Logic Verification and Synthesis using Function Graphs

Yung-Te Lai

CENG Technical Report 93-47

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

December 1993

Logic Verification and Synthesis using Function Graphs

Yung-Te Lai

Deparment of EE-Systems
University of Southern California
Los Angeles, CA. 90089-2562

December 12, 1993

To my parents,

my wife, and

my daughters Shirley and Katherine

Acknowledgments

I am indebted to my advisors Dr. Sarma Sastry and Dr. Massoud Pedram for their continuous support, constant encouragement, and guidance. Without their help, this research would have not been possible.

I would like to express my thanks to my friends King Ho, C. P. Ravikumar, and Tzyh-Yung Wuu for many fruitful discussions. I have had the good fortune of meeting many wonderful people at USC. In particular, I would like to thank Chihshun Ding, Sasan Iman, Wei-Ming Lin, Bahman Nobandegani, Kuo-Rueih Pan, and Hirendu Vaishnav.

The funding for my research was provided by Powell Foundation Grant for Research in VLSI and National Science Foundation's Research Initiation Awards under contract No. MIP-9111206 and MIP-9211668. I gratefully acknowledge that.

I am grateful to my father and mother who encouraged me to pursue academics. Above all, I would like to thank my wife, Jane, and my daughters, Shirley and Katherine, for their love and support, and for coping with the hardships of graduate student life.

Contents

List Of Tables

List Of Figures

Abstract

As scale of integration in VLSI chips increases, designers are paying more attention to the computational efficiency of the CAD tools. The key to developing efficient algorithms is a concise, yet effective, representation of functions. This function representation must satisfy two important attributes: canonicity and compactness. A canonical representation simplifies the detection of function properties such as unateness and symmetry checking. A compact representation ensures the efficiency of function manipulation.

Ordered Binary-Decision Diagram (OBDD) is a compact, canonical, graphical representation of Boolean functions. OBDDs have been used in many tasks encountered in computer aided design, combinatorial optimization, mathematical logic, and artificial intelligence. While OBDDs are very effective for problems which can be solved through symbolic Boolean manipulation, they are not so effective for those requiring arithmetic operations in the integer domain.

This thesis presents a new data structure called Edge-Valued Binary-Decision Diagram (EVBDD) which can represent and manipulate integer functions more efficiently than the OBDD representation. Because Boolean functions are special cases of arithmetic functions, EVBDDs can also be used to represent Boolean functions. With this property, EVBDDs are particularly useful for applications which require both Boolean and integer operations. This thesis also contains a number of applications which demonstrate the effectiveness of the EVBDDs and OBDDs. These applications include the following: showing the equivalence between a Boolean function and an arithmetic function, solving integer linear programming problems, showing the equivalence between two Boolean functions modulo input/output permutation, and function decomposition for the synthesis of multilevel circuits and look-up table based field programmable gate arrays.

Chapter 1

Introduction

1.1 CAD for VLSI Design

As scale of integration in VLSI chips increases, the need for computer-aided design (CAD) tools that can quickly produce correct and near-optimal designs increases. Efficient data structures and algorithms are the key to developing such tools.

The goal of CAD tools is to automate the transformation from the highest level of abstraction in the behavioral domain to the lowest level in the physical domain. There are five different levels of abstraction: *system*, *algorithmic*, *register-transfer*, *logic*, and *physical design* level. Different levels require different synthesis techniques to improve the design quality and verification tools to ensure the design correctness.

System-level synthesis transforms a system-level specification (protocols and processes) to one or more subsystem descriptions at the algorithmic level. The techniques used are system-level partitioning and behavioral transformations [72]. *Proof theory* and *theorem-prover* are very useful for proving the correctness at this level. For example, an arbiter was proved in [5] by using first-order logic and a microprocessor was proved in [24] by using higher-order logic.

The behavior of the algorithmic level is specified as operations and computation sequences on inputs to produce required outputs. Algorithmic-level synthesis, or high-level synthesis, performs the following tasks: functional units of appropriate types and number have to be selected, operations have to be assigned to time slots, and operations need to be assigned to specific functional units. These tasks are called resource allocation, scheduling, and resource assignment, respectively. Examples of CAD tools for high-level synthesis are ADAM [77], CADDY [45], HAL [81], and OLYMPUS [35]. Verification techniques used in this level are the ones used for proving the correctness of software programs [33, 71].

Finite state systems are the behavioral description for the control path at the register-transfer level. Typical synthesis tasks for controllers include the selection of an appropriate controller architecture, state assignment, input/output encoding, and decomposition of controllers into smaller interacting finite state machines. Examples of synthesis tools at this level include CAPPUCINO [34], MUSTANG [36], and NOVA [98]. CIRCAL [73] used a combination of finite state machines and event models to prove the equivalence of finite state systems. Recently, ordered binary-decision diagrams (OBDDs) [16] has been used to model and prove the equivalence of finite state machines [25].

At the logic level, the behavior is described by Boolean functions. Logic synthesis is often divided into two-level and multilevel. Although the two-level synthesis is simpler, the multilevel synthesis usually produces much more compact, thus economic, designs. The tasks performed in this level are finding minimal covers for two-level logic and extracting common sublogic for multilevel logic. The most successful tools for logic synthesis are ESPRESSO [14] (for two-level) and MIS [15] (for multilevel). Showing the equivalence of Boolean functions is conventionally carried out by tautology checking. Because of the canonical and compact properties of OBDDs, verifying Boolean functions using OBDDs has become popular [70].

At the physical design level, the circuit representation of each component is converted into a geometric representation. The tasks performed at this level include partitioning, floorplanning, placement, routing, and compaction. Examples of layout tools are [62, 100], Bear [30], and TimberWolf [89]. Design verification consists of design rule checking and circuit extraction. Design rule checking is a process which verifies that all geometric patterns meet the design rules imposed by the fabrication process. After design rule checking, functionality of the layout is verified by circuit extraction. Examples of these tools are DRC and EXTRACT commands in MAGIC [67].

1.2 Logic Synthesis

Logic synthesis denotes the problem of transforming a specification to an implementation such that some cost functions are minimized. Logic synthesis usually contains two phases: a *technology-independent* step which manipulates general Boolean functions and a *technology-mapping* step which maps Boolean functions into a set of devices in a specific target technology.

The technology-independent phase in logic synthesis is usually divided into two substeps: logic restructuring which identifies common sublogic to produce a near-optimal structure and logic minimization which optimizes the logic with respect to the structure obtained in the previous step. This thesis only considers the problem of identifying common sublogic.

There are two approaches for identifying common sublogic: algebraic and Boolean methods. The algebraic methods are fast because the logic function is represented and manipulated as an algebraic expression. Some optimality may, however, be lost as Boolean identities are not exploited by the algebraic methods.

The algebraic approach is based on the *division* operation, namely, $f = qd + r$ where q, d, and r are the quotient, divisor, and remainder, respectively. The theory of division was studied by Brayton and McMullen [13] and well developed in the MIS package [15]. The problem thus reduces to that of finding good divisors. Because the number of divisors is huge, usually only a subset of the divisors are used. For examples, [15] uses *kernels* (cube-free primary divisors) and [97] uses double- and single-cubes. Division can be carried out by algebraic [15], coalgebraic [52], and Boolean [15] methods. The division-based methods are directly applicable to both single- and multiple-output functions.

The Boolean approach is based on the *decomposition* operation, namely, $f(X, Y) = f'(g(X), Y)$ where the number of inputs of f' is smaller than that of f. The theory of decomposition was pioneered by Ashenhurst [1], Curtis [27], and Roth and Karp [82]. Because Boolean methods usually take more time to operate, an effective representation and efficient manipulation of functions is the key to the success of these methods. For representing functions, Karnaugh maps, which are not a compact representation, are used in [1, 27, 50] and cubes, which are not a canonical representation, are used in [54, 61, 82].

1.3 Logic Verification

Logic verification denotes the problem of showing the equivalence between a specification of the intended behavior and a description of the implemented design. Based on how circuit behavior is modeled, two logic verification approaches can be identified: *logical* and *functional* approaches. In the logical approach, a circuit is viewed as an entity which 'formulates' a predicate by itself [9, 19, 49]. It takes an $\langle input, output \rangle$ pair and returns a value *true* or *false*. Logical inference is the main technique to show the equivalence relation between two behaviors. The main advantage of this approach is that the *proof theory* and *theorem-provers* are well

developed. The expressive power of *higher order logic* [19, 49] and the capability of modeling timing behavior of *interval temporal logic* [79] also make this approach appealing. Some disadvantages of this approach are: not every description is executable [79], the 'false implies everything' problem [19], and it requires experts in theorem proving (instead of the circuit designer) to setup the proof.

In the functional approach, a circuit is viewed as an entity which 'denotes' a function [6, 10, 16, 25]. Usually, it can process inputs to produce outputs. Thus, it provides the capability of simulation. A typical technique applicable for this approach is *transformation*. That is, transforming from a 'source' form (e.g., a FSM) to a 'target' form (e.g., a reduced FSM) based on a set of identity laws or rewrite rules. Although the basic reasoning technique is restricted to identity substitutions, the loss of proving power is not as much as expected [10]. Moreover, circuits modeled by functions are closer to the way designers think, thus it is easier for them to carry out the proofs themselves.

The correctness of a circuit design can only be proved up to the specification used. For example, if the behavior of a 64-bit adder is specified through 65 Boolean functions (64 bits plus carry), then the behavior of arithmetic addition can never be proved. On the other hand, if the specification language allows to specify the operator '+' directly (e.g., '$x + y$'), then the correctness is up to the arithmetic addition. Although OBDDs have become a popular tool for showing the equivalence of Boolean functions, they cannot directly verify the correctness of arithmetic behavior.

When proving the correctness of circuit designs, we may not know in advance the correspondence of inputs between the specification and the implementation. This usually happens when different tools have been used at various stages which have their own naming conventions. A similar situation also arises in the technology-mapping stage of logic synthesis. In this stage, we want to map parts of a Boolean network (synthesized by technology-independent stage) to library

cells which represent circuits in a specific technology. In these cases, we need to permute the inputs in order to find the correspondence between pins on the cell and inputs to the candidate match and show the equivalence. This is referred as *Boolean matching.* Existing Boolean matching techniques [68, 69, 86] are aimed at technology mapping where a Boolean function is usually assumed to have a small number of inputs (e.g., $n \leq 8$). Furthermore, these techniques deal with only single-output Boolean function. For logic verification, these techniques must be extended to functions with large number of inputs and/or outputs.

1.4 Design Data Representation

One key approach to developing efficient CAD tools is to use an effective representation of functions. An effective function representation must satisfy two important attributes: canonicity and compactness. A canonical representation simplifies the detection of function properties (e.g., equivalence checking); A compact representation provides the efficiency of function manipulation.

Truth tables, Karnaugh maps, sum of minterms, and exclusive sum of minterms (fixed-polarity Reed-Muller expressions) are canonical, but not compact, representations of Boolean functions. Applications based on these representations are only practical for small number of inputs. On the other hand, sum of products, product of sums, and factored Boolean expressions are compact, but not canonical, representations of Boolean functions.

Ordered Binary-Decision Diagram (OBDD) [16] is a graphical representation of Boolean functions which is both canonical and compact. With the canonical property of OBDDs, we can easily detect many Boolean properties such as the number of supporting variables, the unateness of variables, and the symmetry between variables. With the compactness property of OBDDs, we can effectively carry out many Boolean operations. For example, tautology checking and complementation

take constant time while conjunction and disjunction take polynomial time in the size of OBDDs. Although the number of nodes in the OBDD representation may be exponential in the input size, OBDDs have a reasonable size in many practical applications [66].

Edge-Valued Binary-Decision Diagrams (EVBDDs) [28] can represent and manipulate integer functions more efficiently. Because Boolean functions are special cases of arithmetic functions, we can also use EVBDDs to represent Boolean functions. With this property, EVBDDs are particularly useful for applications which require both Boolean and integer operations.

1.5 Integer Linear Programming

Integer linear programming (ILP) is an NP-complete [42] problem which appears in many CAD problems. For examples, [55] used a 0-1 ILP technique to solve the resource constrained scheduling problem in high level synthesis, [39] formulated the problem of the optimization of complex data-paths relative to a repertoire of specified operations as a 0-1 ILP problem, and [21] employed a 0-1 ILP technique to partition input variables for PLA decomposition.

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

1.6 Organization of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 contains the syntax, semantics, operations, and properties of the Edge-Valued Binary-Decision Diagrams (EVBDDs) [28]. Because Boolean functions are special cases of arithmetic functions, we also can use EVBDDs to represent Boolean functions. With this property, EVBDDs are particularly useful for applications which require both Boolean and integer operations. Furthermore, EVBDDs enjoy certain properties such as *additive*, *bounding*, and *domain-reducing* properties, which are not seen in OBDDs. With these properties, we can effectively perform arithmetic operations and branch and bound algorithms by using EVBDDs.

Chapters 3 describes a verification paradigm based on the interpretation of designs [51]. For example, we interpret a design in Boolean function form as implementing an arithmetic function. We then can use EVBDDs to show the equivalence between Boolean functions and arithmetic functions. To cope with conditional statements and array data structure, EVBDD is extended to Structured EVBDDs (SEVBDDs) [28].

Chapter 4 presents an OBDD-based Boolean matching algorithm which works for both logic verification and technology mapping. The algorithm is based on checking whether two OBDDs are isomorphic. In particular, the algorithm is carried out in a *level-first* manner which permits significant pruning of the search space. In addition, a set of *filters* which further improve the the performance of the matching algorithm is presented. It also present a method of analyzing the effectiveness of a filter and rank the various filters based on their *effect/cost* ratio.

The application of EVBDDs to solving ILP is presented in Chapters 5. This approach combines benefits of the EVBDD data structure (in terms of subgraph sharing and caching of computation results) with the state-of-the-art ILP solving techniques. In addition, the construction and conjunction of constraints in terms

of EVBDDs are carried out in a divide and conquer manner in order to manage the space complexity.

The theory and practices of OBDD-based function decomposition are investigated in Chapter 6. This provides the basis for extracting common sublogic from multiple-output functions. The set of decomposition algorithms presented handle disjunctive and nondisjunctive decompositions, decomposition on completely and incompletely specified functions, and decomposition for single- and multiple-output functions.

Conclusions are given in Chapter 7.

Chapter 2

Edge-Valued Binary-Decision Diagrams

This chapter first gives a brief review of OBDDs [16]. It then presents the syntax, semantics, operations, and properties of EVBDDs. It also presents the analysis of computational complexity of EVBDD operations and the comparison of representing Boolean functions by EVBDDs and OBDDS.

2.1 Ordered Binary-Decision Diagrams

Definition 2.1.1 [16] An OBDD is a directed acyclic graph consisting of two types of nodes. A *nonterminal* node $\mathbf{v}$ is represented by a 3-tuple $\langle variable(\mathbf{v}), child_l(\mathbf{v}), child_r(\mathbf{v})\rangle$ where $variable(\mathbf{v}) \in \{x_0, \ldots, x_{n-1}\}$. A terminal node $\mathbf{v}$ is either $\mathbf{0}$ or $\mathbf{1}$. There is no nonterminal node $\mathbf{v}$ such that $child_l(\mathbf{v}) = child_r(\mathbf{v})$, and there are no two nonterminal nodes $\mathbf{u}$ and $\mathbf{v}$ such that $\mathbf{u} = \mathbf{v}$. There exist an index function $index(x) \in \{0, \ldots, n-1\}$ such that for every nonterminal node $\mathbf{v}$, either $child_l(\mathbf{v})$ is a terminal node or $index(variable(\mathbf{v})) < index(variable(child_l(\mathbf{v})))$, and either $child_r(\mathbf{v})$ is a terminal node or $index(variable(\mathbf{v})) < index(variable(child_r(\mathbf{v})))$. The function denoted by $\langle x, \mathbf{v}_l, \mathbf{v}_r\rangle$ is $xf_l + \bar{x}f_r$ where f_l and f_r are the functions denoted by $\mathbf{v}_l$ and $\mathbf{v}_r$, respectively. The functions denoted by $\mathbf{0}$ and $\mathbf{1}$ are the constant function 0 and 1, respectively.

The following notation are used throughout this thesis.

1. The left edge of a node represent 1 or the true edge and the right edge represents 0 or the false edge.

2. $\mathbf{v}$ represents both an OBDD node and the OBDD rooted by node $\mathbf{v}$.

3. $index(\mathbf{v})$: the index of the variable associated with node $\mathbf{v}$. If $\mathbf{v}$ is a terminal node, then $index(\mathbf{v}) = n$.

4. $$l\_child(\mathbf{v}, i) = \begin{cases} child_l(\mathbf{v}) & \text{if } index(\mathbf{v}) = i, \\ \mathbf{v} & \text{otherwise.} \end{cases}$$

   $$r\_child(\mathbf{v}, i) = \begin{cases} child_r(\mathbf{v}) & \text{if } index(\mathbf{v}) = i, \\ \mathbf{v} & \text{otherwise.} \end{cases}$$

5. $new\_obdd(x, \mathbf{l}, \mathbf{r})$ returns an OBDD node $\mathbf{v}$ such that $variable(\mathbf{v}) = x$, $child_l(\mathbf{v}) = \mathbf{l}$, and $child_r(\mathbf{v}) = \mathbf{r}$.

Definition 2.1.2 Given an OBDD node $\mathbf{v}$ representing $f(x_0, \ldots, x_{n-1})$ and a bit vector $\langle b_0, \ldots, b_{i-1} \rangle$, the function *eval* is defined as

$$eval(\mathbf{v}, \langle\rangle) = \mathbf{v},$$
$$eval(\mathbf{v}, \langle b_0, \ldots, b_{i-1} \rangle) = \mathbf{v}',$$

where $\mathbf{v}'$ is the OBDD representing function $f(b_0, \ldots, b_{i-1}, x_i, \ldots, x_{n-1})$. When i is known, $eval(\mathbf{v}, p)$ will be used for $eval(\mathbf{v}, \langle b_0, \ldots, b_{i-1} \rangle)$ where $p = 2^{i-1} b_0 + \ldots + 2^0 b_{i-1}$.

Example 2.1.1 The *carry* and *sum* functions of a full adder represented by OBDDs are shown in Fig. 2.1. From Def. 2.1.1, function *carry* is derived as follows:

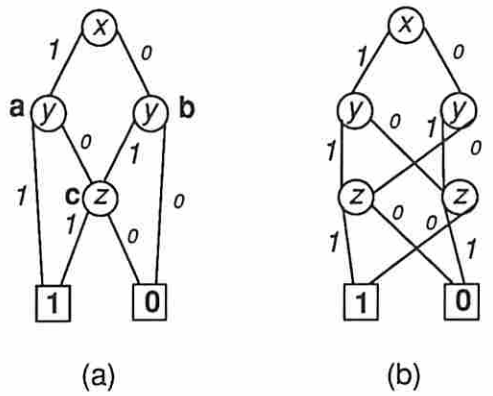


Figure 2.1: OBDD representation of a full adder:(a) carry (b) sum.

$$
\begin{array}{lclcl}
f_{carry} & = & x f_a + \bar{x} f_b & = & xy + x\bar{y}z + \bar{x}yz = xy + xz + yz, \\
f_a & = & y\ 1 + \bar{y} f_c & = & y + \bar{y}z, \\
f_b & = & y f_c + \bar{y}\ 0 & = & yz, \\
f_c & = & z\ 1 + \bar{z}\ 0 & = & z.
\end{array}
$$

From Def. 2.1.2, the evaluation of $x = 1$ in *carry* results node **a**, that is, $carry(1, y, z) = \mathbf{a}$; the evaluation of $x = 1$ and $y = 0$ results node **c**; the evaluation of $x = 1$, $y = 0$, and $z = 1$ results **1**. □

2.2 Definitions

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

f_l and f_r are functions represented by the two children of $\mathbf{v}$. To achieve canonical property, we enforce v_r to be 0.

EVBDDs constructed in the above manner are more related to pseudo Boolean functions [47] which have the function type $\{0,1\}^n \rightarrow integer$. For example, $f(x,y,z) = 3x + 4y - 5xz$ with $x,y,z \in \{0,1\}$ is a pseudo Boolean function, and $f(1,1,0) = 7$ and $f(1,1,1) = 2$. However, for functions with integer variables, we must convert the integer variables to vectors of Boolean variables before using EVBDDs. In the above example, if $x \in \{0,\ldots,5\}$, then $f(x,y,z) = 3(4x_2 + 2x_1 + x_0) + 4y - 5(4x_2 + 2x_1 + x_0)z$ and $f(4,1,1) = -4$.

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

The following definitions describe the syntax and semantics of EVBDDs.

Definition 2.2.1 An EVBDD is a tuple $\langle c, \mathbf{f}\rangle$ where c is a constant value and $\mathbf{f}$ is a directed acyclic graph consisting of two types of nodes:

1. There is a single *terminal node* with value 0 (denoted by $\mathbf{0}$).

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

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

Definition 2.2.2 An EVBDD $\langle c, \mathbf{f} \rangle$ denotes the arithmetic function $c + f$ where f is the function denoted by $\mathbf{f}$. $\mathbf{0}$ denotes the constant function 0, and $\langle x, \mathbf{l}, \mathbf{r}, v \rangle$ denotes the arithmetic function $x(v + l) + (1 - x)r$.

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

Example 2.2.1 Fig. 2.2 shows two arithmetic functions $f_0 = 3 - 4x + 4xy + xz - 2y + yz$ and $f_1 = 4x_0 + 2x_1 + x_2$ represented in EVBDDs. The second function is derived as follows:

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

□

Note that an EVBDD requires only n nonterminal nodes to represent an n variable linear function, for example, Fig. 2.2 (b) is a linear function which can also be interpreted as a 3-bit integer.

Definition 2.2.3 Given an EVBDD $\langle c, \mathbf{f} \rangle$ with variable ordering $x_0 < \ldots < x_{n-1}$, the evaluation of $\langle c, \mathbf{f} \rangle$ with respect to an input pattern $b = \langle b_0, \ldots, b_{i-1} \rangle, 0 \leq i < n$ is defined as follows:

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

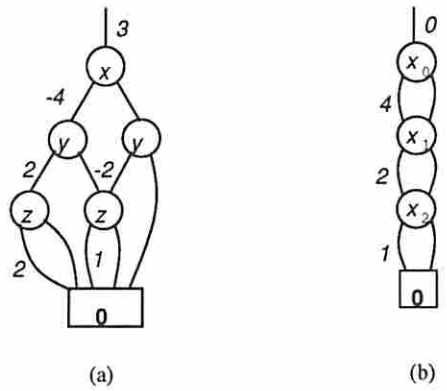


Figure 2.2: Two examples.

From the above definition, function values in an EVBDD representation are obtained by summing edge values (right edge values are always set to 0) along the path associated with the input assignment. For example, in Fig. 2.2 (a), the function value of $x = 1, y = 0$ and $z = 1$ is $3 + (-4) + 0 + 1 = 0$, and the function value of $x_2 = 1, x_1 = 0$ and $x_0 = 1$ in Fig. 2.2 (b) is $0 + 4 + 0 + 1 = 5$.

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

Lemma 2.2.1 Two EVBDDs $\langle c_f, \mathbf{f}\rangle$ and $\langle c_g, \mathbf{g}\rangle$ denote the same function (i.e., $\forall b \in B^n$,
$eval(\langle c_f, \mathbf{f}\rangle, b) = eval(\langle c_g, \mathbf{g}\rangle, b))$, if and only if $c_f = c_g$ and $\mathbf{f}$ and $\mathbf{g}$ are isomorphic.

Proof: Sufficiency: If $c_f = c_g$ and $\mathbf{f}$ and $\mathbf{g}$ are isomorphic, then $\forall b\ eval(\langle c_f, \mathbf{f}\rangle, b) = eval(\langle c_g, \mathbf{g}\rangle, b)$ directly follows from the definitions of isomorphism and *eval*.
Necessity: If $c_f \neq c_g$ then let $b = \langle 0, \ldots, 0\rangle$ be the input assignment to $\mathbf{f}$ and $\mathbf{g}$ (e.g., Fig. 2.3 (a)). We have the following

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

Thus, we focus on the latter condition on $\mathbf{f}$ and $\mathbf{g}$. We want to show if $\mathbf{f}$ and $\mathbf{g}$ are not isomorphic, then $\exists b \in B^n$ such that $eval(\langle 0, \mathbf{f}\rangle, b) \neq eval(\langle 0, \mathbf{g}\rangle, b)$. Without loss of generality, we assume $index(variable(\mathbf{f})) \leq index(variable(\mathbf{g}))$. Let $k = n - index(variable(\mathbf{f}))$, we prove the lemma by induction on k.

Base: When $k = 0$, $\mathbf{f}$ is a terminal node and so is $\mathbf{g}$. Furthermore, $\mathbf{f} = \mathbf{g} = \mathbf{0}$. Thus, $\mathbf{f}$ and $\mathbf{g}$ are isomorphic.

Induction hypothesis: Assume it is true for $n - index(variable(\mathbf{f})) < k$.

Induction: We show that the hypothesis holds for $n - index(variable\mathbf{f}) = k$.

Let $\mathbf{f} = \langle x_{n-k}, \mathbf{f_l}, \mathbf{f_g}, v_f \rangle$.

case 1: $n - index(variable(\mathbf{g})) = k$, that is, $\mathbf{g} = \langle x_{n-k}, \mathbf{g_l}, \mathbf{g_r}, v_g \rangle$.

