Addition and multiplication modulo p, denoted by $+_p$ and $\cdot_p$, are defined as [19]:

$$a +_p b = (a + b) \ mod \ p$$

$$a \cdot_p b = (a \cdot b) \ mod \ p$$

The following is an example of a single pass signature calculation for the graph shown in Figure 5-1, using the prime number $p = 331$ as the modulus for the modular arithmetic. The set of possible values is $\{0, 1, 2, ..., 330\}$. The signature equations for all the nodes in the graph can be determined from the bottom of the graph up, and the equations are:

$$|v_2| = |x_2|$$

$$|v_1| = |x_1| + |1 - x_1| \cdot |v_2|$$

$$|v_0| = |x_0| \cdot |v_1|$$

One possible random assignment of integer values to the variables is $|x_0| = 100$, $|x_1| = 200$, and $|x_2| = 300$. Substituting these values into the signature equations, and using modular arithmetic, the result is $|v_2| = 300$, $|v_1| = 80$ and $|v_0| = 56$. Note that the complement value $|1 - x_1|$ is computed using the formula $1 + p - |x_1|$.

Given a different input ordering, but the same assignment of values to the primary inputs, the signature of the apparently different graph, in Figure 5-2, can be computed, obtaining $|w_0| = 56$ for the graph. This signature is equivalent to the signature obtained

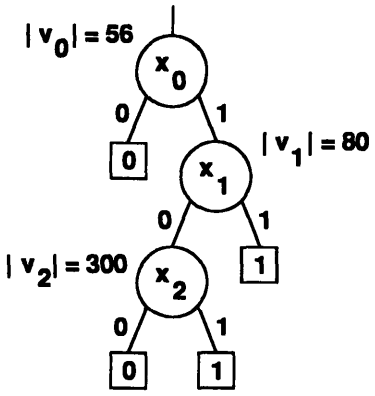


FIGURE 5-1: Example of a 1 pass signature calculation with $|x_0| = 100, |x_1| = 200, |x_2| = 300, p = 331$

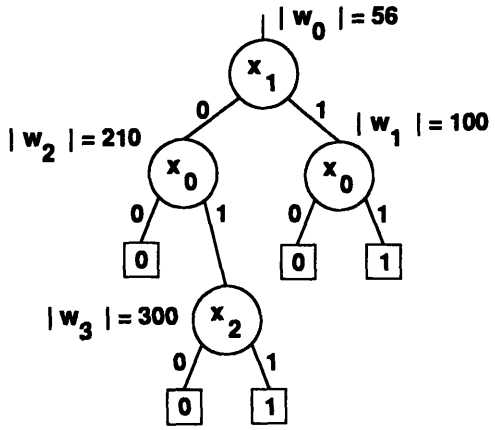


FIGURE 5-2: Example of a 1 pass signature calculation under a different ordering with $|x_0| = 100, |x_1| = 200, |x_2| = 300, p = 331$

for Figure 5-1, so the two graphs are assumed to represent the same function, with a finite probability of error less than $\frac{n}{\|S\|} = \frac{3}{331}$.

5.3 FBD Representation

An FBD is a read-only-once graph with terminal vertices, decision vertices, AND, OR, XOR, NAND, and NOR vertices. The terminal vertices and decision vertices are identical to the vertices defined for ROBDDs. The other types of vertices are called function vertices, and they represent functions under certain constraints. The support of an FBD f, $support(f)$, is the set of decision variables encountered on the paths from root to terminal vertex. An FBD is defined as follows:

Definition 5.3.1 An FBD representing the Boolean function $f^v(x_1, ..., x_n)$ is a directed, acyclic graph with a root vertex v defined recursively with the following types of vertices:

1 A terminal vertex v has no children and it represents the function:

1. $f^v = 1$ if $value(v) = 1$. The signature of the 1 terminal vertex is defined as $|v| = 1$.
2. $f^v = 0$ if $value(v) = 0$. The signature of the 0 terminal vertex is defined as $|v| = 0$.

2 A decision vertex v has a decision variable x_i, index i, a high child $high(v)$ and a low child $low(v)$. The function represented by the decision vertex is:

$$f^v = x_i \cdot f^{high(v)} + \overline{x_i} \cdot f^{low(v)}$$

The signature for the decision vertex is computed as:

$$|v| = |x_i| \cdot |high(v)| + |1 - |x_i|| \cdot |low(v)|$$

3 An AND vertex v has a high child $high(v)$ and a low child $low(v)$. The restriction on the children of the AND vertex is:

$$support(high(v)) \cap support(low(v)) = \phi$$

The function represented by the AND vertex is the Boolean *and* of the children:

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

The signature of the AND vertex is the product of the signatures of the children:

$$|v| = |high(v)| \cdot |low(v)|$$

4 An OR vertex v has a high child $high(v)$ and a low child $low(v)$. The restriction on the children of the OR vertex is that the children must be orthogonal. We have:

$$f^{high(v)} \cap f^{low(v)} = \phi$$

The function represented by the OR vertex is:

$$f^v = f^{high(v)} + f^{low(v)}$$

The signature of the vertex is the sum of the signatures of the children. We have:

$$|v| = |high(v)| + |low(v)|$$

5 A NAND vertex v has a high child $high(v)$ and a low child $low(v)$. The restriction on the children of the NAND vertex is:

$$support(high(v)) \cap support(low(v)) = \phi$$

The function represented by the NAND vertex is:

$$f^v = \overline{f^{high(v)} \cdot f^{low(v)}}$$

The equation for computing the signature of the vertex is:

$$|v| = |1 - |high(v)| \cdot |low(v)||$$

6 A NOR vertex v has a high child $high(v)$ and a low child $low(v)$. The restriction on the children of the NOR vertex is that the children must be orthogonal:

$$f^{high(v)} \cap f^{low(v)} = \phi$$

The function represented by the NOR vertex is:

$$f^v = \overline{f^{high(v)} + f^{low(v)}}$$

The equation for computing the signature of the vertex is:

$$|v| = |1 - |high(v)| - |low(v)||$$

7 A restricted XOR vertex v has two children and the supports of the children must be disjoint:

$$support(high(v)) \cap support(low(v)) = \phi$$

The equation for computing the signature of the vertex is:

$$|v| = |1 - |high(v)|| \cdot |low(v)| + |high(v)| \cdot |1 - |low(v)||$$

The restrictions on the children of the function vertices guarantee the correctness of the signature calculation. NAND and NOR vertices are necessary for node replacements that may occur due to issues discussed in later sections.

In the decision vertex, the function represented by the high child, $f^{high(v)}$, is simply the cofactor of the function with respect to the decision variable of the vertex, x_i. Likewise, the low child is the cofactor of the function with respect to $\overline{x_i}$.

For the function vertex, the high and low children are interchangeable, and the vertex lacks an associated decision variable. Along any path from root to terminal vertex, multiple function vertices can be encountered, however each decision variable will be encountered no more than once. Modular arithmetic with a large prime number as the modulus, such as $p = 32341$, is used to compute the signatures.

In Figure 5-3, we have an example of an FBD. This FBD represents the function $f = x_2 \cdot (\overline{x_1 \oplus x_3}) + x_4 \cdot (x_1 \oplus x_3)$. This FBD representation is valid under the ordering (x_1, x_2, x_3, x_4), and it requires 7 nodes. For each vertex, the left child corresponds to the low child, and the right child corresponds to the high child unless otherwise labeled. The root vertex, v_1, is an OR vertex. The high child represents the function $f^{high(v_1)}$ shown in Equation 5.1. The low child represents the function $f^{low(v_1)}$ shown in Equation 5.2.

$$f^{high(v_1)} = x_2 \cdot (\overline{x_1 \oplus x_3}) \tag{5.1}$$

$$f^{low(v_1)} = x_4 \cdot (x_1 \oplus x_3) \tag{5.2}$$

Note that the functions represented by the children of v_1 are orthogonal.

$$f^{high(v_1)} \cap f^{low(v_1)} = \phi \tag{5.3}$$

Vertices v_2 and v_3 are AND vertices. The children of the vertex v_2 have disjoint support sets.

$$\{support(high(v_2)) = \{x_2\}\} \cap \{support(low(v_2)) = \{x_1, x_3\}\} = \phi \tag{5.4}$$

Edges marked by dots are complemented edges. The low child of vertex v_2 is complemented, and so is the high child of vertex v_5. The equivalent ROBDD representation in Figure 5-4 without function vertices under the same ordering requires 8 nodes. In this case, a better ROBDD can be obtained with the ordering (x_2, x_4, x_1, x_3), but this is not always possible, especially in the case of multiple-output functions that require different orderings for different outputs. The function vertices "simulate" different orderings along different paths in the FBD, allowing smaller representations.

Some additional properties of FBDs are stated in the following theorems. Theorem 5.3.1 shows that the cofactors of two orthogonal functions are also orthogonal. Theorem 5.3.2 shows that the cofactors of two functions with disjoint supports also have disjoint supports. This result guarantees that the FBDs with function vertices at the root can be cofactored by cofactoring the children. The result also has the same type of function vertex at the root, and the children satisfy the required constraints.

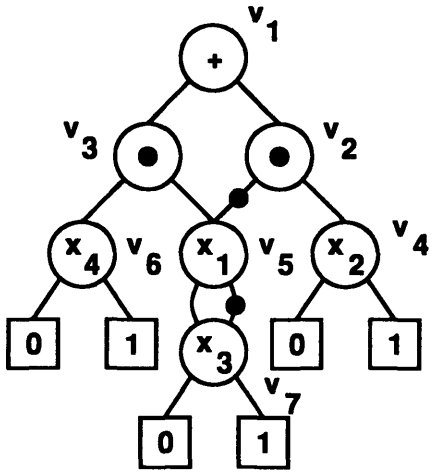


FIGURE 5-3: An FBD that has decision, OR and AND vertices

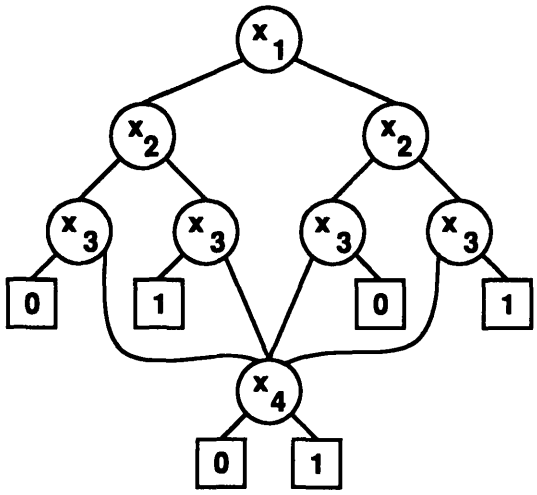


FIGURE 5-4: An equivalent ROBDD for the same function under ordering (x_1, x_2, x_3, x_4)

Theorem 5.3.1 : If $f \cap g = \phi \Rightarrow f_{x_i=b} \cap g_{x_i=b} = \phi$.

Proof:

$$\begin{aligned} & f \cdot g = 0 \\ \Rightarrow \quad & (x_i \cdot f_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}}) \cdot (x_i \cdot g_{x_i} + \overline{x_i} \cdot g_{\overline{x_i}}) = 0 \\ \Rightarrow \quad & x_i \cdot f_{x_i} \cdot g_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}} \cdot g_{\overline{x_i}} = 0 \\ \Rightarrow \quad & f_{x_i} \cdot g_{x_i} = 0 \\ \Rightarrow \quad & f_{\overline{x_i}} \cdot g_{\overline{x_i}} = 0 \end{aligned}$$

■

Theorem 5.3.2 : If $support(f) \cap support(g) = \phi \Rightarrow support(f_{x_i=b}) \cap support(g_{x_i=b}) = \phi$.

Proof: $support(f_{x_i=b}) \subseteq support(f)$ and $support(g_{x_i=b}) \subseteq support(g)$ ■

Proposition 5.3.1 : $f \cap g = \phi \Rightarrow support(f) \cap support(g) \neq \phi$.

Proposition 5.3.1 states that two orthogonal functions must have common support variables. Two functions with nonintersecting supports cannot be orthogonal.

5.4 A Strongly Canonical Form

A strongly canonical form for FBDs can be maintained using the probabilistic equivalence method. Canonicity is maintained by allowing the existence of only one FBD with a particular signature. A *unique table* stores the FBDs associated with all unique signatures in use.

FBDs are reduced as they are created, from the terminal vertices to the root. Before a new vertex, v_j, is created, we calculate the signature of the new vertex and check for the existence of that signature in the unique table. The vertex signatures are calculated using the formulas given for each type of vertex in Definition 5.3.1.

If another node v_i with signature $|v_i|$ is encountered in the unique table, and the signatures $|v_i| = |v_j|$, the nodes v_i and v_j can be assumed equivalent with a quantifiable probability of error. Vertex v_i can be substituted for vertex v_j, or vice versa. Likewise, if $|v_i| = |\overline{v_j}|$, then $v_i \equiv \overline{v_j}$ and the complement of v_j (v_i) can be substituted for v_i (v_j).

If the FBDs represented by v_i and v_j are isomorphic, then no additional probability of error is introduced. However, if the graphs are not isomorphic, the error introduced by the probabilistic equivalence check is still bounded.

5.5 Boolean Operations

The routines necessary for manipulating the FBD representation as a Boolean logic representation are described here. An FBD analog exists for every ROBDD manipulation algorithm. The complexity of the manipulation algorithms is a function of graph size, so an efficient graph representation consumes less memory and is easier to manipulate.

The algorithms, *complement*, *cofactor*, and *apply*, form the complete set of operations necessary for building FBDs from multilevel combinational circuits. Each algorithm operates on FBDs with function vertices to produce a result that is also a valid FBD. During the construction of FBDs, function vertices are also introduced to create smaller graphs by exercising the unordered nature of FBDs.

5.5.1 Complement

The complement of an FBD node is marked by a complement attribute on an edge. This scheme is identical to marking complements in an ROBDD. The signature of a complemented vertex v is simply $|1-|v||$. The complement of an FBD cannot be obtained by interchanging 0 and 1 terminal vertices, due to the presence of function vertices.

5.5.2 Cofactor

The FBD package requires a general cofactor algorithm due to the presence of function vertices. In both the ROBDD and FBD packages, when cofactoring a graph with respect to a decision variable that is not at the root of the graph, a general cofactor is required to compute the result. In the case of FBDs, some graphs have function vertices at the root. In these graphs, a general cofactor is needed, regardless of the decision variable we would like to cofactor against.

The fundamental steps of the general cofactor algorithm are outlined in Figure 5-5. The first step is a pruning step, where we check the support of the FBD f for the decision variable x_i. If the variable is not present in f, the cofactor is simply f. Otherwise, we need to check for other cases.

If the root vertex is a decision vertex, and the decision variable of the root is the variable we wish to cofactor against, then the procedure is done. Simply choose the high or low child, depending on the phase of the variable.

Otherwise, the cofactor procedure is called recursively on the high and low children to obtain the children for the new node. However, before a new node is created, the signature for the node is computed and used in a unique table lookup. If an equivalent

```
FBD_cofactor(f, i, phase)
{
    if (x_i ∉ support(f))
        return (f) ;
    if (type(f) is a decision vertex and index(f) == i) {
        if (phase == 1)
            return (high(f)) ;
        else if (phase == 0)
            return (low(f)) ;
    }
    create vertex v with index index(f) ;
    high(v) = FBD_cofactor (high(f), i, phase) ;
    low(v) = FBD_cofactor (low(f), i, phase) ;
    compute signature |s| of (index(f), low(v), high(v)) ;
    if (result = unique_table_lookup (|s|))) {
        if (size(result) < size(v)) {
            Free low(v) and high(v) ;
            v = result ;
        }
    }
    return (v) ;
}
```

FIGURE 5-5: Pseudocode for the general FBD cofactor algorithm

node was not located, the new node is created.

The general cofactor is a fairly expensive operation if the variable we are cofactoring against is located deep in the FBD f. In the worst case, cofactor may traverse all $size(f)$ nodes in the graph and create $O(size(f))$ new nodes. If the variable is located near the top of the FBD, then the number of recursions required will be small.

The size of the FBD, $f_{x_i=b}$, that represents the cofactor, is bounded by the size of the original function, $size(f_{x_i=b}) \leq size(f)$.

Figure 5-6 shows the result of cofactoring an FBD with respect to $x_1 = 1$. All pointers that point to node v_5 in the original FBD on the left are readjusted to point to the high child, v_7. The cofactored FBD has the same function vertices as the original FBD, and the validity of the function vertices is maintained.

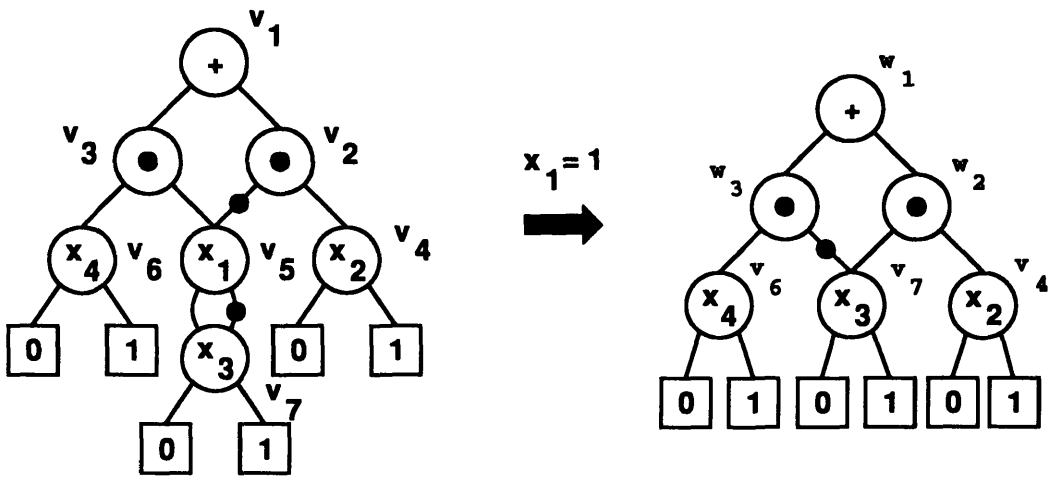


FIGURE 5-6: An example of general cofactor for FBDs

Support Field

In the ROBDD package, the global ordering is an important pruning measure during cofactor. If the variable we are cofactoring against has an index smaller than the index at the root, we automatically know that the variable is not present in the ROBDD, and the result of the cofactor is simply the original function.

For FBDs without a global ordering, this pruning measure is unavailable. Namely, the cofactor algorithm cannot determine a priori if a variable is present in the FBD. Cofactor needs to recurse to the terminal vertices before realizing a variable is not present.

The purpose of the support field for the FBD is to provide the required pruning measure. The support field contains the indices of all decision variables that appear in the FBD. If a variable does not appear in the support, we know the cofactor of f with respect to the variable or its complement is simply f itself.

The support field pruning measure is more effective than the index pruning measure for the general ROBDD cofactor algorithm. If the index of the variable is greater than the index at the root of the ROBDD, but the variable is not present in the ROBDD, the ROBDD cofactor algorithm will recurse until the index pruning measure is satisfied. The FBD cofactor algorithm with the support field will be able to identify the case without further recursions.

The support field, unfortunately, can consume a large amount of memory if the number of inputs is large, because each input requires one bit. The memory requirement can be ameliorated by calculating the support of an FBD only when the support is required, and removing the support field when it is not necessary. This will increase the computation time, but save memory.

5.5.3 Apply

The ITE function is the 3-input If-Then-Else function capable of implementing all 2-input Boolean functions. The ITE algorithm for FBDs is shown in Figure 5-7. The arguments are FBDs representing Boolean functions f, g, and h. The result computed by ITE is an FBD that represents $f \cdot g + \overline{f} \cdot h$.

The procedure starts by checking the inputs for special cases. In addition to the ROBDD special cases, there are also a number of special cases unique to FBDs, for example, $AND(f,g) \cdot f = AND(f,g)$. For a list of special cases, see Appendix A.

If the ITE call does not fit any of the special cases, the arguments need to be canonicalized. The arguments are ordered based upon their signatures. The arguments can now be used to look for a previous result in the computed table hash cache.

If a precomputed result is not available, ITE checks the supports of the arguments f, g and h. If the support of f is disjoint from the supports of g and h, the FBDs can be concatenated. For binary functions, one of the arguments f, g or h is a terminal vertex, so usually only two FBDs need to be concatenated with an AND vertex. If all three FBDs are nontrivial, the FBDs can still be concatenated using two AND vertices and an OR vertex:

$$ITE(f,g,h) = OR(AND(f,g), AND(\overline{f},h))$$

AND Node Introduction

In the 2-input case, AND nodes are introduced during the computation of the *and* of two graphs, f_1 and f_2, with disjoint supports. Since the support information for the input FBDs is stored with the node, the support check is simple, and the result FBD R requires the creation of at most one extra FBD node. Figure 5-8 shows an example of the concatenation of two graphs. f_1 becomes the high child for the AND vertex, and f_2 becomes the low child.

The final size of R is also bounded, namely $size(R) \leq size(f_1) + size(f_2) + 1$. If both f_1 and f_2 are optimal representations, the size of R is guaranteed to be at most one node greater than the smallest-sized FBD. Under a global input ordering, the AND node version of the function may be the smallest possible representation of the result.

Once AND nodes have been introduced, they will be present in subsequent operations. However, they tend to be pushed down towards the bottom of FBDs created with cofactor and ITE, because only decision variables are selected.

The benefit of using AND nodes instead of just concatenating the graphs as shown in Figure 5-9 is that when the graphs are concatenated, variables that appear earlier in the

```
FBD_ITE (f, g, h)
{
    /* special cases */
    if (result = special_cases (f, g, h))
        return (result) ;
    FBD_canonicalize_triple (f, g, h) ;
    if (result = computed_table_lookup (f, g, h))
        return (result) ;
    if (support(f) ∩ (support(g) ∪ support(h)) = φ) {
        result = FBD_concatenate (f, g, h) ;
        return (result) ;
    }
    /* select the top variable to cofactor against */
    x_i = select_variable (f, g, h) ;
    /* recursively call FBD_ITE */
    create vertex v with decision variable x_i ;
    high(v) = FBD_ITE (f_{x_i}, g_{x_i}, h_{x_i}) ;
    low(v) = FBD_ITE (f_{x̄_i}, g_{x̄_i}, h_{x̄_i}) ;
    compute signature |s| of result = (x_i, low(v), high(v)) ;
    if (result = unique_table_lookup (|s|)) {
        if (size(result) < size(v)) {
            free(low(v)) ; free(high(v))
            v = result ;
        }
    }
    if (result = introduce_or (f, g, h)) {
        if (size(result) < size(v)) {
            free(low(v)) ; free(high(v))
            v = result ;
        }
    }
    if (result = retain_or (f, g, h)) {
        if (size(result) < size(v)) {
            free(low(v)) ; free(high(v))
            v = result ;
        }
    }
    insert v = (|f|, |g|, |h|) into computed_table ;
    return (result);
}
```

FIGURE 5-7: Pseudocode for the FBD ITE algorithm

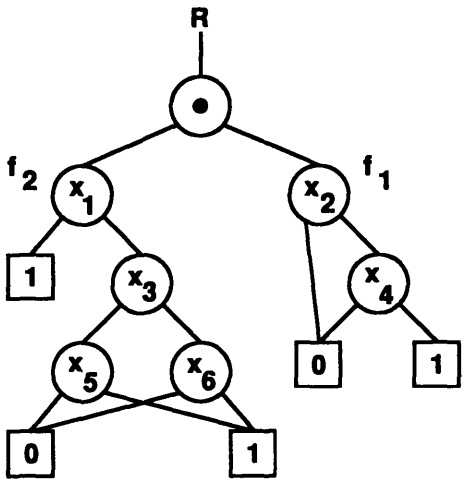


FIGURE 5-8: Concatenation of two graphs with an AND vertex

global ordering may be placed deeply in the result graph, increasing the complexity of further manipulations with this particular graph.