If $v_f \neq v_g$, let $b = \langle \ldots, 0, 1, 0, \ldots \rangle$, that is, $b_{n-k} = 1$ and $b_i = 0, \forall i \neq n-k$, then $eval(\langle 0, \mathbf{f} \rangle, b) = v_f \neq v_g = eval(\langle 0, \mathbf{g} \rangle, b)$ (e.g., Fig. 2.3 (b)). If $v_f = v_g$, then either $\mathbf{f_l}$ and $\mathbf{g_l}$ are nonisomorphic, or $\mathbf{f_r}$ and $\mathbf{g_r}$ are nonisomorphic.

subcase 1: If $\mathbf{f_l}$ and $\mathbf{g_l}$ are not isomorphic, then from $n - index(variable(\mathbf{f_l})) < k$, $n-$ $index(variable(\mathbf{g_l})) < k$, and induction hypothesis, there exists $b = \langle b_0, \ldots, b_{n-1} \rangle$ such that $eval(\langle 0, \mathbf{f_l} \rangle, b) \neq eval(\langle 0, \mathbf{g_l} \rangle, b)$. Let $b' = \langle b'_0, \ldots, b'_{n-1} \rangle$ such that $b'_{n-k} = 1$ and $b'_i = b_i$ for $i \neq n-k$, then $eval(\langle 0, \mathbf{f} \rangle, b') = eval(\langle v_f, \mathbf{f_l} \rangle, b') \neq eval(\langle v_g, \mathbf{g_l} \rangle, b') = eval(\langle 0, \mathbf{g} \rangle, b')$ (e.g., Fig. 2.3 (c)).

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

case 2: $n - index(variable(\mathbf{g})) < k$.

By definition of reduced EVBDD, we cannot have both ($v_f = 0$) and ($\mathbf{f_l}$ and $\mathbf{f_r}$ are isomorphic). If $v_f \neq 0$, let $b_{n-k} = 1$ and $b_i = 0$ for $i \neq n-k$, then $eval(\langle 0, \mathbf{f} \rangle, b) = v_f \neq 0 = eval(\langle 0, \mathbf{g} \rangle, b)$ (e.g., Fig. 2.3 (d)). (Since $\mathbf{g}$ is independent of the first $n-k$ bits.) Otherwise, $\mathbf{f_l}$ and $\mathbf{f_r}$ are not isomorphic and at least one of them is not isomorphic to $\mathbf{g}$. If $\mathbf{f_l}$ ($\mathbf{f_r}$) and $\mathbf{g}$ are not isomorphic, then by induction hypothesis, there exist a b such that $eval(\langle 0, \mathbf{f_l}(\mathbf{f_r}) \rangle, b) \neq eval(\langle 0, \mathbf{g} \rangle, b)$. Again, let $b'_{n-k} = 1(0)$ if $\mathbf{f_l}(\mathbf{f_r})$ is not isomorphic to $\mathbf{g}$, and $b'_i = b_i$ for $i \neq n-k$, $eval(\langle 0, \mathbf{f} \rangle, b') \neq eval(\langle 0, \mathbf{g} \rangle, b')$. □

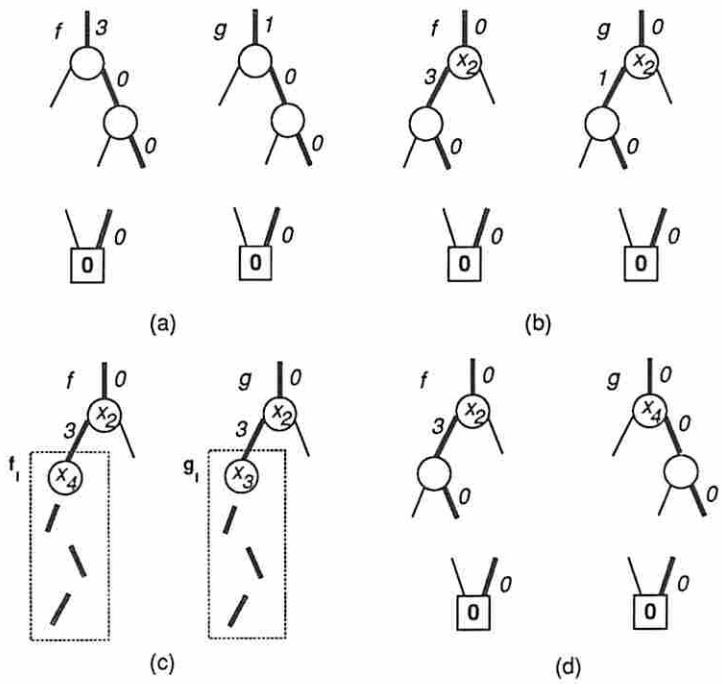


Figure 2.3: Examples for proving canonical property.

2.3 Operations

The following algorithm describes the function (EVBDD) *apply* which takes $\langle c_f, \mathbf{f}\rangle$, $\langle c_g, \mathbf{g}\rangle$, and *op* as arguments and returns $\langle c_h, \mathbf{h}\rangle$ such that $c_h + h \equiv (c_f + f)\ op\ (c_g + g)$ where *op* can be any operator which is closed over the integers.

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

A *comp_table* storing previously computed results is used to achieve computation efficiency. An entry of *comp_table* has the form $\langle f, g, op, h\rangle$ which stands for $f\ op\ g = h$. To compute $f\ op\ g$, we first look up the *comp_table* with key $\langle f, g, op\rangle$, if an entry is found then the last element of the entry h is retrieved as the result; otherwise, we perform *op* on the subgraphs of f and g and store the

result in *comp_table* after the completion of f *op* g. The entries of *comp_table* are used in line 2 and stored in line 18.

After the left and right children have been computed resulting in $\langle c_{h_l}, \mathbf{h_l}\rangle$ and $\langle c_{h_r}, \mathbf{h_r}\rangle$ (lines 14 and 15), if $\langle c_{h_l}, \mathbf{h_l}\rangle = \langle c_{h_r}, \mathbf{h_r}\rangle$, the algorithm returns $\langle c_{h_l}, \mathbf{h_l}\rangle$ to ensure that the case of $\langle x, \mathrm{k}, \mathrm{k}, 0\rangle$ will not occur; otherwise, it returns $\langle c_{h_r}, \langle var, \mathbf{h_l}, \mathbf{h_r}, c_{h_l} - c_{h_r}\rangle\rangle$ to preserve the property of right edge value being 0. There is another table (*uniq_table*) used for the uniqueness property of EVBDD nodes. Before *apply* returns its result, it checks this table through operation *find_or_add* which either adds a new node to the table or returns the node found in the table.

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

Example 2.3.1 An example of $apply(\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, +)$ is shown in Fig. 2.4. Let the variable ordering be $x_0 < x_1$. Fig. 2.4 (a) shows the initial arguments of *apply*; (b) is the recursive call of *apply* on line 14 whose result is (c). Similarly, another call to *apply* on line 15 and its results are shown in (d) and (e). The final result is shown in (f). □

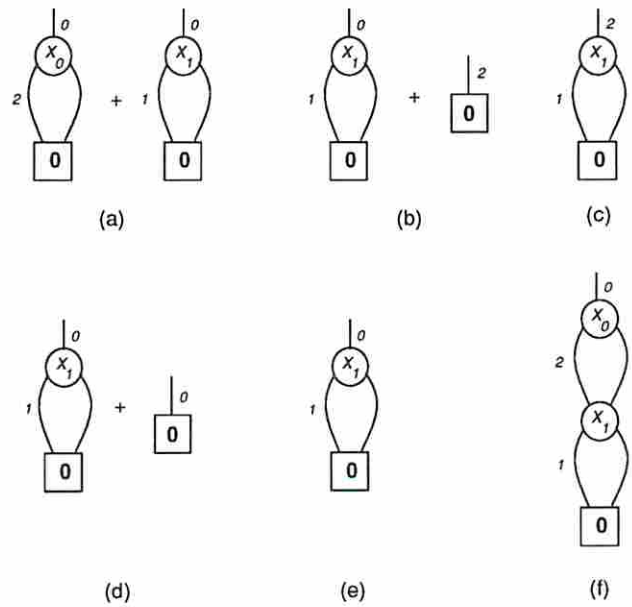


Figure 2.4: Example of the $apply(\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, +)$ operation.

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

```
if (is_commutative(op)) {
    if (index(f) > index(g)||index(f) == index(f) && addr(f) > addr(g))
        swap(⟨c_f, f⟩, ⟨c_g, g⟩);
}
```

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

2.3.1 Complexity Analysis and Flattened EVBDDs

The time complexity of operations in OBDD representation is $O(|\,\mathbf{f}\,| \cdot |\,\mathbf{g}\,|)$ where $|\,\mathbf{f}\,|$ and $|\,\mathbf{g}\,|$ are the number of nodes of OBDDs $\mathbf{f}$ and $\mathbf{g}$. The time complexity of operations in EVBDD representation is however **not** $O(|\,\langle c_f, \mathbf{f}\rangle\,| \cdot |\,\langle c_g, \mathbf{g}\rangle\,|)$ where $|\,\langle c_f, \mathbf{f}\rangle\,|$ and $|\,\langle c_g, \mathbf{g}\rangle\,|$ are the number of nodes of EVBDDs $\langle c_f, \mathbf{f}\rangle$ and $\langle c_g, \mathbf{g}\rangle$. This is because for an internal node $\mathbf{v}$ of $\langle c_f, \mathbf{f}\rangle$ or $\langle c_g, \mathbf{g}\rangle$, *apply* may generate more than one $\langle c_v, \mathbf{v}\rangle$ (lines 4, 5, 11, and 12).

Definition 2.3.1 Given an EVBDD $\langle c_f, \mathbf{f}\rangle$ with variable ordering $x_0 < \ldots < x_{n-1}$ and a node $\mathbf{v}$ of $\mathbf{f}$ with variable x_i, we define the *domain* of $\mathbf{v}$ ($D_{\mathbf{v}}^{eval}$), and the *cardinality* of $\mathbf{v}$ ($\|\,\mathbf{v}\,\|$) as follows:

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

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

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

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

Example 2.3.2 Let $\langle 0, \mathbf{x_0}\rangle$ be the EVBDD in Fig. 2.2.1 (b), then $\|\,\mathbf{x_0}\,\| = 1$, $\|\,\mathbf{x_1}\,| = 2$, $\|\,\mathbf{x_2}\,\| = 4$, $\|\,\mathbf{0}\,\| = 8$, and $\|\,\langle 0, \mathbf{x_0}\rangle\,\| = 15$. The $\langle c_v, \mathbf{v}\rangle$'s for node $\mathbf{x_2}$ are shown in Fig. 2.5. □

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

Definition 2.3.2 A *flattened* EVBDD is a directed acyclic graph consisting of two types of nodes. A *nonterminal* node $\mathbf{v}$ is represented by a 3-tuple $\langle variable(\mathbf{v}), child_l(\mathbf{v}), child_r(\mathbf{v})\rangle$ where $variable(\mathbf{v}) \in \{x_0, \ldots, x_{n-1}\}$. A *terminal* node $\mathbf{v}$ is

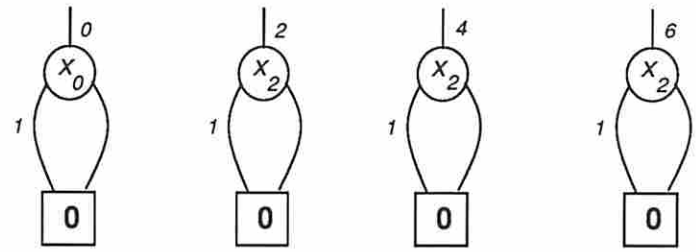


Figure 2.5: The $\langle c_v, \mathbf{v}\rangle$'s of $\mathbf{x_2}$.

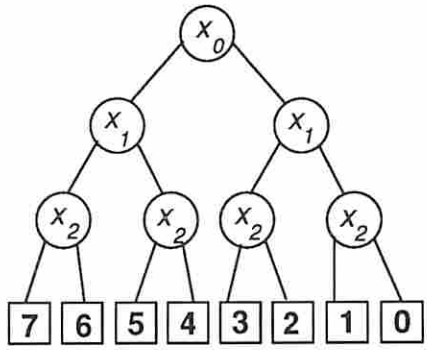


Figure 2.6: An example of flattened EVBDD.

associated with an integer v. *Reduced*, *ordered*, flattened EVBDDs are defined in the same way as OBDDs.

Definition 2.3.3 Given a flattened EVBDD $\mathbf{f}$ with variable ordering $x_0 < \ldots < x_{n-1}$, the evaluation of $\mathbf{f}$ with respect to an input pattern $\langle b_0, \ldots, b_{i-1}\rangle, 0 \leq i < n$ is defined as follows:

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

Example 2.3.3 The flattened EVBDD for the function in Fig. 2.2 (b) is shown in Fig. 2.6. □

From the above definition, flattened EVBDDs are exactly the same as Multi-Terminal OBDDs in [22]. Function values in the flattened EVBDD representation

are obtained in the same way as in the OBDD representation. The flattened EVBDD representation also preserves the canonical property.

Lemma 2.3.1 Two flattened EVBDDs $\mathbf{f}$ and $\mathbf{g}$ denote the same function if and only if they are isomorphic.

Proof: The proof of the canonical property of OBDD representation in [16] can be used to prove the canonical property of flattened EVBDD representation by replacing terminal nodes **0** and **1** by terminal nodes $\mathbf{u}$ and $\mathbf{v}$ where $u \neq v$. □

Lemma 2.3.2 Given a function f represented by an EVBDD $\langle c, \mathbf{f} \rangle$ and a flattened EVBDD $\mathbf{f}'$, $\| \langle c, \mathbf{f} \rangle \| = | \mathbf{f}' |$.

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

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

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

2.3.2 The Additive Property

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

Definition 2.3.4 An operator *op* applied to $\langle c_f, \mathbf{f}\rangle$ and $\langle c_g, \mathbf{g}\rangle$ is said to satisfy the *additive property* if

$$(c_f + f)\ op\ (c_g + g) = (c_f\ op\ c_g) + (f\ op\ g).$$

Examples are $(c_f+f)+(c_g+g)$, $(c_f+f)-(c_g+g)$, $(c_f+f)\times(c+0)$, and $(c_f+f) << (c+0)$ where $<<$ is a left shift operator as in C programming language [63] (i.e., $(c_f + f) \times 2^c$).

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

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

Because the values c_f and c_g can be separated from the functions f and g, the key for this entry in *comp_table* is $\langle\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, -\rangle$. After the computation of $\langle\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, -\rangle$ resulting in $\langle c_h, \mathbf{h}\rangle$, we then add $c_f - c_g$ to c_h to have the complete result of $\langle\langle c_f, \mathbf{f}\rangle, \langle c_g, \mathbf{g}\rangle, -\rangle$. Hence, every operation $\langle\langle c'_f, \mathbf{f}\rangle, \langle c'_g, \mathbf{g}\rangle, -\rangle$ can share the computation result of $\langle\langle 0, \mathbf{f}\rangle, \langle 0, \mathbf{g}\rangle, -\rangle$. This will then increase the hit ratio for caching the computational results. For operators satisfying the additive property, the time complexity of *apply* is $O(|\langle c_f, \mathbf{f}\rangle| \cdot |\langle c_g, \mathbf{g}\rangle|)$ (as opposed to $O(\|\langle c_f, \mathbf{f}\rangle\| \cdot \|\langle c_g, \mathbf{g}\rangle\|)$).

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

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

We also replace lines 2, 16, and 19 of *apply* by the following lines:

$$\begin{array}{ll} 2 & \text{if } (comp\_table\_lookup(\langle c_f, \mathbf{f}\rangle, \langle c_g, \mathbf{g}\rangle, op, \langle c_h, \mathbf{h}\rangle)) \\ & \qquad \text{return } (\langle c_h + c_{fg}, \mathbf{h}\rangle); \\ 16 & \text{if } (\langle c_{h_l}, \mathbf{h_l}\rangle == \langle c_{h_r}, \mathbf{h_r}\rangle) \text{ return } (\langle c_{h_l} + c_{fg}, \mathbf{h_l}\rangle); \\ 19 & \text{return } (\langle c_{h_r} + c_{fg}, \mathbf{h}\rangle); \end{array}$$

For cases of $(c_f + f) \times c$ and $(c_f + f) << c$, we can further separate the processing of edge values. The following pseudo code $times\_c(\langle c_f, \mathbf{f}\rangle, c)$ performs

operation $(c_f + f) \times c$ with time complexity $O(|\mathbf{f}|)$. Note that the new edge value $value(\mathbf{f}) \times c$ is computed in line 5 instead of passing down to the next level in line 3 (cf. line 4 or 11 of *apply*).

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

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

2.3.3 The Bounding Property

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

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

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

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

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

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

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

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

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

Definition 2.3.5 An operator *op* applied to $\langle c_f,\mathbf{f}\rangle$ and $\langle c_g,\mathbf{0}\rangle$ is said to satisfy the *bounding property* if

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

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

As a result, when the maximum or minimum of a function exceeds a boundary value (e.g., c_g in Def. 2.3.5) in an operation, then the result can be determined without further computation.

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

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

2.3.4 The Domain-Reducing Property

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

Definition 2.3.6 Given an EVBDD $\langle c_f, \mathbf{f}\rangle$, the *domain of a node* $\mathbf{v}$ *of* $\mathbf{f}$ *with respect to an operator op* is defined as:

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

Definition 2.3.7 An operator *op* applied to $\langle c_f, \mathbf{f}\rangle$ and $\langle c_g, \mathbf{g}\rangle$ is said to satisfy the *domain-reducing property* if there exist some node $\mathbf{v}$ of $\mathbf{f}$ or $\mathbf{g}$ such that $\mathcal{D}_{\mathbf{v}}^{op} \subset D_{\mathbf{v}}^{eval}$.

An example of this is the following:

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

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

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

Another example is integer division operator with constant divisor:

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

assuming both c_f and c are positive integers for this example. In fact, integer division operator with constant division satisfies the three properties: (c_f/c) satisfies the additive property, $(c_f \ mod \ c)$ satisfies the domain-reducing property, and

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

satisfies the bounding property.

2.4 Representing Boolean Functions

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

$$x \wedge y = xy, \tag{2.1}$$

$$x \vee y = x + y - xy, \tag{2.2}$$

$$x \oplus y = x + y - 2xy, \tag{2.3}$$

$$\overline{x} = 1 - x. \tag{2.4}$$

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

Example 2.4.1 The *sum* and *carry* of a full adder in EVBDD are shown in Fig. 2.7. By using the above equations, we have the following arithmetic functions for *sum* and *carry*:

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

□

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

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

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

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

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

Algorithm A: Converting a Boolean function from OBDD representation to EVBDD representation.

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

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

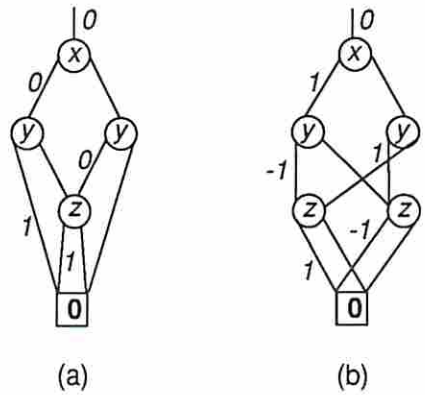


Figure 2.7: A full-adder represented in EVBDDs: (a) *carry* (b) *sum*.

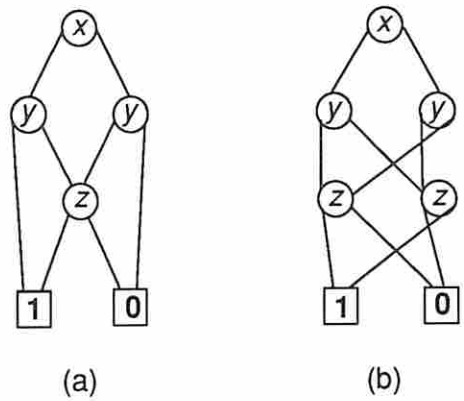


Figure 2.8: A full-adder represented in OBDDs: (a) *carry* (b) *sum*.

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

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

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

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

Example 2.4.2 Fig. 2.8 shows the OBDD representation of *carry* and *sum*. After Algorithm A, they will be converted to the EVBDDs in Fig. 2.7 □

Lemma 2.4.1 Algorithm A converts an OBDD v to either $\langle 0, \mathrm{v}' \rangle$ EVBDD or $\langle 1, \mathrm{v}' \rangle$ EVBDD.

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

Lemma 2.4.2 Algorithm A will neither add nor delete any nonterminal node or edge.

Proof: Directly follows from Algorithm A.

Lemma 2.4.3 Algorithm A preserves functionality. That is, given an OBDD $\mathbf{v}$, if the application of Algorithm A on $\mathbf{v}$ results in an EVBDD $\langle c, \mathbf{v}' \rangle$, then $\mathbf{v}$ and $\langle c, \mathbf{v}' \rangle$ denote the same function.

Proof: Let $\mathbf{v}$ represents a Boolean function with n variables, we prove the lemma by induction on n.

Base: $n = 0$. $\mathbf{v}$ is $\mathbf{0}$ or $\mathbf{1}$ and denotes constant function 0 or 1. From step 1 of Algorithm A, $\langle c, \mathbf{v}' \rangle = \langle 0, \mathbf{0} \rangle$ or $\langle 1, \mathbf{0} \rangle$ and denotes the function $0 + 0 = 0$ or $1 + 0 = 1$.

Induction hypothesis: Assume it is true for $n = 0, \ldots, k-1$.

Induction: Let $\mathbf{v}$ be an OBDD node $\langle x, \mathbf{l}, \mathbf{r} \rangle$ representing the Boolean function $(x \wedge l) \vee (\overline{x} \wedge r)$ where l and r are the functions represented by $\mathbf{l}$ and $\mathbf{r}$ respectively, and each has less than k variables. By induction hypothesis, $\mathbf{l}$ and $\mathbf{r}$ can be converted into $\langle c_l, \mathbf{l}' \rangle$ and $\langle c_r, \mathbf{r}' \rangle$ such that $l = c_l + l'$ and $r = c_r + r'$. From step 2 of Algorithm A, $\mathbf{v}$ will be converted into $\langle c, \mathbf{v}' \rangle$ and we need to show $(x \wedge l) \vee (\overline{x} \wedge r) = c + v'$. In the following, we only prove the correctness of case (b) above, the other three cases can be proved similarly. In case (b), $\langle c_l, \mathbf{l}' \rangle = \langle 0, \mathbf{l}' \rangle$ and $\langle c_r, \mathbf{r}' \rangle = \langle 1, \mathbf{r}' \rangle$. From induction hypothesis, $l = 0 + l' = l'$ and $r = 1 + r'$.

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

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

□

Theorem 2.4.1 Given a Boolean function represented by an OBDD **v** and an EVBDD $\langle c, \mathbf{v}' \rangle$, then **v** and **v'** have the same topology except that the terminal node **1** is absent from the EVBDD **v'** and the edges connected to it are redirected to the terminal node **0**.

Proof: It directly follows from lemmas 2.4.1, 2.4.2, and 2.4.3 and the canonical property of EVBDD representation. □

Lemma 2.4.4 When EVBDDs are used to represent Boolean functions, exactly one of $\langle 0, \mathbf{v} \rangle$ or $\langle 1, \mathbf{v} \rangle$ can be generated during the process of *apply* (lines 4, 5, 9, 13, 14, and 16) where **v** is a nonterminal node.

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

Theorem 2.4.2 Given two OBDDs **f** and **g** and the corresponding EVBDDs $\langle c_f, \mathbf{f}' \rangle$ and $\langle c_g, \mathbf{g}' \rangle$, the time complexity of Boolean operations on EVBDDs (using apply) is $O(|\mathbf{f}| \cdot |\mathbf{g}|)$.

Proof: From lemma 2.4.4, since only one $\langle c, \mathbf{v} \rangle$ can exist for every nonterminal node **v** during the process of *apply*, we have $|\langle c_f, \mathbf{f} \rangle| = \|\langle c_f, \mathbf{f} \rangle\|$ and $|\langle c_g, \mathbf{g} \rangle| = \|\langle c_g, \mathbf{g} \rangle\|$. It is well known that the time complexity of Boolean operations in OBDD

representation is $O(|\mathbf{f}| \cdot |\mathbf{g}|)$ where $|\mathbf{f}|$ and $|\mathbf{g}|$ are the number of nodes in OBDD representation. From theorem 2.4.1, both representations have the same number of nonterminal nodes, thus the complexities are also the same. □

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

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

In the following chapters, several applications of EVBDDs will be presented. For the sake of readability, the flattened form of EVBDDs will also be used. In flattened form, edge values are pushed down to the bottom such that a terminal node is some integer representing the function value.

Chapter 3

Logic Verification

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

When used for logic verification, EVBDDs provide two advantages over OBDDs. First, they allow equivalence checking between Boolean functions and arithmetic functions. Second, they handle hierarchical designs, that is, the implementation of a design can be described using previously verified components rather than having to flatten the design down to the gate level.

This chapter first presents a simple example of how to use EVBDDs to verify the functional behavior of circuit designs and then describes a verification paradigm for proving data paths. In order to verify control paths and do hierarchical verification, EVBDDs are extended to *structured* EVBDDs. Finally, the input variable ordering strategy for logic verification will be discussed. [1]

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

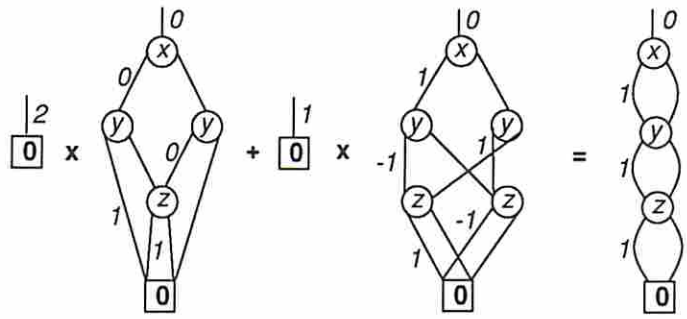


Figure 3.1: EVBDD expression: $2carry + sum$.