Also, an actual sequential concatenation of the two graphs may require the creation of more nodes. The maximum number of nodes that may need to be created by concatenate is $MAX(size(f_1), size(f_2))$ which could be much larger than the one node required for concatenation with an AND vertex.

The Boolean *or* of two graphs f_1 and f_2 can also be computed using an AND vertex and DeMorgan's law, $f_1 + f_2 = \overline{\overline{f_1} \cdot \overline{f_2}}$, if the supports of the graphs are disjoint. The result R is shown in Figure 5-10.

XOR nodes are subject to the same support restriction as AND vertices, and can be introduced when an *xor* or *xnor* function is encountered. The XOR nodes are built in exactly the same fashion as AND vertices, after a support check on the arguments.

Input Variable Ordering

A variable is selected and ITE calls itself with the cofactors as arguments. A difference between the ITE for FBDs and ROBDDs is the use of a general cofactor algorithm during ITE. In the case of ROBDDs, the top index variable, namely the variable that appears earliest in the ordering, will always be located at the root of the arguments to the ITE, if present. While creating FBDs, the variable chosen may not be at the root, and the cofactor call becomes recursive. If the selected variable is located deep in the FBDs, cofactor becomes prohibitively expensive.

The FBDs are not subject to a global input variable ordering, but this presents a number of complications. The determination of the ordering along each path is a difficult

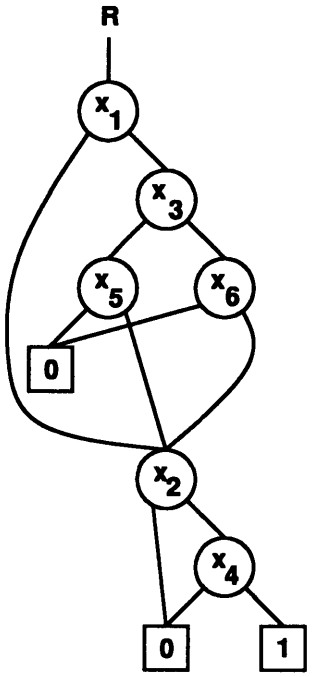


FIGURE 5-9: Concatenation of two graphs without any AND vertices

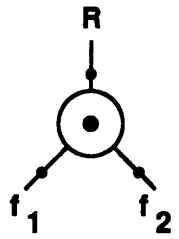


FIGURE 5-10: $R = f_1 + f_2$ with an AND vertex, $support(f_1) \cap support(f_2) = \phi$

problem, and any complicated heuristic for selecting the variable causes the complexity of the ITE algorithm to grow. Also, if the variables can appear in any order, the complexity of the cofactor computation grows.

As a result, a global input variable ordering is useful while building FBDs. The global ordering simplifies the variable selection process, and reduces the number of recursions made by the cofactor algorithm. For our purposes, we simply obtain the global ordering using ordering techniques for ROBDDs.

Even with the global ordering restriction, the constrained function vertices allow for the construction of FBDs that "simulate" free BDDs, because FBDs are essentially unordered read-only-once graphs.

After the results of the recursive calls to ITE are available, a new FBD node is created. The new FBD node is a representation of the result, but now a series of signature checks will determine whether a smaller representation can be easily obtained. The first check is the unique table check. If an equivalent node is located in the table, and the unique table node is a smaller representation, then that representation will be used instead of the newly created node.

OR Node Introduction

The second check is for OR node introduction. OR nodes are introduced during the computation of the *or* of two orthogonal functions f_1 and f_2. The orthogonality of the two functions cannot be tested up front, without computing the intersection of the functions. Hence, once $f_1 + f_2$ has been computed without an OR node at the root, the check for OR node introduction becomes:

$$|f_1 + f_2| = |f_1| + |f_2|$$

If the signature of the result is equivalent to the sum of the signatures of f_1 and f_2, then we assume f_1 and f_2 are orthogonal functions, and an equivalent representation of the result is $OR(f_1, f_2)$, which is an OR vertex with low child, f_1, and high child, f_2. The smaller representation is chosen for use.

For example, suppose we want to calculate $f = f_1 + f_2$ where $f_1 = x_2 \cdot (\overline{x_1 \oplus x_3})$ and $f_2 = x_4 \cdot (x_1 \oplus x_3)$. Clearly, f_1 and f_2 are orthogonal and the representation of $f_1 + f_2$ with an OR node at the root is shown in Figure 5-3, with vertex v_2 representing f_1 and vertex v_3 representing f_2. In this case, we use the OR node representation, because the alternative under ordering (x_1, x_2, x_3, x_4) is larger.

By DeMorgan's law, OR nodes can also be introduced in the case of $f_1 \cdot f_2 = \overline{\overline{f_1} + \overline{f_2}}$ and the condition $\overline{f_1} \cap \overline{f_2} = \phi$ is satisfied. The condition corresponds to orthogonal

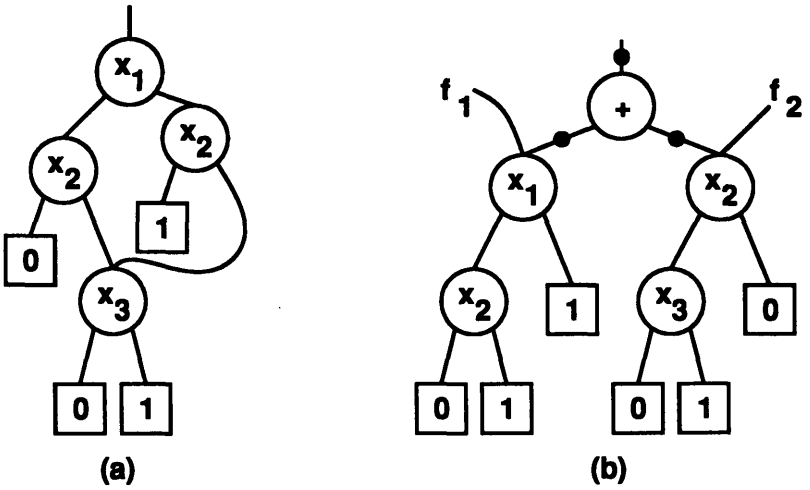


FIGURE 5-11: (a) A representation of $(x_1 + x_2) \cdot (\overline{x_2} + x_3)$ without function vertices. (b) A representation of the same function using an OR vertex at the root

complement functions. In this case, the signature check is:

$$|f_1 \cdot f_2| = |1 - |\overline{f_1}| - |\overline{f_2}||$$

The alternative result is the complement of an OR vertex with children $\overline{f_1}$ and $\overline{f_2}$.

Let $f_1 = x_1 + x_2$ and $f_2 = \overline{x_2} + x_3$. The complements, $\overline{f_1} = \overline{x_1} \cdot \overline{x_2}$ and $\overline{f_2} = x_2 \cdot \overline{x_3}$, have a null intersection, so $\overline{f_1} \cdot \overline{f_2} = 0$. The product of f_1 and f_2 is $x_1 \cdot \overline{x_2} + x_2 \cdot x_3$. An FBD that represents the function without an OR vertex at the root is shown in Figure 5-11(a). The alternative representation is illustrated in Figure 5-11(b). The signature check consists of building the representation shown in Figure 5-11(a), and comparing the signature against the signature for the representation in Figure 5-11(b), which assumes the orthogonal children requirement is met. If the signatures are equal, then we assume the representation in Figure 5-11(b) is valid. The new FBD simply uses the existing FBDs for f_1 and f_2 in addition to the OR vertex at the root.

The introduction of an OR vertex requires the use of the probabilistic equivalence check to determine whether the two different representations are actually equivalent. If an OR vertex is also introduced, a finite probability of error is introduced. This error is equivalent to the error introduced by a unique table hit, and less than the upper bound of $\left(\frac{n}{\|S\|}\right)^k$.

OR Node Retention

Once OR nodes have been introduced, the OR nodes tend to bubble down during subsequent calculations because ITE always selects decision variables. This is not always

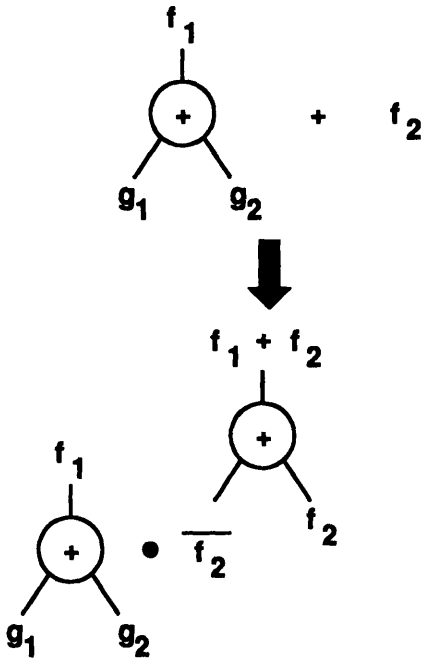


FIGURE 5-12: OR node retention while computing $f_1 + f_2$

desirable, since an orthogonal FBD version of the function may actually be more compact. However, the actual size of the FBD cannot be determined without building the FBD.

OR node retention is attempted only if an OR node is already present at the root of at least one of the FBDs representing functions f_1 and f_2. The functions are not necessarily orthogonal. Assume the root vertex v, of the FBD representing the function f_1, is an OR vertex.

When computing the Boolean *or* of f_1 and f_2, an OR node can be retained at the root vertex by computing the result as:

$$f_1 + f_2 = f_1 \cdot \overline{f_2} + f_2$$

The resulting FBD, $OR(f_1 \cdot \overline{f_2}, f_2)$ is an OR vertex with a low child representing $f_1 \cdot \overline{f_2}$ and a high child representing f_2. This result is depicted in Figure 5-12. To determine which version of the result should be smaller, one version needs to be calculated first, and the size of this version is used as a pruning measure while calculating the second version. If the size of the second version ever exceeds some limit, the calculation is abandoned. Otherwise, the smaller version will be utilized.

The size of an FBD can be calculated by doing a depth-first search. The size calculation is linear in the number of nodes in the FBD, but it becomes expensive when the size function is called recursively. As a result, the size of an FBD may not be calculated on every function recursion, but perhaps every m recursions, where typically, $m = 4$.

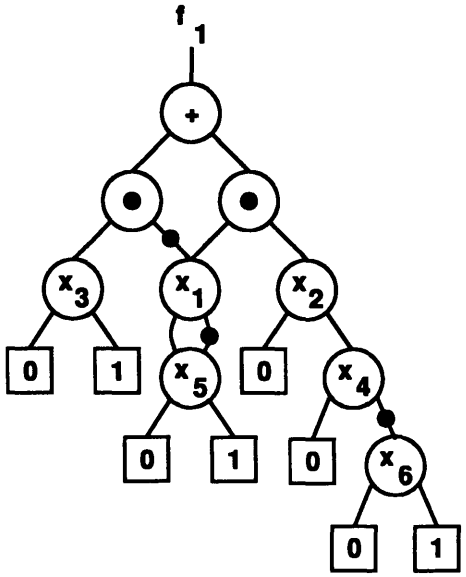


FIGURE 5-13: 8-node FBD representation of $f_1 = (x_1 \cdot \overline{x_5} + \overline{x_1} \cdot x_5) \cdot (x_2 \cdot x_4 \cdot \overline{x_6}) + (x_1 \cdot x_5 + \overline{x_1} \cdot \overline{x_5}) \cdot x_3$ using an OR vertex at the root

The following is an example of OR node retention in the case of $f_1 + f_2$. Under the ordering $\{x_1, x_2, x_3, x_4, x_5, x_6\}$ let:

$$f_1 = (x_1 \cdot \overline{x_5} + \overline{x_1} \cdot x_5) \cdot (x_2 \cdot x_4 \cdot \overline{x_6}) + (x_1 \cdot x_5 + \overline{x_1} \cdot \overline{x_5}) \cdot x_3$$

$$f_2 = x_6$$

An FBD representing f_1 is shown in Figure 5-13. If we calculate $f = f_1 + f_2$ without retaining OR nodes the result is shown in Figure 5-14. There are no OR nodes present in the result FBD, and 14 nodes are required in the representation under the ordering restriction. However, if an OR vertex is retained at the root, the result FBD in Figure 5-15 has 11 nodes, and can be computed quickly.

Figure 5-16 depicts the OR node retention in the case of computing *and*. OR nodes can be retained when computing the *and* of the two functions $f = f_1 \cdot f_2$. Without loss of generality, assume the FBD representing f_1, has an OR vertex, v at the root. The children of v represent the functions g_1 and g_2. The resulting FBD represents the function:

$$f_1 \cdot f_2 = g_1 \cdot f_2 + g_2 \cdot f_2$$

The result FBD retains an OR vertex at the root, and the children represent the functions $g_1 \cdot f_2$ and $g_2 \cdot f_2$.

If the functions are represented by complement edges to an OR vertex, the Boolean operation can be rewritten using DeMorgan's Law to fit either the Boolean *or* case or the *and* case.

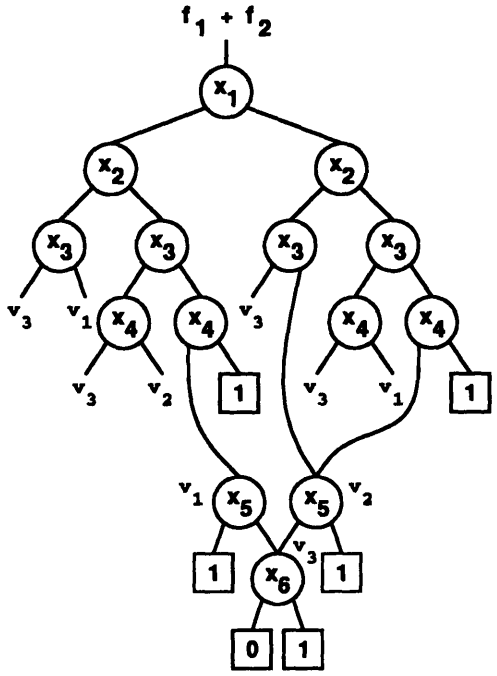


FIGURE 5-14: 14-node FBD representation of $f = f_1 + f_2$ without OR vertices

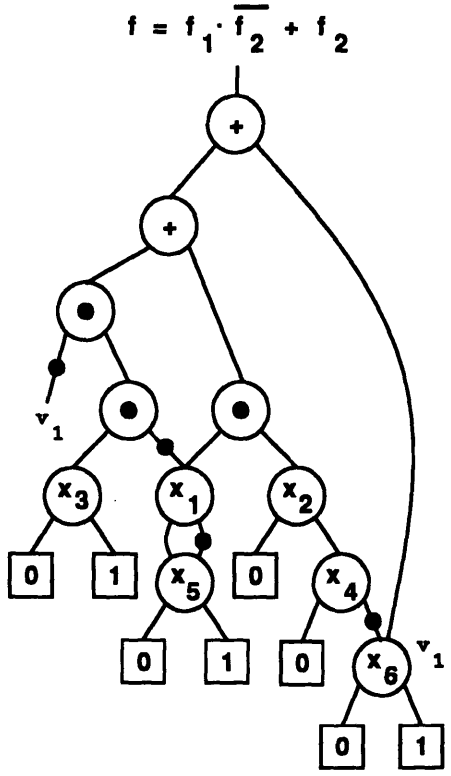


FIGURE 5-15: 11-node FBD representation of $f = f_1 + f_2$ with OR vertices

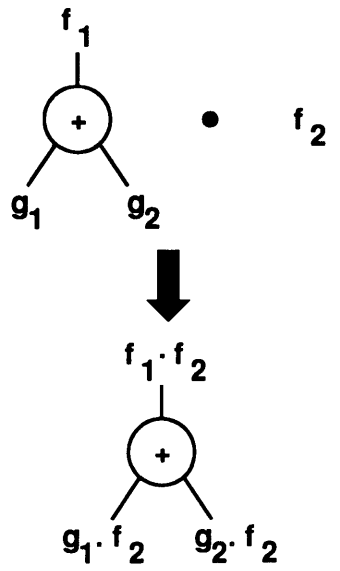


FIGURE 5-16: OR node retention while computing $f_1 \cdot f_2$

OR vertices can also be introduced in the above cases even when neither the root vertex of f_1 nor f_2 is an OR vertex. However, the introduction is difficult to control, and computationally expensive. The introduction of excess OR vertices may result in larger FBD representations.

Redundant Vertices

Without OR vertices, the supports of two FBDs representing two equivalent Boolean functions should be identical. If the supports are not identical, the FBDs cannot be equivalent, even if they have the same signature.

In the case of OR vertices, there is a possibility of redundant variables. A variable x_i is redundant in an FBD f if the Boolean function does not rely on x_i. So, given two FBDs f and f' that represent equivalent Boolean functions, if only f contains redundant vertices and f' does not, this implies that $support(f') \subset support(f)$. Redundant variables can be removed using the general cofactor operation discussed in Section 5.5.2 because $f_{x_i} = f_{\overline{x_i}} = f$.

An example of an FBD with a redundant variable, x_2 is shown in Figure 5-17. The function is $x_1 \cdot x_2 \cdot x_3 \; + \; x_1 \cdot \overline{x_2} \cdot x_3$ and the equivalent reduced function is $x_1 \cdot x_3$. The redundant variables can be removed from the function to create a reduced function.

Redundant variables are recognized when we have two different nodes with equivalent signatures and different supports. The supports should be identical, so all support variables not contained in the intersection are redundant. Redundant variables, once

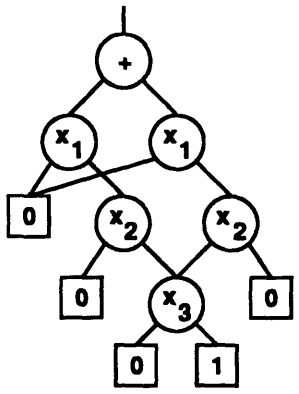


FIGURE 5-17: Function that is redundant in variable x_2

```
repair_node (ut_node, new_node) {
    if (support(ut_node) ⊆ support(new_node))
        /* The unique table node has the smaller support */
        return (ut_node) ;
    if (support(ut_node) ⊃ support(new_node)) {
        /* Replace the unique table node with the new node */
        ut_node = replace_node (ut_node, new_node) ;
        return (ut_node) ;
    }
    /* Use cofactor to eliminate redundant variables */
    cube = support(ut_node) ⊕ support(new_node) ;
    ut_node = FBD_cofactor_wrt_cube (ut_node, cube) ;
    return (ut_node) ;
}
```

FIGURE 5-18: Pseudocode for redundant variable removal

identified, can be eliminated by cofactor.

Figure 5-18 shows the algorithm for eliminating redundant variables when an existing node is recovered from the unique table. If the support of the existing node, *ut_node*, is contained by the support of the new node, *new_node*, the existing node can be used. Otherwise, if the support of *new_node* is completely contained by *support*(*ut_node*), the *new_node* can replace the existing node in the unique table. The parents of the node do not have to be updated because the memory location for the node is the same, just the node data is updated.

Node replacement is allowed only if the signatures are identical, and in some cases, NAND or NOR vertices may be required to maintain this condition. If the supports are not contained, the existing node, *ut_node*, is cofactored with respect to a cube that represents

```
replace_node (ut_node, new_node) {
    /* Update the index variable */
    index(ut_node) = index(new_node) ;
    /* Update the vertex type */
    type(ut_node) = type(new_node) ;
    /* Update the children */
    high(ut_node) = high(new_node) ;
    low(ut_node) = low(new_node) ;
    /* Update the support */
    support(ut_node) = support(new_node) ;
    return (ut_node) ;
}
```

FIGURE 5-19: Pseudocode for node replacement

the redundant variables all set to 0 or 1. The cofactor algorithm is similar to the single variable cofactor algorithm, only the cofactor is performed until no more variables are left in the cube.

The pseudocode for the node replacement algorithm is shown in Figure 5-19. The *new_node* data is simply copied into the unique table node, *ut_node*, leaving all other memory locations untouched. The signature of *ut_node* does not need to be updated either, because we only allow the replacement of nodes with equal signatures. Note that the support field of *ut_node* will change, and this will corrupt the support of the existing parents of *ut_node*, but the supports can be updated when the corrupted nodes are used.

We need the NAND and NOR vertices because of redundant vertices. A node can only be replaced by another node with the same signature, and sometimes, with function vertices this is not possible without the NAND and NOR vertices.

For example, in Figure 5-20 the function associated with the root OR vertex labeled v_1 is $g = \overline{f_1} + \overline{f_2}$. The function f_1 is associated with vertex v_2 and is

$$f_1 = x_1 \cdot \overline{x_2} + \overline{x_1} \cdot \overline{x_3} + \overline{x_4}.$$

The OR vertex v_2 represents:

$$f_2 = x_1 \cdot \overline{x_2} + \overline{x_1} \cdot \overline{x_3} + x_4$$

When the function f_1 is cofactored with respect to $x_4 = 1$, we discover that the resulting function is represented by an OR node that is equivalent to the complement of vertex v_1. The FBD is shown in Figure 5-21. We want to substitute the complement

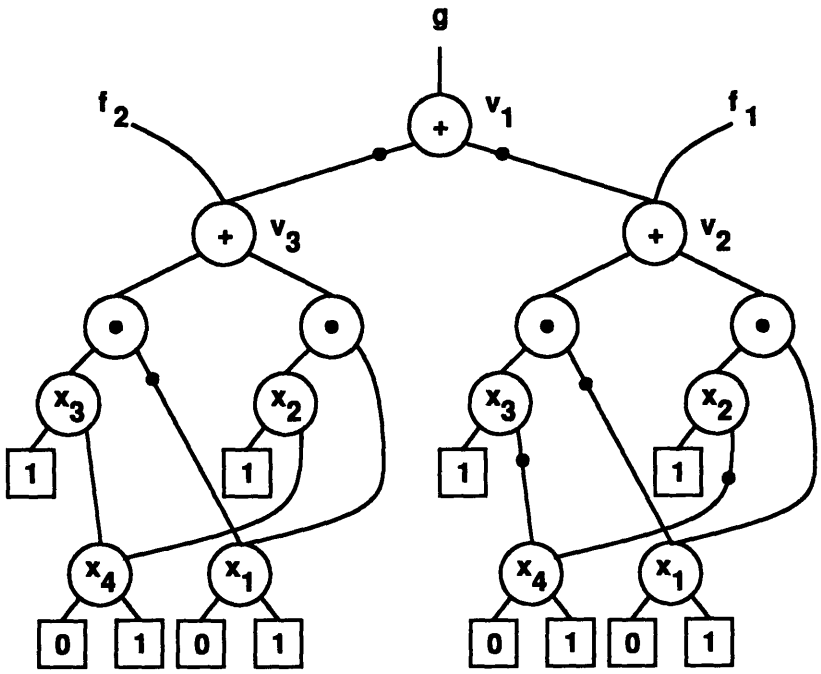


FIGURE 5-20: An FBD with redundant variable x_4

of node w_4 into vertex v_1. In order to make the substitution correctly, we have to use the FBD shown in Figure 5-22, with a NOR node as the root vertex w_5. The function represented by this FBD is equivalent to g.