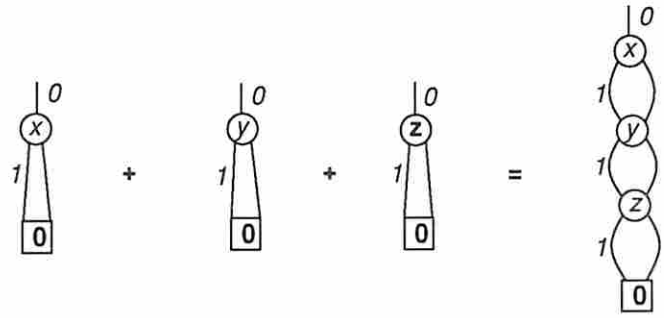


Figure 3.2: EVBDD expression: $x + y + z$.

Example 3.0.3 We prove that $carry(x, y, z)$ and $sum(x, y, z)$ implement the full adder $x + y + z$. That is, with the interpretation of $\langle carry, sum \rangle$ as a 2-bit integer, we show $2carry + sum = x + y + z$. Given a gate-level (Boolean) description of a full adder, it is easy to construct the EVBDD representation of the *carry* and *sum* functions as shown in Fig. 2.7. Carrying out the expression $2carry + sum$ results in the rightmost EVBDD shown in Fig. 3.1. On the other hand, the specification of the arithmetic behavior of the full adder, $x + y + z$, represented in EVBDDs is shown in Fig. 3.2. The equivalence between $2carry + sum$ and $x + y + z$ can then be checked by comparing the two rightmost EVBDDs in Figures 3.1 and 3.2. □

As shown in the above example, the implementation of a design is described by Boolean functions while its behavioral specification is described as an arithmetic function. The equivalence checking between two different levels of abstraction is carried out by using one representation – EVBDD.

3.1 The Verification Paradigm

We are given the following:

1. The description of an implementation:

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

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

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

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

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

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

4. The description of a specification:

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

   where function f specifies the intended behavior of the implementation.

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

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

Using the example in the previous section, we have:

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

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

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

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

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

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

As presented in Sec. 2.3.4, operations modulo and integer division can be effectively carried out in EVBDDs. An application of the above synthesis procedure is in logic verification without variable binding. For example, we can specify a 64-bit adder as '$x + y$' while the variable sets in the implementation are a's and b's. In this case, we first convert the arithmetic expression into a vector of Boolean functions and then use Boolean matching [57] to perform the equivalence checking.

Example 3.1.1 The design (*imp*) is a 64-bit 3-level carry lookahead adder which has 129 inputs, 65 outputs, and 420 logic gates. The intended behavior (*spec*) is specified as:

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

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

□

3.2 Structured EVBDD

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

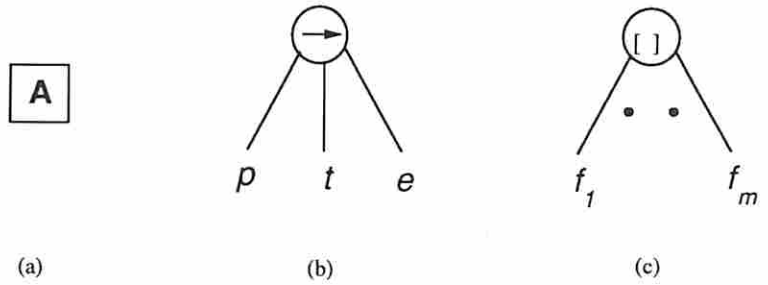


Figure 3.3: Graphical representation of SEVBDDs.

Definition 3.2.1 SEVBDDs are recursively defined as follows:

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

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

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

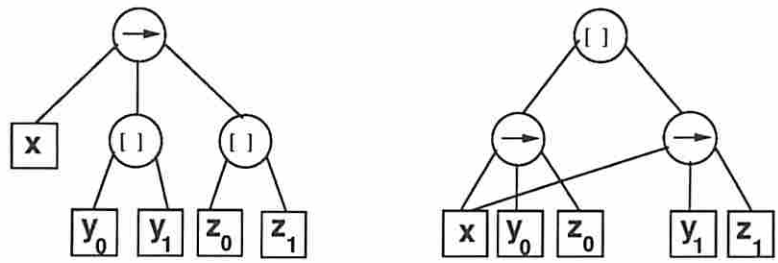
In graphical representation of SEVBDDs, terminal nodes are atomic type SEVBDDs (Fig. 3.3 (a)) and there are two types of nonterminal nodes: a *conditional node* which has three children (Fig. 3.3 (b)) and a *vector node* which has an indefinite number of nodes (Fig. 3.3 (c)).

Example 3.2.1 Let x, y, z, y_0, y_1, z_0, and z_1 be EVBDDs. Consider

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

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

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



(a) (b)

Figure 3.4: Examples of SEVBDDs.

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

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

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

SEVBDDs in groups 1 and 2 are atomic type SEVBDDs; Those in groups 3 and 4 are conditional type SEVBDDs; Those in groups 5 and 6 are vector type SEVBDDs. Note that the SEVBDDs in groups 2 and 3 represent a 1-bit multiplexer while the SEVBDDs in groups 4, 5, and 6 represent two 1-bit multiplexers which have the same control signal x. The graphical representation of those in groups 4 and 5 are shown in Fig. 3.4 (a) and (b), respectively. □

Definition 3.2.2 The *type graph* of an SEVBDD **f** is obtained by replacing all terminal nodes of **f** by a unique terminal node **A**.

Example 3.2.2 The type graphs of the SEVBDDs in Fig. 3.4 are shown in Fig. 3.5.

□

SEVBDD would be a canonical representation if two SEVBDDs denote the same function if and only if they are isomorphic. This is however not true because we can have two SEVBDDs denoting the same function which have different types (e.g.,

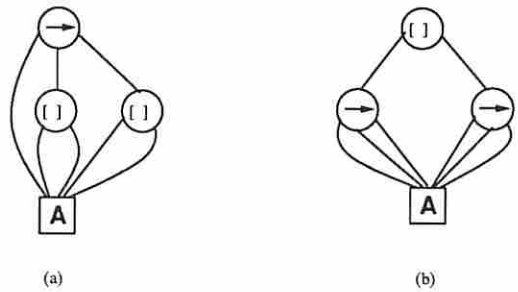


Figure 3.5: Examples of type graph of SEVBDDs.

Ex. 3.2.1). However, with proper restrictions, SEVBDDs can still have the canonical property. That is, if two SEVBDDs satisfy those conditions then they denote the same function if and only if they are isomorphic. In the following, two conditions are defined such that the subset of SEVBDDs which satisfy these conditions have the canonical property.

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

Definition 3.2.3 Two SEVBDDs are *isotypic* if their type graphs are isomorphic. Equivalently, two SEVBDDs f and g are *isotypic* if

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

Example 3.2.3 In Ex. 3.2.1, the SEVBDDs in groups 1 and 2 are isotypic; the SEVBDDs in group 3 are isotypic but none of them is isotypic to that of 4; SEVBDDs in groups 5 and 6 are not isotypic. □

Note that two SEVBDDs which are isotypic but are not isomorphic, may still denote the same function. In Ex. 3.2.1, the SEVBDDs in group 3 are isotypic but are not isomorphic, yet they all denote the same function. Given an SEVBDD $p \rightarrow t; e$,

for any input assignment b such that $p(b) = 1$, the function value of $e(b)$ will not influence the result; similarly, if $p(b) = 0$, then $t(b)$ is irrelevant. Therefore, we can use operators $cofactor_1(p, t)$ and $cofactor_0(p, e)$ to transform t and e to t' and e' such that if $p(b) = 1$, then $t'(b) = t(b)$ and $e'(b) = 0$; if $p(b) = 0$, then $t'(b) = 0$ and $e'(b) = e(b)$. Consequently, we obtain a reduced form $(p \rightarrow t'; e')$ for $p \rightarrow t; e$. Operator $cofactor_1(p, t)$ is carried out in a similar way to the *restrict* operator in [25] except for the following differences: When $p = 0$, *restrict* returns *error* while $cofactor_1$ returns 0; *Restrict* applies to Boolean functions while $cofactor_1$ applies to arithmetic and Boolean functions.

The second condition for SEVBDDs to have the canonical property is to be *reduced* as defined in the following:

Definition 3.2.4 An SEVBDD is *reduced* if

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

In Ex. 3.2.1, the SEVBDDs in groups 1 and 2 are reduced; the last SEVBDD in group 3 and the one in group 6 are also reduced.

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

Definition 3.2.5 The function $level$: SEVBDD $\rightarrow$ *integer* is defined recursively as:

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

In Ex. 3.2.1, the SEVBDDs in groups 1 and 2 have level 0; the SEVBDDs in group 3 have level 1, and the SEVBDDs in groups 4, 5, and 6 have level 2.

Lemma 3.2.1 If two SEVBDDs f and g are isotypic and reduced, then f and g denote the same function if and only if they are isomorphic.

Proof: Necessity: It is trivial to show that if f and g are isomorphic, then they denote the same function.
Sufficiency: If f and g are not isomorphic, then they denote different functions, that is, $\exists b \in B^n$ such that $eval(f, b) \neq eval(g, b)$. This is shown by induction on $level(f)$.
Base: $level(f) = 0$, then both f and g are EVBDDs. This is true from lemma 2.2.1.
Induction hypothesis: Assume it is true for $level(f) < k$.
Induction:
Case 1: $f = (p \rightarrow t_f; e_f)$ and $g = (p \rightarrow t_g; e_g)$.
Since f and g are not isomorphic, then t_f and t_g are not isomorphic and/or e_f and e_g are not isomorphic. If t_f and t_g are not isomorphic then by induction hypothesis there exists b such that $eval(t_f, b) \neq eval(t_g, b)$. Because t_f and t_g are reduced, that is, if $eval(p, b) = 0$ then $eval(t_f, b) = eval(t_g, b) = 0$ by operator $cofactor_1$, we have $eval(p, b) = 1$. Thus, $eval(f, b) = eval(t_f, b) \neq eval(t_g, b) = eval(g, b)$. By a similar reasoning, we can show that the induction step holds true when e_f and e_g are not isomorphic.
Case 2: $f = [f_1, \ldots, f_m]$ and $g = [g_1, \ldots, g_m]$.
There exist $1 \leq i \leq m$ such that f_i and g_i are not isomorphic and from induction hypothesis there exists b such that $eval(f_i, b) \neq eval(g_i, b)$. Then, $eval(f, b) = \langle \ldots, eval(f_i, b), \ldots \rangle \neq \langle \ldots, eval(g_i, b), \ldots \rangle = eval(g, b)$. □

After proving that isotypic and reduced SEVBDDs enjoy the canonical property, we need procedures for converting an SEVBDD from one form to another and/or reducing an SEVBDD. Operator $cofactor_1$ and $cofactor_0$ are used for converting

from atomic (EVBDDs) to conditional form. To convert from conditional to atomic form, we use operator *ite* which is nearly the same as the one described in [12] except that the *ite* operator used here also applies to arithmetic functions. Operator *ite* takes a conditional SEVBDD such as $(p \rightarrow t; e)$ (t and e are EVBDDs) as argument and returns an EVBDD f such that $(p \rightarrow t; e)$ and f denote the same function. The following pseudo codes are for converting the forms of SEVBDDs and reducing SEVBDDs, where $cofactor\_s_1$, $cofactor\_s_0$, and ite_s are SEVBDD versions of $cofactor_1$, $cofactor_0$, and ite, respectively.

```
convert(f, g) /* converting g to the same form as that of f */
/* assuming f and g have the same number of outputs, e.g., both f and g */
/* are atomic form or vector form with the same number of elements */
{
    if (f is an EVBDD)
        if (g is an EVBDD) return(g);
        if (g == (p → t; e)) return(ite(p, convert(f, t), convert(f, e)));
    else if (f == (p → t; e))
        return(p → convert(t, cofactor_s1(p, g)); convert(e, cofactor_s0(p, g)));
    else      /* f = [f1, ..., fm] */
        if (g == (p → t; e))
          return(ite_s(p, convert(f, t), convert(f, e)));
        else return([convert(f1, g1), ..., convert(fm, gm)]);
}
```

```
reduce(f)
{
    if (f is an EVBDD) return(f);
    else if (f == (p → t; e))
        return(reduce(p) → reduce(cofactor_s1(p, t)); reduce(cofactor_s0(p, e)));
    else return([reduce(f1), ..., reduce(fm)]);
}
```

```
cofactor_s1(p, t)
{
    if (t is an EVBDD) return(cofactor1(p, t));
    else if (t == p' → t'; e')
        return(cofactor_s1(p, p') → cofactor_s1(p, t'); cofactor_s1(p, e'));
    else /* t = [t1, ..., tm] */
        return([cofactor_s1(p, t1), ..., cofactor_s1(p, tm)]);
}
```

```
ite_s(p, t, e)      /* assuming t and e are isotypic */
{
    if (p is a conditional SEVBDD) return(ite_s(ite_s(p) → t; e));
    if (t is an EVBDD) return(ite(p, t, e));
    if (t == (pt → tt; et) && e == (pe → te; ee))
        return(ite_s(p, pt, pe) → ite_s(p, tt, te); ite_s(p, et; ee));
    if (t == [t1, ..., tm] && e == [e1, ..., em])
        return([ite_s(p, t1, e1), ..., ite_s(p, tm, em)]);
}
```

To show the equivalence between a specification and an implementation described in two different forms, we need to convert from one form to another. The following experimental results use the specification as the target form. This is because specifications usually have more compact representations than that of implementations have. For example, a specification of '$x \leq y \rightarrow x+y; x-y$' where x and y are n-bit integers, requires $3n$, $2n$, and $2n$ nonterminal nodes for representing $x \leq y$, $x+y$, and $x-y$, respectively. On the other hand, a gate implementation of the above specification requires $n+1$ Boolean functions in which the ith function (for generating i^{th} bit) requires at least $2i$ nonterminal nodes, and the carry function (bit) requires at least $2n$ nonterminal nodes. Thus, it requires at least $n(n+3)$ nonterminal nodes. The following two examples verify SN74L85 and SN74181 chips [94], where the first one is a 4-bit comparator and the second one is a 4-bit ALU.

Example 3.2.4 The implemented design (*imp*) is the SN74L85 chip [94] which is a 4-bit comparator. This chip has 11 inputs, 3 outputs and 33 gates. The specification (*spec*) of the design may be described as:

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

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

Example 3.2.5 The implementation is the SN74181 chip which is a 4-bit ALU [94]. A partial specification is given below. Note: *un_comp*, *two* and *unsigned* perform type coercion. *un_comp* results in an unsigned integer, with the most significant bit being complemented. *two* means that the result is to be a two's complement integer.

```
SN74181(M, S, A, B, Cin)
 unsigned M, Cin;
 unsigned(4) S, A, B;
{
 if (M = 0)
   if (S = 0) return((un_comp (5)) A + (− Cin));
        ⋮
   else if (S = 3) return((two(5)) − Cin);
        ⋮
   else if (S = 6) return((un_comp(5)) A−B−Cin);
        ⋮
 else
   if (S = 0) return((unsigned (4)) not(A));
   else if (S=1) return((unsigned(4)) not(A or B));
        ⋮
}
```

Note that it is allowed that the interpretation of the same outputs to different number systems as well as different sizes in different branches of conditional statements.

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

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

Example 3.2.6 The design is a 64-bit comparator implemented through serial connection of 16 SN74L85s. A net-list description used for this design is as follows:

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

where a net list has the form of : *output_name module_name input_name_list.* The specification of this design is the same as the one in Example 3.2.4 except that the size declaration is changed from 4 to 64. Generation of implementation and specification SEVBDDs take 0.26 and 0.39 seconds respectively, and the proof takes 3.35 seconds. □

Example 3.2.7 The design is a 64-bit ripple-carry adder implemented through serial connection of 16 SN74181s. The specification of this design is exactly the same as the one used in Example 3.1.1. A net-list description used for this design is as follows:

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

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

Time to generate the SEVBDDs for the implementation and specification are 2.09 and 0.16 seconds, respectively and time to verify their equivalence is 0.98 second. Note that generation of implementation SEVBDD takes longer time while verification takes less time than the case in Example 3.1.1. This is because, here, 16 SEVBDDs are generated each with the sum of 4 bits instead of 64 SEVBDDs each with the sum of 1 bit. □

3.3 Ordering Strategy

The conditional type of SEVBDDs provides information for determining the ordering of input variables. For example, for SEVBDD $(p \rightarrow t; e)$, variables occurring in p are assigned lower indices compared to those in t and e. This ordering strategy matches the suggestion (controlling variables should be put on top of OBDDs) in [16]. It is more difficult to identify controlling variables in a Boolean expression. In addition, variables with larger integer coefficients are assigned lower indices compared to those with smaller integer coefficients. This ordering strategy also matches the

observation in [16], and is easier to identify from arithmetic expressions than from Boolean expressions.

Chapter 4

Boolean Matching

The problem of Boolean matching is defined as follows:

> Given two vectors of Boolean functions $\langle f_1(x_0, \ldots, x_{n-1}), \ldots, f_m(x_o, \ldots, x_{n-1})\rangle$ and $\langle g_1(y_0, \ldots, y_{n-1}), \ldots, g_m(y_0, \ldots, y_{n-1})\rangle$, does there exist two bijections $\pi_O : \{f_1, \ldots, f_m\} \rightarrow \{g_1^*, \ldots, g_m^*\}$ and $\pi_I : \{x_0, \ldots, x_{n-1}\} \rightarrow \{y_0^*, \ldots, y_{n-1}^*\}$ such that $f_k(x_0, \ldots, x_{n-1}) = \pi_O(f_k)(\pi_I(x_0), \ldots, \pi_I(x_{n-1}))$, for $1 \leq k \leq m$, where $g_k^* = g_k$ or $\overline{g_k}, 1 \leq k \leq m, y_j^* = y_j$ or $\overline{y_j}, 0 \leq j < n$?

When such bijections do exist, $\langle f_1, \ldots, f_m\rangle$ and $\langle g_1, \ldots, g_m\rangle$ are said to be *input-negation, input-permutation, output-negation*, and *output-permutation equivalent* or *npnp-equivalent* for short.

A straightforward way to determine whether two Boolean functions are npnp-equivalent is to explicitly enumerate the set of all bijections, of which there are $2^m m! 2^n n!$ in number, and perform tautology checking. This is clearly not practical. This chapter presents a matching algorithm based on the isomorphism checking on OBDDs which can be carried out in the time proportional to the sizes of OBDDs.

This chapter first describes the basic algorithm for checking input-permutation equivalence based on two OBDDs and then discusses how it can be extended to

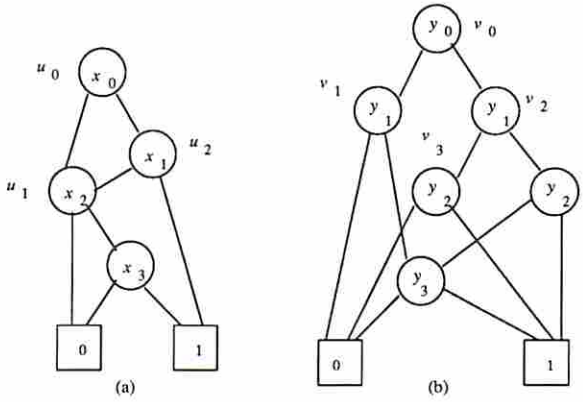


Figure 4.1: A simple example.

check output-permutation, input- and output-negation equivalences. The various filters that speed up the process will be presented in the following sections.

Given two functions f and g, we first construct their OBDDs, $obdd_f$ and $obdd_g$. Then $obdd_g$ is incrementally transformed to the same form as $obdd_f$. The transformation is carried out by permuting a subset of variables of g while performing a check for isomorphism.

To check whether $obdd_f$ and $obdd_g$ are isomorphic, the function graphs are searched in a *level-first* manner, that is, pairs of nodes to be mapped are added to a set $\mathcal{M}$ and elements of this set are processed in the order of the index of the node in $obdd_f$. As an example, consider the OBDDs shown in Fig. 4.1. Using a breadth-first search, the order of processing the elements is $\langle u_0, v_0\rangle, \langle u_1, v_1\rangle, \langle u_2, v_2\rangle, \ldots$ while using a level-first search, the order would be $\langle u_0, v_0\rangle, \langle u_2, v_2\rangle, \langle u_1, v_1\rangle, \ldots$.

At each level i, we perform the following operations. For every $\langle u, v\rangle$ in $\mathcal{M}$ such that $index(u) = i$, we first check $index(v)$. If $index(v) = i$, we then check if $\langle child_l(u), child_l(v)\rangle$ and $\langle child_r(u), child_r(v)\rangle$ will cause *conflicts*. A conflict between two nodes exists if both of them are terminal nodes but have different values, or if one of them is a terminal node while the other one is not. If there is no conflict, then we proceed to the next level. If $index(v) \neq i$ or there is a conflict between $child_l(u)$ and $child_l(v)$ or between $child_r(u)$ and $child_r(v)$, then

we perform the *replace_root* operation as shown in Fig. 4.2. In Fig. 4.2(a), the right child of u and v cause a conflict. Thus, we make y_j to be the new root of the subtree previously rooted at y_i (Fig. 4.2(b)).

After performing *replace_root*, we carry out another check for isomorphism starting at level i for the new permutation $\langle ..., y_{i-1}, y_j, y_i, y_{i+1}, ..., y_{j-1}, y_{j+1}, ..., \rangle$. Note that all the $(n-i-1)!$ permutations of $\langle y_0, \ldots, y_i, *, *, \ldots, * \rangle$ (with $y_0, \ldots, y_i$ fixed in the first $i+1$ positions) have been discarded. If y_j still fails, then we try y_k which has not yet been tried as a new root. This *replace_root* operation can be repeated until no y_l can be the new root, at which point we backtrack to level $i-1$ and perform a *replace_root* operation on y_{i-1}. Proceeding in this manner, an isomorphism is found if we advance to level n, or we conclude that no isomorphism exists when we backtrack to level -1.

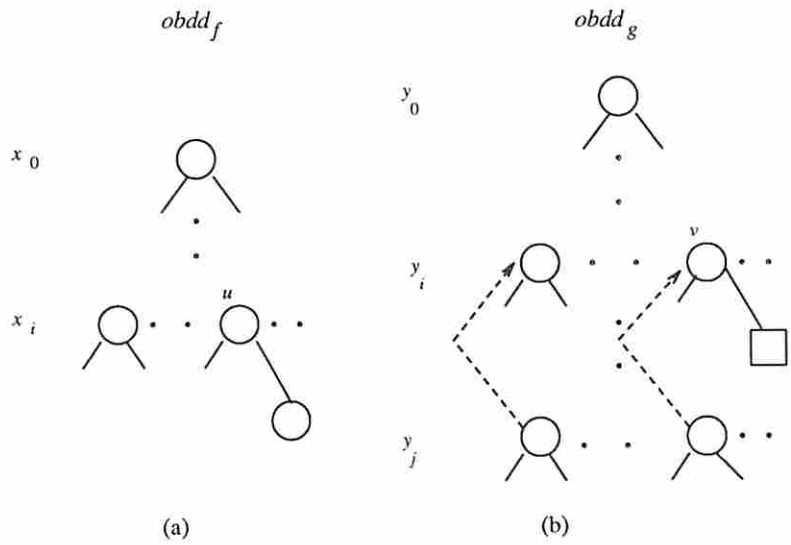


Figure 4.2: The operation of replace_root.

In the following procedure *replace_root*, *cofactor* creates two OBDDs, $obdd_l$ and $obdd_r$, such that the functions h, l and r denoted by $obdd_h, obdd_l$ and $obdd_r$ satisfy $h = zl + \bar{z}r$. The procedure *new_obdd* creates a new OBDD rooted by a node v with $name(v) = z, child_l(v) = obdd_l$, and $child_r(v) = obdd_r$. For every variable z' with

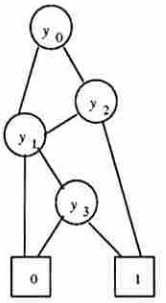


Figure 4.3: The new $obdd_g$ after replace_root.

$index(z) > index(z') \geq index(obdd_h)$, procedure *update_index* changes its index to $index(z') + 1$ and then changes the index of z to $index(obdd_h)$.

```
replace_root(obdd_h, z)
{
   if (index(obdd_h) ≥ index(z)) return(obdd_h);
   else {
      ⟨obdd_l, obdd_r⟩ = cofactor(obdd_h, z);
      obdd_new = new_obdd(z, obdd_l, obdd_r);
      update_index(obdd_new, index(z));
      return(obdd_new);
   }
}
```

Example 4.0.1 As an example of how the above Boolean matching algorithm works, consider the OBDDs shown in Fig. 4.1. Initially, $\mathcal{M}$ consists of $\{\langle u_0, v_0\rangle\}$. At level 0, we first check if $index(v_0) = 0$, and then add $\{\langle u_1, v_1\rangle, \langle u_2, v_2\rangle\}$ to the set $\mathcal{M}$. At level 1, since $index(u_2) = 1$, we check if the index of v_2 is 1, as it is in this case. We then find that $child_r(u_2)$ and $child_r(v_2)$ cause a conflict. Thus we perform *replace_root* operation on variable y_2. After replace_root, we have the new $obdd_g$ as shown in Fig. 4.3 and the steps outlined above are carried out. □

For output-permutation equivalence, two vectors of OBDDs are used for isomorphism checking. To handle input-negation equivalence, the conflict checking of $\langle child_l(u), child_l(v)\rangle$ and $\langle child_r(u), child_r(v)\rangle$ is replaced by the checking of $\langle child_l(u), child_r(v)\rangle$ and $\langle child_r(u), child_l(v)\rangle$. In the basic algorithm, the terminal node with value 1(0) is isomorphic with the terminal node with value 1(0), while in the case of output-negation equivalence checking, the terminal node with value 1(0) is isomorphic with the terminal node with value 0(1).

4.1 Matching Filters

An *output variable filter* is a function $F^{out} : \{f_1, \ldots, f_m\} \rightarrow r$, $r \in R(F^{out})$, where $R(F^{out})$ is some finite set, and f_i is a Boolean function on n variables.

Similarly, an *input variable filter* is a function $F^{in} : \langle f(x_0, \ldots, x_{n-1}), x_i\rangle \rightarrow r$, $x_i \in \{x_0, \ldots, x_{n-1}\}$, $r \in R(F^{in})$ where $R(F^{in})$ is some finite set.

To speed up the task of Boolean matching, we examine a set of output and input variable filters which are derived from the necessary conditions for *npnp*-equivalence, that is, they have the properties that if $F^{out}(f_i) \neq F^{out}(g_j^*)$ then $\langle \ldots, f_i, \ldots\rangle$ and $\langle \ldots, g_j^*, \ldots\rangle$ cannot result in *npnp*-equivalent under $g_j^* = \pi_O(f_i)$, and if $F^{in}(f, x_i) \neq F^{in}(g, y_j^*)$ then $f(\ldots, x_i, \ldots) \neq g(\ldots, y_j^*, \ldots)$ under $y_j^* = \pi_I(x_i)$. An example of a simple filter is the size of the on-set.

All filters presented in this section are computed using the OBDD representation. A filter can be applied to the entire OBDD prior to the execution of Boolean matching, or to the subgraph of an OBDD during the execution of Boolean matching. Clearly, there is a tradeoff between how *expensive* it is to compute a filter and how *effective* it is in pruning the search space. Section 4.1.7 describes a way to measure the effectiveness of a filter and provides a comparison of the various filters based on an *effect/cost* ratio.

4.1.1 Cardinality of Dependence Set

The *dependence* set of a function $f(x_0, \ldots, x_{n-1})$, denoted as $Dep(f)$, is the set $\{x_i \mid \exists$ two assignments $B_1 = \langle \ldots, b_{i-1}, 1, b_{i+1}, \ldots \rangle$ and $B_0 = \langle \ldots, b_{i-1}, 0, b_{i+1}, \ldots \rangle$ such that $f(B_1) \neq f(B_0)\}$.

$F^{out}_{dep}(f) = \mid Dep(f) \mid$ is the cardinality of the dependence set of f with the range $R(F_{dep}) = \{0, 1, \ldots, n\}$. The algorithm for computing the dependence set is shown below. Dep can be computed in $O(m)$ time, where m is the number of nodes of an OBDD.

```
Dep(obdd)
{
  if (index(obdd) == n) return(φ);
  else {
    left = Dep(child_l(obdd));
    right = Dep(child_r(obdd));
    return(left ∪ right ∪ index(obdd));
  }
}
```

F^{out}_{dep} can be used during Boolean matching by applying it to subtrees rooted at nodes that are being compared. It can also be applied to the outputs of a multiple-output circuit. The latter is particularly useful for circuits that implement arithmetic functions where almost every output function results in a unique value of F^{out}_{dep}. An input variable filter based on the dependence set is $F^{in}_{dep}(f, x_i) = true(false)$, if x_i is (is not) in $Dep(f)$.

4.1.2 Cardinality of On-set

The *on-set* of a function $f(x_0, \ldots, x_{n-1})$ is the set of input assignments where $f = 1$. F_{on}^{out} is the cardinality of the on set of f and can be computed in $O(m)$ time and $R(F_{on}^{out}) = \{0, 1, \ldots, 2^n\}$.

```
F_on^out(f)
{
  if (index(f) == n) return(value(f));
  else {
    l = F_on^out(child_l(f)) * 2^(index(child_l(f))-index(f)-1)
    r = F_on^out(child_r(f)) * 2^(index(child_r(f))-index(f)-1)
    return(l + r);
  }
}
```

F_{on}^{out}, as defined above, can be used to reduce the number of possible mappings between the outputs of two multiple-output circuits. It can also be used to reduce the number of mappings among input variables. For this we define a function $F_{on}^{in}(f, x_i) = F_{on}^{out}(f(x_0, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_{n-1}))$. If $F_{on}^{in}(f, x_{i_1}) = \ldots = F_{on}^{in}(f, x_{i_k}) = F_{on}^{in}(g, y_{j_1}) = \ldots = F_{on}^{in}(g, y_{j_k})$, then the set of variables $\{x_{i_1}, \ldots, x_{i_k}\}$ must be mapped to the set $\{y_{j_1}, \ldots, y_{j_k}\}$. This property can be used when attempting to match x_i to y_j during Boolean matching. Furthermore, for two functions $f(x_0, \ldots, x_{n-1})$ and $g(y_0, \ldots, y_{n-1})$ to be p-equivalent, the *multi-sets* $\{F_{on}^{in}(f, x_0), \ldots, F_{on}^{in}(f, x_{n-1})\}$ and $\{F_{on}^{in}(g, y_0), \ldots, F_{on}^{in}(g, y_{n-1})\}$ must be the same. Thus, a new filter $F_{on\_s}^{out}(f)$ can be defined as $\{F_{on}^{in}(f, x_0), \ldots, F_{on}^{in}(f, x_{n-1})\}$.

4.1.3 Sizes of Distance k

Let $D_k(f)$ denote the set of pairs of one-points of $f(x_0, \ldots, x_{n-1})$ whose Hamming distance is k. Then $F_{d_k}^{out} = \mid D_k(f) \mid$ and $R(F_{d_k}^{out}) = \{0, 1, \ldots, C_k^n 2^{n-1}\}$. $F_{d_k}^{out}$

represents a set of filters. $F_{d_k}^{out}$ can be applied to the outputs of a multi-output circuit prior to executing Boolean matching, thereby reducing the number of possible mappings between the outputs of two circuits. $F_{d_k}^{out}$ can also be used while performing Boolean matching. The following algorithms show how to determine $F_{d_n}^{out}$ and $F_{d_{n-1}}^{out}$. In general, the computation of $F_{d_k}^{out}$ can be carried out in the same way as of $F_{d_{n-1}}^{out}$.

```
F^out_dn(obdd)
{
  if (index(obdd) == n) return(0);
  else {
    dn = F^out_on(obdd_and(child_l(obdd), swap(child_r(obdd))));
    if (index(child_l(obdd)) > index(child_r(obdd)))
      return(dn * 2^(index(child_r(obdd)) - index(obdd) - 1));
    else return(dn * 2^(index(child_l(obdd)) - index(obdd) - 1));
  }
}
```

```
swap(obdd)
{
  if (index(obdd) == n) return(obdd);
  else return(new_obdd (variable(obdd), swap(child_r(obdd)),
    swap(child_l(obdd))));
}
```

In $F_{d_n}^{out}$, both F_{on}^{out} and *swap* take $O(m)$ time, but the operation of $obdd\_and(u, v)$ takes $O(m^2)$ time. Thus, the complexity of $F_{d_n}^{out}$ is $O(m^2)$.

```
F^out_{d_{n-1}}(obdd)
{
  n1 = 0;
  for (i = 0; i < n; i++) {
    cofactor(obdd, x_i, obdd_l, obdd_r);
    left = F^out_{d_n}(update_index(obdd_l, x_i));
    right = F^out_{d_n}(update_index(obdd_r, x_i));
    n1 = n1 + left + right;
  }
  return(n1);
}
```

Both $cofactor$ and $update\_index$ take $O(m)$ time while $F^{out}_{d_n}$ takes $O(m^2)$ time, thus the complexity of $F^{out}_{d_{n-1}}$ is $O(nm^2)$. In general, the complexity of computing distance $n-k$ is $O(C^n_k m^2)$. Similar to $F^{in}_{on}(f, x_i)$, we define $F^{in}_{d_k}(f, x_i) = F^{out}_{d_k}(f(\ldots, x_{i-1}, 1, x_{i+1}, \ldots))$.

4.1.4 Unateness of Input Variables

A variable x_i is *monotone increasing* (*monotone decreasing*) in a function f if for all input assignments of $x_j \neq x_i$ $f(\ldots, x_{i-1}, 0, x_{i+1}, \ldots) \leq (\geq) f(\ldots, x_{i-1}, 1, x_{i+1}, \ldots)$. A variable x_i is *binate* in f if x_i is neither monotone increasing nor monotone decreasing.

The unateness of an input variable can be used as a filter $F^{in}_{unate}(f, x_i) \in \{inc, dec, binate\}$. This property can also be used as an output variable filter in the following way. Let $S_{inc}(f) = \{x_i \mid F^{in}_{unate}(f, x_i) = inc\}$. $S_{dec}(f)$ and$S_{binate}(f)$ are defined similarly. Then the cardinalities of these sets define the filter $F^{out}_{unate}(f) = \langle \mid S_{inc} \mid, \mid S_{dec} \mid, \mid S_{binate} \mid \rangle$ with $R(F^{out}_{unate}) = \{\langle n, 0, 0\rangle, \langle n-1, 1, 0\rangle, \langle n-2, 1, 1\rangle, \ldots, \langle 0, 0, n\rangle\}$ and $\mid R(F^{out}_{unate}) \mid = C^{n+2}_n$.

The function $F^{in}_{unate}(f, x_i)$ can be computed in $O(m^2)$ as shown below:

```
F^in_unate(obdd, x_i)
{
  cofactor(obdd, x_i, obdd_l, obdd_r);
  if (obdd_operator(obdd_l, obdd_r, ≥)) return(inc);
  else if (obdd_operator(obdd_l, obdd_r, ≤)) return(dec);
  else return(binate);
}
```

4.1.5 Symmetry Classes of Input Variables

Two variables x_i and x_j are *symmetric* if they can be interchanged without changing the function value, that is, $f(\ldots, x_i, \ldots, x_j, \ldots) = f(\ldots, x_j, \ldots, x_i, \ldots)$. We can determine if x_i and x_j are symmetric by testing if $f(\ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_{j-1}, 1, x_{j+1}, \ldots) = f(\ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_{j-1}, 0, x_{j+1}, \ldots)$. The following algorithm returns *true* if x_i and x_j are symmetric under the function denoted by *obdd*.

```
symmetry(obdd, x_i, x_j)
{
  cofactor(obdd, x_i, obdd_{l_i}, obdd_{r_i});
  cofactor(obdd_{l_i}, x_j, obdd_{l_i l_j}, obdd_{l_i r_j});
  cofactor(obdd_{r_i}, x_j, obdd_{r_i l_j}, obdd_{r_i r_j});
  return(obdd_{l_i r_j} == obdd_{r_i l_j});
}
```

Let $S_{sym}(f)$ be the maximal symmetry classes of f, that is,

$S_{sym}(f) = \{X_1, \ldots, X_k\}, \sum X_i = \{x_0, \ldots, x_{n-1}\}, X_i \cap X_j = \phi$, x_i and x_j are symmetric if and only if $x_i \in X_l$ and $x_j \in X_l$, for some l.

The input and output variable filters based on $S_{sym}(f)$ are

$$\begin{aligned} F^{out}_{sym}(f) &= \{\mid X_1 \mid, \ldots, \mid X_k \mid\} \text{ and} \\ F^{in}_{sym}(f, x_i) &= \mid X_l \mid, \text{ where } x_i \in X_l. \end{aligned}$$

The ranges of these filters are $R(F^{out}_{sym}) = \{\{1, \ldots, 1\}, \{1, \ldots, 1, 2\}, \ldots, \{n\}\}$ and $R(F^{in}_{sym}) = \{1, \ldots, n\}$. The cardinality of $R(F^{out}_{sym})$ is the number of decompositions of the integer n.

4.1.6 Use of Filters

This section explains, by the way of a small example, how the various filters can be used. Consider the following Boolean functions each of which has three inputs a, b, and c, where

$$\begin{aligned} f(a,b,c) &= a\bar{b} + \bar{b}c + \bar{a}b\bar{c}, \\ g_0(a,b,c) &= a\bar{b} + \bar{a}bc, \\ g_1(a,b,c) &= \bar{b}c + b\bar{c}, \\ g_2(a,b,c) &= a\bar{b} + \bar{b}c + \bar{a}c, \\ g_3(a,b,c) &= \bar{a}b + \bar{a}c + a\bar{b}\bar{c}, \text{ and} \\ g_4(a,b,c) &= a\bar{b} + ac + \bar{b}c. \end{aligned}$$

We want to find out which of g_i's are p-equivalent to f. Applying F^{out}_{dep} to the above functions results in

$$F^{out}_{dep} : \langle f, g_0, g_1, g_2, g_3, g_4 \rangle = \langle 3, 3, 2, 3, 3, 3 \rangle,$$

which prunes g_1. Then we apply F^{out}_{on} and get

$$F^{out}_{on} : \langle f, g_0, g_2, g_3, g_4 \rangle = \langle 4, 3, 4, 4, 4 \rangle.$$

This filter prunes function g_0. Next, we apply F_{on}^{in} to the rest of functions:

$$\begin{aligned} F_{on}^{in}(f(a,b,c)) &= \langle 2,1,2\rangle, \\ F_{on}^{in}(g_2) &= \langle 2,1,3\rangle, \\ F_{on}^{in}(g_3) &= \langle 1,2,2\rangle, \text{ and} \\ F_{on}^{in}(g_4) &= \langle 3,1,3\rangle. \end{aligned}$$

By comparing the multi-sets of F_{on}^{in} we know that both g_2 and g_4 cannot be p-equivalent to f. Furthermore, variable b of f can only be mapped by variable a of g_3 because they have the same F_{on}^{in} value. This reduces the number of possible mappings among the inputs between f and g_3 to 2! instead of 3!.

Alternatively, instead of using F_{on}^{in} as an output filter we can use $F_{d_i}^{out}$ to prune g_2 and g_4 because

$$\begin{aligned} F_{d_1}^{out}\langle f,g_2,g_3,g_4\rangle &= \langle 2,3,2,3\rangle, \\ F_{d_2}^{out}\langle f,g_2,g_3,g_4\rangle &= \langle 3,2,3,3\rangle, \text{ and} \\ F_{d_3}^{out}\langle f,g_2,g_3,g_4\rangle &= \langle 1,1,1,0\rangle. \end{aligned}$$

4.1.7 Comparison of Filters

As mentioned earlier, there is a tradeoff between the cost of applying a filter and how effective it is in pruning the search space. To measure a filter's effectiveness we need to examine the distribution of values of the filter over the sample space of all Boolean function of n variables. A filter partitions the set of all 2^{2^n} functions of n variables into equivalence classes, where two functions belong to the same equivalence class if they have the same filter value.

Let $\eta(F,n)$ be the number of equivalence classes formed by F and $N(F,n,k)$ denote the number of Boolean functions f of n variables such that $F(f)=k, 0 \leq k \leq \eta(F,n)$.

Let $P(F,n)$ be the probability that two arbitrarily chosen functions f and g of n variables result in $F(f)=F(g)$. $P(F,n)$ can be used as measure of a filter's *effectiveness*. The condition $P(F_1,n) < P(F_2,n)$ implies that two arbitrary

functions will be less likely to have the same filter value under F_1 than under F_2. Thus, F_1 is more likely than F_2 to declare that the two non-p-equivalent functions are indeed not p-equivalent. Once we know the number of equivalence classes $\eta(F,n)$ and the cardinality of each class, then $P(F,n)$ is easily computed. This is given in the following lemma.

Lemma 4.1.1

$$P(F,n) = \frac{\sum_{k=0}^{\eta(F,n)} N^2(F,n,k) + 2^{2^n}}{(2^{2^n}+1)2^{2^n}} \quad . \tag{4.1}$$

Proof : The proof rests on the following fact. Given a set of m elements, the number of ways of selecting j of them without regard to order and with repetitions allowed is C_j^{m+j-1} (binomial coefficient). Therefore the total number of ways of selecting two functions from the set of all 2^{2^n} functions of n variables without regard to order and with repetitions allowed is $T_n = (2^{2^n}+1)2^{2^n}/2$. Using the same argument, the number of ways of selecting two functions from one of the $\eta(F,n)$ equivalence classes which has $N(F,n,k)$ elements is

$$A_{n,k} = \frac{(N(F,n,k)+1)N(F,n,k)}{2}.$$

Therefore,

$$P(F,n) = \frac{\sum_{k=0}^{\eta(n,k)} A_{n,k}}{T_n}.$$

The result follows since $\sum_k^{\eta(F,n)} N(F,n,k) = 2^{2^n}$. □

As an example, consider F_{on}^{out}. Then $\eta(F_{on}^{out},n) = 2^n$, $N(F_{on}^{out},n,k) = C_k^{2^n}$, and $P(F_{on}^{out},n)$ is given by

$$P(F_{on}^{out},n) = \frac{\begin{pmatrix} 2^{n+1} \\ 2^n \end{pmatrix} + 2^{2^n}}{(2^{2^n}+1)2^{2^n}} \asymp \frac{1}{\sqrt{2^n\pi}}. \tag{4.2}$$

Filter F	Cost	$\lvert R(F)\rvert - 1 = \eta(F,n)$
F_{dep}	$O(m)$	n
F_{on}	$O(m)$	2^n
F_{dk}	$O(C_k^n m^2)$	$C_k^n 2^{n-1}$
F_{unate}	$O(m^2)$	C_n^{n+2}
F_{sym}	$O(n^2 m)$	$O(n^{-1}e^{n^{1/2}})$

Table 4.1: Comparison of filters.

For F_{dep}^{out}, $\eta(F_{dep}^{out}, n) = n$ and $N(F_{dep}^{out}, n, k)$ is given by

$$N(F_{dep}^{out}, n, k) = \binom{n}{k} \sum_{i=0}^{k} (-1)^i \binom{k}{i} 2^{2^{k-i}}. \tag{4.3}$$

It can been shown that $P(F_{on}^{out}, n) < P(F_{dep}^{out}, n)$, which implies that F_{on}^{out} would be more effective than F_{dep}^{out}.

Similarly, we can determine the forms of $P(F, n)$ for each of the filters described in this section. In this way it is possible to order the filters based on increasing values of $P(F, n)$. A complete analysis of the filters is beyond the scope of this section. However, it is possible to get another equivalent but highly simplified measure of a filter's effectiveness. This is simply the cardinality of $R(F)$. For the set of filters discussed in this section, the order obtained using the simplified measure is the same as the one based on $P(F, n)$. Table 4.1.7 shows the *cost* of applying each filter and the cardinality of its range.

Based on the simplistic measure of a filter's effectiveness, we note that

1. F_{on} has the best effect/cost ratio, a property that is supported by experimental results 4.5.

2. Although the effect/cost ratio of F_{dep} is not as good as F_{on} and F_{d_k}, it is very inexpensive to compute. Furthermore, the computation of F_{on} and F_{d_k} requires determining the dependence set.

3. The effect/cost ratio of F_{dk} is moderate for extreme values of k with respect to n.

Note that in the above analysis, each filter is analyzed independently of other filters. A much more accurate analysis is possible if the effect of one filter is taken into account in evaluating another.

4.2 Don't Care Sets

This section presents the effect of Boolean functions with don't cares on Boolean matching. A Boolean function f with don't cares is denoted by $\langle f_{on}, f_{dc} \rangle$ where the on-set of f_{on} is the on-set of f and the on-set of f_{dc} is the don't care set of f. Before extending the matching algorithm, we have the following definitions and lemmas.

Definition 4.2.1 Two Boolean functions f and g are *unifiable*, denoted by $f \cong g$, if $f_{on} \subseteq g_{on} + g_{dc}$ and $g_{on} \subseteq f_{on} + f_{dc}$. That is, there exist some don't care assignments such that f and g can become equal.

Definition 4.2.2 Given two Boolean functions f and g which are unifiable, the *maximum unifier* of f and g, denoted by $f \ \mu \ g$, is is the function derived by the minimum don't care assignment on f and g such that they become equal.

Lemma 4.2.1 If $f \cong g$ then $f \ \mu \ g \equiv \langle h_{on}, h_{dc} \rangle$ where $h_{on} = f_{on} + g_{on}$ and $h_{dc} = f_{dc} g_{dc}$.

Lemma 4.2.2 If $f \cong g$ then $f \ \mu \ g \equiv g \ \mu \ f$.

If $f \cong g \wedge f \cong h \wedge g \cong h$ then $(f \ \mu \ g) \ \mu \ h = f \ \mu \ (g \ \mu \ h)$.

If $f \cong g \wedge (f \ \mu \ g) \ncong h$ then $(f \ncong h \vee g \ncong h)$.

Lemma 4.2.2 has the following important implication: the unification process can be carried out in arbitrary order without backtracking. More specifically, when a

backtrack occurs in the matching algorithm, it is due to a wrong match not the wrong order of unification process.

With the above definitions and lemmas, the extension of the matching algorithm to cover don't care sets is straightforward. A function f now is represented by two OBDDs $obdd_{on}$ and $obdd_{dc}$. The conflict checking in the basic algorithm is extended to check the unifiable property between two nodes. For example, if a node u of f is mapped to two different nodes v and v' of g during the matching process, then we check if v and v' are unifiable. If they cannot be unified, a conflict results; otherwise we have a possible match between u and v'' where v'' is the maximum unifier of v and v'.

Since a Boolean function with don't cares actually represents a set of Boolean functions, a filter of such a function then becomes a set of values instead of a single one. For example, $F_{on}^{out}(\langle f_{on}, f_{dc}\rangle)$ becomes $\{F_{on}^{out}(f_{on}), \ldots, F_{on}^{out}(f_{on}) + F_{on}^{out}(f_{dc})\}$. Consequently, the testing of filter values between two nodes is changed to $r_1 \cap r_2 \neq \phi$, where r_1 and r_2 are sets of filter values.

Note, two variables may be symmetric under one don't care assignment but not symmetric under another assignment. Thus, the *filtering properties* of F_{sym} (and F_{unate}) are lost when don't cares are present.

4.3 Filters Based on Spectral Coefficients

This section presents a set of filters based on spectral coefficients. The main purpose of spectral methods [99] is to transform Boolean functions from Boolean domain into another domain so that the transformed functions have more compact implementations. It was conjectured that these methods would provide a unified approach to the synthesis of analog and digital circuits [96]. Although spectral techniques have solid theoretical foundation, until recently they did not receive much attention due to their expensive computation times. With new applications

in fault diagnosis, spectral techniques have recently invoked interest [53]. New computational methods have been proposed. In [40], a technique based on arrays of disjoint ON- and DC-cubes is proposed. In [96], a cube-based algorithm for linear decomposition in spectral domain is proposed.

Recently, [23] proposed two OBDD-based methods for computing spectral coefficients. The first method was to treat integers as bit vectors and integer operations as the corresponding Boolean operations. The main disadvantage of this representation is that arithmetic operations must be performed bit by bit which is very time consuming. The second method employed a variation of OBDD called Multi-Terminal Binary Decision Diagrams (MTBDDs) [22] which are exactly the same as the flattened form of EVBDDs. The major problem with using MTBDDs is the space requirement when the number of distinct coefficients is large.

The work on spectral techniques is motivated by the following observations. First, it is well-known that OBDD-based algorithms have outperformed cube-based algorithms in many logic synthesis and verification applications. Second, spectral methods are based on orthogonal transform techniques in which arithmetic operations and integer domain are used rather than Boolean operations and Boolean domain.

This section presents an EVBDD-based algorithms for computing Hadamard (sometimes termed Walsh-Hadamard) spectrum [99]. In this approach, the matrix representing Boolean function values used in spectral methods is represented by EVBDDs. This takes advantage of compact representation through subgraph sharing. The transformation matrix and the transformation itself are carried out through EVBDD operations. Thus, the benefit of caching computational results is achieved. The algorithms presented here include both the transformation from Boolean domain to spectral domain and the operations within the spectral domain itself.

The Hadamard transformation is carried out in the following form:

$$T^n \, Z^n \;=\; R^n, \tag{4.4}$$

where T^n is a $2^n \times 2^n$ matrix called *transformation matrix*, Z^n is a $2^n \times 1$ matrix which is the truth table representation of a Boolean function, and R^n is a $2^n \times 1$ matrix which is the spectral coefficients of a Boolean function.

Different transformation matrices generate different spectra. In this section, we use the Hadamard transformation matrix [99] which has a recursive structure as follows:

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

and

$$T^0 \;=\; 1.$$

Example 4.3.1 The spectrum of function $f(x, y) = x \oplus y$ is computed as

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

The *order* of each spectral coefficient r_i (i^{th} row of R^n) is the number of 1's in the binary representation of $i, 0 \leq i \leq 2^n - 1$. For example, r_{00} is the zeroth-order coefficient, r_{01} and r_{10} are the first-order coefficients, and r_{11} is the second-order coefficient. Let $R_i^n = \{$ multi-set of the absolute value of r_k's where r_k is an ith-order coefficient of $R^n\}$, $0 \leq i \leq n$. In Example 4.3.1, $R_0^2 = \{2\}$, $R_1^2 = \{0, 0\}$, and $R_2^2 = \{2\}$. An operation on f and its R^n which does not modify the sets R_i^n is referred as an *invariance* operation. Given a function $f(x_1, \ldots, x_n)$ with spectrum

R^n, three invariance operations on f and R^n are as follows (formal proofs may be found in [38]):

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

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

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

Lemma 4.3.1 Two Boolean functions are input-negation, input-permutation, and output-negation equivalent (npn-equivalent) only if their R_i^n's are equivalent.

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

From Lemma 4.3.1, we have a new set of filters: R_i^n. The following shows how to use EVBDDs to represent Boolean functions in spectral domains. The resulting data structure is called an SPBDD. The function value of an input pattern $b_0 \ldots b_{n-1}$ in SPBDDs is an element of R_i^n if the pattern contains i 1's.

4.3.1 Spectral EVBDD (SPBDD)

The major problem with Equation 4.4 is that all matrices involved are of size 2^n. Therefore, only functions with a small number of inputs can be computed. To overcome this difficulty, we use EVBDDs to represent both Z^n and R^n. When an EVBDD is used to represent R^n, it is referred as SPectral EVBDD, or SPBDD for

short. The difference between EVBDDs and SPBDDs is in the semantics, not the syntax. A path in EVBDDs corresponds to a function value while a path in SPBDDs corresponds to a spectral coefficient. The matrix multiplication by T^n is implicitly carried out in the transformation from Z^n to R^n (i.e., from EVBDD to SPBDD).