FBDs with a decision or XOR vertex, v, at the root are adjusted to the correct signature phase during node replacement simply by adjusting the children. Figure 5-23 represents the adjustment pictorially. The adjustment is possible because the following relationships are true:

$$\overline{x_i \cdot f^{high(v)} + \overline{x_i} \cdot f^{low(v)}} = x_i \cdot \overline{f^{high(v)}} + \overline{x_i} \cdot \overline{f^{low(v)}}$$

$$\overline{f^{high(v)} \oplus f^{low(v)}} = \overline{f^{high(v)}} \oplus f^{low(v)} = f^{high(v)} \oplus \overline{f^{low(v)}}$$

5.6 Results

The results of applying the FBD package to some benchmarks are summarized in Tables 5-1 and 5-2. The same global ordering was given to the FBD package and the ROBDD package implemented in the program `sis` [50]. In most cases, the same global ordering was used for all outputs, except examples marked with an asterisk required different orderings for different outputs in order to create graphs. For these examples, any further manipulation of the output graphs requires the FBD package. All CPU times are reported in seconds on a SPARCstation 10.

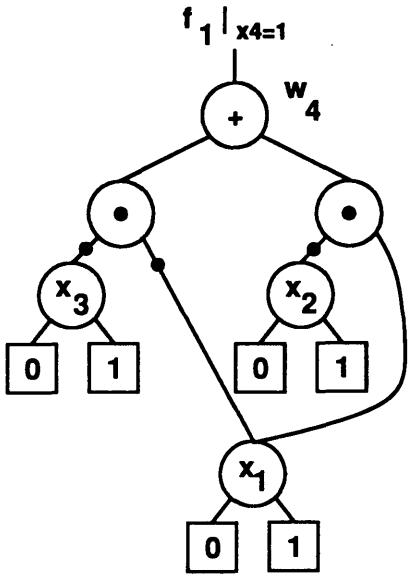


FIGURE 5-21: $\overline{g} = f_1 \mid_{x_4} = x_1 \cdot \overline{x_2} + \overline{x_1} \cdot \overline{x_3}$

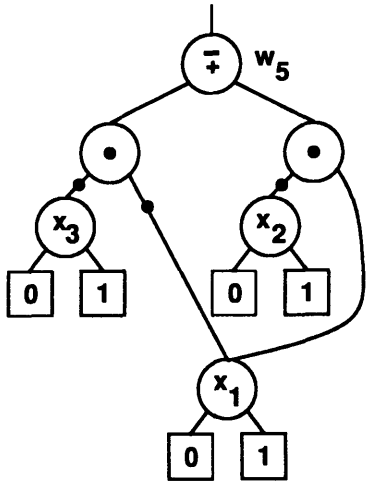


FIGURE 5-22: $f^{w_5} = \overline{x_1} \cdot x_3 + x_1 \cdot x_2$

FIGURE 5-23: Phase adjustment for decision and XOR vertices

EX	#i	#o	ROBDD		FBD	
			Nodes	Time	Nodes	Time
C432	36	7	31,195	26	31,195	28
C499	41	32	33,214	14	33,214	11
C880	60	26	7,761	2	7,761	3
C1355	41	32	33,214	14	33,214	20
C1908	33	25	12,734	9	12,734	12
C2670	233	140	57,767	21	57,767	32
C3540	50	22	88,652	82	88,652	116
C5315	178	123	26,129	11	26,129	14
C6288(14)	32	32	115,607	535	115,607	1,599
C7552	207	108	8,356	5	8,356	6
i10	257	224	880,254	306	880,254	602
s5378	199	213	6,854	2	6,854	1
s9234*	247	250	1,069,633	632	1,069,633	1,786
s13207	700	790	13,892	9	13,892	5
s15850	611	684	53,612	26	53,612	28
s35932	1,763	2,048	6,190	36	6,190	5
s38584	1,464	1,730	40,817	56	40,817	20
hwb16	16	1	614	0.4	614	0.6
hwb32	32	1	40,126	36	40,126	36
hwb64	64	1	unable	unable	unable	unable
mm16	67	64	3,308	4	3,308	3
mm30	123	120	11,071	15	11,071	14
mult12	24	24	610,093	950	610,093	4,632
rotate32	69	1	unable	unable	unable	unable

Table 5-1: Comparison of CPU time required to build ROBDDs using the ROBDD package and the FBD package

Table 5-1 shows a comparison of the ROBDDs created by the ROBDD package versus the ROBDDs created by the FBD package under the same global ordering without function vertices.

The first column gives the name of the benchmark. The next two columns show the number of inputs and outputs in the circuit. The two columns labeled ROBDD show the number of nodes necessary to represent the circuit as an ROBDD, and the CPU time required to build the ROBDDs. The last two columns labeled FBD shows the number of nodes and the CPU times obtained from the FBD package.

The FBD package without any OR/AND/XOR vertices and the same global ordering produces exactly the same ROBDDs 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.

Table 5-2 compares FBDs with function vertices against ROBDDs. Each row in Table 5-2 shows the graph 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**.

FBDs built using a global input ordering and AND/XOR vertices appear next in the table. The last two columns in the table show the node sizes and times for FBDs with a global input ordering and OR/AND/XOR function vertices. As function vertices are added, the sizes of FBDs decrease by as much as a factor of 11 in certain cases (**hwb32**). In example **C1908**, the FBD with function vertices is slightly larger than the ROBDD because the AND vertices do not maximize node sharing in this case.

Most of the examples are from the ISCAS-85 and ISCAS-89 benchmark set. **mm** is the min-max function. The example **hwb** is the hidden weighted bit function described in [13]. $O(n^2)$-sized FBDs for this function can be created using OR vertices, whereas any ROBDD representation has $O(1.14^n)$ vertices. **rotate32** is the function from [22]. FBDs of $O(n^2)$ size for this function can be created using AND vertices, whereas any ROBDD representation has $\Omega(2^{\frac{n}{2}})$ vertices. **C2670** and **C5315** also benefit significantly from OR vertices.

In example **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 probabilistic equivalence method does introduce a finite amount of error into the experiments, but this error is bounded by the expressions given in Chapter 4, and decreases exponentially with the number of passes made. During our experiments, we used a field size of $\|S\| = 32341$ and $k = 3$ passes. We never encountered an error in our experiments.

5.7 Conclusion

An FBD representation and a strongly canonical form for the representation using probabilistic equivalence were presented. Practical FBD algorithms for probabilistically constructing and manipulating FBDs were given, along with mechanisms for introducing and retaining function vertices.

The finite probability of error associated with the FBDs built for a circuit is upper

EX	ROBDD		AND/XOR		OR/AND/XOR	
	Nodes	Time	Nodes	Time	Nodes	Time
C432	31,195	26	29,004	37	29,004	36
C499	33,214	14	33,214	16	33,214	16
C880	7,761	2	6,851	3	6,851	4
C1355	33,214	14	33,214	25	33,214	29
C1908	12,734	9	12,765	15	12,765	16
C2670	57,767	21	39,468	37	28,633	36
C3540	88,652	82	84,409	144	87,575	157
C5315	26,129	11	23,789	14	2,987	17
C6288(14)	115,607	535	115,607	2022	115,607	2,107
C7552	8,356	5	7,511	6	7,511	6
i10	880,254	306	733,746	620	638,890	671
s5378	6,854	2	3,521	1	3,348	1
s9234*	1,069,633	632	1,065,835	1,658	398,211	1,665
s13207	13,892	9	4,035	3	4,001	4
s15850	53,612	26	31,278	27	28,644	29
s35932	6,190	36	5,731	8	5,731	9
s38584	40,817	56	35,854	35	27,209	35
hwb16	614	0.4	618	0.9	533	0.9
hwb32	40,126	36	40,127	43	3,474	66
hwb64	unable	unable	unable	unable	11,599	70
mm16	3,308	4	3,276	4	3,276	4
mm30	11,071	15	10,983	17	10,983	19
mult12	610,093	950	610,112	6344	610,644	6,492
rotate32	unable	unable	2,117	0.3	2,117	0.4

Table 5-2: Results comparing FBDs with function vertices against ROBDDs

bounded by the expression:

$$ut\_hits \times \left(\frac{n}{\|S\|}\right)^k$$

previously derived in Section 4.3.

FBDs are provably more efficient than ROBDDs for certain classes of circuits. Experimental results also show that FBDs produce smaller graphs than ROBDDs for a number of general benchmark circuits. The experimental results were obtained using a global input ordering and restricted function vertices. Despite the constraints, the FBD results were still smaller than ROBDDs, and the FBD package is both practical in terms of memory use and CPU time.

The presence of function vertices allowed the graphs to effectively "simulate" un-

ordered graphs in the examples. The quality of the results does depend on the global input ordering and the form of the circuit specification. The global ordering can be improved by incorporating more ROBDD global ordering techniques such as dynamic variable re-ordering [48].

6

Free Boolean Diagrams Based Upon Polynomial Hashing

6.1 Introduction

In the FBD package with OR and AND vertices, the values in the algebraic field used to probabilistically determine equivalence were chosen as non-negative integers. These values can be chosen as something other than integers, as long as the properties of fields are still satisfied. A MOD-2 polynomial field with polynomials as elements of the field can be used in the equivalence scheme [30, 31]. Multiplication and addition can be performed in the MOD-2 polynomial field, and the additional characteristic of the field, $a + a = 0$, allows for an efficient $\oplus$ representation.

We built an XOR-FBD package based upon the polynomial field hashing scheme. This scheme uses decision and terminal vertices, the restricted support AND vertex and a new $\oplus$ vertex [52]. The $\oplus$ vertex is now unrestricted, meaning there are no constraints on the children of the vertex. The unrestricted $\oplus$ vertex completely subsumes the OR and XOR vertices defined in Chapter 5.

The first section in this chapter describes the characteristic 2 polynomial fields, and the operations defined on the field. The second section presents the representation definition, followed by the changes in the manipulation algorithms. The manipulation algorithms are similar, although, the support field was eliminated. Experimental results and a summary conclude the chapter.

6.2 Probabilistic Equivalence

The probabilistic equivalence check described by Blum et al. [8] is defined for an algebraic field. One such field is the field used in Chapter 5, non-negative integers defined over modular-p integer arithmetic, where p is prime. Another field is a set of polynomials of degree less than m defined over modular-$p(x)$ arithmetic, where $p(x)$ is a prime and irreducible polynomial of degree m. We will now proceed to describe addition and multiplication defined over the polynomial field.

6.2.1 MOD-2 Polynomial Field

The modulus for the polynomial field with 2^m elements is a prime and irreducible polynomial $p(x)$ of degree m [42]. The elements of the field are polynomials with degree less than m. Modulo-2 addition and multiplication are defined on the polynomials in the field. Given two polynomials:

$$p_0(x) = a_{m-1}x^{m-1} + \ldots + a_1x + a_0$$

$$p_1(x) = b_{m-1}x^{m-1} + \ldots + b_1x + b_0$$

The coefficients are restricted to either 0 or 1, that is:

$$a_i, b_i \in \{0, 1\}$$

The *bitwise xor* of the coefficients is defined by the truth table shown in Table 6-1. The addition of the two polynomials $p_0(x)$ and $p_1(x)$ is defined as:

$$p_0(x) + p_1(x) = (a_{m-1} \hat{} b_{m-1})x^{m-1} + \ldots + (a_1 \hat{} b_1)x + (a_0 \hat{} b_0)$$

The addition of two polynomials yields a polynomial of degree less than m.

The multiplication of the two polynomials yields a result with degree less than $2m$. The result is mapped into a polynomial with degree less than m by dividing the result with the modulus $p(x)$ and taking the remainder.

The set of polynomials with degree less than $m = 4$ is shown in Table 6-2. The first column is the actual polynomial, the second column shows the bit encoding for the coefficients in each polynomial. The least significant bit a_0 represents the coefficient in front of $x^0 = 1$. The most significant bit a_3 represents the coefficient associated with x^3. The third column, labeled index, is used to compute multiplication.

The polynomials in the node signature are implemented as packed bits. The bit i is set if the coefficient, a_i, corresponding to the polynomial x^i is set. For example, the packed

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Table 6-1: Truth table for the bitwise xor function

Polynomial	a_3	a_2	a_1	a_0	index
0	0	0	0	0	na
1	0	0	0	1	0
x	0	0	1	0	1
$x + 1$	0	0	1	1	12
x^2	0	1	0	0	2
$x^2 + 1$	0	1	0	1	9
$x^2 + x$	0	1	1	0	13
$x^2 + x + 1$	0	1	1	1	7
x^3	1	0	0	0	3
$x^3 + 1$	1	0	0	1	4
$x^3 + x$	1	0	1	0	10
$x^3 + x + 1$	1	0	1	1	5
$x^3 + x^2$	1	1	0	0	14
$x^3 + x^2 + 1$	1	1	0	1	11
$x^3 + x^2 + x$	1	1	1	0	8
$x^3 + x^2 + x + 1$	1	1	1	1	6

Table 6-2: Bit representation of polynomials with degree less than 4

field 0110 corresponds to the polynomial $p = x^2 + x^1$. This packed bit representation allows us to perform addition using a bitwise xor operation. For example, given the polynomials $x^2 + 1$ and $x^2 + x + 1$, from Table 6-2 their addition is the polynomial x, because $0101 \hat{} 0111 = 0010$.

We illustrate the multiplication method with an example. In Figure 6-1, the multiplication of polynomials $x^2 + 1$ and $x^2 + x + 1$, using bitwise xor to add the intermediate products, yields $x^4 + x^3 + x + 1$. Now we have to use the chosen irreducible, prime polynomial $p(x) = x^4 + x^3 + 1$ of degree 4 [42] to compute modulus $p(x)$ of the result $x^4 + x^3 + x + 1$. The modulus can be obtained by continually bitwise xor'ing $p(x)$ with the given polynomial until the degree of the result is less than $m = 4$. This is equivalent to subtracting $p(x)$ until the result has degree less than m. In the example,

$$
\begin{array}{rl}
 & x^2 + x + 1 \\
\times & x^2 \quad + 1 \\
\hline
 & x^2 + x + 1 \\
+ \quad x^4 + x^3 + x^2 & \\
\hline
 & x^4 + x^3 \quad + x + 1
\end{array}
$$

FIGURE 6-1: Multiplication of two polynomials

one bitwise xor suffices:

$$(x^4 + x^3 + x + 1)\hat{}(x^4 + x^3 + 1) = x$$

Therefore the multiplication of $x^2 + 1$ and $x^2 + x + 1$ in $GF(2^4)$ under the prime, irreducible polynomial $x^4 + x^3 + 1$ is x.

The multiplication algorithm can be converted into two table lookups and one integer addition step. Given the two polynomials, p_1 with index $index(p_1)$ and p_2 with index $index(p_2)$, we look up their indices from the index table, sum the indices, and use the modular sum to determine the polynomial associated with the index of the result. The index of the result is:

$$index(p_1 \cdot p_2) = (index(p_1) + index(p_2)) \; mod \; (2^m - 1)$$

The result of the multiplication is the polynomial associated with $index(p_1 \cdot p_2)$, and the polynomial can be located by using its index to access an inverse index table. The multiplication of any polynomial with the zero polynomial is defined to be 0.

Table 6-2 is the index table for $GF(2^4)$ and $p(x) = x^4 + x^3 + 1$. To compute the multiplication of $p_1 = x^2 + 1$ and $p_2 = x^2 + x + 1$, we look up the indices for the polynomials in Table 6-2, and find $index(p_1) = 9$ and $index(p_2) = 7$. The index of the result is $(9 + 7) \; mod \; 15 = 1$. The inverse index table for $GF(2^4)$ is shown in Table 6-3. The polynomial associated with 1 is simply x. This result is the same result obtained using the standard modular-$p(x)$ multiplication method in the previous example.

6.2.2 Probabilistic Equivalence Test Using Polynomials

A probabilistic equivalence test for free BDDs with $\oplus$ nodes using polynomial fields with characteristic 2 is described in [30]. The field is a polynomial field P. The values of the polynomial field are polynomials $p_i(x)$.

index	polynomial
0	0001
1	0010
2	0100
3	1000
4	1001
5	1011
6	1111
7	0111
8	1110
9	0101
10	1010
11	1101
12	0011
13	0110
14	1100

Table 6-3: Inverse index table for $GF(2^4)$

If we also include AND nodes, the algorithm for computing the polynomial associated with a node v is shown in Figure 6-2. The procedure is the same as the procedure in Chapter 5, except we assign polynomial values instead of integer values. The computation for determining error bounds is still valid.

The first step is to assign random polynomials to the inputs of the circuit. Then, given two graphs, G_1 and G_2, the graphs are parsed from the terminal vertices to the root, computing a signature for each node. The rules for computing the signature of a node are determined by the type of node.

If the signatures of the graphs are different, G_1 and G_2 represent different functions. If the signatures are the same, G_1 and G_2 are assumed equivalent with error less than $\left(\frac{n}{\|S\|}\right)^k$, where n is the number of inputs, $\|S\|$ is the cardinality of the field and k is the number of passes.

An example of a one pass signature calculation for a graph using a polynomial field is shown in Figure 6-3. The signatures for the vertices are polynomials. The size of field is a large number, usually 2^{16}.

6.2.3 Properties of Polynomial Signatures

An interesting characteristic of the MOD-2 polynomial field is $|a| + |a| = 0$, implying $|a| = -|a|$. Consequently, the value of the complement is simply $|\overline{a}| = 1 + |a|$. By using

Equivalence ($G_1(x_1, x_2, ..., x_n)$, $G_2(x_1, x_2, ..., x_n)$)
{
Given a polynomial field P with at least $2n$ elements ;
Assign k polynomials from P to each variable x_i ;
For (all vertices v from terminal vertices to the root in G_1 and G_2)
if (v is an $\oplus$ vertex)
$|v| = |high(v)| + |low(v)|$;
else **if** (v is an AND vertex)
$|v| = |high(v)| \cdot |low(v)|$;
else
$|v| = |x_i| \cdot |high(v)| + |1 - x_i| \cdot |low(v)|$;
if ($|G_1| == |G_2|$)
return (G_1 and G_2 are equivalent with probability of error $< \left(\frac{1}{2}\right)^k$) ;
else **if** ($|G_1| \neq |G_2|$)
return (G_1 and G_2 are definitely not equivalent) ;
}

FIGURE 6-2: Probabilistic equivalence test using polynomials

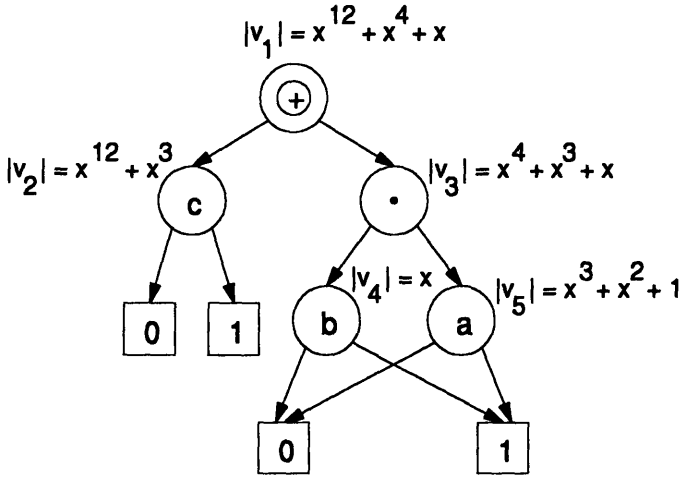


FIGURE 6-3: An example of polynomial signature calculation

a polynomial field with characteristic 2, the signature calculation of $a \oplus b$ reduces to the sum of the signatures of the arguments a and b:

$$|a \oplus b| = |a| + |b| \tag{6.1}$$

The validity of Equation 6.1 can be verified by expanding the sum:

$$|a| + |b| = |a \cdot b| + |a \cdot \overline{b}| + |a \cdot b| + |\overline{a} \cdot b|$$

$$|a| + |b| = |a \cdot \overline{b}| + |\overline{a} \cdot b|$$

6.3 XOR-FBD Representation

The ease of signature calculation of *exclusive-or* is the justification for the $\oplus$ node introduced in the following definition. The terminal, decision, AND and NAND vertex are equivalent to the vertices defined in the FBDs with integer-valued signatures.

Definition 6.3.1 An XOR-FBD representing the Boolean function $f^v(x_1, ..., x_n)$ is a directed, acyclic graph with a root vertex v and defined recursively with five types of vertices:

1 A terminal vertex v has no children and it represents the function $f^v = 1(0)$ and the vertex has signature $|v| = 1(0)$.

2 A decision vertex v has an index i, decision variable x_i, a high child, $high(v)$ and a low child, $low(v)$. The function represented by the decision vertex is:

$$f^v = x_i \cdot f^{high(v)} + \overline{x_i} \cdot f^{low(v)}$$

The signature for the decision vertex is computed as:

$$|v| = |x_i| \cdot |high(v)| + |\overline{x_i}| \cdot |low(v)|$$

3 An AND vertex v has a high child, $high(v)$ and a low child, $low(v)$. The restriction on the children of the AND vertex is:

$$support(high(v)) \cap support(low(v)) = \phi$$

The function represented by the AND vertex is:

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

The equation for computing the signature of the vertex is:

$$|v| = |high(v)| \cdot |low(v)|$$

4 An $\oplus$ vertex v has a high child, $high(v)$ and a low child, $low(v)$. There are NO restrictions on the children of the vertex. The function represented by the $\oplus$ vertex is:

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

The equation for computing the signature of the vertex is:

$$|v| = |high(v)| + |low(v)|$$

5 A NAND vertex v has a high child, $high(v)$ and a low child, $low(v)$. The restriction on the children of the NAND vertex is:

$$support(high(v)) \cap support(low(v)) = \phi$$

The function represented by the NAND vertex is:

$$f^v = \overline{f^{high(v)} \cdot f^{low(v)}}$$

The equation for computing the signature of the vertex is:

$$|v| = |1 - |high(v)| \cdot |low(v)||$$

Since $a \oplus b = a + b$ when $a \cap b = \phi$, the $\oplus$ vertex completely subsumes the OR and XOR vertices of Chapter 5. Unlike the XOR vertices described in Section 5.3, there are no restrictions on the children of the $\oplus$ vertex. A maximum of one additional node needs to be created when computing the *xor* of any two functions.