Z^n and R^n are recursively defined as follows:

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

and

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

Then, Equation 4.4 can be rewritten as:

$$\begin{bmatrix} T^{n-1} & T^{n-1} \\ T^{n-1} & -T^{n-1} \end{bmatrix} \begin{bmatrix} Z_0^{n-1} \\ Z_1^{n-1} \end{bmatrix} = \begin{bmatrix} T^{n-1}Z_0^{n-1} + T^{n-1}Z_1^{n-1} \\ T^{n-1}Z_0^{n-1} - T^{n-1}Z_1^{n-1} \end{bmatrix} = \begin{bmatrix} R_0^{n-1} \\ R_1^{n-1} \end{bmatrix} \quad (4.5)$$

Equation 4.5 then is implemented through EVBDD [1] as :

$$\tau(\langle x, Z_1^{n-1}, Z_0^{n-1}\rangle) = \langle x, \tau(Z_0^{n-1}) - \tau(Z_1^{n-1}), \tau(Z_0^{n-1}) + \tau(Z_1^{n-1})\rangle, \quad (4.6)$$

$$\tau(1) = 1, \quad (4.7)$$

$$\tau(0) = 0. \quad (4.8)$$

where τ is the transformation function which converts an EVBDD representing a Boolean function to an SPBDD. To show the above equations correctly implement Equation 4.5, we prove the following lemma.

[1] For the sake of readability, flattened EVBDD is used in this section.

Lemma 4.3.2 Let τ : EVBDD $\rightarrow$ SPBDD as defined in Equations 4.6-4.8, then τ implements T, that is, $\tau(f^n) = T^n f^n$, where f^n is an n-input function. Or, equivalently, $\tau(\langle x_n, Z_1^{n-1}, Z_0^{n-1}\rangle) = \langle x_n, R_1^{n-1}, R_0^{n-1}\rangle$.

Proof: By induction on input size n.

Base: $n = 0$, $f^0 = c$ is a constant function, $c \in \{0, 1\}$,

$\tau(c) = c$ by Equations 4.7 and 4.8,

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

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

Induction:

$$\begin{array}{lll} & \tau(\langle x^k, Z_1^{k-1}, Z_0^{k-1}\rangle) & \\ = & \langle x^k, \tau(Z_0^{k-1}) - \tau(Z_1^{k-1}), \tau(Z_0^{k-1}) + \tau(Z_1^{k-1})\rangle & \text{by Equation 4.6} \\ = & \langle x^k, T^{k-1}Z_0^{k-1} - T^{k-1}Z_1^{k-1}, T^{k-1}Z_0^{k-1} + T^{k-1}Z_1^{k-1}\rangle & \text{by IH} \\ = & \langle x^k, R_1^{k-1}, R_0^{k-1}\rangle & \text{by Equation 4.5} \end{array}$$

□

Example 4.3.2 The exclusive-or function in Example 4.3.1 is redone in terms of EVBDD representation.

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

□

Pseudo code *evbdd_to_spbdd*$(ev, level, n)$ is the implementation of Equation 4.5. Because of the following situation, this procedure requires *level* and n as parameters:

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

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

```
evbdd_to_spbdd(ev, level, n)
{
1     if (level == n) return ev;
2     if (ev == 0) return 0;
3     if (index(ev) > level) {
4         sp = evbdd_to_spbdd(ev, level + 1, n);
5         left = 0;
6         right = evbdd_add(sp, sp)
7         return new_evbdd(level, left, right);
8     }
9     sp_l = evbdd_to_spbdd(child_l(ev), level + 1, n);
10    sp_r = evbdd_to_spbdd(child_r(ev), level + 1, n);
11    left = evbdd_sub(sp_r, sp_l);
12    right = evbdd_add(sp_r, sp_l);
13    return new_evbdd(level, left, right);
}
```

4.3.2 Boolean Operations in Spectral Domain

This section shows how to perform Boolean operations in SPBDDs. It first presents the algorithm for performing Boolean conjunction in SPBDDs by the following definition.

Definition 4.3.1 Given two SPBDDs f and g, the operator $\wedge$ is carried out in the following way:

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

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

Lemma 4.3.3 $(f+g) \wedge (i+j) = f \wedge i + f \wedge j + g \wedge i + g \wedge j$, where $f, g, i, j \in$ SPBDD. (Note that $+$'s may be replaced by $-$'s.)

Proof: by induction on the size of inputs.

Base: $n = 0$, f, g, i, and j are constant functions.

$$
\begin{aligned}
& (f+g) \wedge (i+j) && \text{from LHS} \\
= \; & (f+g) \times (i+j) && \text{by Def. 4.3.1} \\
= \; & f \times i + f \times j + g \times i + g \times j && \text{by distributive laws of } \times \text{ and } +
\end{aligned}
$$

$$
\begin{aligned}
& f \wedge i + f \wedge j + g \wedge i + g \wedge j && \text{from RHS} \\
= \; & f \times i + f \times j + g \times i + g \times j && \text{by Def. 4.3.1}
\end{aligned}
$$

Induction hypothesis (IH): assume it is true for $0 \leq n < k$.

Induction: let $f = \langle x_k, f_l, f_r \rangle$, $g = \langle x_k, g_l, g_r \rangle$, $i = \langle x_k, i_l, i_r \rangle$, and $j = \langle x_k, j_l, j_r \rangle$, where f_l, f_r, g_l, g_r, i_l, i_r, j_l, and j_r have at most $k-1$ inputs.

$$
\begin{aligned}
& \langle x_k, f_l, f_r \rangle \wedge \langle x_k, i_l, i_r \rangle + \langle x_k, f_l, f_r \rangle \wedge \langle x_k, j_l, j_r \rangle + \\
& \langle x_k, g_l, g_r \rangle \wedge \langle x_k, i_l, i_r \rangle + \langle x_k, g_l, g_r \rangle \wedge \langle x_k, j_l, j_r \rangle && \text{from RHS} \\
= \; & \langle x_k, (f_l \wedge i_r + f_r \wedge i_l)/2, (f_l \wedge i_l + f_r \wedge i_r)/2 \rangle + \\
& \langle x_k, (f_l \wedge j_r + f_r \wedge j_l)/2, (f_l \wedge j_l + f_r \wedge j_r)/2 \rangle + \\
& \langle x_k, (g_l \wedge i_r + g_r \wedge i_l)/2, (g_l \wedge i_l + g_r \wedge i_r)/2 \rangle + \\
& \langle x_k, (g_l \wedge j_r + g_r \wedge j_l)/2, (g_l \wedge j_l + g_r \wedge j_r)/2 \rangle && \text{by Def 4.3.1} \\
= \; & \langle x_k, (f_l \wedge i_r + f_r \wedge i_l)/2 + (f_l \wedge j_r + f_r \wedge j_l)/2 + \\
& \quad (g_l \wedge i_r + g_r \wedge i_l)/2 + (g_l \wedge j_r + g_r \wedge j_l)/2, \\
& \quad (f_l \wedge i_l + f_r \wedge i_r)/2 + (f_l \wedge j_l + f_r \wedge j_r)/2 + \\
& \quad (g_l \wedge i_l + g_r \wedge i_r)/2 + (g_l \wedge j_l + g_r \wedge j_r)/2 \rangle && \text{by } + \text{ in SPBDD} \\
= \; & \langle x_k, (f_l \wedge i_r + f_r \wedge i_l + f_l \wedge j_r + f_r \wedge j_l \\
& \quad + g_l \wedge i_r + g_r \wedge i_l + g_l \wedge i_r + g_r \wedge j_l)/2, \\
& \quad (f_l \wedge i_l + f_r \wedge i_r + f_l \wedge j_l + f_r \wedge j_r \\
& \quad + g_l \wedge i_l + g_r \wedge i_r + g_l \wedge j_l + g_r \wedge j_r)/2 \rangle && \text{by } + \text{ and } /
\end{aligned}
$$

$$
\begin{array}{rll}
 & (\langle x_k, f_l, f_r\rangle + \langle x_k, g_l, g_r\rangle) \wedge (\langle x_k, i_l, i_r\rangle + \langle x_k, j_l, j_r\rangle) & \text{from LHS} \\
= & \langle x_k, f_l + g_l, f_r + g_r\rangle \wedge \langle x_k, i_l + j_l, i_r + j_r\rangle & \text{by + in SPBDD} \\
= & \langle x_k, ((f_l + g_l) \wedge (i_r + j_r) + (f_r + g_r) \wedge (i_l + j_l))/2, & \\
 & \quad ((f_l + g_l) \wedge (i_l + j_l) + (f_r + g_r) \wedge (i_r + j_r))/2\rangle & \text{by Def 4.3.1} \\
= & \langle x_k, (f_l \wedge i_r + f_l \wedge j_r + g_l \wedge i_r + g_l \wedge j_r & \\
 & \quad + f_r \wedge i_l + f_r \wedge j_l + g_r \wedge i_l + g_r \wedge j_l)/2, & \\
 & \quad (f_l \wedge i_l + f_l \wedge j_l + g_l \wedge i_l + g_l \wedge j_l & \\
 & \quad + f_r \wedge i_r + f_r \wedge j_r + g_r \wedge i_r + g_r \wedge j_r)/2\rangle & \text{by IH}
\end{array}
$$

□

Theorem 4.3.1 Given two Boolean functions f and g represented in EVBDDs, $\tau(f \cdot g) = \tau(f) \wedge \tau(g)$, where $\cdot$ is the conjunction operator in Boolean domain.

Proof: by induction on the size of inputs.

Base: $n = 0$, f and g are terminal nodes or constant functions.

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

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

Induction hypothesis (IH): assume it is true for $0 \leq n \leq k - 1$.

Induction: let $f = \langle x_k, f_l, f_r\rangle$ and $g = \langle x_k, g_l, g_r\rangle$ where f_l , f_r, g_l and g_r have at most $k - 1$ inputs.

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

$$
\begin{aligned}
& \tau(x_k, f_l, f_r\rangle) \wedge \tau(\langle x_k, g_l, g_r\rangle) && \text{(RHS)} \\
= \; & \langle x_k, \tau(f_r) - \tau(f_l), \tau(f_r) + \tau(f_l)\rangle \wedge \\
& \quad \langle x_k, \tau(g_r) - \tau(g_l), \tau(g_r) + \tau(g_l)\rangle && (\tau) \\
= \; & \langle x_k, ((\tau(f_r) - \tau(f_l)) \wedge (\tau(g_r) + \tau(g_l)) + \\
& \quad (\tau(f_r) + \tau(f_l)) \wedge (\tau(g_r) - \tau(g_l)))/2, \\
& \quad ((\tau(f_r) - \tau(f_l)) \wedge (\tau(g_r) - \tau(g_l)) + \\
& \quad (\tau(f_r) + \tau(f_l)) \wedge (\tau(g_r) + \tau(g_l)))/2\rangle && (\wedge) \\
= \; & \langle x_k, (\tau(f_r) \wedge \tau(g_r) + \tau(f_r) \wedge \tau(g_l) - \\
& \quad \tau(f_l) \wedge \tau(g_r) - \tau(f_l) \wedge \tau(g_l) + \\
& \quad \tau(f_r) \wedge \tau(g_r) - \tau(f_r) \wedge \tau(g_l) + \\
& \quad \tau(f_l) \wedge \tau(g_r) - \tau(f_l) \wedge \tau(g_l))/2, \\
& \quad (\tau(f_r) \wedge \tau(g_r) - \tau(f_r) \wedge \tau(g_l) - \\
& \quad \tau(f_l) \wedge \tau(g_r) + \tau(f_l) \wedge \tau(g_l) + \\
& \quad \tau(f_r) \wedge \tau(g_r) + \tau(f_r) \wedge \tau(g_l) + \\
& \quad \tau(f_l) \wedge \tau(g_r) + \tau(f_l) \wedge \tau(g_l))/2\rangle && \text{(LEMMA 4.3.3)} \\
= \; & \langle x_k, \tau(f_r) \wedge \tau(g_r) - \tau(f_l) \wedge \tau(g_l), \\
& \quad \tau(f_r) \wedge \tau(g_r) + \tau(f_l) \wedge \tau(g_l)\rangle && (+, -, /)
\end{aligned}
$$

□

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

$$
\begin{aligned}
f \vee g &= f + g - f \wedge g, && (4.9) \\
f \oplus g &= f + g - 2 \times (f \wedge g), && (4.10) \\
\bar{f} &= J^n - f, && (4.11)
\end{aligned}
$$

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

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

These operations $\vee, \oplus$, and $^{-}$ are from [53] with minor modification to match the τ operation. Operations $+$, $-$, and $\times$ are carried out in the same way as in EVBDDs (Sec. 2.3).

4.4 Filters Based on Prime Implicants

This section describes a set of filters based on the complete set of prime implicants of a function and presents OBDD-based data structures and algorithms for efficient computation of this set of filters.

Definition 4.4.1 Given a function $f(x_0, \ldots, x_{n-1})$, let

$$\begin{aligned}
Primes(f) &= \{p \mid p \text{ is a prime implicant of } f\}, \\
P_{f,x} &= \{p \mid p \in Primes(f) \wedge p \subseteq x\}, \\
P_{f,\bar{x}} &= \{p \mid p \in Primes(f) \wedge p \subseteq \bar{x}\}, \\
P_{f,x_\epsilon} &= \{p \mid p \in Primes(f) \wedge p \not\subseteq x \wedge p \not\subseteq \bar{x}\}, \\
P_c(f,x) &= \langle min(\mid P_{f,x} \mid, \mid P_{f,\bar{x}} \mid), max(\mid P_{f,x} \mid, \mid P_{f,\bar{x}} \mid), \mid P_{f,x_\epsilon} \mid\rangle, \text{ and} \\
P_{count}(f) &= \{\!\{P_c(f,x) \mid x \in support(f)\}\!\}.
\end{aligned}$$

where $\{\!\{\ \}\!\}$ represents a multi-set. We have the following propositions. Let F^{out} and F^{in} denote output and input filters as defined in Sec. 4.1.

Proposition 4.4.1 $\mid Primes(f) \mid$ is an F^{out}.

Proposition 4.4.2 $P_c(f,x)$ is an F^{in}.

Proposition 4.4.3 $P_{count}(f)$ is an F^{out}.

To compute the complete set of prime implicants of a function, a variation of OBDD was developed which is inspired by the Coudert and Madre's work [26]. In this variant of OBDD, each nonterminal node has three children *left*, *center* and *right* and denotes the function $xf_l + f_c + \bar{x}f_r$ where $f_c = f_l f_r$. The resulting OBDD is referred as the Consensus OBDD, or CBDD for short.

The computation of $Primes(f)$ is carried out in two steps. In the first step, we generate all prime implicants as well as some nonprime implicants based on the consensus operation. In the second step, we remove the nonprime implicants.

To perform the first step, we must convert OBDDs to CBDDs. The conversion is described by the following algorithm *obdd_to_cbdd*.

```
obdd_to_cbdd(f)
{
  if (f == 0) return 0;
  if (f == 1) return 1;
  f_c = obdd_and(f_l, f_r);
  return cbdd_construct(variable(f), obdd_to_cbdd(f_l),
                        obdd_to_cbdd(f_c), obdd_to_cbdd(f_r));
}
```

where *cbdd_construct* builds a nonterminal node with root $variable(\mathbf{f})$ and three children.

After the operation *obdd_to_cbdd*, the resulting CBDD contains not only all prime implicants but also some nonprime implicants. In the second step, we remove the nonprime implicants from the CBDD using the following formula:

$$\begin{aligned} Primes(f) &= (Primes(f_l) - Primes(f_l f_r)) \cup (Primes(f_r) - Primes(f_l f_r)) \\ &\quad \cup Primes(f_l f_r) \end{aligned}$$

The operation is carried out by the *cbdd_to_pbdd* procedure. The resulting OBDD is referred as Prime OBDD, or PBDD for short, in which every path to a terminal node 1 corresponds to a prime implicant. Note that, a PBDD f contains all the prime implicants of function f and its center child contains all the prime implicants of function $f_l f_r$, but its left (right) child neither contains all the prime implicants of f_l (f_r) nor contains all the prime implicants of $f_l - f_r$ ($f_r - f_l$). This is because the 'subtract' (*pbdd_sub*) operation is not closed under PBDDs.

```
cbdd_to_pbdd(f)
{
  if (f == 0) return 0;
  if (f == 1) return 1;
  h_l = cbdd_to_pbdd(f_l);
  h_r = cbdd_to_pbdd(f_r);
  new_c = cbdd_to_pbdd(f_c);
  new_l = pbdd_sub(h_l, new_c);
  new_r = pbdd_sub(h_r, new_c);
  return pbdd_construct(variable(f), new_l, new_c, new_r);
}
```

```
pbdd_sub(f, g)
{
  if (f == 0) return 0;
  if (g == 0) return f;
  if (f == g) return 0;
  if (variable(f) > variable(g))
    return pbdd_sub(f, g_c);
  if (variable(f) < variable(g))
    return pbdd_construct(variable(f), f_l, pbdd_sub(f_c, g), f_r);
  return pbdd_construct(variable(f), pbdd_sub(f_l, g_l),
                        pbdd_sub(f_c, g_c), pbdd_sub(f_r, g_r));
}
```

Filters described in propositions 4.4.1, 4.4.2, and 4.4.3 can be easily constructed using the PBDDs. For example, $\mid Primes(f) \mid$ is equivalent to computing the number of 1-paths in the PBDD for f.

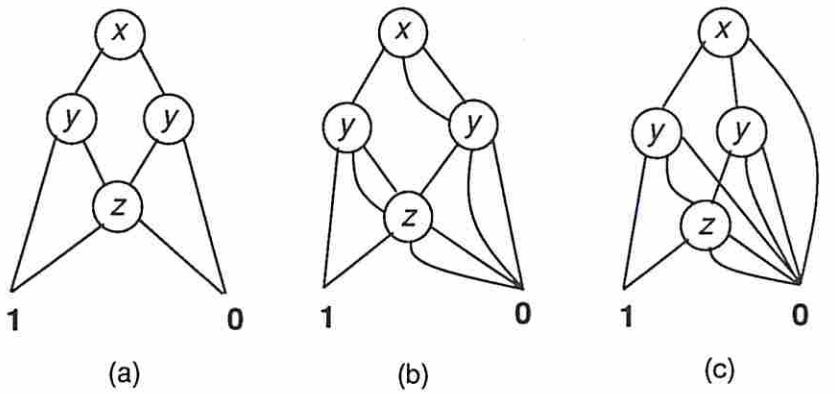


Figure 4.4: Function *carry* represented by (a) OBDD (b) CBDD (c) PBDD.

Example 4.4.1 The function *carry* of a full adder represented by OBDD, CBDD, and PBDD is shown in Fig. 4.4. The CBDD representation of *carry* in Fig. 4.4 (b) is derived by applying procedure *obdd_to_cbdd* on the OBDD in Fig. 4.4 (a), and the PBDD in Fig 4.4 (c) is derived by applying procedure *cbdd_to_pbdd* on the one in Fig. 4.4 (b).

The set of prime implicants of *carry* is $\{xy, yz, xz\}$. Note that the implicants of *carry* represented by 4.4 (b) are $\{xy, x\bar{y}z, xz, yz, \bar{x}yz\}$ where $x\bar{y}z$ and $\bar{x}yz$ are not prime implicants.

The set of filters defined in this section for *carry* is as follows:

$$\begin{aligned} |\, Primes(carry)\,| &= 3, \\ P_c(carry, x) &= \langle 0, 2, 1\rangle, \\ P_c(carry, y) &= \langle 0, 2, 1\rangle, \\ P_c(carry, z) &= \langle 0, 2, 1\rangle, \text{ and} \\ P_{count}(carry) &= \{\!\{\langle 0, 2, 1\rangle, \langle 0, 2, 1\rangle, \langle 0, 2, 1\rangle\}\!\}. \end{aligned}$$

□

4.5 Experimental Results

The Boolean matching algorithm and the various filters have been tested on the MCNC and ISCAS benchmark circuits. For each circuit we constructed two OBDDs.

The second OBDD was generated by randomly permuting and renaming the variables of the first. Table 4.2 shows the characteristics of each circuit, the number of nodes in each OBDD and the execution times (in CPU seconds on a Sparc Station II with 64 MB of memory) for the Boolean matching algorithm with and without the use of various filters. The following legend describes the data in each column.

1. **A** : Boolean matching with no filters used.

2. **B** : F_{on}^{in} computed and checked during Boolean matching.

3. **C** : $F_{dep}^{out}, F_{on}^{out}, F_{d_n}^{out}, and F_{d_1}^{out}$ used in the given order as long as more than two outputs have the same filter value. These filters are computed and checked prior to Boolean matching.

4. **D** : F^{out} filters are used as in **C** (before matching) and then F_{on} is used as in **B** (during matching).

	in,out	A	B	C	D
sao2	10, 4	12.39	0.85	0.73	0.40
rd73	7, 3	1.85	0.10	0.07	0.12
rd84	8, 4	6.27	0.22	0.02	0.08
misex1	8, 7	13.13	12.52	0.03	0.09
misex2	25, 18	—	—	3.68	1.87
misex3	14, 14	—	—	13.25	6.08
apex1	45, 45	—	—	—	165.80
apex2	39, 3	—	221.42	—	62.47
duke2	22, 29	—	—	3.50	2.54
c432	36, 7	—	1279.72	29.55	34.10

Table 4.2: Experimental results of Boolean matching.

The results displayed in Table 4.2 are summarized below.

- Comparing columns **A** and **B**, we find that in those cases where the Boolean matching was completed in a reasonable amount of time, the application of F_{on}^{out} resulted in speed up ranging from 19 to 28 times.

- Comparing columns **A** and **C**, we find a speed up ranging from 17 to 430 times. We also note that for circuits *misex2*, *misex3*, *duke2*, and *c432* the matching algorithm without any filters had to be aborted. However, use of the filters F_{dep}^{out}, F_{on}^{out}, $F_{d_n}^{out}$, and $F_{d_1}^{out}$, resulted in successful completion within a very short period of time.

- When F_{on}^{in} was applied in conjunction with the filters in $\{F_{dep}^{out}, F_{on}^{out}, F_{d_n}^{out}, F_{d_1}^{out}\}$, then the matching algorithm was successful in all cases.

- For circuits with a small number of inputs, the filters in $\{F_{dep}^{out}, F_{on}^{out}, F_{d_n}^{out}, F_{d_1}^{out}\}$ seem to be more effective than F_{on}^{in}. However, F_{on}^{in} is very effective for larger circuits.

The above results are generated using arbitrary variable orderings for the initial OBDDs. Using any of the other ordering strategies that have been reported [12], we expect further improvement in performance.

Table 4.3 shows the results of some benchmarks represented in both EVBDD and SPBDD forms. Column 'EVBDD' depicts the size and time required for representing and constructing a circuit using EVBDDs while column 'SPBDD' depicts the size of SPBDDs and the time required for converting from EVBDDs to SPBDDs. In average, the ratio of the number of nodes required for representing SPBDDs over that of EVBDDs is 6.8, and the ratio of the conversion time for SPBDDs over the construction time of EVBDDs is 41.

The performance of a filter depends on its capability of pruning (effectiveness) and its computation time (cost). Experimental results of [23] show that this filter

is quite good because it rejected all unmatchable functions that were encountered. However, according to results of Table 4.3, this filter is relatively expensive to compute when comparing with other filters [57]. Therefore, this filter should be used only after other filters have failed to prune the search space.

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

Table 4.3: Experimental results of SPBDDs.

In Table 4.4, columns 'OBDD', 'CBDD', and 'PBDD' depict the time required for constructing a circuit using OBDDs, CBDDs, and PBDDs, respectively. Column '# Primes' depicts the number of prime implicants of a circuit. Since the construction of the PBDD filters requires operations similar to those performed while constructing F_{d1}^{out}, we should apply these filters only after F_{d1}^{out} has failed to prune the search space.

	OBDD	CBDD	PBDD	# Primes
9symml	0.17	0.77	0.05	1680
c8	0.08	0.2	0.07	228
cc	0.02	0.08	0.04	52
cmb	0.04	0.11	0.01	26
comp	0.09	80.73	0.1	196606
cordic	0.06	9.3	0.06	1742
count	0.11	0.39	0.16	336
cu	0.02	0.07	0.05	29
f51m	0.06	0.18	0.09	94
lal	0.05	0.21	0.08	102
mux	0.08	0.84	0.16	81
parity	0.02	0.08	0.01	32768
pcle	0.03	0.08	0.08	81
pcler8	0.04	0.12	0.09	119
pm1	0.02	0.04	0.01	37
sct	0.05	0.18	0.11	78
ttt2	0.19	0.51	0.12	160
unreg	0.04	0.12	0.1	112
x2	0.01	0.05	0.03	33
z4ml	0.06	0.15	0.03	59
b9	0.08	0.32	0.14	146
alu2	0.58	2.6	0.47	328
alu4	2.88	38.28	9.51	1980
term1	1.11	7.52	1.44	330
apex7	0.28	1.66	0.99	1027
cht	0.08	0.14	0.08	120
example2	0.19	0.41	0.28	461

Table 4.4: Experimental results of PBDDs.

Chapter 5

Integer Linear Programming

Integer Linear Programming (ILP) is an NP-hard problem [42] that appears in many applications. Most of existing techniques for solving ILP such as branch-and-bound [64, 31, 76] and cutting-plane methods [43, 44] are based on the linear programming (LP) method. While they may sometimes solve hundreds of variables, they cannot guarantee to find an optimal solution for problems with more than , say, 50 variables. It is believed that an effective ILP solver should incorporate integer or combinatorial programming theory into the linear programming method [4].