The AND vertices are identical to the AND vertices in the FBD package of Chapter 5. The NAND vertex is simply the complement of the AND vertex.

An example of an XOR-FBD is shown in Figure 6-4. The XOR-FBD represents the function $f = (x_1 \cdot \overline{x_5} + \overline{x_1} \cdot x_5) \oplus (x_1 \cdot x_2 \cdot x_3 + x_1 \cdot \overline{x_2} \cdot x_4)$. The vertex v_1 is an $\oplus$ node, and the children of the vertex are unrestricted. The vertex v_2 is a restricted AND vertex. The other vertices labeled with variables are decision vertices, and the terminal vertices are labeled 0 or 1.

6.4 A Strongly Canonical Form

The strongly canonical form for XOR-FBDs with AND and $\oplus$ vertices is maintained in the same manner as with FBDs and integer value signatures in Section 5.4. The sole difference is the use of polynomial fields with the probabilistic equivalence check.

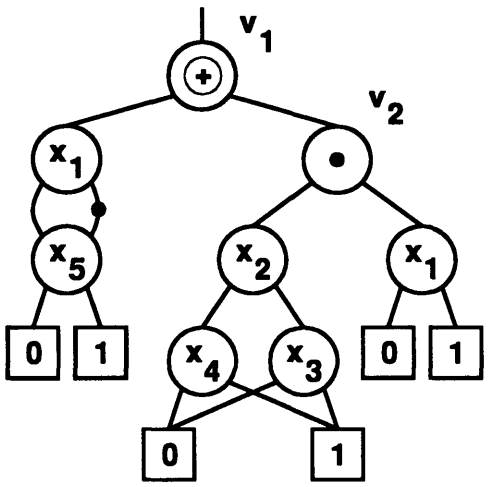


FIGURE 6-4: An XOR-FBD that has decision, $\oplus$ and AND vertices

6.5 Boolean Operations

The XOR-FBD package with $\oplus$ and AND function vertices is very similar to the FBD package with OR and AND function vertices. The $\oplus$ vertices are extremely efficient, and completely subsume the OR vertices in the FBD package. The XOR-FBD package does not have an *apply* or *ITE* operation, but relies solely on *and* and *xor* calls. Since *and*, *xor* and *complement* form a complete basis in Boolean space, all Boolean functions can be mapped into the available functions.

Also, the nodes in the XOR-FBD package do not carry support variables. Instead of maintaining support information, the package assumes a global ordering restriction is satisfied, and the index of the top decision variable is a sufficient pruning condition for the manipulation algorithms.

If the root vertex is a decision vertex, the top variable is simply the decision variable of the root. The index of a function vertex is simply the smallest index of the children. So, the index of a function vertex is simply the lowest index decision variable in the XOR-FBD rooted at the function vertex. For example, the index of the AND function vertex, v_2, in Figure 6-4 is 1, for variable x_1.

6.5.1 Cofactor

The cofactor algorithm is similar to the FBD cofactor algorithm, the only significant difference is we have $\oplus$ vertices instead of OR vertices. The cofactor of an $\oplus$ vertex representing the function $f = f_1 \oplus f_2$ is valid under the Shannon expansion for the

function:

$$f_1 \oplus f_2 = x_i \cdot (f_1 |_{x_i} \oplus f_2 |_{x_i}) + \overline{x_i} \cdot (f_1 |_{\overline{x_i}} \oplus f_2 |_{\overline{x_i}})$$

We also use the top variable, stored in the *top_index* field as the pruning condition for cofactor when a global input ordering is enforced, so the supports are no longer necessary in this case.

6.5.2 XOR

The *xor* of two functions f_1 and f_2 is computed simply by creating an $\oplus$ vertex, v, and setting the low child, $low(v) = f_1$, and setting the high child, $high(v) = f_2$.

Similarly, the *xnor* of the two functions is created simply by creating the $\oplus$ vertex, v and by letting $high(v) = f_2$ and $low(v) = \overline{f_1}$.

6.5.3 AND

The fundamental algorithm for creating XOR-FBDs in this package version is the two-input Boolean *and* operation. The two-input *or* operation can be converted into an *and* using DeMorgan's Law.

The pseudocode for the *and* computation is shown in Figure 6-5. Given two XOR-FBDs f and g, the first step in the computation of the *and* is to check for the usual special cases. These cases are enumerated in Appendix A.

The next step is to check for a previously computed result to the *and* in the computed table. If an entry matches the signatures of f and g, we can use the result in the entry as the result to the *and* without further computation. Since we are dealing with only two functions, f and g, and the *and* is symmetric, canonicity of the input vector is easy to maintain. The canonicity requirement is $f < g$.

Additionally, even if $f \cdot g$ cannot be located in the computed table, we can make some additional checks for $\overline{f} \cdot g$, $f \cdot \overline{g}$, or $\overline{f} \cdot \overline{g}$. If any of these pairs are located in the computed table, $\oplus$ vertices can be used to compute a result (see Appendix A).

Input Variable Ordering

If a result is not found in the computed table, we need to compute the cofactors of f and g to call **fbd-and** recursively. To maintain a global ordering, we can simply select the smallest index at the root of f and g.

Typically, the global input ordering is determined in advance using existing ROBDD variable ordering techniques. When a global input variable ordering is assumed, namely when all variables that appear along a path satisfy the global ordering, this simplifies

```
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 ⊕ vertex v with
            f̄ and g as children ;
        }
    }
    if (f ≡ ⊕(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 ⊕ 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 6-5: Procedure to compute $f \cdot g$

a number of computations, and a support field for each node is not required. Instead, each node maintains a *top_index* field. For decision vertices, this field is the index field, it contains the index of the decision variable associated with the vertex. The index associated with a function vertex is simply the smallest variable that appears in the entire XOR-FBD. Additional flags are needed to identify the type of vertex. This *top_index* field is sufficient for pruning the recursive cofactor calls, and for handling redundant vertices.

Despite the global input ordering assumption, XOR-FBDs still have a degree of freedom not available to ROBDDs. The function vertices "emulate" unordered paths. Namely XOR-FBDs with function vertices effectively permit results that will be similar to XOR-FBDs with unordered variables.

AND Node Introduction

Since the support field is no longer available on the XOR-FBD nodes, AND vertices can no longer be introduced by checking supports, as in Section 5.5.3.

Instead, when computing $f \cdot g$, AND vertices are introduced using the signature check:

$$|f \cdot g| = |f| \cdot |g|$$

After computing the *and* by selecting decision vertices, if we discover the resulting signature $|f \cdot g|$ is equivalent to the product of the signatures of the inputs then we assume the supports of f and g are disjoint, and hence f and g can be concatenated with an AND vertex, v, by setting $low(v) = f$ and $high(v) = g$.

The probability of error introduced by this check, if an AND vertex is actually created, is equivalent to the error introduced by a unique table hit. The additional error is less than $\left(\frac{n}{\|S\|}\right)^k$, where n is the number of inputs, $\|S\|$ is the cardinality of the field and k is the number of passes.

This AND vertex introduction scheme is more expensive in terms of computation time and memory use, because additional *and* recursions must be made to discover the concatenation, whereas with support fields, the concatenation is discovered a priori without creating any new nodes.

However, the memory consumed by the extra recursions is low, because the intermediate results are discarded as soon as the intermediate concatenation is discovered. Also, we save memory because the support field is very expensive when the number of inputs is large.

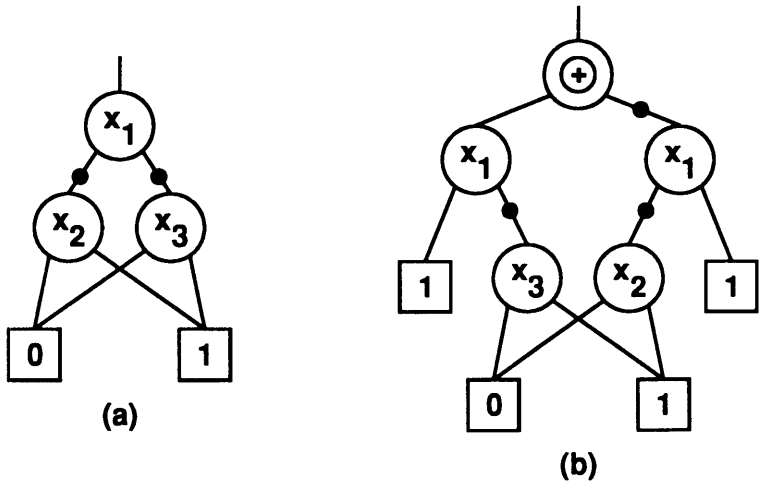


FIGURE 6-6: (a) FBD that represents the function $x_1 \cdot \overline{x_3} + \overline{x_1} \cdot \overline{x_2}$. (b) FBD that represents the same function, but with an $\oplus$ vertex at the root

$\oplus$ Node Introduction

$\oplus$ vertices are introduced during the computation of Boolean *and* if the following check is satisfied:

$$|f \cdot g| = 1 + |f| + |g|.$$

The signature $1 + |f| + |g|$ is associated with the *xnor* of functions f and g:

$$\overline{f \oplus g} = f \cdot g + \overline{f} \cdot \overline{g}$$

If this check is satisfied, this implies:

$$f \cdot g = f \cdot g + \overline{f} \cdot \overline{g}$$

This is true if the complement of f is orthogonal to the complement of g:

$$\overline{f} \cap \overline{g} = \phi,$$

The result can be replaced by an $\oplus$ vertex with high child, $\overline{f}$, and low child, g.

For a specific example, let $f = x_1 + \overline{x_2}$ and $g = \overline{x_1} + \overline{x_3}$. The Boolean *and* of f and g is $x_1 \cdot \overline{x_3} + \overline{x_1} \cdot \overline{x_2}$. The complements of f and and g do not intersect. Two possible representations of the *and* are shown in Figure 6-6(a) and Figure 6-6(b). The graph in Figure 6-6(b) has an $\oplus$ vertex at the root, and children $\overline{f}$ and g. The second representation can be computed with the creation of one additional node, the $\oplus$ vertex at the root, and no other nodes because the FBDs for the inputs f and g already exist.

If we use a one pass signature, and we assign $|x_1| = x$, $|x_2| = x^2$ and $|x_3| = x^3$, we see that the signature of the graph in Figure 6-6(a) is $x^4 + x^3 + x^2 + 1$. The signature

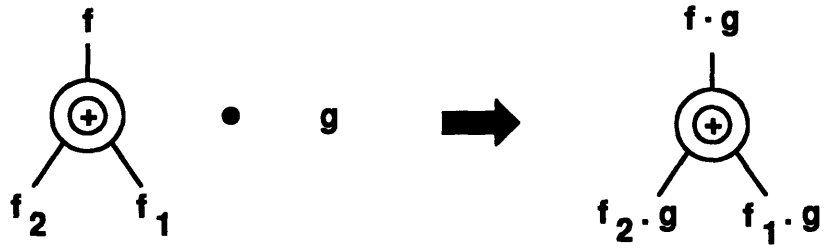


FIGURE 6-7: Example of $\oplus$ retainment at the root

of the graph in Figure 6-6(b) is also the same, and it has signature $1 + |f| + |g| = x^4 + x^3 + x^2 + 1$.

$\oplus$ Node Retention

We check for $\oplus$ vertex retainment during the calculation of $f \cdot g$ if there is an $\oplus$ vertex at the root of either f or g.

Without loss of generality, assume the root of f is an $\oplus$ vertex, f_1 and f_2 are the high and low child, respectively. The *and* can be computed by selecting decision variables and bubbling the $\oplus$ vertex down, or by creating another $\oplus$ vertex with high child $f_1 \cdot g$ and low child $f_2 \cdot g$, as in Figure 6-7.

For a more specific example, consider $f = (x_2 \cdot x_3) \oplus x_4$ and $g = x_1 \cdot \overline{x_2} + x_3$. Possible representations for the functions are shown in Figure 6-8(a). The result of the product is $x_1 \cdot \overline{x_2} \cdot x_4 + x_2 \cdot x_3 \cdot \overline{x_4} + \overline{x_2} \cdot x_3 \cdot x_4$. A representation of the result without $\oplus$ vertex retainment at the root is shown in Figure 6-8(b). The result with an $\oplus$ vertex is shown in Figure 6-8(c), and this FBD requires the construction of just 2 nodes, the AND vertex and the new $\oplus$ vertex.

As shown in Figure 6-9, any complement edges can be moved into the children, using:

$$\overline{f_1 \oplus f_2} = \overline{f_1} \oplus f_2$$

After the computation of the two representations of the result, the smaller result is chosen for use, and the alternate version is discarded. No additional error is introduced by node retention, other than the error introduced by the equivalence checks during the creation of the result.

Redundant Vertices

Redundant vertices are handled differently in the XOR-FBD package. Redundant vertices were discovered in Chapter 5 when there are two versions of the same node

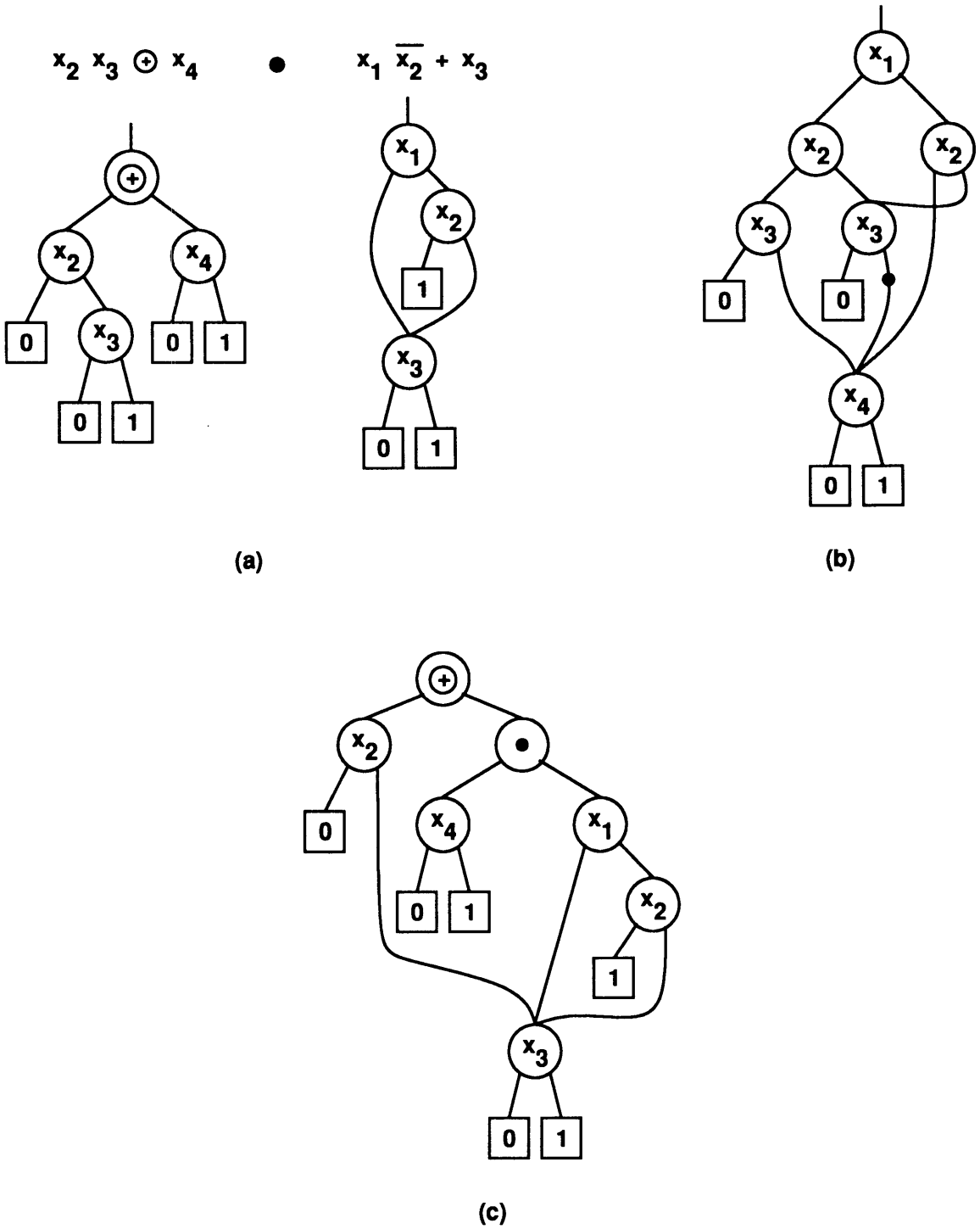


FIGURE 6-8: (a) FBDs representing $f = (x_2 \cdot x_3) \oplus x_4$ and $g = x_1 \cdot \overline{x_2} + x_3$. (b) The result without function vertex retainment. (c) The result with the $\oplus$ vertex retained

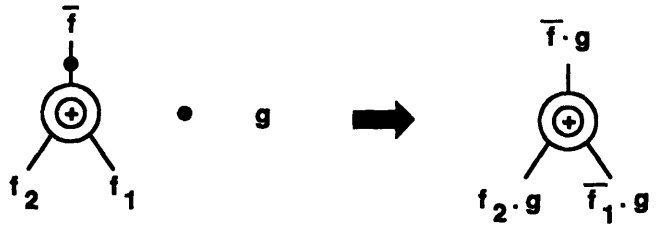


FIGURE 6-9: Example of $\oplus$ retainment with complement edges

with different supports. Namely, there are two nodes v_1 and v_2 such that $support(v_1) \neq support(v_2)$ but the functions represented by the nodes are equivalent so $f^{v_1} \equiv f^{v_2}$ and $|v_1| = |v_2|$. In the XOR-FBD package, these vertices are not actively removed from the XOR-FBD, but rather the two XOR-FBDs are compared, and the XOR-FBD with the larger top index is used. The XOR-FBD with the smaller top index is redundant in that variable and is discarded.

For example, during the *and* operation, before constructing the node v_2, an equivalent node v_1 is located in the unique table. However, if $top\_index(v_1) < top\_index(v_2)$, we need to use the v_2 representation of the function instead of the v_1 representation. This is easily accomplished by replacing v_1 with v_2 without changing the address of the node v_1, so all parents of v_1 still refer to the correct function.

The input ordering restriction along each path is also satisfied after the replacement, because v_2 satisfies the ordering requirement, and all variables in v_2 are guaranteed to have greater index than the index of v_1. All parents of v_1 must have index less than $index(v_1)$.

In Figure 6-10, variable x_3 is the top variable in the FBD rooted at vertex v_1. The internal node, w_1, is the root of an FBD that can contain more redundant variables, but these variables do not have to be removed at this point. Vertex v_1 can be replaced by vertex v_2, which represents the same function without using variable x_3. The global ordering requirement is still satisfied, because all paths from root to vertex v_2 have seen only variables with index less than 3, and all variables seen after v_2 have index greater than 3.

Redundant vertices cannot be handled in this manner without the assumption of a global input ordering. If the inputs are not subject to a global ordering, a support field is required to prune the number of cofactor calls, and redundant vertices must be removed when discovered. The lack of an overall ordering implies that no assumptions should be made concerning the variables that have already been seen before reaching a certain node.

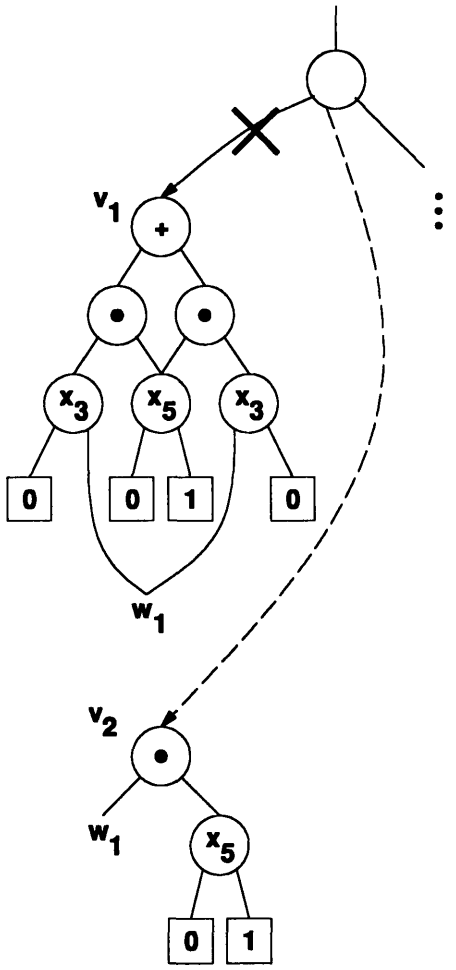


FIGURE 6-10: Removal of a redundant top variable, x_3

The NAND vertex is necessary for correctly replacing AND vertices that contain redundant variables. The NAND vertex allows us to replace an AND vertex without introducing complement edges, and altering the parents of the vertex. An $\otimes$ vertex is not necessary due to the symmetry in the *xor* function. Namely, $\overline{a \oplus b} = a \oplus \overline{b} = \overline{a} \oplus b$. The complement of an $\oplus$ vertex can be obtained by simply complementing one of the children.

6.6 Results

The results shown in Table 6-4 were obtained using the XOR-FBD package with unrestricted $\oplus$ and support-restricted AND function vertices. The benchmark circuits are the same circuits used to obtain the results of Section 5.6.

The same global ordering was given to all the packages. Examples marked with an asterisk require different orderings for different outputs to create XOR-FBDs or ROBDDs.

The first column gives the name of the example circuit. The next two columns labeled ROBDD time and nodes show the number of nodes in the ROBDDs representing the outputs of the circuit and the time required to obtain the results using the package in `sis`. Columns labeled FBD show the results for the circuits using the FBD package described in Chapter 5. The columns labeled AND/$\oplus$ show the results obtained using the XOR-FBD package with $\oplus$ and AND nodes. The last column is a count of the number of function vertices present in the final XOR-FBD results.

Example **C499** showed significant improvement over both the ROBDD package and the previous FBD package. The improvement is attributable to the preponderance of XOR gates in the circuit specification.

The benchmarks show that in a majority of examples, the final XOR-FBDs actually used a number of function vertices. The function vertices were able to decrease the number of graph nodes required to represent the circuit, by up to a factor of 11 in the case of **hwb32**. The XOR-FBDs also required less memory to build the smaller final results for the benchmark circuits.

The multipliers are still hard for XOR-FBDs with function vertices. **mult12** is a 12-bit multiplier, and **C6288(14)** is the 14th output of the 16-bit integer multiplier.

The results were obtained using 4 passes and field size $\|S\| = 65,536$. No errors were encountered in the examples.

EX	ROBDD		FBD		AND/⊕		#
	Nodes	Time	Nodes	Time	Nodes	Time	F.V.
C432	31,195	26	29,004	36	28,629	43	2,411
C499	33,214	14	33,214	16	5,354	17	60
C880	7,761	2	6,851	4	6,861	4	1,109
C1355	33,214	14	33,214	29	33,214	92	24
C1908	12,734	9	12,765	16	12,553	26	365
C2670	57,767	21	28,633	36	21,805	55	171
C3540	88,652	82	87,575	157	87,617	114	1,906
C5315	26,129	11	2,987	17	2,898	36	664
C6288(14)	115,607	535	115,607	2107	115,607	1,522	0
C7552	8,356	8	7,511	7	7,515	9	47
i10	880,254	306	638,890	671	680,100	471	50,804
s5378	6,854	2	3,348	1	3,599	2	706
s9234*	1,069,633	632	398,211	1665	824,958	742	262,342
s13207	13,892	9	13,892	5	5,054	6	472
s15850	53,612	26	28,644	29	28,928	40	5,036
s35932	6,190	36	5,731	9	5,731	3	290
s38584	40,817	56	27,209	35	24,808	22	3,003
hwb16	614	0.4	533	0.9	614	0.5	0
hwb32	40,126	36	3,474	66	3,470	177	29
hwb64	unable	unable	11,599	70	11,599	70	64
mm16	3,308	4	3,276	4	3,263	4	37
mm30	11,071	15	10,983	19	10,983	17	64
mult12	610,093	950	610,093	6,492	610,112	2,344	0
rotate32	unable	unable	2,117	0.4	2,117	0.3	992

Table 6-4: Results using the XOR-FBD package

6.7 Conclusion

The difference between the XOR-FBD package in this chapter and the previous FBD package is the use of sets of polynomials as signatures, instead of sets of integers. This leads to the introduction of the unrestricted ⊕ vertex.

The global input ordering restriction is still used to build the XOR-FBDs. Under the ordering restriction, certain assumptions can be made, and the XOR-FBD vertices no longer carry support information on the XOR-FBD.

The OR vertices are now completely subsumed by the unrestricted ⊕ vertices. The *or* of two orthogonal functions is equivalent to the *xor* of the orthogonal functions.

XOR-FBDs with ⊕ and AND function vertices are shown to benefit the representation

of the general benchmark circuits. These XOR-FBDs subsume the FBDs with restricted OR, AND and XOR function vertices.

Circuits that may especially benefit from an XOR-FBD representation are circuits with XOR gates, especially if the XOR gates are near the outputs. Examples of such functions are parity trees with other functions as the inputs.

7

FBD Package Implementation

7.1 Introduction

The efficient implementation of an FBD package is based upon the ROBDD package of [9]. Similar structures and routines are used with modifications for signatures, supports, and function vertices. The structures and routines also support unordered FBDs, a feature which is used in Chapter 8.

This chapter first discusses the fundamental FBD vertex data structure. Sections 7.3 and 7.4 detail the operation of the hash tables and hash caches essential for efficient FBD manipulations. The Boolean operations are described in Section 7.5. A discussion of the impact of several parameters on FBD size, memory requirement and computation time is presented in Section 7.6. The automatic garbage collection scheme and amortized memory usage by the FBD package are also discussed, followed by a conclusion.

7.2 FBD structures

Each vertex in the FBD is represented by a vertex structure that has one pointer to the high child and one pointer to the low child. The FBD vertex also uses 16 bits to represent the index of the decision variable in the case of a decision vertex, or the smallest index present in the FBD in the case of a function vertex. The number of pointers to the vertex is maintained in a saturating reference count that goes up to 255. The type of vertex is maintained, distinguishing between the different types of function vertex, and the decision and terminal vertex.

The vertex structure also maintains a pointer to the next node in the unique table collision list, and an array of words for storing the signature of the vertex. The size of

the signature is determined by the number of passes used. Each value in the signature uses 16 bits, so two passes are represented by one 32-bit word. The number of passes is relevant for bounding the error probability. When using unordered FBDs, a support field is required and an additional pointer is necessary to maintain the array of integers that form the packed support field.

Complemented edges are marked using the least significant bit in the pointer to a vertex. The complement attribute requires no additional memory.

7.3 Basic Hash Tables

The unique table stores all vertices with unique signatures. The unique table consists of an array of bins, where each bin contains the head of a singly-linked list of vertices. Only uncomplemented nodes are stored in the unique table lists.

Given a signature, the unique table is checked for that signature in the following fashion. The first word of the signature is used to hash into a bin in the table. The list in the bin is traversed, comparing signatures, until a match is found, or until the list has been fully traversed.

The lists are currently not sorted because we assume the collision lists are very short, around four nodes on average. If the lists become long (in the case of a size limit on the unique table), the entries can be sorted by signature.

During a search, if a matching signature was found, this is called a unique table hit. In general, the hit rates on the unique table are quite high, over 70% for most examples. This implies we are dealing with mostly equivalent nodes when creating FBDs. Many of the hits also involve identical nodes, so a large percentage of the unique table hits do not even contribute to the error probability.

The computed table is a hash-based cache that stores the results of previous *apply* or *and* computations. The key used to access cache entries is the signatures of the arguments to the *apply* or *and* function. It is important to use portions of the signatures of all arguments to the function call in the key, to properly distribute the entries in the cache.

If the signatures of parameters for the current function call match the signatures of the cache table entry, the result stored in the entry is returned. The computed table for FBDs does not store any actual node pointers, but rather uses the signatures as both the result and the key.

This means that the entries in the computed table do not need to be flushed. Also, any amount of node replacement can take place without affecting the validity of the table

entries. The computed table hit rate for the example circuits varied, from 17% to 45%.

Even with the seemingly low hit rates, the computed table is essential for building FBDs efficiently. The presence of the table reduces computation time from possibly exponential to guaranteed polynomial. Also, the current caching scheme simply replaces an old computed table entry with the new entry in the event of a collision. It seems that the cache may benefit somewhat from some simple caching schemes, such as a replacement scheme based upon the number of recursions saved by that entry and the depth of the entry, or perhaps a scheme that retains entries that are hit frequently.

If node replacement in the unique table is allowed only if the alternate representations have identical signatures, pointers can be used to access the caches, in place of signatures. This saves memory, but the cache entries need to be checked for dead nodes during garbage collection.

7.4 Additional Hash Tables

Additional hash caches may be necessary if the ordering requirement is removed for FBDs, or in the case of the multiple-input *and* used in the Boolean satisfiability problem (see Chapter 8).

Lifting the global input ordering restriction on FBDs implies that the general cofactor procedure will be called extensively, and the recursive nature of the function will adversely impact computation time. As a result, a cofactor table similar to the computed table may help reduce the computation time significantly.

The multiple-input *and* also requires the use of an AND computed table to reduce computation time. This computed table simply stores the results of the multiple-input *and*, and is used in exactly the same way as the regular computed table.

7.5 Operations

The cofactor operation is recursive and general. The *top_index* of the FBD is used for pruning if a global input ordering is enforced. Otherwise, the support field is necessary for pruning purposes.

We allow the replacement of a node in the unique table by an equivalent vertex that is either smaller or has fewer redundant vertices. There is a possibility that we will replace a node that is actively being cofactored and this presents a problem.

During the cofactoring of a function vertex with a redundant variable, care must be taken when replacing nodes. We have to insure that we detect the replacement of the

node we are cofactoring. During the process of cofactoring a node, if we discover the node has been *replaced*, this implies no further work is required, the node has already been cofactored, and the result without redundant vertices has been stored in the current node location.

The *apply* and *and* operations are straightforward. We use the general cofactor algorithm to calculate the arguments for further recursions. We perform signature checks for function vertex introduction and retainment. When calculating multiple representations for the same arguments, the size of the first result is used to terminate extra calculations if the size of the alternate has exceeded the size of the current result.

7.6 Parameters

Three parameters were used during the creation of FBDs. These parameters are tied to limiting function vertex creation and retention. The parameters, *intro_m, retain_m* and *or_limit*, affect the computation time, intermediate memory requirements, and the size of the results.

The check for introducing disjoint OR vertices or unrestricted $\oplus$ vertices is performed every *intro_m* recursions of *apply* or *and*, starting with the root call. The check for function vertex retention is also performed every *retain_m* recursions. When deciding between two FBD versions of the same function, the smaller representation is chosen, otherwise the first result is used. A representation is smaller if it has at least *or_limit* fewer nodes than the alternative.

Choosing a smaller value for the parameters increases the number of FBD size calculations and the number of *apply* or *and* recursions, increasing the computation time. However, the smaller representations may improve the results of the computation, and reduce the memory requirement for creating the FBDs. Typical values are:

$$intro\_m = 8$$

$$retain\_m = 8$$

$$or\_limit = 400$$

More function vertices are introduced as the values get smaller.

Table 7-1 shows some of the results obtained for example circuit **C5315** under different parameters. The first column shows the value of *or_limit* for the runs. The three sets of three columns show the results for $retain\_m = intro\_m = 8, 4, 1$. The columns labeled "Node" give the number of nodes required to implement the FBDs for the outputs

or_limit	$m = 8$			$m = 4$			$m = 1$		
	Nodes	Active	Time	Nodes	Active	Time	Nodes	Active	Time
400	14,694	18,968	39.3	14,694	18,292	56.3	14,861	18,003	135
100	4,627	8,676	39.0	4,938	8,001	51.7	6,179	7,934	111.9
5	2,898	6,100	35.7	2,898	3,982	40.7	2,907	3,345	92.5

Table 7-1: Effect of the parameters on circuit example **C5315**

of the circuits. "Active" shows the number of active nodes required to build the FBDs, and gives an indication of the memory required to build the results. "Time" is the CPU time, in seconds, used to build the graphs.

The results improve as *or_limit* is decreased, resulting in an 80% difference between the FBD sizes when the parameter decreases from 400 to 5. The frequency of the checks for function vertex introduction and retention does not affect the final results significantly, but does influence the memory required to build the FBDs. The number of active nodes during FBD creation is a measure of memory usage. The more function vertex introduction and retention checks we make, the smaller the active node requirement. There is a 45% difference between $m = 8$ and $m = 1$ in the last row. However, the computation time also increased 159%, showing the tradeoff between memory usage and computation time.

The main consumer of computation time, when *intro_m* and *retain_m* are small, is FBD size calculation. The size can be calculated in $O(size(G))$, where $size(G)$, is the size of the graph, but when this function is repeated recursively, the cost of computing the size becomes very high. For example, almost 60% of the time required for **C1908** with $m = 8$ and $or\_limit = 400$ is used by the size calculation. Reducing the number of size computations should significantly decrease the computation time.

However, as long as function vertices are still introduced and retained, the FBDs are still smaller than ROBDDs. In the limit, when function vertex introduction and retainment no longer take place, and a global input ordering is enforced, the FBD representation is equivalent to the ROBDD representation.

7.7 Garbage Collect

The number of pointers to nodes are maintained using reference counters, as in the efficient ROBDD package. The reference counters saturate at 255, but this is not a common occurrence, except in the case of the terminal vertex. Garbage collection removes

all nodes with zero reference counters, and all unique table entries that refer to the physical address of these nodes.

During garbage collection, if fixed limits on the table sizes are not exceeded, the unique table and the computed table are resized and the number of bins is doubled.

Two schemes were used to test for the need to garbage collect. The first scheme garbage collects when either the load factor on the computed table exceeds a limit, or when the unique table load factor is greater than 4. The load factor is the ratio of entries to the number of bins. In the second scheme, garbage collection is automatically performed when the current number of nodes in the unique table exceeds the previous maximum by 4 and the computed table usage does not initiate a garbage collect.

The first method collects garbage more frequently, and the number of bins in the tables increases faster. The frequency of garbage collect and the table sizes affect FBD computation time. Although removing unreferenced nodes may slow computation time because the nodes have to be recalculated, the expansion of the tables speeds up the program by enlarging the hash cache and the unique table.

More bins in the unique table and the computed table will decrease computation time. The collision lists in the unique table are shorter and hence require less traversal time when searching the unique table. A larger computed table means that more previous results entries are stored in the cache, reducing computation time.

However, this means more nodes are stored, so to limit memory usage, we also placed physical limits on the number of bins in the unique table and the computed table. Once the limits are exceeded, the computation time for manipulating the FBDs increases significantly.

7.8 Memory Usage

The amortized memory cost per ROBDD node for the ROBDD package of [9] is 22 bytes per node. This cost is obtained assuming each node requires 16 bytes of memory and the cost of the each unique table bin, computed table bin and computed table entry is distributed over 4 nodes.

The amortized memory usage per FBD node in the package of Chapter 5 is 54 bytes per node, under the same assumptions as the calculation for ROBDD nodes (4 nodes in each unique table collision list, and the same number of bins in the unique table and the computed table). The calculation also assumes 3 passes for the signature computation, but not the memory necessary for the support information. Experimental results show that the total memory usage, including all overhead, is around 60 bytes per node.

The XOR-FBD package improved the amortized cost per FBD node to 32 bytes per node, under the same assumptions for the hash cache and hash table, but the number of passes has been increased to 4. If 6 passes are used, the memory cost increases to 39 bytes per vertex, because of the extra word required to store two more passes, and the increase in the size of the computed table entries. Experimental results confirm that XOR-FBD nodes consume 32 bytes per node.

7.9 Conclusion

The ROBDD data structures and speedups, such as the computed table, can be generalized to work for FBDs using signatures. The ROBDD algorithms can also be modified to account for signatures, function vertices, and the lack of a global input ordering.

Adjustable parameters also limit the frequency of function vertex introduction and retainment. An additional parameter controls how much smaller the alternate representation has to be before it is used. The setting of the parameters affects the size of the result, the memory required to compute the result, and the computation time.

The FBD package uses automatic garbage collection to resize the hash tables, and to remove nodes that are not in use. An efficient FBD package averages 32 bytes of memory per node, about 1.45 times the ROBDD memory cost of 22 bytes per node.

In summary, the FBD package is able to mimic the functionality of the efficient ROBDD package. Although the FBD package is computationally more expensive and the amortized memory cost per node is higher, the FBDs can be more efficient than ROBDDs if the FBDs are small enough, because the memory usage and graph algorithm complexity depend on graph size.

8

Combinatorial Optimization Application

8.1 Introduction

This chapter describes the application of FBDs to solve combinatorial optimization problems. A multiple-input *and* function is presented that computes the conjunction of an array of functions. We use "dynamic" variable selection to exploit the unordered nature of FBDs. AND vertices are used to concatenate FBDs with disjoint supports. We show that the unordered FBDs outperform the conventional ROBDD-based methods in terms of both memory usage and CPU time.

The specific problem addressed is optimal layout stated as a Boolean satisfiability problem. Two-layer dogleg channel routing, multi-layer dogleg channel routing, two-way partitioning, one-dimensional and two-dimensional placement problems can be transformed into Boolean functions, where the satisfiability of the function shows that an optimal assignment exists, and a satisfying assignment is an optimal assignment [21].

The problem and transformation method will be reviewed in Section 8.2, followed by a description of the FBD-based method for solving the resulting Boolean satisfiability problem, in Section 8.3. Results for some benchmarks are presented in Section 8.4. Conclusions are presented in Section 8.5.

8.2 Optimal Layout Stated as a Boolean Satisfiability Problem

Devadas presents a method for solving the optimal layout problem exactly and more efficiently than exhaustive search, by transforming the layout problem into a Boolean

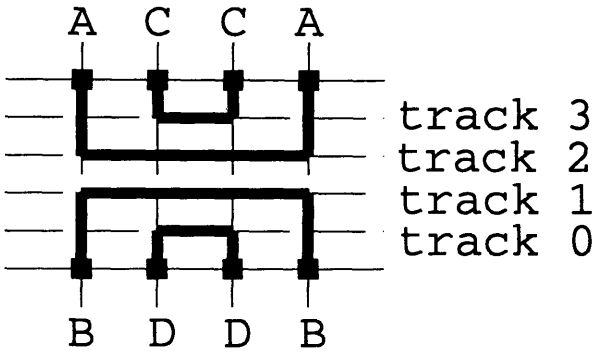


FIGURE 8-1: An example of a channel with 4 nets and 4 tracks

satisfiability problem [21]. Although the problems are NP-complete, it is still possible to obtain an optimal solution for large problems in reasonable time and memory.

The method transforms the layout problems into a Boolean function whose inputs grow linearly or quasi-linearly with problem size. The satisfiability of the Boolean function provides the answer to the decision problem, "can a given channel be routed in $\leq D$ tracks" and in the process of determining satisfiability, the satisfiable assignment of tracks to nets is also discovered. Two-layer channel routing and the transformation technique for two-layer channel routing will be reviewed here.

8.2.1 Two-Layer Channel Routing

The two-layer wiring model assumes that all routes on a layer run in only one direction, either vertical or horizontal. A channel is composed of horizontal and vertical tracks, with two rows of terminals, one at the top of the channel and one at the bottom of the channel. The terminals are assigned nets which need to be routed in a limited number of tracks in the channel. Initially, no doglegging is permitted, so each route has only one horizontal segment.

An example of a channel with four horizontal tracks, four columns and four nets, A, B, C, D is shown in Figure 8-1. The tracks are numbered from the bottom of the channel to the top, in ascending order. The columns are numbered from left to right. The first column on the left is labeled column 1. The terminals are assigned nets A, B, C, D. In the figure, a valid route for this channel is shown. Net A is routed in track 2. Net B is routed in track 1. Track 3 has been assigned to net C and track 0 has been assigned to net D.

The channel routing problem is converted into a Boolean satisfiability problem by forming the conjunction of Boolean functions that represent the constraints on the route.

A given channel can be routed in $\leq D$ tracks if the associated Boolean function is satisfiable.

Assume there are N nets that need to be routed in $\leq D$ horizontal tracks. Each net i has $M = \lceil \log_2(D) \rceil$ Boolean variables $\{v_{i1}, v_{i2}, ..., v_{iM}\}$. These variables represent the binary encoding for the number of the track in which this net will be routed. This means the Boolean function that will be tested for satisfiability is a function of $N \cdot \lceil \log_2(D) \rceil$ input variables. The Boolean function is the conjunction of routing constraints. The three classes of constraints are described in the sequel.

The interval of a net i is defined as $Interval(i) = [l_i, r_i]$, where l_i is the leftmost column position of net i and r_i is the rightmost position. Nets with overlapping intervals cannot be placed in the same track and this requirement forms the first constraint. For the net pair (i, j) if $Interval(i) \cap Interval(j) \neq \phi$ then we require

$$v_{i1} \oplus v_{j1} + v_{i2} \oplus v_{j2} + ... + v_{iP} \oplus v_{jP} \tag{8.1}$$

to be true under the assignment.

The maximum number of functions possible from this constraint is $\frac{N(N-1)}{2}$ because for N nets there are $\frac{N(N-1)}{2}$ unique net pairs, and it is possible for all intervals to overlap.

The second type of constraint, column requirements in the channel, can be expressed in terms of a directed graph, known as the Vertical Constraint Graph (VCG). An edge from net i to net j in the VCG represents the appearance of net i on the top terminal of a column that has net j on the bottom terminal. The vertical routing constraint requires that net j must be routed on a lower numbered track than i, so the corresponding constraint is the logic function that computes:

$$(v_{i1}\ v_{i2}\ ...\ v_{iM})\ >\ (v_{j1}\ v_{j2}\ ...\ v_{jM}), \tag{8.2}$$

where $(v_{i1}\ v_{i2}\ ...\ v_{iM})$ and $(v_{j1}\ v_{j2}\ ...\ v_{jM})$ represent bit vectors.

Due to transitivity, if edges (i, j) and (j, k) both exist, edge (i, k) is represented implicitly and some amount of reduction is applicable to the VCG, reducing the number of functions derived from this constraint. The maximum number of functions from this constraint is the total number of columns in the channel, since each column gives rise to at most one constraint. The vertical constraints may also subsume some of the inequality constraints in Equation 8.1, because $>$ also implies $\neq$. In this case, not all of the functions associated with Equation 8.1 need to be represented.

The third constraint prevents the placement of routes in invalid tracks. This constraint is necessary if $D < 2^M$, yielding $2^M - D$ possible invalid tracks. At most $(2^M - D)N$ functions are associated with the third constraint.

One function, Equation 8.3, is built for each unused bit vector $[c_{k1}\ c_{k2}\ ...\ c_{kM}]$. Depending upon track numbering assignments, unused bit vectors can often be combined for fewer Boolean functions. We have:

$$v_{i1} \oplus c_{k1} + v_{i2} \oplus c_{k2} + ... + v_{iM} \oplus c_{kM} \tag{8.3}$$

The conjunction of all the functions derived from the preceding three constraints forms the Boolean function whose satisfiability proves that the route is possible in $\leq D$ tracks. Some of the interval constraints may be subsumed by vertical constraints, but the resulting Boolean satisfiability problem may still be very large, and unsolvable for larger problems in reasonable time and memory.

8.3 FBDs Applied to Boolean Satisfiability

We examine the application of unordered FBDs to Boolean satisfiability problems derived from channel routing. The FBDs are not subject to a global input ordering restriction, and variables are actively ordered during the construction of FBDs.

The satisfiability of a Boolean function is determined by creating the FBD for the function and comparing the FBD signature with 0. A traversal of the nodes in the graph produces a satisfying assignment.

The unordered nature of the FBDs used to determine satisfiability requires the use of the support fields for each node in the FBD. The support information is necessary to quickly compute the general cofactor necessary when working with unordered graphs.

8.3.1 Multiple-Input And

The basis for the efficient FBD-based satisfiability check is the multiple-input *and*. The multiple-input *and* accepts a vector of subfunctions, and computes the conjunction of the subfunctions simultaneously. The conventional ROBDD algorithms are 2-input algorithms, where the Boolean operations are 2-input operations. The computation of an n-input *and* using 2-input operations would require $n-1$ *and* operations.

The multiple-input *and* computes the n-input *and* function recursively, with one root call. The algorithm for the multiple-input *and* is shown in Figure 8-2.

The algorithm is called with n subfunctions. The algorithm first checks the subfunctions for the 0 terminal vertex since a 0 subfunction means the conjunction is unsatisfiable. If the subfunctions are all non-zero, a lookup is performed in the *and computed table*. The key to this hash cache is the vector of subfunctions, and a valid entry returns the result of a previous multiple-input *and* computation with the same arguments.