Jeong et al. [59] describe an OBDD-based approach for solving the 0-1 programming problems. This approach does not, however, use OBDDs for integer related operations such as conversion from linear inequality form of constraints into Boolean functions and optimization of nonbinary goal functions. Consequently, the caching of computation results is limited to only Boolean operations (i.e., for constraint conjunction).

This chapter presents an approach for solving the ILP by combining benefits of the EVBDD data structure (in terms of subgraph sharing and caching of computation results) with the state-of-the-art ILP solving techniques. The approach uses a minimization operator in EVBDD which computes the optimal solution to a given goal function subject to a constraint function. In addition, the construction

and conjunction of constraints in terms of EVBDDs are carried out in a divide and conquer manner in order to manage the space complexity.

5.1 Background

An ILP problem can be formulated as follows:

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

The first equation is referred as the *goal function* and the second equation is referred as *constraint functions*. It is assumed that the problem to be solved is a *minimization* problem. A *maximization* problem can be converted to a minimization problem by changing the sign of coefficients in the goal function.

There are three classes of algorithms for solving ILP problems [93]. The first class is known as the branch-and-bound method [64, 31, 76]. This method usually starts with an optimum continuous LP solution which forms the first *node* of a search tree. If the initial solution satisfies the integer constraints, it is the optimum solution and the procedure is terminated. Otherwise, we split on variable x (with value x^* from the initial solution) and create two new subproblems: one with the additional constraint $x \leq \lfloor x^* \rfloor$ and the other with the additional constraint $x \geq \lfloor x^* \rfloor + 1$. Each subproblem is then solved using the LP methods, e.g., the simplex method [32] or the interior point method [60]. A subproblem is pruned if there are no feasible solutions, the feasible solution is inferior to the best one found, or all variables satisfy the integer constraints. In the last case, the feasible solution becomes the new best solution. The problem is solved when all subproblems are processed. Most commercial programs use this approach [65].

The second class is known as the implicit enumeration technique which deals with 0-1 programming [2, 3, 91]. Initially, all variables are *free*. Then, a sequence of *partial solutions* is generated by successively *fixing* free variables, i.e., setting free variables to 0 or 1. A *completion* of a partial solution is a solution obtained by fixing all free variables in the partial solution. The algorithm ends when all partial solutions are completions or are discarded. The procedure proceeds similar to the branch-and-bound except that it solves a subproblem using the *logical tests* instead of the LP. A logical test is carried out by inserting values corresponding to a given (partial or complete) solution in the constraints. A complete solution is feasible if it satisfies all constraints. A partial solution is pruned if it cannot reach a feasible solution or could only produce an inferior feasible solution (compared to the current best solution). One advantage of this approach is that we can use *partial order* relations among variables to prune the solution space. For example, if it is established that $x \leq y$, then portions of the solution space which correspond to $x = 1$ and $y = 0$ can be immediately pruned [11, 48].

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

The third class is known as the cutting-plane method [43, 44]. Here, the integer variable constraint is initially dropped and an optimum continuous variable solution is obtained. The solution is then used to chop off the solution space while ensuring that no feasible integer solutions are deleted. A new continuous solution

is computed in the reduced solution space and the process is repeated until the continuous solution indeed becomes an integer solution. Due to the machine round-off error, only the first few cuts are effective in reducing the solution space [93].

5.2 A Model Algorithm

This section first shows a straightforward method to solve the ILP problem using EVBDDs. It then describes how to improve this method in this and the following sections.

Example 5.2.1 For the sake of readability, flattened EVBDDs (see Sec. 2.3.1) are used in this example.

The following is a 0-1 ILP problem:

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

We first construct an EVBDD for the goal as shown in Fig. 5.1 (a). We then construct the constraints. The left hand side of constraint (1) represented by an EVBDD is shown in Fig. 5.1 (b). After the relational operator $\leq$ has been applied on constraint (1), the resulting EVBDD is shown in Fig. 5.1 (c). Similarly, EVBDDs for constraint (2) are shown in Fig. 5.1 (d) and (e). The conjunction of two constraints, Fig. 5.1 (c) and (e), results in the EVBDD in Fig. 5.1 (f) which represents the solution space of this problem. A feasible solution corresponds to a path from the root to 1.

We then *project* the constraint function c onto the goal function g such that for a given input assignment X, if $c(X) = 1$ (feasible) then $p(X) = g(X)$; otherwise $p(X) = infeasible\_value$. For minimization problems, the $infeasible\_value$ is any value which is greater than the maximum of g, and for maximization problems,

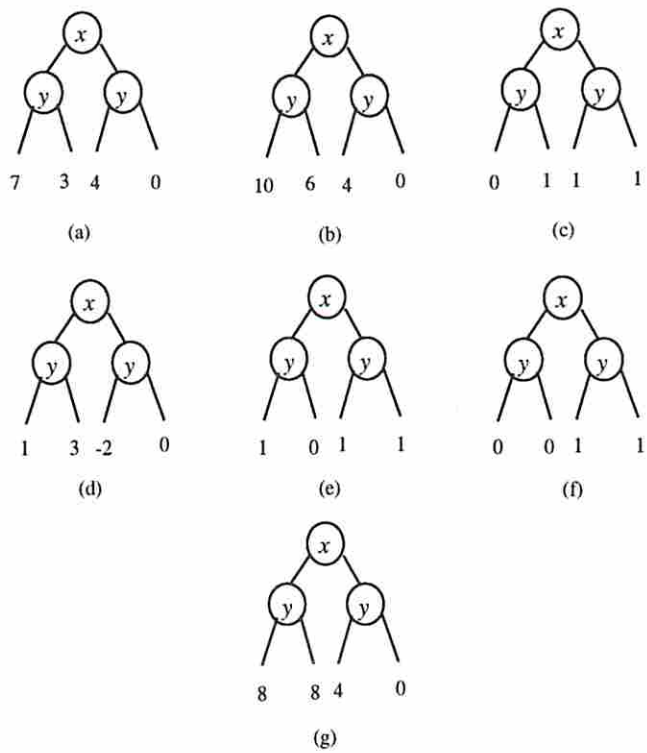


Figure 5.1: A simple example (using flattened EVBDDs and OBDDs).

the $infeasible\_value$ is any value which is smaller than the minimum of g. This example uses 8 as the $infeasible\_value$. Thus, in Fig. 5.1 (g), the two leftmost terminal values have been replaced by value 8. The last step in solving the above ILP problem is to find the minimum in Fig. 5.1 (g) which is 0. □

The above approach has three problems:

1. Converting a constraint from inequality form to a Boolean function may require exponential number of nodes;

2. Even if all constraints can be constructed without using excessive amounts of memory, conjoining them altogether at once may create too big an EVBDD; and

3. The operator *projection* is useful when we want to find all optimal solutions. However, in many situations, we are interested in finding *any* optimal

solution. Thus, full construction of the final EVBDD (e.g., Fig. 5.1 (g)) is unnecessary.

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

Every constraint is converted to the form $AX - b \leq 0$. Thus, it only need one operator *leq*0 (Sec. 2.3.3) to perform the conversion. $AX < b$ is converted to $AX - b + 1 \leq 0$ (since all coefficients are integer); $AX \geq b$ is converted to $-AX + b \leq 0$; and $AX = b$ is converted to two constraints $AX - b \leq 0$ and $-AX + b \leq 0$.

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

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

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

```
ilp_minimize(goal, const, lower_bound, upper_bound, n_supp, c_size)
{
1    if (max(goal) < lower_bound) return;
2    if (min(goal) ≥ upper_bound) return;
3    if (∃c ∈ const such that minimize(goal, c, upper_bound) == 0) return;
4    new_const = conjunction_const(const, c_size);
5    if (new_const has only one element and is in Boolean form) {
6        (void) minimize(goal, new_const, upper_bound);
7    }
8    else {
9    ⟨⟨goal_l, new_const_l⟩, ⟨goal_r, new_const_r⟩⟩ =
         divide_problem(goal, new_const, n_supp);
10   ilp_minimize(goal_l, new_const_l, lower_bound, upper_bound, n_supp, c_size);
11   ilp_minimize(goal_r, new_const_r, lower_bound, upper_bound, n_supp, c_size);
12   }
}
```

Figure 5.2: Pseudo code for *ilp_minimize*.

Similarly, the new goal and constraint functions for the second subproblem are the right children of the root nodes of the current goal and constraints.

The main algorithm, *ilp_minimize*, employs a branch-and-bound technique as shown in Fig. 5.2. In addition to goal and constraint functions, *n_supp*, and *c_size*, there are two parameters which are used as bounding condition: *Lower bound* is either given by the user or computed through linear relaxation or Lagrangian relaxation methods; *Upper bound* represents the best feasible solution found so far. The initial value of the upper bound is the maximum of the goal function plus 1.

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

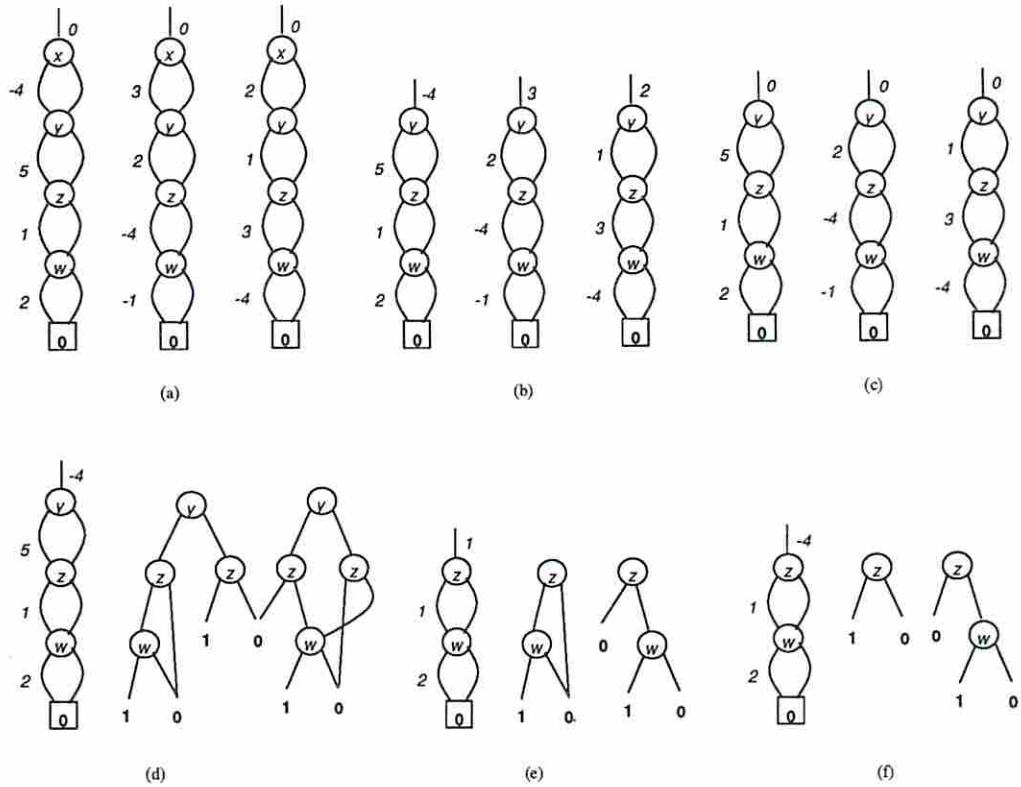


Figure 5.3: An example for conjoining constraints.

Example 5.2.2 We want to solve the following problem:

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

1. The initial goal and constraint EVBDDs are shown in Fig. 5.3 (a). Suppose both parameters *n_supp* and *c_size* are set to 4.

2. Since the number of supporting variables in the constraint EVBDDs is not less than 4, we divide the problem into two subproblems: one with $x = 1$ (Fig. 5.3 (b)) and the other with $x = 0$ (Fig. 5.3 (c)). The final solution is the minimum of solutions to these two subproblems.

3. Next, we want to solve the subproblem with $x = 1$. Since the number of supporting variables in constraint EVBDDs is smaller than *n_supp*, we convert the constraint EVBDDs into Boolean functions by carrying out operation *leq*0 (Fig. 5.3 (d)).

4. Since the size of constraint EVBDDs are not less than *c_size*, we divide the problem into two subproblems: one with $y = 1$ (Fig. 5.3 (e)) and the other with $y = 0$ (Fig. 5.3 (f)).

5. Now, we want to solve the subproblem with $y = 1$. Since the size of both constraint EVBDDs are less than *c_size*, we conjoin them together and then solve this subproblem using the *minimize* operator (Sec. 5.3).

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

5.3 The Operator *minimize*

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

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

The parameter b serves two purposes: it increases the hit ratio for computation caching and is a bounding condition for pruning the problem space. To achieve the first goal, an entry of the computed table used by *minimize* has the form

$\langle g, c, \langle b, v \rangle \rangle$ where v is set to the minimum of g which satisfies c and is less than b. If there is no feasible solution (with respect to g and c) which is less than b, then v is set to b.

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

Lines 9-17 perform the table lookup operation. If the lookup succeeds, no further computation is required; otherwise, we traverse down the graph in lines 19-26 in the same way as *apply*. Since *minimize* satisfies the additive property (Sec. 2.3.2), we subtract c_g from *bound* to obtain a new local bound (*local_bound*) in line 9. c_g will be added back to *bound* in lines 13 or 32 if a new solution is found.

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

1. $entry.value < entry.bound$, i.e., a smaller value v was previously found with respect to g, c, and $entry.bound$ (i.e., the minimization of g with respect to c has been solved and the result is $entry.value$).

   (a) If $entry.value < local\_bound$, then $entry.value$ is the solution we wanted.

   (b) Otherwise, the best we can find under g and c is $entry.value$ which is inferior to *local_bound*, so we return with no success.

2. $entry.bound = entry.value$, i.e., there was no feasible solution with respect to g, c, and $entry.bound$ (i.e., there is no stored result for the minimization of g with respect to c and $entry.bound$).

   (a) If $local\_bound \leq entry.bound$, then we cannot possibly find a solution better than $entry.bound$ for g under c. Therefore, we return with no success.

   (b) Otherwise, no conclusion can be drawn and further computation is required. Although there is no better feasible solution than $entry.bound$, it does not imply that there will be no better solution than $local\_bound$.

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

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

```
minimize(⟨c_g, g⟩, ⟨c_c, c⟩, bound)
{
1    if (⟨c_c, c⟩ == ⟨0, 0⟩) return 0;
2    if (min(⟨c_g, g⟩) ≥ bound) return 0;
3    if (⟨c_g, g⟩ == ⟨c_g, 0⟩) {
4       bound = c_g;
5       return 1;    }
6    if (⟨c_c, c⟩ == ⟨1, 0⟩) {
7       bound = min(⟨c_g, g⟩);
8       return 1;    }
9    local_bound = bound − c_g;
10   if (comp_table_lookup(⟨0, g⟩, ⟨c_c, c⟩, entry)) {
11      if (entry.value < entry.bound) {
12          if (entry.value < local_bound) {
13              bound = entry.value + c_g;
14              return 1;   }
15          else return 0;   }
16      else {
17          if (local_bound ≤ entry.bound) return 0;   }   }
18   entry.bound = local_bound;
19   ⟨c_gl, gl⟩ = ⟨value(g), child_l(g)⟩;
20   ⟨c_gr, gr⟩ = ⟨0, child_r(g)⟩;
21   if (index(variable(c)) ≤ index(variable(g))) {
22      ⟨c_cl, cl⟩ = ⟨c_c + value(c), child_l(c)⟩;
23      ⟨c_cr, cr⟩ = ⟨c_c, child_r(c)⟩;   }
24   else {   ⟨c_cl, cl⟩ = ⟨c_cr, cr⟩ = ⟨c_c, c⟩;   }
```

```
if (min(g_l) ≤ min(g_r)) {
    t_ret = minimize(⟨c_gl, g_l⟩, ⟨c_cl, c_l⟩, local_bound);
    e_ret = minimize(⟨c_gr, g_r⟩, ⟨c_cr, c_r⟩, local_bound);   }
else {
    e_ret = minimize(⟨c_gr, g_r⟩, ⟨c_cr, c_r⟩, local_bound);
    t_ret = minimize(⟨c_gl, g_l⟩, ⟨c_cl, c_l⟩, local_bound);   }
if (t_ret || e_ret) {
    bound = local_bound + c_g;
    entry.value = local_bound;
    comp_table_insert(⟨0, g⟩, ⟨c_c, c⟩, entry);
    return 1;   }
else {
    entry.value = entry.bound;
    comp_table_insert(⟨0, g⟩, ⟨c_c, c⟩, entry);
    return 0;   }
}
```

Example 5.3.1 We want to minimize the goal function $-4x+5y+z+2w$ subject to the constraint $(xz\bar{w} \vee \bar{x}yz\bar{w} \vee \bar{x}\bar{y}z \vee \bar{x}\bar{y}\bar{z}w = 1)$ shown in Fig. 5.4. For the sake of readability, the goal function is represented in EVBDD while the constraint function is represented in OBDD. The initial *upper bound* is $max(goal)+1 = 0+5+1+2+1 = 9$. The reason for plus 1 is to recognize the case when there are no feasible solutions.

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

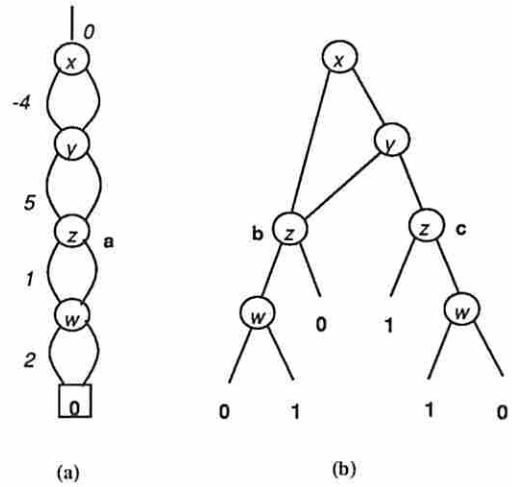


Figure 5.4: An example for the *minimize* operator.

$\langle \mathbf{a}, \mathbf{b}, \langle 8, 1 \rangle \rangle$ as an entry into the computed table and recalculate the *upper bound* as $-4 + 5 + 1 + 0 = 2$.

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

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

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

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

5.4 Discussion

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

Search Strategy

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

Branching Rule

This parameter refers to the selection of next variable to branch. Various selection criteria which have been proposed use *priorities* [75], *penalties* [37, 95], *pseudo-cost* [7], and *integer infeasibility* [3] conditions. Currently, FGILP uses the same variable ordering as the one used to create EVBDDs because it simplifies the implementation. When the variable selected does not correspond to the variable ordering of EVBDD, operation *cofactor* (instead of $child_l$ and $child_r$) should be used.

Bounding Procedure

The most important component of a branch-and-bound method is the bounding procedure. The better the bound, the more pruning of the search space. The most frequently used bounding procedure is to use the linear programming method. Other procedures which can generate better bounds, but are more difficult to implement include the cutting planes, Lagrangian relaxation [41, 90], and disjunctive

programming [4]. The bounding procedure used in FGILP is similar to the one proposed in [2]. The most pruning takes place at line 3 of the code for *ilp_minimize*. This pruning rule however has two weak points. First, it is carried out on each constraint one at a time. Thus, it is only a 'local' method. Second, it can only be applied to a constraint which is in the Boolean form. The other bounding procedures described above are 'global' methods which are directly applicable to the inequality form.

Logical Tests

It is believed that logical tests may be as important as the bounding procedure [80]. In addition to partial ordering of variables, a particularly useful class of tests, when available, are those based on *dominance* [56, 59]. Currently, FGILP employs no logical tests. It is believed that the inclusion of logical tests in FGILP will improve its performance.

Despite the fact that there are many improvements which can be made to FGILP, the performance of FGILP, as it is now, is already comparable to that of LINDO [88] which is one of the most widely used commercial tools [80] for solving ILP problems.

5.5 Experimental Results

FGILP has been implemented in C under the SIS environment. Table 5.1 shows experimental results on ILP problems from MIPLIB [74]. It also shows the results of LINDO [88] (a commercial tool) on the same set of benchmarks. FGILP was run under SPARC station 2 (28.5 MIPS) with 64 MB memory while LINDO was run under SPARC station 10 (101.6 MIPS) with 384 MB memory. In Table 5.1, column 'Problem' lists the name of problems, columns 'Inputs' and 'Constraints' indicate the number of input variables and constraints, and columns 'FGILP' and 'LINDO'

are the running time in seconds for obtaining the optimal solution shown in the last column.

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

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

2. EVBDDs with smallest size are conjoined first.

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

The parameters used for the problems in Table 5.1 are summarized below:

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

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

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

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

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

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

Table 5.1: Experimental results of ILP problems.

Chapter 6

Function Decomposition

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

The function decomposition theory was studied by Ashenhurst [1], Curtis [27], and Roth and Karp [82]. In Ashenhurst-Curtis method, functions are represented by Karnaugh maps and the decomposability of functions are determined from the number of distinct columns in the map. In Roth-Karp method, functions are represented by cubes and the decomposability of functions are determined from the cardinality of compatible classes. Recently, researchers [8, 20, 29, 83] have used OBDDs to determine decomposability of functions. However, most of these works only consider single-output Boolean functions.

Definition 6.0.1 A function $f(x_0, \ldots, x_{n-1})$ is said to be *decomposable* under *bound set* $\{x_0, \ldots, x_{i-1}\}$ and *free set* $\{x_{i-s}, \ldots, x_{n-1}\}, 0 < i < n, 0 \leq s$ if f can be transformed to $f'(g_0(x_0, \ldots, x_{i-1}), \ldots, g_{j-1}(x_0, \ldots, x_{i-1}), x_{i-s}, \ldots, x_{n-1})$, where $0 < j < i - s$. If s equals 0 then it is *disjunctively decomposable*; otherwise, it is *nondisjunctively decomposable.* If j equals 1 then it is *simply decomposable.* Function f' is referred as the *F-function* and each g_i is referred as a *G-function.*

6.1 Disjunctive Decomposition

Definition 6.1.1 Given an OBDD $\mathbf{v}$ representing $f(x_0, \ldots, x_{n-1})$ with variable ordering $x_0, \ldots, x_{n-1}$ and bound set $B = \{x_0, \ldots, x_{i-1}\}$, we define

$$cut\_set(\mathbf{v}, B) \;=\; \{\mathbf{u} \mid \mathbf{u} = eval(\mathbf{v}, p), 0 \leq p < 2^i\}.$$

In the above definition, each element in $cut\_set(\mathbf{v}, B)$ corresponds to a distinct column in the Ashenhurst-Curtis decomposition charts [1, 27]. Furthermore, $\lceil \log_2 \mid cut\_set(\mathbf{v}, B) \mid \rceil$ determines the minimum number of G-functions required for a decomposition of f under B.

Example 6.1.1 Let $f = x_0x_1x_2x_3 + x_0x_1x_2\bar{x}_3\bar{x}_4 + x_0x_1\bar{x}_2\bar{x}_4 + x_0\bar{x}_1x_2\bar{x}_4 + x_0\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_0x_1x_2\bar{x}_4 + \bar{x}_0x_1\bar{x}_2\bar{x}_3 + \bar{x}_0\bar{x}_1x_2x_3 + \bar{x}_0\bar{x}_1x_2\bar{x}_3x_4 + \bar{x}_0\bar{x}_1\bar{x}_2\bar{x}_3$, the OBDD representation and the decomposition chart of $f(x_0, x_1, x_2, x_3, x_4)$ with respect to the bound set $\{x_0, x_1, x_2\}$ and the free set $\{x_3, x_4\}$ are shown in Fig. 6.1 (a) and (b) respectively. In the OBDD representation, $cut\_set(\mathbf{f}, \{x_0, x_1, x_2\}) = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$. Nodes **a**, **b**, and **c** correspond to distinct columns 1100, 1010, and 1011 respectively. Since there are three distinct columns f is not simply decomposable under the bound set $\{x_0, x_1, x_2\}$ and the free set $\{x_3, x_4\}$. □

When bound variables are on the top of OBDDs, the computation of cut_sets is straightforward. The following procedure assumes that bound variables B are already on the top of OBDDs. The time complexity of computing cut_sets depends on the size of OBDDs.

```
cut_set(v, B)
{
  if (index(v) > index(last(B))) return({ v });
  else return( cut_set(child_l(v), B) ∪ cut_set(child_r(v), B) );
}
```

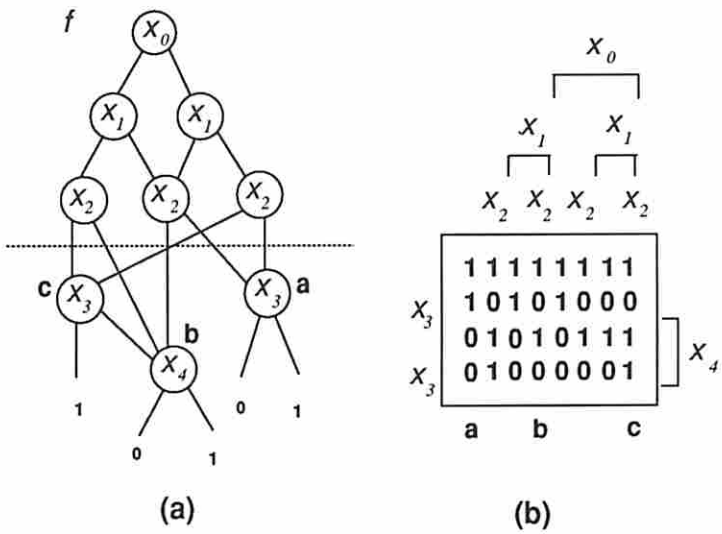


Figure 6.1: A function represented in (a) OBDD and (b) decomposition chart.