```
n-and (P = {f_1, f_2, ..., f_n})
{
    if (f_i == 0;  1 ≤ i ≤ n)
        return (0) ;
    if (result = and_computed_table_lookup(P))
        return (result) ;
    partition P into k sets S_1, ..., S_k such that
            support(S_i) ∩ support(S_j) = φ and S_i ∩ S_j = φ for all 1 ≤ i, j ≤ k, i ≠ j ;
    for (i = 1 to k) {
        x_i = select_variable(S_i) ;
        create vertex v with decision variable x_i ;
        high(v) = n-and(S_i |_{x_i = 1}) ;
        low(v) = n-and(S_i |_{x_i = 0}) ;
        compute signature |s| of result = (x_i, low(v), high(v)) ;
        if (result = unique_table_lookup(|s|)) {
            if (size(result) < size(v)) {
                Free low(v) and high(v) ;
                v = result ;
            }
        }
        r_i = concatenate(v, r_{i-1}) ;
    }
    insert r_k = (f_1,..., f_n) into and_computed_table ;
    return (r_k) ;
}
```

FIGURE 8-2: Procedure to compute a multiple-input *and*

Disjoint Support Sets

The set of input functions, $P = \{f_1, ..., f_n\}$, is partitioned into k subsets, $S_1, ..., S_k$. The support of the subset S_i, $support(S_i)$, is simply the disjunction of the supports of the FBDs contained. The rule for partitioning the subsets is that the subsets must have pairwise disjoint supports:

$$support(S_i) \cap support(S_j) = \phi; \quad 1 \leq i, j \leq k, i \neq j$$

The partition is discovered by building an undirected graph that describes the connectivity of the supports of the input FBDs, and performing a depth-first search on the graph to discover unconnected components. The nodes in the graph correspond to the inputs of the *and*. An edge exists between two nodes i and j, if the functions f_i and f_j share common support variables. The nodes in each component subgraph form a subset that is going to be disjoint in supports from any other component.

Multiple-input *and* is called recursively on each subset of FBDs. The results of the k multiple-input *ands* are concatenated together, using AND vertices, to form the final FBD. If we discover a 0 FBD during the creation of FBDs for the subsets, calculations terminate immediately because the final result is also the 0 terminal vertex.

Input Variable Ordering

The FBD for a subset is created by calling the **n-and** procedure on the cofactors of the functions in the given subset. A variable needs to be selected as the variable to cofactor against, and this variable is chosen based upon the properties of the constraint functions which are the inputs to the multiple-input *and*.

The problem we would like to solve is the conjunction of subfunctions. The subfunctions fall into one of the three types of constraint categories discussed in the previous section. The first two categories produce functions of the form $f_1 = (p < q)$ and $f_2 = (p \neq q)$, where p and q are integers, encoded by the M variables $\{p_1, p_2, ..., p_M\}$ and $\{q_1, q_2, ..., q_M\}$.

Subfunctions of type f_1 and f_2 have $O(M)$-sized FBDs if the variables are ordered in the following way:

$$\{p_1, q_1, p_2, q_2, ..., p_M, q_M\} \tag{8.4}$$

This is called the interleaved ordering. However, we are dealing with the conjunction of a number of these types of functions, and there are no restrictions on the number of constraints the nets p or q will appear in, although each pair, (p, q), can only appear once in a satisfiable route without doglegging.

As the number of constraints and nets increases, other orderings become more efficient. The non-interleaved ordering shown in Equation 8.5, which groups all variables that encode a particular net together, becomes more effective.

$$\{p_1, p_2, p_M, ..., q_1, q_2, ..., q_M\}, \tag{8.5}$$

The non-interleaved ordering is a better ordering when there are more constraints because the valid track assignments will need to be fully enumerated for each net, regardless of the input ordering for the FBD, so the FBDs will be smaller if the encoding bits for each net are placed together.

Our first ordering strategy uses the interleaved ordering shown in Equation 8.4 to build $O(M)$-sized FBDs for the individual subfunctions. We switch to a non-interleaved ordering when we compute the conjunction of the subfunctions. The variables associated with each net will be placed together, and the order in which the nets appear will be the original order of appearance in the interleaved ordering.

Articulation Points

We also use articulation points to dynamically select variables. By choosing the variables associated with the articulation points, we may break up the set of input functions into more disjoint sets.

A vertex is an articulation point in the graph if the removal of the vertex results in at least two disconnected, non-empty subgraphs. The nodes in the graph are the nets in the constraints, and an edge exists between two nodes i and j if the nets n_i and n_j are involved in the same constraint. There are three possibilities for the relationship between the assignments to n_i and n_j:

$$n_i < n_j \quad \text{or}$$

$$n_i > n_j \quad \text{or}$$

$$n_i \neq n_j$$

By selecting the variables in articulation points first, we disconnect the graph into more unconnected components. The FBDs for the unconnected components can be created and attached using AND vertices.

The remainder of this section reviews articulation points, and the algorithm for finding the points in a graph. Vertices 3 and 5 in Figure 8-3 are articulation points. The resulting graphs after the deletion of the articulation points are shown in Figure 8-4.

An efficient algorithm based upon depth-first search (DFS) for detecting articulation points was developed in [33]. The algorithm is based upon the observation that if an

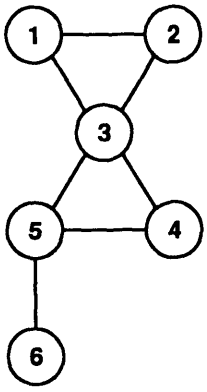


FIGURE 8-3: A support graph

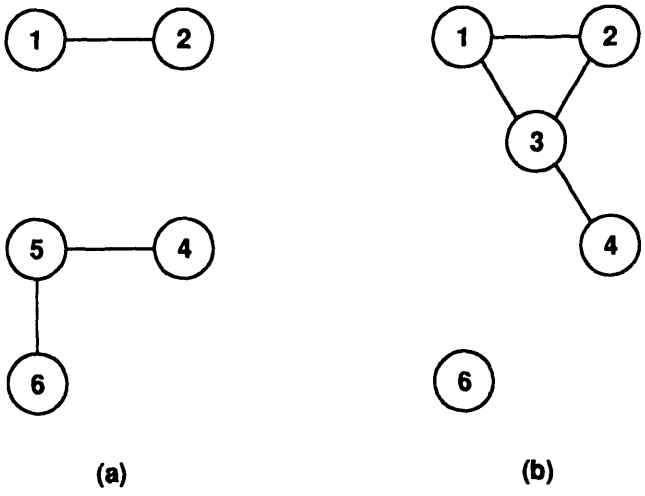


FIGURE 8-4: The support graph (a) after the elimination of articulation point 3 and (b) after the elimination of articulation point 5

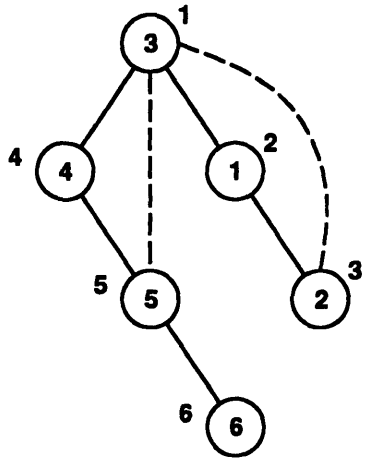


FIGURE 8-5: Depth-first spanning tree

edge (u, v) exists, without loss of generality we can assume vertex u is visited first, and it is easy to see that the depth-first search of u is not complete until the vertex v has been visited. This means that u will always be an ancestor of vertex v in the depth-first spanning tree.

The depth-first spanning tree for Figure 8-3 is shown in Figure 8-5. The order in which the nodes are visited is indicated by the number outside each vertex. This number is called the depth-first number or DFN. Back edges in the DFS spanning tree are indicated by the dashed lines. This spanning tree is for a search that starts at node 3.

A vertex u in the DFS spanning tree is not an articulation point if after u is removed from the tree, each child w of u can still reach an ancestor of u using the descendants of w and back edges. This implies that the graph is still connected after the removal of u, so u cannot be an articulation point. The root vertex is an articulation point if it has more than one child in the DFS spanning tree. A child of the root vertex can only reach another child through the root vertex.

The algorithm for determining articulation points uses two numbers for each vertex u, $DFN(u)$ and $L(u)$. $DFN(u)$ is simply the depth-first number for u. $L(u)$ is the lowest depth node that can be reached from u using descendants of u and back edges. The articulation points are nodes u with a child w such that $L(w) \geq DFN(u)$.

Figure 8-6 shows the algorithm for performing the depth-first search and the assignment of $L(u)$ concurrently. Note that the argument v is the parent of u. Each node in the spanning tree can have multiple children, but only one parent. The arrays DFN and L are global. Initially, $depth = 0$ and $DFN(u) = -1$ for all u. The initialization takes place before calling the articulation point algorithm. In the algorithm, $DFN(u)$ and $L(u)$ are assigned to the current depth. Then, for each child w, if w has not been

```
find_art (u, v, depth)
{
    /* u is the current vertex, v is the parent of u */
    DFN(u) = L(u) = depth ;
    for (w adjacent to u) {
        if (DFN(w) == -1) {
            /* The node has not been visited yet, do so now */
            find_art (w, u, depth + 1) ;
            /* Assign L(u) to either L(u) or L(w), whichever is smaller */
            L(u) = MIN(L(u), L(w)) ;
            if (L(w) ≥ DFN(u))
                return (u is an articulation point) ;
        } else if (w ≠ v) {
            /* Found a back edge, assign L(u) to the DFN(w) if it is smaller */
            L(u) = MIN(L(u), DFN(w)) ;
        }
    }
}
```

FIGURE 8-6: Pseudocode for finding articulation points

reached yet, **find_art** is called recursively and then $L(u)$ becomes $L(w)$ if it is smaller. If $L(w) \geq DFN(u)$, then u is an articulation point. If w has already been reached, we have located a back edge, and $L(u) = DFN(w)$ only if $DFN(w) < L(u)$.

Concatenation

After we obtain the result FBDs for the disjoint sets, the FBDs are concatenated together using AND vertices. The concatenation of FBDs with disjoint supports created from the disjoint sets of functions is important for reducing the size of the resulting output FBD.

8.4 Results

We obtained some results using FBDs to check the satisfiability of channel routing functions. The FBD package used support fields, general cofactor, and the **n-and** multiple-input *and* function. The results were obtained on a SPARCstation 10 and an IBM 590.

Table 8-1 compares the ROBDD-based satisfiability check against the multiple-input *and* FBD check, without any dynamic variable ordering. The first column gives the name

Function	n	m	c	ROBDD			N-AND		
				Nodes	Active	Time	Nodes	Active	Time
2a	19	3	68	318	11,377	3	266	587	0.5
2b	24	3	96	73	76,285	26	73	998	77
ex1	6	2	9	24	57	0.02	24	46	0.01
ex2	6	2	10	34	84	0.03	24	51	0.02
ex3	12	4	68	4,599	2,903,274	875	871	1,778	111
ltest	16	3	13	96	332	0.07	96	300	0.06
nasty5.5	9	3	30	28	2,216	0.3	28	299	0.13
nasty7.5	11	3	34	34	2,450	0.3	34	335	0.11

Table 8-1: Comparison of ROBDDs and N-AND without dynamic ordering

of the function. The second column lists the number of nets in the routing problem and the third column shows the number of encoding bits required for each net. The fourth column is the number of constraints in the conjunction.

The next six columns show the final number of nodes in the result graphs, the maximum number of nodes required to compute the final result, and the time it took to compute the results, in seconds. The first set of 3 columns shows the statistics obtained from the ROBDD package. The second set shows the results obtained from the FBD package using the multiple-input *and* without dynamic variable ordering.

Examples **2a, ex1, ex2, ex3** and **ltest** are satisfiable. **2b, nasty5.5** and **nasty7.5** are unsatisfiable examples. The multiple input *and* greatly improves the unsatisfiable examples because by considering all the constraints concurrently, the conflicting constraints are going to be discovered sooner.

The multiple-input *and* greatly improves the active node count, which is the number of nodes required to find the result. The final result of the output is also smaller in the satisfiable examples. In the unsatisfiable examples, the final number of nodes reported is simply the number of inputs plus 1 more node for the 1/0 terminal vertex.

The active node count is the primary indicator of the effectiveness of FBDs in solving the Boolean satisfiability problem. The ROBDD-based method allows the intermediate nodes, built while trying to compute the 2-input *and*, to grow. The growth in the intermediate nodes may become so large, a final result is never obtained by the ROBDD method within reasonable memory usage and CPU time limits.

Table 8-2 compiles the results for the same functions using multiple-input *and* and dynamic variable ordering. The first column gives the name of the function. The first set of 3 columns gives the results using just the multiple-input *and* and the given global

Function	N-AND			INTER			ART		
	Nodes	Active	Time	Nodes	Active	Time	Nodes	Active	Time
2a	266	587	0.5	268	543	1.13	228	506	0.6
2b	73	998	77	73	932	174	73	912	271
ex1	24	46	0.01	24	48	0.02	24	50	0.02
ex2	24	51	0.02	24	53	0.01	24	56	0.03
ex3	871	1,778	111	872	1,421	248	877	1,437	1,129
ltest	96	300	0.06	96	204	0.1	96	220	0.1
n5.5	28	299	0.13	28	264	0.3	28	265	0.2
n7.5	34	335	0.11	34	281	0.2	34	282	0.2

Table 8-2: N-AND results with two types of dynamic ordering

input ordering. The second set of 3 columns shows the results obtained when the bits in the net encodings are interleaved to build the constraint functions, but the bits for the same net are later ordered together while computing the conjunction of constraints. The third set of 3 columns shows the results obtained when in addition to the first dynamic ordering scheme, we also select the variables in the articulation points before selecting any other variables.

We are primarily interested in how much memory was required to build the result of the satisfiability problem. The dynamic variable ordering is not useful on the smaller examples because more nodes need to be created due to the different orderings. However, on the larger examples, and especially on the satisfiable examples, the dynamic variable ordering reduces the intermediate memory usage.

Selecting articulation points also decreases the memory usage for **2a** and **2b** over just the first dynamic scheme. For some examples (but not the examples in the table), the articulation points will significantly decrease the memory usage. This occurs in examples where there are a few shared nodes that prevent the function sets from breaking up into disjoint sets. However, more comprehensive experiments are necessary to justify the use of articulation points.

8.5 Conclusion

We applied unordered FBDs to the problem of optimal two-layer routing stated as a Boolean satisfiability problem. We showed that the multiple-input *and* function and dynamic variable selection during the creation of FBDs can significantly decrease memory usage.

The multiple input *and* partitions the inputs of the *and* into sets such that the supports of the sets do not intersect. The FBDs for the disjoint sets are constructed by dynamically selecting decision variables. The variables are selected by grouping the variables associated with the same net, and by selecting variables to maximize the number of disjoint support sets. AND vertices are used to construct the final FBD from the disjoint support FBDs.

The FBD-based method significantly outperforms ROBDD-based methods, by producing results using less memory and CPU time. The dynamic variable selection method and multiple-input *and* also extend to other combinatorial optimization problems in addition to producing more compact FBDs when computing the *and* of FBDs with completely disjoint, or mostly disjoint supports.

9

Complexity

9.1 Introduction

The complexity of ROBDDs and free BDDs has been a subject of interest because bounds on the complexity of the BDD representations of certain functions lend insight to the usefulness and also the limitations of BDDs as a representation. A compact BDD representation is both memory efficient and easy to manipulate.

Exponential lower bounds for ROBDDs under any global input ordering have been proven for integer multipliers, the hidden weighted bit function and the shift rotate function [13, 22].

A number of bounds have been obtained for the read-only-once branching program model for a number of circuit classes [1, 25, 29, 44, 54, 55]. The read-only-once branching program is simply the free BDD representation of Boolean functions. Additional results were proven for classes of $\Omega - BP1s$, also known as branching programs with function vertices [45].

This chapter reviews some fundamental definitions. Existing complexity results and some interesting proof techniques for various forms of BP1s are summarized. The results are then extended to FBD complexity to gain an understanding of the power of the FBD representation.

FBDs are quite powerful, even under the constraints of restricted function vertices. Any function representable by a free BDD can also be represented by an FBD with the same order of complexity. For certain functions, a polynomial-sized FBD is still possible, despite exponential lower bounds for both the ROBDD and free BDD representation.

9.2 Definitions

A branching program (BP) is a directed, acyclic graph with decision vertices and terminal vertices. These vertices are exactly the vertices defined for ROBDDs. An $\Omega - BP$ is a branching program with function vertices that represent two-input Boolean functions. There are no restrictions on the children of the function vertices.

The size of a BP, P, is given by $size(P)$. The number of levels or the depth of P is denoted by $length(P)$. The maximum width of any level is called $width(P)$.

An oblivious BP is leveled, all paths from a source to a target node have the same length, and all nodes on the same level have the same decision variable, or none at all.

A read-only-once branching program (BP1) is subject to the restriction that each decision variable appears at most once along any path from root to terminal vertex. The terms free BDD and BP1 are interchangeable. An ROBDD is a BP1 that is restricted to the same ordering along each path.

An $\Omega - BP1$ is subject to the read-only-once restriction. There are no restrictions on the number of function vertex appearances along the paths in an $\Omega - BP1$. Examples of $\Omega - BP1s$ are the negation ($\Omega = \{\neg\}$),conjunctive ($\Omega = \{\wedge\}$), disjunctive ($\Omega = \{\vee\}$), and parity ($\Omega = \{\oplus\}$) BP1s. These function vertices have no restrictions on their children, and will be represented by symbols only. The restricted function vertices used by FBDs will be denoted by AND, OR and so forth.

An input vector, $w \in \{0,1\}^n$, is accepted if the graph, P, evaluates to 1 under w. An accepted set is the set of all input vectors such that P evaluates to 1. A rejected set, $w' \in \{0,1\}^n$ is the set of input vectors such that P evaluates to 0. The accepted set is an ON-set for the function represent by P while the rejected set is an OFF-set.

Two graphs are *computationally equivalent* if the sizes are different by no more than a constant factor, and the same sets are accepted. A $\{\neg\}$-BP1 is equivalent to a BP1 with complement attributes on the edges. It was shown that $\{\neg\}$-BP1s have the same computational power as "ordinary" BP1s. An ordinary BP1 can represent a $\{\neg\}$-BP1, P, with size no more than $2 \times size(P)$ [45].

A relevant result concerning relationships between the complexity classes of the BP1s is restated below [45]. The class of polynomial-sized $\Omega - BP1s$ is denoted by $\mathcal{P}_{\Omega-BP1}$.

Proposition 9.2.1

$$\mathcal{P}_{BP1} \subset \mathcal{P}_{\{\vee\}-BP1}, \mathcal{P}_{\{\wedge\}-BP1}, \mathcal{P}_{\{\oplus\}-BP1}.$$

If we obtain a polynomial-sized upper bound for a BP1, then there will also exist a polynomial-sized $\Omega - BP1$. Likewise, if we obtain an exponential lower bound for the $\Omega - BP1$, the BP1 and ROBDD will be at least as large as the $\Omega - BP1$.

9.3 Upper and Lower Bounds

Two types of bounds on the sizes of the graphs are of interest, the exponential lower bound and the polynomial upper bound.

An exponential lower bound for a computation model shows that independently of any factors (the ordering of decision variables, the function specification, etc.), the size of the model will grow exponentially with the number of inputs to the function. In this case, the exponential growth is inherent to the the model, and unavoidable. A function that exhibits exponential growth as a graph is typically difficult to represent in such a form due to physical memory and time constraints.

A polynomial upper bound shows that the function we are studying can be represented by a graph that is polynomial size in the number of inputs. The growth rate of the graph is linear in the number of inputs. A polynomial-sized representation constitutes an "efficient" representation.

A compilation of some complexity results obtained from various sources is shown in Table 9-1. The first column gives the name of the function. The remaining 5 columns are labeled by the type of graph representation. Each entry is either a lower bound or an upper bound on the size of representation for the type of graph in that column for the particular function in that row.

The upper bounds are upper limits on a minimal-sized graph representation. The lower bounds show that any graph, in that class, that represents the function under consideration must be at least as large as the lower bound. A description of each function in the table, and the result, is given in the following sections.

9.3.1 Hidden Weighted Bit Function

The n-input hidden weighted bit function, denoted by HWB_n, is an example of a function that requires an exponential-sized ROBDD, and a polynomial-sized BP1. A pictorial representation of the function is given in Figure 9-1. The function has n inputs, $(x_1, ..., x_n)$ and a single output labeled HWB_n. The weight of the inputs, $wt(x) = \{$ *the number of ones on the inputs* $\}$, controls the select lines of the multiplexor. The i^{th}

Function	ROBDD	BP1	$\{\vee\} - BP1$	$\{\wedge\} - BP1$	$\{\oplus\} - BP1$
HWB_n	1.14^n	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
$mult_n$	1.09^n				
$cl_{n,n/2}$	$2^{o(n)}$	$2^{n/3-o(n)}$			
$rotate_n$	$2^{n/2}$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
f_n	$2^{\Omega(n)}$	$2^{\Omega(n)}$	$2^{\Omega(n)}$	$O(n^2)$	
$\overline{f_n}$	$2^{\Omega(n)}$	$2^{\Omega(n)}$	$O(n^2)$	$2^{\Omega(n)}$	
$\oplus\ cl_{n,3}$	$2^{\Omega(N)}$	$2^{\Omega(N)}, N = \binom{n}{2}$			$O(n^3)$

Table 9-1: Some lower and upper bounds for special functions

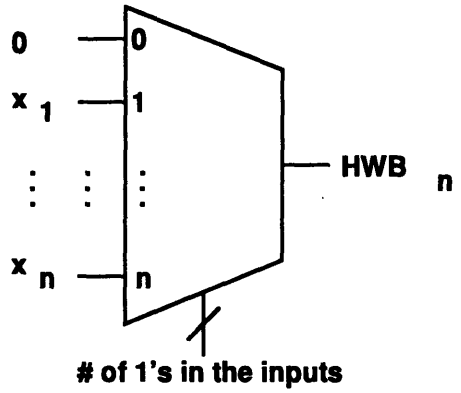


FIGURE 9-1: n-input hidden weighted bit function

input x_i is passed to the output if there are i ones in the inputs.

$$HWB_n(x_1, ..., x_n) = \begin{cases} 0, & wt(x_1, ..., x_n) = 0 \\ x_{wt(x_1,...,x_n)}, & wt(x_1, ..., x_n) > 0 \end{cases} \tag{9.1}$$

The exponential lower bound $\Omega(1.14^n)$ on the ROBDD representation was obtained by using the fooling set argument which partitions the graph into two sections. The amount of communication necessary between the two partitions to evaluate the function determines the lower bound [13]. The $O(n^2)$ upper bound on the BP1 representation for the HWB function [29] is obtained by defining the following functions:

$$F_{i:j} = \begin{cases} 0, & wt(x_i, ..., x_j) = 0 \\ x_{i\ +\ wt(x_i,...,x_j)\ -\ 1}, & wt(x_i, ..., x_j) > 0 \end{cases} \tag{9.2}$$

$$G_{i:j} = \begin{cases} 1, & wt(x_i, ..., x_j) = j - i + 1 \\ x_{i\ +\ wt(x_i,...,x_j)}, & wt(x_i, ..., x_j) < j - i + 1 \end{cases} \tag{9.3}$$

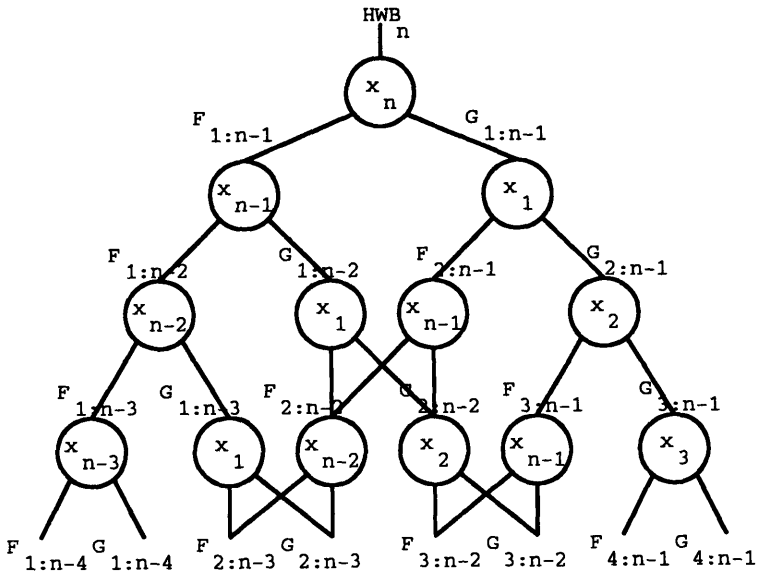


FIGURE 9-2: $O(n^2)$ BP1 for the HWB function

Figure 9-2 shows the BP1 for the HWB function $HWB_n = F_{1:n}$ is constructed recursively from $F_{i:j}$ and $G_{i:j}$ using $F_{i:i} = G_{i:i} = x_i$ and the expansions:

$$F_{i:j} = x_j \cdot G_{i:j-1} + \overline{x_j} \cdot F_{i:j-1} \tag{9.4}$$

$$G_{i:j} = x_i \cdot G_{i+1:j} + \overline{x_i} \cdot F_{i+1:j} \tag{9.5}$$

By using different orderings along different paths in the BP1, enough sharing is possible in the subfunctions for a polynomial-sized BP1.

9.3.2 Integer Multiplier

The $n \times n$ integer multiplier denoted by $mult_n$ has $2n$ inputs $A = \{a_{n-1}, ..., a_0\}$ and $B = \{b_{n-1}, ..., b_0\}$ and $2n-1$ outputs labeled $Y = \{y_{2n-1}, ..., y_0\}$. a_{n-1} and b_{n-1} are the most significant bits, and a_0 and b_0 are the least significant bits. The outputs form the product of the two input integers. An exponential lower bound for ROBDDs that represent multipliers is $\Omega(1.09^n)$ [13].

Gergov was able to show that any linear length deterministic, nondeterministic, co-nondeterministic and MOD_p oblivious ordered BDD that represents integer multiplication must have exponential size [28]. This class includes leveled, read-k general BDDs such as IBDDs [7], and ordered, MOD_2-BP1s [31].

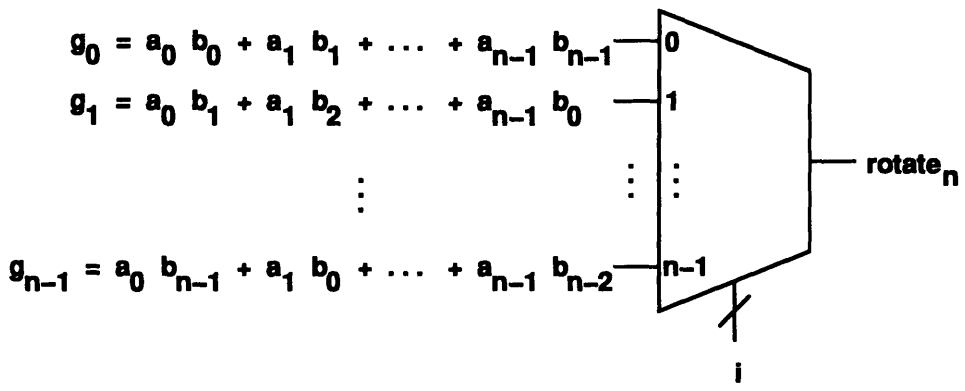


FIGURE 9-3: n-way shift rotate function with $2n + \log n$ inputs

9.3.3 Clique Functions

The clique function $cl_{n,k}$ is defined on $N = \binom{n}{2}$ input variables. The function is defined for an n-node, undirected, graph $G = (V_n, E)$. The variable $x_{i,j} = 1$ if the edge $(i,j) \in E$. Otherwise, $x_{i,j} = 0$. The function $cl_{n,k} = 1$ if there exists a k-clique in the graph G. A clique is a subset of nodes in G such that all pairs of nodes are connected by an edge. In a k-clique, there are k nodes and $\binom{k}{2}$ edges.

Wegener shows exponential lower bounds for the clique function if $k = n/2$ [55]:

$$BP1(cl_{n,n/2}) = \Omega(2^{n/3-o(n)})$$

9.3.4 Shift Rotate Function

The shift rotate function shown in Figure 9-3 has $2n + \lceil log(n) \rceil$ inputs and a single output described by:

$$f = g_i \quad \text{if} \quad value(mux_1, ..., mux_{\lceil log(n) \rceil}) = i \tag{9.6}$$

$$g_i = \sum_{j=0}^{n-1} (a_j \cdot b_{(j+i) \bmod n}) \; ; \quad 0 \leq i < n \tag{9.7}$$

The function uses the select lines to choose between one of n different Achilles Heel functions. Each g_i function requires a different input variable ordering to attain a linear ROBDD. The ROBDD for $rotate_n$ under any global input variable ordering has size $\Omega(2^{n/2})$ [22]. The function has an $O(n^2)$ BP1, obtained by ordering the mux select inputs at the top, and using the appropriate input variable ordering for each subfunction g_i. One ordering that produces a size $2n$ graph for the function g_i is:

$$\{a_0, b_i, a_1, b_{i+1 \bmod n}, ..., a_{n-1}, b_{i+n-1 \bmod n}\}.$$

9.3.5 Permutation Matrix Function

f_n is the permutation matrix function defined for an $n \times n$ matrix A. There are n^2 entries in the matrix A, denoted by $x_{i,j}$, where i corresponds to the row index and j is the column index. The function $f_n(x_{(0,0)} \ldots x_{(n-1,n-1)}) = 1$ only if the matrix A is a permutation matrix, namely it has only one 1 in each row and column, and all other entries are 0.

The permutation function, f_n, has a $2^{\Omega(n)}$ $\{\vee\} - BP1$ representation and an $O(n^2)$ $\{\wedge\} - BP1$ representation. The complement permutation function, $\overline{f_n}$, has a $2^{\Omega(n)}$ $\{\wedge\} - BP1$ representation and an $O(n^2)$ $\{\vee\} - BP1$ representation. The bounds obtained for the permutation function f_n and its complement, $\overline{f_n}$ were proven in [45]. By Proposition 9.2.1, BP1s for the permutation function have exponential size.

9.3.6 Triangle Clique Function

The triangle clique function, $\oplus$ $cl_{n,3}$ is 1 if the number of triangles or 3-node cliques in an undirected graph $G(x)$ is odd. The BP1 exponential lower bound for the triangle clique function was proven by Ajtai et al. [1]. The $\{\oplus\} - BP1$ uses $\oplus$ nodes to compute the function in $O(n^3)$ [45].

Any function that has a two-level exclusive-or sum-of-products form with a polynomial number of product terms has a polynomial $\{\oplus\} - BP1$ representation. The $\{\oplus\} - BP1$ is more powerful than ROBDDs and the exclusive-or sum-of-products form.

9.4 ROBDD Fooling Set Argument

The lower bounds for the integer multiplier, shift rotate and the HWB functions were obtained using the fooling set argument [13]. The technique is a global technique that can be applied to many classes of functions to determine ROBDD complexity. Let f_n be a single output function with inputs X. Define a subset of key inputs $Y \subseteq X$ and a balance parameter $0 < \omega < 1$. The inputs X are partitioned into two sets L and R such that the fraction of Y in L equals ω. Typically, $\omega = 0.5$.

The size of the fooling set is determined by the number of left assignments l to the variables in L such that for two distinct assignments l and l', there exists a right assignment r that distinguishes between l and l'. Namely, $f(l \cdot r) \neq f(l' \cdot r)$. The lemma that appeared in [13] is repeated here:

Lemma 9.4.1 : If for every balanced partition (L,R), the function f_n has a fooling set $A_{OBDD}(L,R)$ with at least c^n elements, $c > 1$, then any OBDD representing f must have $\Omega(c^n)$ vertices.

9.5 FBD Complexity

Let Ω_0 = { AND, NAND, OR, NOR, XOR }, where the function vertices are restricted by the rules defined in the FBD representation. The AND, NAND, XOR vertices have children with disjoint support and the OR, NOR vertices have orthogonal children. Let Ω_1 = { AND, NAND, $\oplus$ } where the $\oplus$ vertex is an unrestricted *exclusive-or*, and the AND and NAND vertices must have children with disjoint supports.

Theorem 9.5.1 : For each $\Omega_0 - FBD$ ($\Omega_1 - FBD$) F that represents the function f, there is a computationally equivalent $\Omega_0 - FBD$ ($\Omega_1 - FBD$) F' that represents the function $\overline{f}$.

Proof: For the $\Omega_0 - FBD$, the complement of an AND(OR) node can be obtained by replacing the AND(OR) vertex with an NAND(NOR) vertex and vice versa. The complement of a decision vertex is obtained by complementing the children. The complement of an XOR vertex is obtained by simply complementing one child because $\overline{a \oplus b} = \overline{a} \oplus b$. The complement of the 1(0) terminal vertex is the 0(1) terminal vertex.

In the $\Omega_1 - FBD$, the same rules apply to AND, NAND, decision and terminal vertices. The complement of an $\oplus$ vertex is obtained by complementing one child.

The rest follows by induction, and the size of the $\Omega_0 - FBD$ ($\Omega_1 - FBD$) that represents the complemented function has size no greater than twice the original $\Omega_0 - FBD$ ($\Omega_1 - FBD$) size.

■

The complement of an ordinary BP1 can be obtained by simply exchanging 1 and 0 terminal vertices. The computation of the complement of an FBD is complicated by the function vertices, but the complement can still be computed by exchanging vertices. An FBD for the complement function with size less than twice the size of the FBD for the uncomplemented function can be constructed.

Theorem 9.5.2 : $\Omega_0 - FBDs$ and { Ω_0, $\neg$ }-$FBDs$ are computationally equivalent. $\Omega_1 - FBDs$ and { Ω_1, $\neg$ }-$FBDs$ are computationally equivalent.

Proof: An FBD without complement attributes, P', can be constructed from an FBD with complement attributes, P, and the new graph will have size less than $2 \times size(P)$. A vertex, v, in P that is used in both complemented and uncomplemented form needs to be replicated. Call the replicated vertex v'. All parents that used v in complemented form will now point to v'. v' is created using the rules stated in the proof of Theorem 9.5.1. ■

Although the complement attribute is not included in Ω_0 or Ω_1, the results for the FBDs are entirely valid for FBDs with the complement attributes because they are computationally equivalent. The FBD without complement attributes will be no more than 2 times larger than the FBD with complement attributes.

Theorem 9.5.3 : An $\Omega_0 - FBD$ can be represented by an $\Omega_1 - FBD$ of the same size, within a constant factor.

Proof: To convert the $\Omega_0 - FBD$, P_0, into an $\Omega_1 - FBD$, P_1, the following procedure can be used. The AND and NAND vertices are equivalent. The OR vertex can be replaced directly with an $\oplus$ because the children must be disjoint. The NOR vertex can be replaced by an $\oplus$ vertex, and by complementing one child. The XOR vertex can simply be replaced by an $\oplus$ vertex. The sizes of the FBDs are different by no more than a constant factor, $size(P_1) \leq 2 \times size(P_0)$. ■

$\Omega_0 - FBDs$ can be represented by $\Omega_1 - FBDs$ with the same size, within a constant factor. The converse, however, is not true because the $\oplus$ vertex is unrestricted and cannot be directly replaced. It is also possible for a function to have a much smaller representation as an $\Omega_1 - FBD$.

Theorem 9.5.4 : Any BP1 can be represented by an $\Omega_0 - FBD$ or $\Omega_1 - FBD$ of the same size, within a constant factor, even under a global input ordering requirement for the $\Omega_0 - FBD$ and $\Omega_1 - FBD$.

Proof: Each decision vertex in a BP1, can be replaced by an OR vertex and two AND vertices to obtain an $\Omega_0 - FBD$ subject to any global input ordering. Likewise, each decision vertex in a BP1 can be replaced by an $\oplus$ vertex and two AND vertices to obtain an $\Omega_1 - FBD$. Figure 9-4 shows a pictorial representation of the transformation.

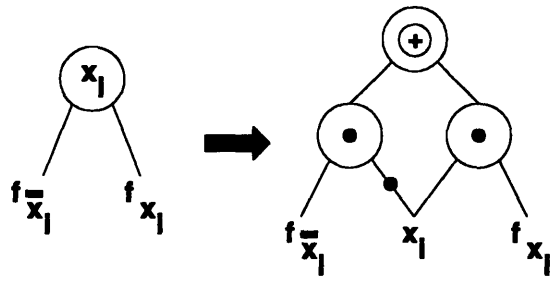


FIGURE 9-4: Transformation of a decision vertex into an OR ($\oplus$) vertex and two AND vertices

The FBD will be valid under the global input ordering, and its size will be no more than 3 times the size of the BP1. If no global input ordering is required for FBD, the FBD is equivalent to the BP1. ■

Theorem 9.5.5 : Any $\{\oplus\} - BP1$ can be represented by an $\Omega_1 - FBD$ of the same size, within a constant factor, even under a global input ordering requirement for the $\Omega_1 - FBD$.

Proof: The $\oplus$ vertices are equivalent. Decision vertices can be replaced by the combination of one $\oplus$ vertex and two AND vertices. This means the $\Omega_1 - FBD$ will be less than three times the size of the $\{\oplus\} - BP1$, and the global input ordering restriction is still satisfied for the $\Omega_1 - FBD$, if present. ■