To move a bound variable x to the top of an OBDD, we carry out $new\_obdd(x, \mathbf{f_x}, \mathbf{f_{\overline{x}}})$ where $\mathbf{f_x}$ and $\mathbf{f_{\overline{x}}}$ are the cofactor of f with respect to x and $\overline{x}$ respectively. In the worst case, both $\mathbf{f_x}$ and $\mathbf{f_{\overline{x}}}$ have about the same size as that of $\mathbf{f}$. Thus, moving a variable to the top may double the size of an OBDD. To move bound variables to the top is therefore practical only for small size of bound sets. An algorithm for computing the cut_set without moving the bound variables to the top will be presented in Sec. 6.3.

It is clear that the computation of cut_sets of all 2^n bound sets is very expensive. However, in practical applications, we need compute the cut_sets of C_k^n bound sets where k is a small number such as 4 or 5. The time complexity of computing the cut_sets of n choose k bound sets is then $O(n^k m)$ and the space complexity is $O(2^k m)$ where m is the size of an OBDD.

The following algorithm shows how to perform the disjunctive decomposition

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

directly on an OBDD.

Algorithm *decompose* (pseudo code *decompose*):

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

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

2. Encode each node in the cut_set by $\lceil \log_2 k \rceil = j$ bits (Fig. 6.2 (a)). Let the encoding of $\mathbf{u_q}$ be q.

3. Construct $\mathbf{v_{f'}}$ to represent function f':
   Replace the top part of $\mathbf{v_f}$ by a new set of variables $g_0, \ldots, g_{j-1}$ such that $eval(\mathbf{v_{f'}}, q) = \mathbf{u_q}$ for $0 \leq q < k-1$, $eval(\mathbf{v_{f'}}, q) = \mathbf{u_{k-1}}$ for $k-1 \leq q < 2^j$ (Fig. 6.2 (b) and pseudo code *decomp_f*).

4. Construct $\mathbf{v_{g_p}}$'s to represent g_p's, $0 \leq p < j$:
   Replace each node $\mathbf{u}$ with encoding $b_0, \ldots, b_{j-1}$ in the cut_set by terminal node $\mathbf{b_p}$ (Fig. 6.2 (c) and pseudo code *decomp_g*). For example, in Fig. 6.2 (c) $\mathbf{u_1}$ is replaced by terminal node **1** in the construction of $\mathbf{v_{g_{j-1}}}$ and by terminal node **0** in the construction of other $\mathbf{v_{g_p}}$'s.

The idea behind Algorithm *decompose* is the following: For any input pattern m in the bound set, the evaluation of m in function f will result at a node in the cut_set with encoding e. The evaluation of m on these g_p functions will produce the function values e. Then, the evaluation of e in function f' will also end at the same node in the cut_set. Thus, the composition of f' and g_p's is equivalent to f. If $k < 2^j$, then not every j-bit pattern is used for the encoding of the cut_set. Function g_p's can never generate those function values which correspond to the patterns not used for encoding, thus we can assign arbitrary node to these pattern in function f'. In step 4, we assign them to the last node in the cut_set ($\mathbf{u_{k-1}}$). Alternatively, we can assign don't-cares instead.

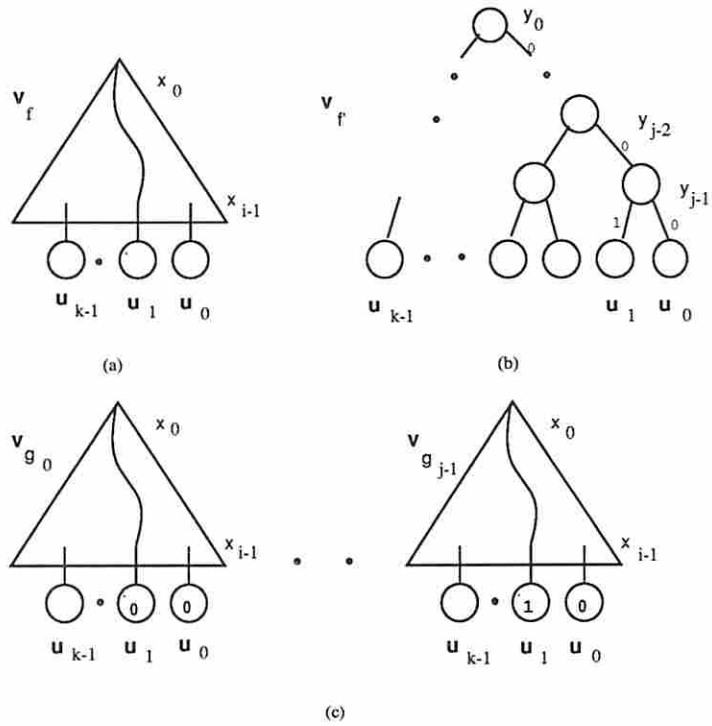


Figure 6.2: Disjunctive decomposition.

In step 2 of Algorithm *decompose*, we use an arbitrary encoding which is not unique. Different encodings will result in different decompositions. Different encoding strategies can be used for different applications. For example, in the mapping of G-functions to k-input LUT FPGAs where a k-input LUT can implement any Boolean function of up to k variables, any encoding strategy would be the same if the bound-set size is k.

$decompose(\mathbf{v}_f, cutset)$ $\quad$ $/* \; cutset = \{\mathbf{u}_0, \ldots, \mathbf{u}_{k-1}\}, 2^{j-1} < k \leq 2^j \; */$

{

$\quad \mathbf{v_{f'}} = decomp\_f(cutset, \{y_{j-1}, \ldots, y_0\});$

$\quad$ for $(p = 0; p < j; p++)$

$\quad\quad \mathbf{v}_{g_p} = decomp\_g(\mathbf{v}_f, cutset, p);$

$\quad$ return $\mathbf{v}_{f'}$ and $\mathbf{v}_{g_p}$'s;

}

```
decomp_f(cutset, ids)
  cutset: an array of OBDD nodes with 2^(j-1) < length(cutset) ≤ 2^j
  return: an OBDD v_f' with variables y_0, ..., y_{j-1}, x_i, ..., x_{n-1}
{
  ptr_id = 0;
  while (length(cutset) > 1) {   /* do until cutset becomes a root node */
    id = ids[ptr_id ++];
    p = 0;        /* pointer to cutset */
    ptr = 0;        /* pointer to newset */
    while (p < length(cutset)) {
      if (p == length(cutset) - 1) {   /* the last element of cutset */
        newset[ptr ++] = cutset[p];   /* just move it to newset */
        p ++;        /* p is increased by 1 */
      }
      else {   /* create a new node with new variable y_id */
        newset[ptr ++] = new_bdd(id, cutset[p + 1], cutset[p]);
        p += 2;      /* p is increased by 2 */
      }
    }
    for (q = 0; q < ptr; q ++)        /* move nodes from newset to cutset */
      cutset[q] = newset[q];
  }
  return(cutset[0]);        /* cutset[0] contains v_f' */
}
```

```
decomp_g(v, cutset, p)
  v: an OBDD
  cutset: an array of OBDD nodes with 2^(j-1) < length(cutset) ≤ 2^j
  p: an integer between 0 and j - 1
  return: v_gp
{
  for (q = 0; q < length(cutset); q++) {
    v_q = cutset[q];
    if (bit(p, q) == 0)    /* p^th bit from right of q */
      v = replace_node(v, v_q, 0);    /* replace v_q in v by 0 */
    else v = replace_node(v, v_q, 1);    /* replace v_q in v by 1 */
  }
  return v;    /* as v_gp */
}
```

Example 6.1.2 As an example of how *decompose* works, consider the OBDDs shown in Fig. 6.3. Since x_4-node in $\mathbf{f}$ has encoding 01, it has been replaced by terminal nodes $\mathbf{0}$ and $\mathbf{1}$ in $\mathbf{g_0}$ and $\mathbf{g_1}$ respectively. The evaluation of $x_0 = 1$, $x_1 = 0$, and $x_2 = 1$ in $\mathbf{f}$ ends at x_4-node in the cut_set. The evaluation of the same pattern 101 in $\mathbf{g_0}$ and $\mathbf{g_1}$ produce function values 0 and 1 for new variables g_0 and g_1. Then, the evaluation of 01 in $\mathbf{f}'$ also ends at the same x_4-node.

Because there is no encoding 11 in the cut_set, variables g_0 and g_1 can never be 11. We can assign arbitrary value for this pattern. In this example, we assign the left x_3-node so that the left g_1-node can be reduced in $\mathbf{f}'$. □

Lemma 6.1.1 Given an OBDD $\mathbf{v_f}$ with variable ordering $x_0 < \ldots < x_{n-1}$ representing $f(x_0, \ldots, x_{n-1})$, a bound set $B = \{x_0, \ldots, x_{i-1}\}$, the $cut\_set(\mathbf{v_f}, B) = \{\mathbf{u_0}, \ldots, \mathbf{u_{k-1}}\}$, and algorithm *decompose* returning OBDDs $\mathbf{v_{f'}}, \mathbf{v_{g_0}}, \ldots, \mathbf{v_{g_{j-1}}}$,

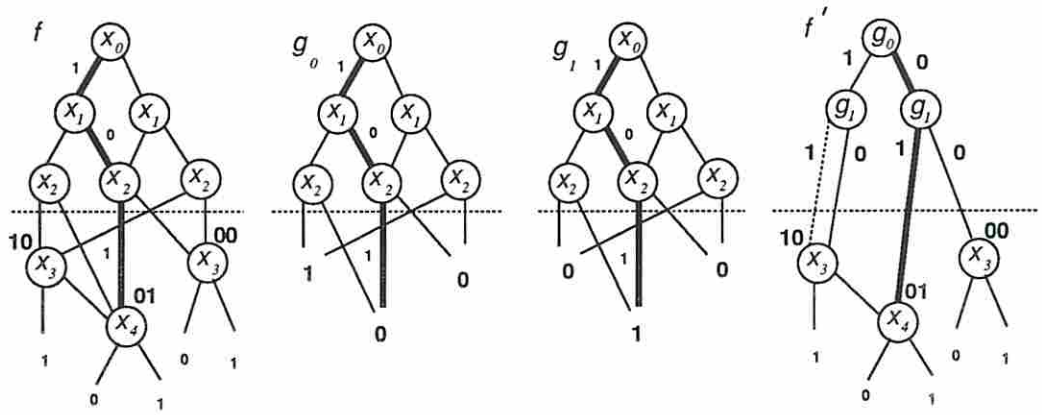


Figure 6.3: An example of disjunctive decomposition.

then

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

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

Proof: Consider the behavior of an input pattern $\langle b_0, \ldots, b_{i-1} \rangle$ on $\mathbf{v_f}, \mathbf{v_{f'}}$ and $\mathbf{v_{gp}}$'s. Suppose $\mathbf{u_q}$ with encoding q is the node we reach in $\mathbf{v_f}$ using the input pattern, that is, $eval(\mathbf{v_f}, \langle b_0, \ldots, b_{i-1} \rangle) = \mathbf{u_q}$, where $\mathbf{u_q} = f(b_0, \ldots, b_{i-1}, x_i, \ldots, x_{n-1})$. Since $\mathbf{u_q}$ has been replaced by the pth bit of q in $\mathbf{v_{gp}}$, $eval(\mathbf{v_{gp}}, \langle b_0, \ldots, b_{i-1} \rangle) = \mathbf{b_{qp}}$, that is, $g_p(b_0, \ldots, b_{i-1}) = b_{q_p}$. Because of the way we construct $\mathbf{v_{f'}}$, $eval(\mathbf{v_{f'}}, \langle b_{q_0}, \ldots, b_{q_{j-1}} \rangle)$ $= \mathbf{u_q}$, that is, $f'(b_{q_0}, \ldots, b_{q_{j-1}}, x_i, \ldots, x_{n-1}) = \mathbf{u_q} = f(b_0, \ldots, b_{i-1}, x_i, \ldots, x_{n-1})$. Thus, $f'(g_0(b_0, \ldots, b_{i-1}), \ldots, g_{j-1}(b_0, \ldots, b_{i-1}), x_i, \ldots, x_{n-1}) = f(b_0, \ldots, b_{i-1}, x_i, \ldots, x_{n-1})$, for any input pattern in the bound set. □

6.2 Nondisjunctive Decomposition

Before describing how to perform nondisjunctive decomposition based on the OBDD representation, the concept of *cut_set* is extended in the following definition.

Definition 6.2.1 Let $R = \{x_0, \ldots, x_{s-1}\}$, $S = \{x_s, \ldots, x_{i-1}\}$, and $T = \{x_i, \ldots, x_{n-1}\}$, $0 < s < i < n$. Given an OBDD $\mathbf{v}$ representing $f(x_0, \ldots, x_{n-1})$, a bound set $R \cup S$, and a free set $S \cup T$, we define

$$cut\_set\_nd(\mathbf{v}, R, S, p) = \{eval(\mathbf{w}, p) \mid \mathbf{w} \in cut\_set(\mathbf{v}, R)\},$$

where $0 \leq p < 2^{|S|}$.

With the above definition, $cut\_set(\mathbf{v}, B)$ can be represented by $cut\_set\_nd(\mathbf{v}, B, \phi, 0)$. The following presents a pseudo code for computing cut_set_nd and an example of it.

```
cut_set_nd(v, R, S, p)        /* 0 ≤ p < 2^|S| */
{
  if (index(v) < index(head(S)))
    return(cut_set_nd(child_l(v), R, S, p) ∪ cut_set_nd(child_r(v), R, S, p));
  else if (index(head(S)) ≤ index(v) ≤ index(last(S))) {
    q = 2^(index(last(S)) − index(v));
    if (q ≤ p)        /* then traverse down through left edge */
      return(cut_set_nd(child_l(v), R, S, p − q));
    else return(cut_set_nd(child_r(v), R, S, p));
  }
  else return({v});
}
```

Example 6.2.1 The OBDD in Figure 6.1 (a) has

$$cut\_set\_nd(\mathbf{f}, \{x_0, x_1\}, \{x_2\}, 0) = \{\mathbf{a}, \mathbf{b}\},$$
$$cut\_set\_nd(\mathbf{f}, \{x_0, x_1\}, \{x_2\}, 1) = \{\mathbf{b}, \mathbf{c}\},$$
$$cut\_set\_nd(\mathbf{f}, \{x_0\}, \{x_1, x_2\}, 0) = \{\mathbf{a}\},$$
$$cut\_set\_nd(\mathbf{f}, \{x_0\}, \{x_1, x_2\}, 1) = \{\mathbf{b}, \mathbf{c}\},$$
$$cut\_set\_nd(\mathbf{f}, \{x_0\}, \{x_1, x_2\}, 2) = \{\mathbf{a}, \mathbf{b}\}, \text{ and}$$
$$cut\_set\_nd(\mathbf{f}, \{x_0\}, \{x_1, x_2\}, 3) = \{\mathbf{b}, \mathbf{c}\}. \qquad \square$$

Algorithm *decomp_nd* (pseudo code *decomp_nd*):

Given a function f represented in an OBDD $\mathbf{v_f}$, a bound set $\{x_0, \ldots, x_s, \ldots, x_{i-1}\}$, and a free set $\{x_s, \ldots, x_{i-1}, \ldots, x_{n-1}\}$, a nondisjunctive decomposition with respect to the given bound set and free set is carried out in the following steps:

1. Compute $cut\_set\_nd(\mathbf{v_f}, R, S, r)$ for $0 \leq r < 2^{|S|}$ where $R = \{x_0, \ldots, x_{s-1}\}$ and $S = \{x_s, \ldots, x_{i-1}\}$ (Fig. 6.4 (a)). Let $cut\_set\_nd(\mathbf{v_f}, R, S, r) = \{\mathbf{u_{r,0}}, \ldots, \mathbf{u_{r,l}}\}$, $max\{\mid cut\_set\_nd(\mathbf{v_f}, R, S, r) \mid\} = k$, and $j = \lceil \log_2 k \rceil$.

2. Construct $\mathbf{v_{f'}}$ to represent function f' in two steps (Fig. 6.4 (b) and pseudo code *decomp_f_nd*):

   (a) Construct $\mathbf{v_q}$, $0 \leq q < k$, such that $eval(\mathbf{v_q}, r) = \mathbf{u_{q,r}}$ where $\mathbf{u_{q,r}}$ is the qth element in $cut\_set\_nd(\mathbf{v_f}, R, S, r)$ or the last element if $q > \mid cut\_set\_nd(\mathbf{v_f}, R, S, r) \mid$.

   (b) Construct $\mathbf{v_{f'}}$ such that $eval(\mathbf{v_{f'}}, q) = \mathbf{v_q}$ for $0 \leq q < k-1$ and $eval(\mathbf{v_{f'}}, q) = \mathbf{v_{k-1}}$ for $k-1 \leq q < 2^j$.

3. Construct $\mathbf{v_{gp}}$'s to represent g_p's for $0 \leq p < j$ (Fig. 6.4 (c) and pseudo code *decomp_g_nd*):

   Replace each node $\mathbf{u_{q,r}}$ (q^{th} node of $cut\_set\_nd(\mathbf{v_f}, R, S, r)$) from $\mathbf{v_f}$ by the terminal node whose value is b_{q_p} where b_{q_p} is the pth bit from the least significant bit of integer q.

Note that, a node $\mathbf{u}$ may be the ith element of $cut\_set\_nd(\mathbf{v_f}, R, S, r_1)$ and the jth element of $cut\_set\_nd(\mathbf{v_f}, R, S, r_2)$ which requires different encodings for $\mathbf{u}$. This will not cause a problem because we can first duplicate the node $\mathbf{u}$ and then assign each copy a different encoding. An alternative statement is that if paths p_1 and p_2 both end at node $\mathbf{u}$ and require different encodings b_1 and b_2, then we let p_1 end at terminal node $\mathbf{b_1}$ and p_2 end at terminal node $\mathbf{b_2}$.

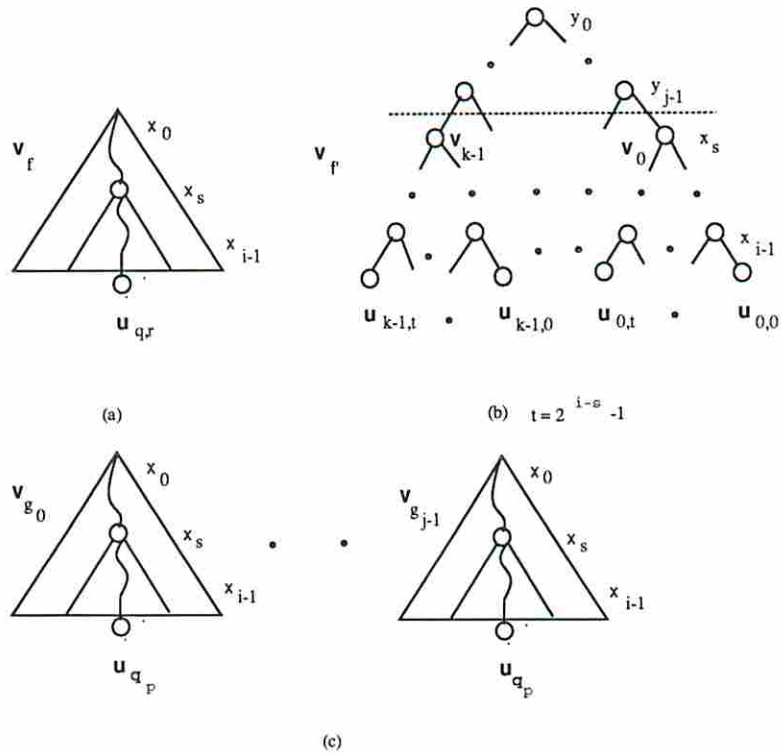


Figure 6.4: Nondisjunctive decomposition.

$decomp\_nd(\mathbf{v_f}, \{x_0, \ldots, x_{s-1}\}, \{x_s, \ldots, x_{i-1}\})$

$\{$

for $(r = 0; r < 2^{i-s}; r++)$

$\quad cutsetnd_r = cut\_set\_nd(\mathbf{v_f}, \{x_0, \ldots, x_{s-1}\}, \{x_s, \ldots, x_{i-1}\}, r)$

$k = max\{|\, cutsetnd_r \,|\};$

$j = \lceil \log_2 k \rceil;$

$\mathbf{v_{f'}} = decomp\_f\_nd(\{cutsetnd's\}, \{x_{i-1}, \ldots, x_s\}, \{y_{j-1}, \ldots, y_0\});$

for $(p = 0; p < j; p++)$

$\quad \mathbf{v_{gp}} = decomp\_g\_nd(\mathbf{v_f}, \{cutsetnd's\}, p);$

return $\mathbf{v_{f'}}$ and $\mathbf{v_{gp}}$'s;

$\}$

$decomp\_f\_nd(cutsetnds, idx, idy)$

$cutsetnds$:$\{cutsetnd_r \mid 0 \leq r < 2^{i-s}\}$ with $max\{\mid cutsetnd_r \mid\} = k$; and

$\lceil \log_2 k \rceil = j$;

$cutsetnd_{q,r}$:q^{th} element of $cutsetnd_r$, or the last element of $cutsetnd_r$

if $q > \mid cutsetnd_r \mid$;

$idx : \{x_{i-1}, \ldots, x_s\}$;

$idy : \{y_{j-1}, \ldots, y_0\}$;

return: an OBDD $\mathbf{v}_{f'}$ with variables $y_0, \ldots, y_{j-1}, x_s, \ldots, x_{i-1}, \ldots, x_{n-1}$;

```
{
    for (q = 0; q < k; q++)
      v_q = decomp_f({cutsetnd_{q,r} | 0 ≤ r < 2^{i-s}}, idx);
    v_f' = decomp_f({v_q | 0 ≤ q < k}, idy);
    return v_f';
}
```

$decomp\_g\_nd(\mathbf{v_f}, cutsetnds, p)$

return: $\mathbf{v_{gp}}$;

```
{
    for (r = 0; r < 2^{i-1}; r++) {
        if (eval(v_f, r) == cutsetnd_{q,p})
            let eval(v_gp, r) = bit(q, p);
    }
    return v_{g_p};
}
```

Example 6.2.2 One possible nondisjunctive decomposition of the OBDD in Fig. 6.1 (a) with respect to the bound set $\{x_0, x_1, x_2\}$ and the free set $\{x_2, x_3, x_4\}$ is shown in Fig. 6.5. In this decomposition, we use the following coding: {a =

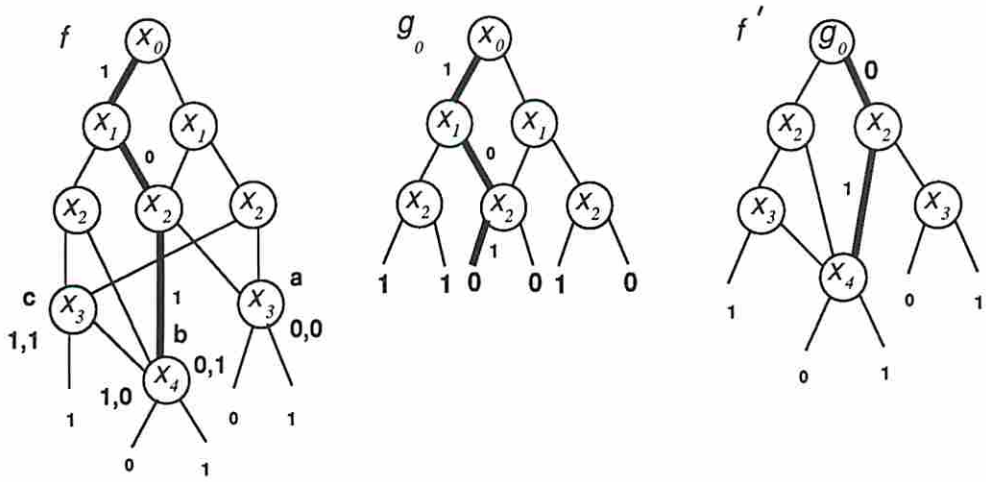


Figure 6.5: An example of nondisjunctive decomposition.

$\mathbf{u_{0,0}}, \mathbf{b} = \mathbf{u_{0,1}}\} = cut\_set\_nd(\mathbf{v_f}, \{x_0, x_1\}, \{x_2\}, 0)$ and $\{\mathbf{c} = \mathbf{u_{1,1}}, \mathbf{b} = \mathbf{u_{1,0}}\} = cut\_set\_nd(\mathbf{v_f}, \{x_0, x_1\}, \{x_2\}, 1)$. □

Lemma 6.2.1 Given an OBDD $\mathbf{v_f}$ with variable ordering $x_0, \ldots, x_{n-1}$ representing $f(x_0, \ldots, x_{n-1})$, and $k = max\{|\ cut\_set\_nd\ (\mathbf{v_f}, \{x_0, \ldots, x_{s-1}\}, \{x_s, \ldots, x_{i-1}\}, r)\ |\ 0 \leq r < 2^{i-s}\}$, $2^{j-1} < k \leq 2^j$, the algorithm *decomp_nd* returns $j+1$ OBDDs $\mathbf{v_{f'}}, \mathbf{v_{g_0}}, \ldots, \mathbf{v_{g_{j-1}}}$ such that $f(x_0, \ldots, x_{n-1}) = f'(g_0(x_0, \ldots, x_{i-1}), \ldots, g_{j-1}(x_0, \ldots, x_{i-1}), x_{s-1}, \ldots, x_{i-1}, \ldots, x_{n-1})$ where $f', g_0, \ldots, g_{j-1}$ are the functions denoted by $\mathbf{v_{f'}}, \mathbf{v_{g_0}}, \ldots, \mathbf{v_{g_{j-1}}}$, respectively.