Theorem 9.5.4 and Theorem 9.5.5 show that $\Omega_1 - FBDs$ with and without a global input ordering restriction are as powerful as $\{\oplus\} - BP1s$, and more powerful than ordinary BP1s. $\Omega_0 - FBDs$ are as powerful as ordinary BP1s.

Let Ω_0' = { AND, OR, NOR }. The NAND and XOR vertices are not necessary, because they can be replaced by AND vertices and OR vertices. Let Ω_1' = { AND, $\oplus$ }. The NAND vertex is replaced by an AND vertex and an $\oplus$ vertex. An $\Omega_1' - FBD$ can represent a function with the same order of complexity as an $\Omega_1 - FBD$.

Theorem 9.5.6 : $\Omega_1' - FBD$ and $\Omega_1 - FBD$ are computationally equivalent.

Proof: An $\Omega_1 - FBD$ can represent an $\Omega_1' - FBD$. An $\Omega_1' - FBD$ can represent an $\Omega_1 - FBD$ by transforming the NAND vertex, v with children $high(v) = b$ and $low(v) = a$, into an $\oplus$ vertex, v_1 and an AND vertex, v_2. $high(v_1) = \overline{b}$ and $low(v_1) = v_2$.

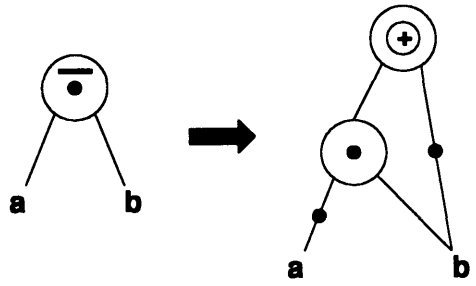


FIGURE 9-5: Transformation for converting a NAND into an AND vertex and an $\oplus$ vertex

The children of the AND vertex are $high(v_2) = b$ and $low(v_2) = \overline{a}$. The complement edges can be eliminated using the rules for decision and $\oplus$ vertices. The transformation is shown in Figure 9-5, and it represents:

$$\overline{a \cdot b} = \overline{a} + \overline{b} = \overline{a} \cdot b + \overline{b} = \overline{a} \cdot b \oplus \overline{b}$$

■

Lemma 9.5.1 : $\Omega_0' - FBD$ and $\Omega_0 - FBD$ are computationally equivalent.

Ordered $\Omega_0 - FBDs$ can represent unordered free BDDs, with graphs no more than 3 times the size of the free BDD. Ordered $\Omega_1 - FBDs$ can represent free BDDs, within the same order of complexity, by using the restricted AND vertices. The support restriction on the AND vertices, necessary for valid signature calculations, limits $\Omega_1 - FBDs$ to the same complexity class as $\{\oplus\} - BP1s$.

Theorem 9.5.7 : $\Omega_1' - FBDs$ and $\{\oplus\} - BP1s$ are computationally equivalent.

Proof: An $\{\oplus\} - BP1$ can be represented by an $\Omega_1' - FBD$, by Theorem 9.5.5 and Theorem 9.5.6. An $\Omega_1' - FBD$ can also be represented by an $\{\oplus\} - BP1$, by bubbling out the AND vertices to obtain an $\{\oplus\} - BP1$.

Given an $\Omega_1' - FBD$, G, start from the terminal vertices, in a levelized fashion, so we never have to bubble an AND vertex through another AND vertex. When an AND vertex, v with $high(v) = b$ and $low(v) = a$, is encountered, the AND vertex can be bubbled down to the terminal vertices using the following rules to account for the possible scenarios.

Without loss of generality, assume we are going to keep b and bubble the AND vertex through the FBD represented by a. If a is a terminal vertex, then either

$$1 \cdot b = b \quad \text{or}$$

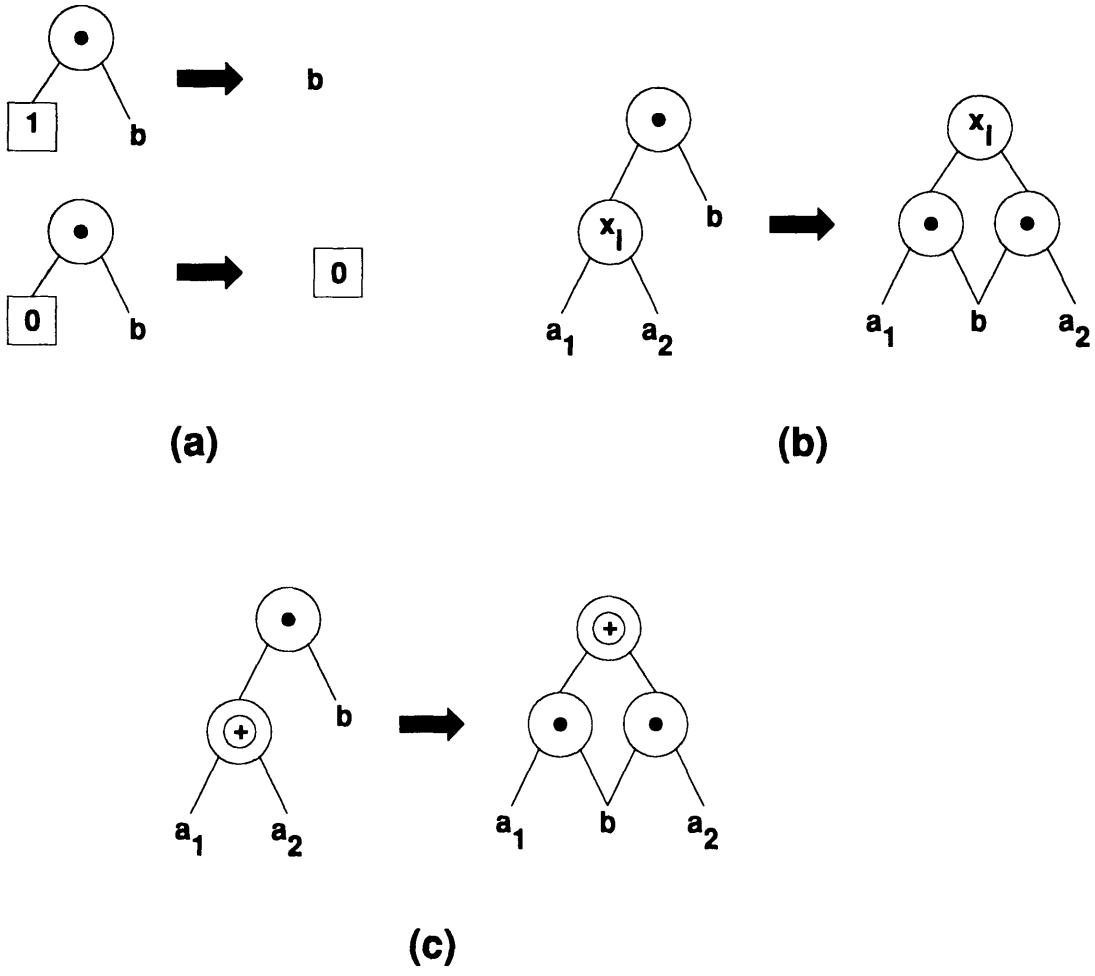


FIGURE 9-6: Rules for bubbling down an AND vertex, based upon the different types of children

$$0 \cdot b = 0.$$

This transformation is depicted in Figure 9-6(a).

Otherwise, a is either an $\oplus$ vertex or a decision vertex. a has high child, a_2, and low child, a_1. If a is a decision vertex, the AND vertex is pushed into the children as follows:

$$b \cdot (\overline{x_i} \cdot a_1 + x_i \cdot a_2) = \overline{x_i} \cdot (a_1 \cdot b) + x_i \cdot (a_2 \cdot b)$$

If a is an $\oplus$ vertex, then we can use the distributive law.

$$b \cdot (a_1 \oplus a_2) = (b \cdot a_1) \oplus (b \cdot a_2)$$

The transformation for the decision vertex is shown in Figure 9-6(b) and for the $\oplus$ vertex is shown in Figure 9-6(c).

After an AND vertex is removed, we go to the next AND vertex, until all AND vertices are gone, and the result is an $\{\oplus\} - BP1$ of the same size, representing the same function. ■

The bounds on the FBD representation of classes of functions will be at least as good as the bounds for ROBDDs and BP1s for those functions. $\Omega_1 - FBDs$ and $\{\oplus\} - BP1s$ are equivalent. In many cases, the actual FBD representation will be smaller than the ROBDD representation, and the FBD representation is more manipulable than the BP1 representations. The $\Omega_1 - FBD$ representation is still in the same class as the $\{\oplus\} - BP1$ representation, because the support restricted AND vertex does not introduce another degree of freedom over unordered variables in the free BDD. The relationships are summarized in Proposition 9.5.1.

Proposition 9.5.1

$$\mathcal{P}_{ROBDD} \subset \mathcal{P}_{BP1} \subset \mathcal{P}_{\Omega_0 - FBD} \subset \mathcal{P}_{\Omega_1 - BP1} = \mathcal{P}_{\{\oplus\} - BP1}.$$

9.6 FBD Time Complexity

9.6.1 FBD Evaluation Complexity

Given an ROBDD R with n decision variables and an input vector $v(x_1, ..., x_n) = \{0,1\}^n$, the time to evaluate the value of the function for the input vector is $O(n)$ because

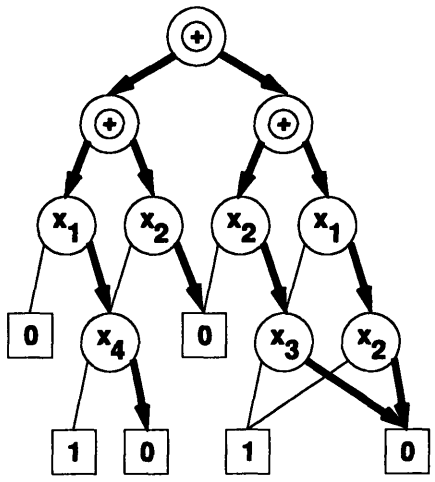


FIGURE 9-7: An FBD that may require the traversal of all nodes for evaluation

$depth(R) \leq n$. At most n decision vertices will be evaluated to determine the value of a function given an input vector.

The time it takes to evaluate an FBD for the same function will vary, and it is not necessarily bounded by n. The depth of an FBD can exceed n and the function vertices make the FBD a non-deterministic BP1. Multiple paths may have to be traversed to evaluate the function. In the worst case, all nodes in the FBD need to be tested, so the complexity of evaluating the value of FBD G is $O(size(G))$.

Figure 9-7 shows an FBD that requires the traversal of all nodes in the FBD to evaluate the function given the input vector $x_1 = 1$, $x_2 = 1$, $x_3 = 1$, and $x_4 = 1$. Both children of the $\oplus$ function vertices need to traversed to determine the value at the function vertex. The paths that are traversed are shown in bold, and we see that all nodes need to be evaluated before the value of the function is determined to be 0.

9.6.2 Cofactor Complexity

The complexity of general FBD cofactor on an FBD of size $size(G)$ is $O(size(G))$. The free BDD general cofactor algorithm is equivalent. A difference between cofactoring free BDDs and FBDs arises when using AND nodes and the FBD length is less than the free BDD length.

Two equivalent graphs are shown in Figure 9-8 and Figure 9-9. Although the FBD in Figure 9-9 is one node larger than the free BDD in Figure 9-8, the cofactor of the FBD with respect to x_4 requires 2 steps and the creation of one node, another AND node. The cofactor of the free BDD with respect to x_4 requires four steps, because we have to traverse 3 nodes before reaching the vertex with decision variable x_4. The result requires

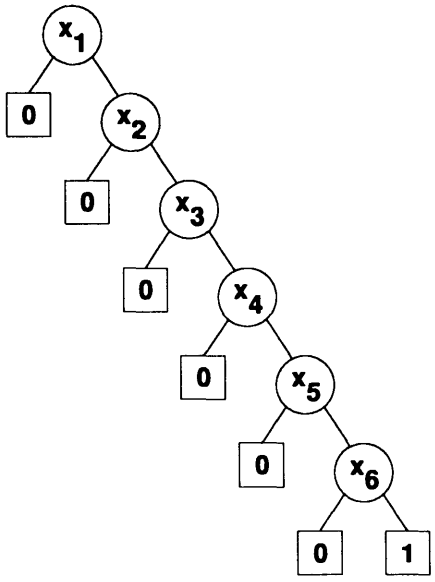


FIGURE 9-8: A free BDD that represents $x_1 \cdot x_2 \cdot x_3 \cdot x_4 \cdot x_5 \cdot x_6$

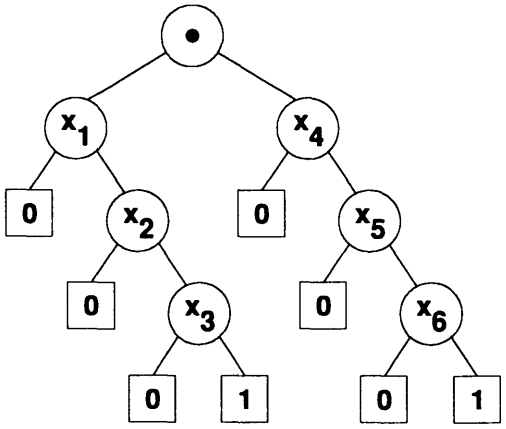


FIGURE 9-9: An FBD that represents $x_1 \cdot x_2 \cdot x_3 \cdot x_4 \cdot x_5 \cdot x_6$

the creation of three nodes.

9.6.3 Signal Probability Evaluation

A disjoint cover of a function, f, can be derived from the paths of an ROBDD representing the function. Given the static signal probabilities of the inputs of f, the static signal probability of f itself can be computed with a single traversal of the ROBDD. The signal probability of a decision vertex v, $prob(v)$, with decision vertex x_i, high child $high(v)$, and low child $low(v)$ is given by

$$prob(v) = prob(x_i) \cdot prob(high(v)) + (1 - prob(x_i)) \cdot prob(low(v)).$$

The signal probability of a free BDD can also be evaluated with a single traversal of the graph, using the same equation. The probability of an $\Omega_0 - FBD$ can also be computed, using the appropriate probability equations for the function vertices. The equations are simply the equations for computing the integer signatures of the function vertices. The computation is valid because of the restrictions on the function vertices.

The signal probability of $\Omega_1 - FBDs$, however, cannot be easily computed from the graph, due to the unrestricted $\oplus$ vertices. The signal probability computation is simply signature calculation using a set of non-negative real numbers between 0 and 1 and real number arithmetic. This field does not have characteristic two, the requirement for $\oplus$ vertices, so the signal probabilities are difficult to calculate. The $\oplus$ vertices have to be removed by performing the *xor* operation, and the $\Omega_1 - FBDs$ have to be converted into $\Omega_0 - FBDs$.

9.7 Conclusion

Some functions, such as the hidden weighted bit function and the shift rotate function, have exponential-sized ROBDDs under any input ordering, but polynomial-sized free BDD and FBD representations. Other functions, such as the triangle clique function, have exponential-sized free BDD representations, but polynomial-sized $\{\oplus\}$−BP1 and FBD representations. Some functions, such as integer multipliers, seem to be difficult for the BP1 representations, but this is still an open problem.

We reviewed bounds on the sizes of ROBDDs, free BDDs, and $\{\oplus\}$−BP1s for certain classes of functions. We showed that these bounds also apply to FBDs because FBDs are as powerful as these BP1s, even with the restricted function vertices and the global input ordering.

10

Conclusion

This thesis presented a graph-based Boolean logic representation called the Free Boolean Diagram (FBD). In addition to the decision vertices and terminal vertices in ROBDDs, FBDs allow for some types of function vertices, and unordered variables along the paths, as long as each variable is seen no more than once along the path. However, function vertices may appear multiple times along the paths.

Since the deterministic equivalence of these graphs is a difficult problem, the representation uses a probabilistic equivalence check to determine FBD equivalence in polynomial time. The type of arithmetic used in the check determines the type of signatures, and the function vertices.

If modular-p integer arithmetic is used, the allowed function vertices are AND, OR, XOR, NAND, and NOR. The AND, XOR, and NAND vertices impose a support restriction on the children: the supports of the children must be disjoint. The OR and NOR vertices must have orthogonal children. The restrictions on the function vertices are necessary to maintain the validity of the signature calculation and the equivalence check.

If modular-p polynomial arithmetic is used, the allowed function vertices are AND, NAND, and $\oplus$. The AND and NAND vertices have children with disjoint supports, but the $\oplus$ vertex computes the *exclusive-or* of the two children without imposing any restrictions on the children.

The probabilistic equivalence check identifies two FBDs equivalent with a finite and bounded probability of error. The error introduced by the check is guaranteed to be less than $\left(\frac{n}{\|S\|}\right)^k$, where n is the number of inputs, $\|S\|$ is the cardinality of the field, and k is the number of passes. The probability that we made an error while creating the final FBDs for a circuit is less than $ut\_hits \times \left(\frac{n}{\|S\|}\right)^k$, where ut_hits is the number of hits to the unique table.

The error bound associated with the probabilistic equivalence check can be adjusted by changing the number of passes or the size of the field used by the equivalence check. However, even with current constraints, the error bound is extremely small. The equivalence check will also become more efficient and the error bound will decrease even more, as the speed of machines increases, and as the datapath width on these machines increases.

A strongly canonical form for FBDs is maintained by allowing the existence of only one FBD representation for each unique signature. The unique signatures in use are stored in the unique table. If two nodes have the same signature, we assume the nodes are equivalent and choose the smaller representation.

The fundamental FBD algorithms necessary for creating FBD representations from multilevel combinational circuits and for manipulating the representations were presented. The FBD algorithms, *complement*, *cofactor*, *apply*, *and* and *xor* were given. We showed that the algorithms work with function vertices and unordered variables. The complexity of the algorithms is related to the size of the FBD representations being manipulated.

The *complement* of an FBD is obtained using complement attributes. The FBD *cofactor* algorithm is a general cofactoring method that handles function vertices and does not require a global input ordering to compute the cofactor. Procedures *apply* and *and* use the general cofactor to make recursive function calls. We also introduce and retain function vertices in the FBDs during these functions. The *xor* of two functions is easily computed using the function vertices.

The FBD representation and manipulation algorithms were implemented in two FBD software packages. We were able to use these packages to create and manipulate FBDs for benchmark circuits. The FBD package mimics the behavior of the ROBDD package, and uses modifications of the ROBDD structures and algorithms.

For the sake of efficiency, the FBDs we created still enforced a global input ordering. However, even with the global ordering, and the restricted function vertices, we were still able to obtain FBDs that were smaller than the ROBDDs for many benchmarks. This demonstrates the power of the FBD representation in the general case. The function vertices in the FBD allow for essentially unordered graphs.

Theoretically, FBDs with function vertices are more powerful than both ROBDDs and free BDDs. We show that any free BDD has a computationally equivalent FBD, even if a global ordering is enforced. Some functions, such as the triangle clique function, have exponential free BDD representations under any ordering, yet still have a polynomial-sized FBD representation.

We successfully applied the FBDs to both combinational logic verification and Boolean

satisfiability. The increased power of the FBD representation over ROBDDs translates into more powerful logic verification methods. For Boolean satisfiability, we were able to obtain significant improvements in the memory required to determine an answer to large satisfiability problems. The memory improvement was obtained by using AND vertices and by exploiting the characteristics of the problem to dynamically determine a free ordering for the FBDs.

Multipliers are still difficult for FBDs and although the complexity of FBDs representing multipliers has not been proven to be exponential, indications are that multipliers will be difficult to represent. Additional work of interest includes work in improving the efficiency of algorithms for determining the satisfiability and equivalence for multiplier-type circuits. Efficient methods for the verification of large multipliers using general BDDs, probabilistic methods, and hierarchical verification is an ongoing area of research.

Future work with FBDs includes the incorporation of dynamic variable re-ordering [48] to modify the global input ordering. FBDs can be applied to sequential logic verification [15, 20]. Additional work can be done with the relaxation of the ordering requirement, and with finding other methods to exploit the power of FBDs during the construction of FBDs from circuits.

A

Special Cases

A.1 Introduction

The appendix enumerates some of the early termination conditions for ITE, *and*, and additional computed table checks. Section A.2 lists the early termination cases checked at the beginning of the ROBDD ITE algorithm. Section A.3.1 names some of the early termination cases encountered during FBD ITE when at least one of the input FBDs has an AND vertex at the root. Section A.3.2 lists the termination cases for ITE calls on FBDs with OR vertices at the root, and Section A.3.3 does the same for FBDs with XOR vertices at the root.

Section A.4 shows the basic termination cases for computing the *and* of two FBDs. The computed table special cases in Section A.5 show functions that can be created quickly from existing computed table results.

A.2 ROBDD ITE Special Cases

$$ITE(1, g, h) = g$$

$$ITE(0, g, h) = h$$

$$ITE(f, 1, 0) = f$$

$$ITE(f, 0, 1) = \overline{f}$$

A.3 FBD ITE Special Cases

A.3.1 AND Vertex Special Cases

$$AND(f,g) \cdot f = AND(f,g)$$

$$AND(f,g) + f = f$$

$$AND(f,g) \oplus f = AND(f,\overline{g})$$

$$AND(f,g) \cdot \overline{f} = 0$$

$$AND(f,g) + \overline{f} = !AND(f,\overline{g})$$

$$AND(f,g) \oplus \overline{f} = !AND(f,\overline{g})$$

$$AND(f,g) \cdot AND(f,h) = AND(f,g \cdot h)$$

$$AND(f,g) \cdot !AND(f,h) = AND(f,g \cdot \overline{h})$$

$$AND(f,g) \cdot AND(\overline{f},h) = 0$$

$$AND(f,g) \cdot !AND(\overline{f},h) = AND(f,g)$$

$$AND(f,g) + AND(f,h) = AND(f,g + h)$$

$$AND(f,g) + !AND(f,h) = !AND(f,\overline{g} \cdot h)$$

$$AND(f,g) + !AND(\overline{f},h) = !AND(\overline{f},h)$$

$$AND(f,g) + !AND(f,h) = !AND(f,\overline{g} \cdot h)$$

$$AND(f,g) + !AND(\overline{f},h) = !AND(\overline{f},h)$$

$$AND(f,g) + AND(f,h) = AND(f,g + h)$$

$$AND(f,g) \oplus AND(f,h) = AND(f,g \oplus h)$$

$$AND(f,g) \oplus !AND(f,h) = !AND(f,g \oplus h)$$

A.3.2 OR Vertex Special Cases

$$OR(f,g) \cdot f = f$$
$$OR(f,g) + f = OR(f,g)$$
$$OR(f,g) \oplus f = \overline{f} \cdot g = g$$
$$OR(f,g) \cdot \overline{f} = \overline{f} \cdot g = g$$
$$OR(f,g) + \overline{f} = 1$$
$$OR(f,g) \oplus \overline{f} = f + \overline{g} = \overline{g}$$
$$OR(f,g) + OR(\overline{f},h) = 1$$
$$OR(f,g) + !OR(\overline{f},h) = OR(f,g)$$

A.3.3 XOR Vertex Special Cases

$$XOR(f,g) \cdot f = AND(f,\overline{g})$$
$$XOR(f,g) + f = !AND(\overline{f},\overline{g})$$
$$XOR(f,g) \oplus f = g$$
$$XOR(f,g) \cdot \overline{f} = AND(\overline{f},g)$$
$$XOR(f,g) + \overline{f} = !AND(f,g)$$
$$XOR(f,g) \oplus \overline{f} = \overline{g}$$
$$XOR(f,g) \oplus XOR(f,h) = g \oplus h$$
$$XOR(f,g) \oplus XOR(\overline{f},h) = !(g \oplus h)$$
$$XOR(f,g) \oplus !XOR(f,h) = !(g \oplus h)$$
$$XOR(f,g) \oplus !XOR(\overline{f},h) = g \oplus h$$

A.4 FBD AND Special Cases

$$0 \cdot g = 0$$
$$1 \cdot g = 1$$
$$f \cdot 0 = 0$$
$$f \cdot 1 = 1$$

A.5 FBD AND Computed Table Special Cases

$$f \cdot g = XOR(g, \overline{f} \cdot g)$$

$$f \cdot g = XOR(f, f \cdot \overline{g})$$

$$f \cdot g = XOR(XOR(f,g), \overline{\overline{f} \cdot \overline{g}})$$

Bibliography

[1] M. Ajtai, L. Babai, P. Hajnal, J. Komlos, P. Pudlak, V. Rodl, E. Szemeredi, and G. Turan. Two lower bounds for branching programs. In *Proceedings of* 18^{th} *ACM STOC*, pages 30–38, 1986.

[2] S. B. Akers. Binary Decision Diagrams. In *IEEE Transactions on Computers*, volume C-27, pages 509–516, June 1978.

[3] P. Ashar, A. Ghosh, and S. Devadas. Boolean Satisfiability and Equivalence Checking Using General Binary Decision Diagrams. *INTEGRATION, the VLSI Journal*, 13:1–16, May 1992.

[4] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and their Applications. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 188–191, November 1993.

[5] C. L. Berman. Ordered Binary Decision Diagrams and Circuit Structure. In *Proceedings of the Int'l Conference on Computer Design: VLSI in Computers*, pages 392–395, October 1989.

[6] J. Bern, J. Gergov, C. Meinel, and A. Slobodová. Boolean Manipulation with Free BDD's. First Experimental Results. In *Technical Report*. University of Trier, Trier, Germany, 1993.

[7] J. Bitner, J. Jain, M. Abadir, J. Abraham, and D. Fussell. Efficient Verification of Multiplier and other Difficult Functions using IBDDs. In *IEEE Custom Integrated Circuits Conference*, pages 5.5.1–5.5.5, May 1992.

[8] 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.

[9] 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.

[10] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[11] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

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

[13] R. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

[14] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[15] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of the* 27^{th} *Design Automation Conference*, pages 46–51, June 1990.

[16] K. Butler, D. Ross, R. Kapur, and R. Mercer. Heuristics to Compute Variable Orderings for Efficient Manipulation of Ordered Binary Decision Diagrams. In *Proceedings of* 28^{th} *Design Automation Conference*, pages 417–420, June 1991.

[17] K. Cho and R. Bryant. Test Pattern Generation for Sequential MOS Circuits by Symbolic Fault Simulation. In *Proceedings of the* 26^{th} *Design Automation Conference*, pages 418–423, June 1989.

[18] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping. In *Proceedings of* 30^{th} *Design Automation Conference*, pages 54–60, June 1993.

[19] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Mass., 1990.

[20] 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.

[21] S. Devadas. Optimal Layout Via Boolean Satisfiability. *Int'l Journal of Computer-Aided VLSI Design*, 2(2):251–262, 1990.

[22] S. Devadas. Comparing Two-Level and Ordered Binary Decision Diagram Representations of Logic Functions. *IEEE Transactions on Computer-Aided Design*, 12(5):722–723, May 1993.

[23] S. Devadas, A. Ghosh, and K. Keutzer. *Logic Synthesis.* McGraw-Hill, New York, 1994.

[24] A. W. Drake. *Fundamentals of Applied Probability Theory.* McGraw-Hill, New York, 1967.

[25] P. E. Dunne. Lower bounds on the complexity of 1-time only branching programs. In *Symposium on Fundamentals of Computation Theory*, pages 90–99, New York, 1985. Springer-Verlag.

[26] 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.

[27] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 2–5, November 1988.

[28] J. Gergov. Time-Space Tradeoffs for Integer Multiplication on Various Types of Input Oblivious Sequential Machines. In *Technical Report, University of Trier, Germany*, 1994.

[29] J. Gergov and C. Meinel. Efficient Boolean Manipulation with OBDD's can be Extended to FBDD's. In *Technical Report 92-10.* University of Trier, Trier, Germany, June 1992.

[30] 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.

[31] 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.

[32] A. Ghosh, S. Devadas, K. Keutzer, and J. White. Estimation of Average Switching Activity in Combinational and Sequential Circuits. In *Proceedings of the 29th Design Automation Conference*, pages 253–259, June 1992.

[33] E. Horowitz and S. Sahni. *Fundamentals of computer algorithms.* Computer Science Press, Potomac, MD, 1978.

[34] N. Ishiura, H. Sawada, and S. Yajima. Minimization of Binary Decision Diagrams Based on Exchanges of Variables. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 472–475, November 1991.

[35] J. Jain, J. Bitner, D. Fussell, and J. Abraham. Probabilistic Design Verification. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 468–471, November 1991.

[36] J. Jain, J. Bitner, D. Fussell, and J. Abraham. Functional Partitioning for Verification and Related Problems. In *Proceedings of the Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems*, pages 210–226, March 1992.

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

[38] S-W. Jeong, B. Plessier, G. Hachtel, and F. Somenzi. Extended BDDs: Trading Off Canonicity for Structure in Verification Algorithms. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 464–467, November 1991.

[39] K. Karplus. Using If-then-else DAGs for Multi-Level Logic Minimization. In C. L. Seitz, editor, *Advanced Research in VLSI*, pages 101–117. MIT Press, 1989.

[40] Y. Lai, M. Pedram, and S. Vrudhula. FGILP: An Integer Linear Program Solver Based on Function Graphs. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 685–689, November 1993.

[41] C. Y. Lee. Representation of Switching Circuits by Binary Decision Programs. In *Bell System Technical Journal*, volume 38, pages 985–999, July 1959.

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

[43] 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.

[44] W. Masek. A fast algorithm for the string editing problem and decision graph complexity. In *MIT S.M. Thesis.* Department of Electrical Engineering and Computer Science, Cambridge, Mass., May 1976.

[45] C. Meinel. *Modified Branching Programs and Their Computational Power.* Springer-Verlag, Berlin, 1989.

[46] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proceedings of the* 27^{th} *Design Automation Conference*, pages 52–57, June 1990.

[47] S. Rudeanu. *Boolean Functions and Equations.* North-Holland, Amsterdam, 1974.

[48] R. Rudell. Dynamic Variable Ordering in Ordered Binary Decision Diagrams . In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 42–47, November 1993.

[49] H. Savoj, R. Brayton, and H. Touati. Extracting Local Don't-Cares for Network Optimization. In *Proceedings of the International Conference on Computer-Aided Design*, pages 514–517, November 1991.

[50] 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.

[51] 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.

[52] A. Shen, S. Devadas, and A. Ghosh. Probabilistic Manipulation of Boolean Functions Using Free Boolean Diagrams. In *Submitted to IEEE Transactions on Computer-Aided Design*, May 1994.

[53] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines Using BDD's. In *Proc. of Int'l Conference on Computer-Aided Design*, pages 130–133, November 1990.

[54] I. Wegener. *The Complexity of Boolean Functions.* B. G. Teubner, Stuttgart, 1987.

[55] I. Wegener. On the Complexity of Branching Programs and Decision Trees for Clique Functions. *Journal of the ACM*, 35(2):461–471, April 1988.