Proof: Consider the behavior of an input pattern $\langle b_0, \ldots, b_s, \ldots, b_{i-1}\rangle$ on $\mathbf{v_f}, \mathbf{v_{f'}}$ and $\mathbf{v_{gp}}$'s. Let $eval(\mathbf{v_f}, \langle b_0, \ldots, b_s, \ldots, b_{i-1}\rangle) = \mathbf{u_{q,r}}$, where $r = 2^{i-s-1}b_s + \ldots + 2^0 b_{i-1}$. Since $\mathbf{u_{q,r}}$ has been replaced by $\mathbf{b_{qp}}$ in $\mathbf{v_{gp}}$, we have $eval(\mathbf{v_{gp}}, \langle b_0, \ldots, b_s, \ldots, b_{i-1}\rangle) = b_{q_p}$. Then from $eval(\mathbf{v_{f'}}, \langle b_{q_0}, \ldots, b_{q_{j-1}}\rangle) = \mathbf{v_q}$ and $eval(\mathbf{v_q}, r) = \mathbf{u_{q,r}} = eval(\mathbf{v_q}, \langle b_s, \ldots, b_{i-1}\rangle)$, we have

$eval(\mathbf{v_{f'}}, \langle b_{q_0}, \ldots, b_{q_{j-1}}, b_s, \ldots, b_{i-1}\rangle) = \mathbf{u_{q,r}} = eval(\mathbf{v_f}, \langle b_0, \ldots, b_s, \ldots, b_{i-1}\rangle)$. □

6.3 Cut_set In Place

This section considers only disjunctive decomposition. It first presents an algorithm for computing an *encoded decomposition chart* from the truth table of a function. It then shows a similar algorithm which computes cut_sets without moving bound variables to the top of OBDDs.

Definition 6.3.1 Given a function f with n variables and a bound set B with size $i < n$, the *encoded decomposition chart* of f with respect to B is a vector c with length 2^i where each distinct element of c corresponds to a distinct column of the decomposition chart of f.

Definition 6.3.2 Operator *encode* is defined as follows:

$$encode(\langle v_{0,0}, \ldots, v_{0,2^i-1}\rangle, \ldots, \langle v_{m-1,0}, \ldots, v_{m-1,2^i-1}\rangle = \langle v_{m,0}, \ldots, v_{m,2^i-1}\rangle$$

where $v_{m,0} = 0$, and $v_{m,p} = q$ if $\langle v_{0,p}, \ldots, v_{m-1,p}\rangle$ is the qth distinct m-tuple of $\langle v_{0,0}, \ldots, v_{m-1,0}\rangle, \ldots, \langle v_{0,2^i-1}, \ldots, v_{m-1,2^i-1}\rangle$.

Example 6.3.1 The decomposition chart shown in Fig. 6.6 (a) has x_i and x_j as the free set. We show how to convert the decomposition chart into an encoded decomposition chart one variable at a time. With respect to variable x_i, we encode the first two rows and the last two rows in Fig. 6.6 (a) by:

$$encode(\langle 1,1,1,0,1,1,1,0\rangle, \langle 1,1,1,0,1,0,1,0\rangle) \quad \text{and}$$
$$encode(\langle 0,0,1,1,0,0,1,0\rangle, \langle 0,0,1,0,1,0,1,1\rangle), \quad \text{respectively.}$$

The result is shown in Fig. 6.6 (b). With respect to variable x_j, we carry out $encode(\langle 0,0,0,1,0,2,0,1\rangle, \langle 0,0,1,2,3,0,1,3\rangle)$ and derive the encoded decomposition chart as shown in Fig. 6.6 (c). □

Definition 6.3.3 An *encoded decomposition vector* (*edv*) is a vector

$$\langle [e_{0,0}, \ldots, e_{0,2^j-1}], \ldots, [e_{2^i-1,0}, \ldots, e_{2^i-1,2^j-1}]\rangle$$

where $j \geq 0, i \geq 0, e_{k,l}$ is an integer. Each element of an *edv*, $[e_{k,0}, \ldots, e_{k,2^j-1}]$, is called an *encoded decomposition column* (*edc*).

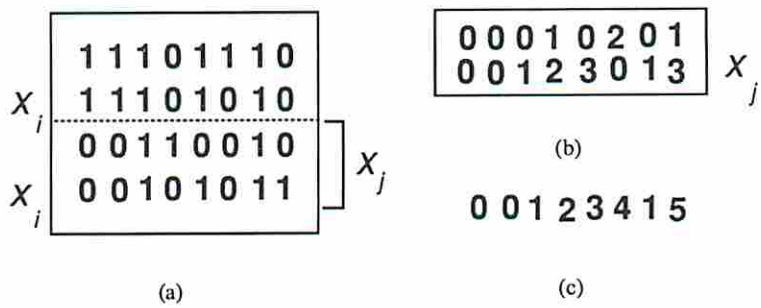


Figure 6.6: An example of coded decomposition chart.

Pseudo code *gen_edc* constructs the encoded decomposition chart of a given function and bound set as described before. Initially, we have an *edv* containing only one *edc* which is the truth table representation of the given function (line 1). Each variable, from x_0 to x_{n-1} (line 2), is processed in the following way: If x_p is in the bound set, we use *include* to double the number of *edc*'s by breaking each *edc* into two *edc*'s such that each new *edc* corresponds to either $x_p = 0$ or $x_p = 1$ (line 3); otherwise, we use *exclude* to cut the size of each *edc* and to perform the encode operation in the same way as in Ex. 6.3.1 (line 4). After all variables have been processed, the number of *edc*'s will be 2^i where i is the size of the bound set (due to the calling of *include* i times), and each *edc* contains only one element (due to the calling of *include* and *exclude* n times in total). We then *flatten* the *edv* to get the encoded decomposition chart.

$gen\_edc(f, B)$

f: a Boolean function with n variables represented by the truth table form as $[m_0, \ldots, m_{2^n-1}]$ where $m_0, m_1, \ldots, m_{2^{n-1}-1}$ are the minterms correspond to $x_0 = 0$ and $m_0, m_2, \ldots, m_{2^n-2}$ are the minterms correspond to $x_{n-1} = 0$;

B: a bound set;

```
{
1     edv = ⟨[m_0, ..., m_{2^n-1}]⟩;
2     for (p == 0; p++; p < n) {
3         if (x_p ∈ B) edv = include(edv);
4         else edv = exclude(edv);
5     }
6     return(flatten(edv));
}
```

$include(\langle[e_{0,0}, \ldots, e_{0,2^j-1}], \ldots, [e_{2^i-1,0}, \ldots, e_{2^i-1,2^j-1}]\rangle)$

$\{$

$\quad return(\langle[e_{0,0}, \ldots, e_{0,2^{j-1}-1}], [e_{0,2^{j-1}}, \ldots, e_{0,2^j-1}], \ldots,$

$\qquad [e_{2^i-1,0}, \ldots, e_{2^i-1,2^{j-1}-1}], [e_{2^i-1,2^{j-1}}, \ldots, e_{2^i-1,2^j-1}]\rangle);$

$\}$

$exclude(\langle[e_{0,0}, \ldots, e_{0,2^j-1}], \ldots, [e_{2^i-1,0}, \ldots, e_{2^i-1,2^j-1}]\rangle)$

$\{$

$\quad \langle d_{0,0}, \ldots, d_{0,2^{j-1}-1}, d_{1,0}, \ldots, d_{1,2^{j-1}-1}, \ldots, d_{2^i-1,0}, \ldots, d_{2^i-1,2^{j-1}-1}\rangle =$

$\qquad encode(\langle e_{0,0}, \ldots, e_{0,2^{j-1}-1}, e_{1,0}, \ldots, e_{1,2^{j-1}-1}, \ldots, e_{2^i-1,0}, \ldots, e_{2^i-1,2^{j-1}-1}\rangle,$

$\qquad\quad \langle e_{0,2^{j-1}}, \ldots, e_{0,2^j-1}, e_{1,2^{j-1}}, \ldots, e_{1,2^j-1}, \ldots, e_{2^i-1,2^{j-1}}, \ldots, e_{2^i-1,2^j-1}\rangle);$

$\quad return(\langle[d_{0,0}, \ldots, d_{0,2^{j-1}-1}], [d_{1,0}, \ldots, d_{1,2^{j-1}-1}], \ldots, [d_{2^i-1,0}, \ldots, d_{2^i-1,2^{j-1}-1}]\rangle);$

$\}$

$$
\begin{aligned}
&flatten(\langle [e_0], \ldots, [e_{2^i-1}] \rangle) \\
&\{ \\
&\qquad \text{return}(\langle e_0, \ldots, e_{2^i-1} \rangle); \\
&\}
\end{aligned}
$$

Example 6.3.2 The application of *gen_edc* to the function in Ex. 6.1.1 with bound set $\{x_0, x_2, x_4\}$ is summarized below:

1. Initially, we have $\langle$[1100 1011 1100 1010 1100 1010 1010 1011]$\rangle$.

2. Since x_0 is in the bound set, we execute *bound_v* and have $\langle$[1100 1011 1100 1010], [1100 1010 1010 1011]$\rangle$.

3. After applying $free\_v$ on x_1, we have

$$
\begin{array}{lllll}
V_0: & 1100\ 1011 & 1100\ 1010 & & \\
V_1: & 1100\ 1010 & 1010\ 1011 & \Rightarrow & \langle [0011\ 0102], [0231\ 0103] \rangle. \\
C: & 0011\ 0102 & 0231\ 0103 & &
\end{array}
$$

4. After processing x_2, we have $\langle [0011], [0102], [0231], [0103] \rangle$.

5. The computation of $free\_v$ on x_3 is:

$$
\begin{array}{lllllll}
V_0: & 00 & 01 & 02 & 01 & & \\
V_1: & 11 & 02 & 31 & 03 & \Rightarrow & \langle [00], [12], [34], [15] \rangle. \\
C: & 00 & 12 & 34 & 15 & &
\end{array}
$$

6. After the processing of x_4 and $flatten$, we have the encoded decomposition chart $\langle 0, 0, 1, 2, 3, 4, 1, 5 \rangle$ which implies that there are 6 different distinct columns in the decomposition chart as shown in Fig. 6.7.

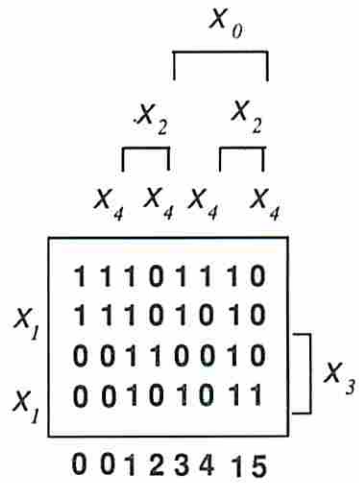


Figure 6.7: The decomposition chart with respect bound set $\{x_0, x_2, x_4\}$.

The following pseudo code *cut_set_ip* is the OBDD version of *gen_edc*.

```
cut_set_ip(node_vector, B, p)
{
1   if (p > index(last(B))) return(node_vector);
2   if (x_p ∈ B) return(include_obdd(node_vector, B, p));
3   else return(exclude_obdd(node_vector, B, p));
}
```

```
include_obdd(⟨v_0, ..., v_{2^i-1}⟩, B, p)
{
  return(cut_set_ip(⟨r_child(v_0, p), l_child(v_0, p), ..., r_child(v_{2^i-1}, p),
                    l_child(v_{2^i-1}, p)⟩, B, p + 1));
}
```

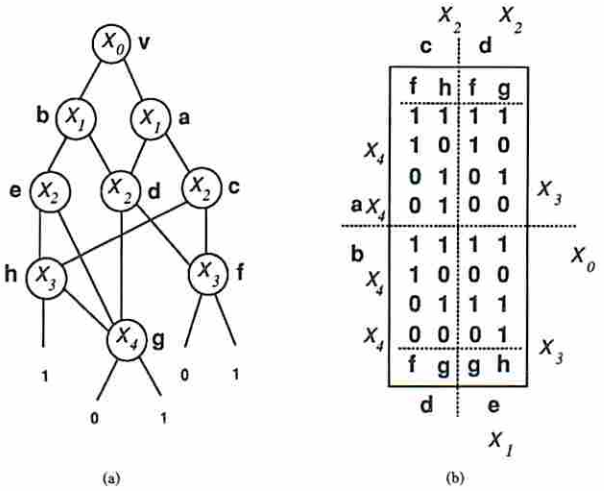


Figure 6.8: An example for cut_set in place.

$exclude\_obdd(\langle \mathbf{v_0}, \ldots, \mathbf{v_{2^i-1}} \rangle, B, p)$

$\{$

$\quad \text{return}(encode(cut\_set\_ip(\langle r\_child(\mathbf{v_0}, p), \ldots, r\_child(\mathbf{v_{2^i-1}}, p)\rangle, B, p+1),$

$\qquad\qquad cut\_set\_ip(\langle l\_child(\mathbf{v_0}, p), \ldots, l\_child(\mathbf{v_{2^i-1}}, p)\rangle, B, p+1)));$

$\}$

Example 6.3.3 The execution of *cut_set_ip* on the OBDD shown in Fig. 6.8 (a) with bound set $B = \{x_1, x_2\}$ is summarized below:

initial:	$cut\_set\_ip(\langle \mathbf{v} \rangle, B, 0)$,
$p = 0$:	$encode(cut\_set\_ip(\langle \mathbf{a} \rangle, B, 1), cut\_set\_ip(\langle \mathbf{b} \rangle, B, 1))$,
$p = 1$:	$encode(cut\_set\_ip(\langle \mathbf{c}, \mathbf{d} \rangle, B, 2), cut\_set\_ip(\langle \mathbf{d}, \mathbf{e} \rangle, B, 2))$,
$p = 2$:	$encode(cut\_set\_ip(\langle \mathbf{f}, \mathbf{h}, \mathbf{f}, \mathbf{g} \rangle, B, 3), cut\_set\_ip(\langle \mathbf{f}, \mathbf{g}, \mathbf{g}, \mathbf{h} \rangle, B, 3))$,
$p = 3$:	$encode(\langle \mathbf{f}, \mathbf{h}, \mathbf{f}, \mathbf{g} \rangle, \langle \mathbf{f}, \mathbf{g}, \mathbf{g}, \mathbf{h} \rangle)$,
result:	$\langle 0, 1, 2, 3 \rangle$.

The decomposition chart is shown in Fig. 6.8 (b). □

6.4 Computing Cut_sets for All Possible Bound Sets

The previous sections showed how to perform function decomposition directly on OBDDs when a bound set is given. This section will show how to compute the *cut_sets* for all bound sets of (single-output) Boolean functions using EVBDDs. The method is based on the encoding of columns as integers.

Initially, every variable is in the free set. For each variable x_i, we perform the following two operations:

1. *include*: include x_i in the bound set to derive a new cut_set, and
2. *exclude*: partially encode the columns such that distinct columns are given unique codes and variable x_i is permanently excluded from the bound set.

Example 6.4.1 Fig. 6.9 (a) shows a decomposition chart where variable x is in the free set and a, b, c, d, e, f, and g are Boolean values. To perform the *include* operation, we move the two bottom rows to the left of the top two rows such that x now is in the bound set (Fig. 6.9 (b)). To perform the *exclude* operation, we encode bit vectors $\langle c, a\rangle$, $\langle d, b\rangle$, $\langle g, e\rangle$, and $\langle h, f\rangle$ as $2c + a$, $2d + b$, $2g + e$, and $2h + f$, respectively (Fig. 6.9 (c)). The coded decomposition chart preserves the distinctness of columns, that is, column $\langle a, b, c, d\rangle$ is distinct from column $\langle e, f, g, h\rangle$ if and only if column $\langle 2c + a, 2d + b\rangle$ is distinct from column $\langle 2g + e, 2h + f\rangle$. Furthermore, variable x is absent from the encoded decomposition chart and will never be included in the bound set. □

Given an EVBDD **v** with the top variable x_i, the right and left children of **v** correspond to the top and bottom halves of rows in the decomposition chart. Thus, operations *include* and *exclude* in the EVBDD representation are carried out in the following way:

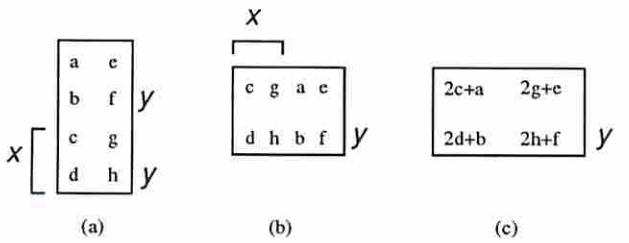


Figure 6.9: Operations *include* and *exclude* in the decomposition chart.

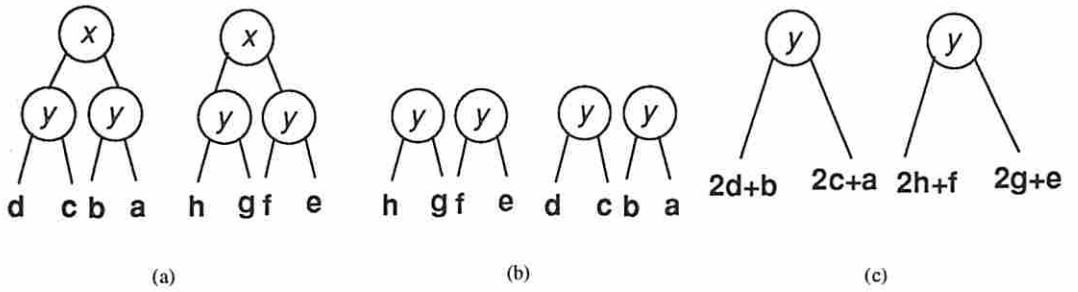


Figure 6.10: Operations *include* and *exclude* in the EVBDD representation.

1. *include*: construct the set $\{child_l(\mathbf{v}), child_r(\mathbf{v})\}$, and
2. *exclude*: construct an EVBDD representing $2^{2^i} \times child_l(\mathbf{v}) + child_r(\mathbf{v})$ where 2^{2^i} is to ensure that the resulting EVBDD has a unique encoded representation.

Example 6.4.2 Fig. 6.10 (a) is the EVBDD representation of the decomposition chart in Fig. 6.9 (a). The corresponding operations *include* and *exclude* are shown in Fig. 6.10 (b) and (c), respectively. □

Pseudo code *cut_set_all* computes the cardinality of the cut_set for every possible bound set of a given function. The routine returns the set $\{\langle b, k\rangle \mid b$ is a bound set and k is the cardinality of the cut_set of $b\}$. Initially, we have $i = 0$ and $node\_set = \{\mathbf{v}\}$ where $\mathbf{v}$ is the EVBDD representing the given function. This corresponds to the bound set $B = \phi$ and free set X where X is the set of input variables. If $i = n$, then we reach the terminal case and *node_set* is the (encoded) cut_set for bound set B (line 1); otherwise, we perform *include* and *exclude* operations with respect to variable x_i (lines 2 and 3). We repeat the process for variable

x_{i+1} in lines 4 and 5. In line 6, the union of $\langle b, k \rangle$'s from lines 4 and 5 is returned. Pseudo code *include* and *exclude* perform the include and exclude operations for a set of EVBDD nodes.

```
cut_set_all(node_set, B, i)
{
1    if (i == n) return({⟨B, | node_set |⟩});
2    inc_set = include(node_set, i);
3    exc_set = exclude(node_set, i);
4    inc = cut_set_all(inc_set, B ∪ {x_i}, i + 1);
5    exc = cut_set_all(exc_set, B, i + 1);
6    return(inc ∪ exc);
}
```

```
include(node_set, i)
{
1    new_set = φ;
2    for each node u ∈ node_set {
3        if (index(variable(u)) == i)
4            new_set = new_set ∪ {child_l(u), child_r(u)};
5        else /* index(u) > i */
6            new_set = new_set ∪ {u};
7    }
8    return new_set;
}
```

```
exclude(node_set, i)
{
1    new_set = φ;
2    for each node u ∈ node_set {
3        if (index(variable(u)) == i)
4            new_set = new_set ∪ {2^(2^i) × child_l(u) + child_r(u)};
5        else /* index(u) > i */
6            new_set = new_set ∪ {2^(2^i) × u + u};
7    }
8    return new_set;
}
```

Example 6.4.3 Fig. 6.11 (a) shows a function represented by both a truth table and a flattened EVBDD. Initially, the bound set is empty. The application of *include* and *exclude* with respect to variable x_0 are shown in Fig. 6.11 (b) and (c), respectively. In Fig. 6.11 (b), the bound set is $\{x_0\}$ and the cardinality of the cut_set is 2; In Fig. 6.11 (c), the bound set is ϕ with cut_set size 1.

The application of *include* and *exclude* on Fig. 6.11 (b) with respect to variable x_1 results in Fig. 6.12 (a) and (b) with bound sets $\{x_0, x_1\}$ and $\{x_0\}$ and cut_set sizes 2 and 2, respectively. The application of *include* and *exclude* on Fig. 6.11 (c) with respect to variable x_1 results in Fig. 6.12 (c) and (d) with bound sets $\{x_1\}$ and ϕ and cut_set sizes 2 and 1, respectively.

The application of *include* and *exclude* on Fig. 6.12 (b) with respect to variable x_2 results in Fig. 6.13 (a) and (b) with bound sets $\{x_0, x_2\}$ and $\{x_0\}$ and encoded cut_sets $\{\mathbf{0}, \mathbf{1}, \mathbf{5}, \mathbf{4}\}$ and $\{\mathbf{5}, \mathbf{12}\}$, respectively. In Fig. 6.13, the top row shows the encoded decomposition charts, the second row shows the encoded cut_sets, and the third row shows the decomposition charts. The encodings used for the bottom row in Fig. 6.13 (a) and (b) are $1 \times row1 + 4 \times row2$ and $1 \times row1 + 2 \times row2 + 4 \times row3 + 8 \times row4$, respectively. □

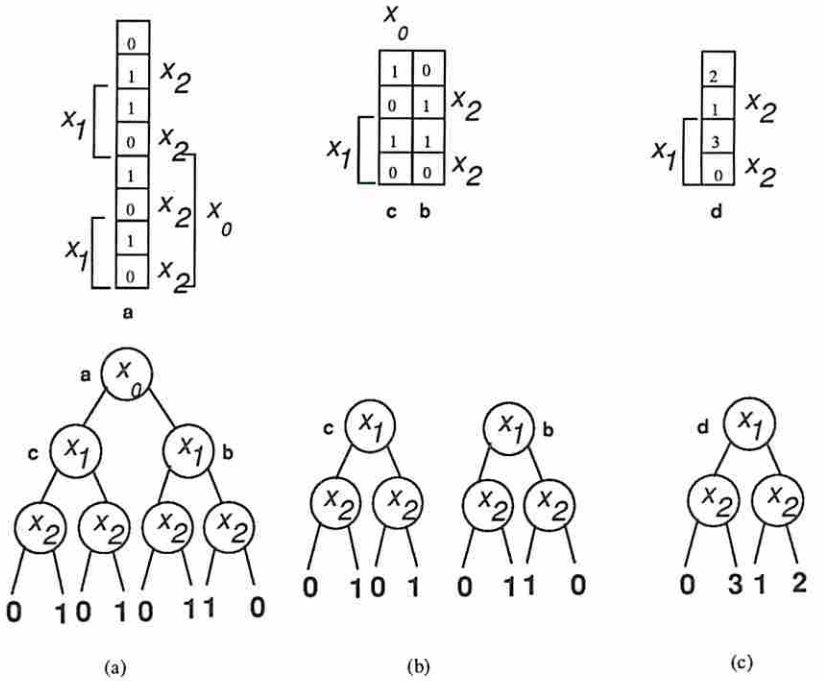


Figure 6.11: An example of the application of *cut_set_all*.

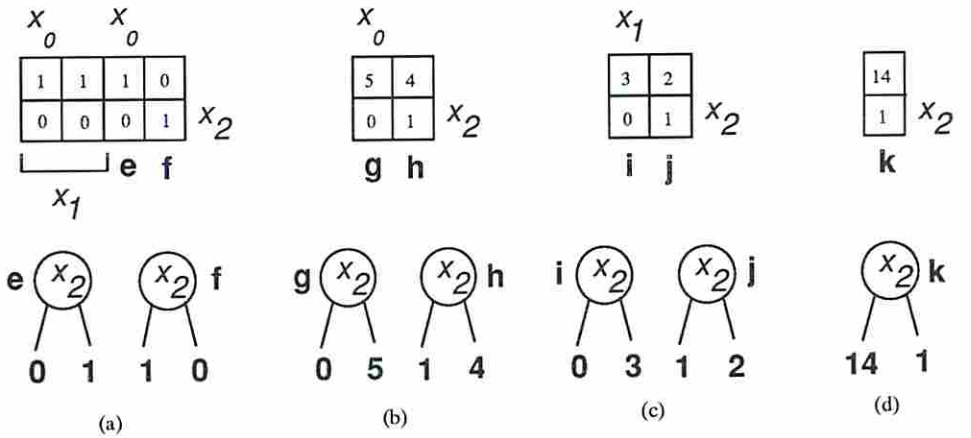


Figure 6.12: Example continued.

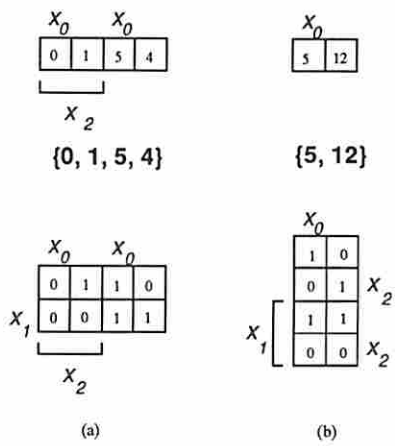


Figure 6.13: Example continued.

Since there are 2^n different bound sets for an n variable function, the computation of the *cut_set* for every bound set is very expensive. If we replace line 1 in *cut_set_all* by

$$1 \text{ if } (level == n \mid\mid \mid var\_set \mid \leq k) \text{ return}(\{\langle B, \mid node\_set \mid\rangle\}),$$

then *cut_set_all* becomes a routine for computing the cardinality of the cut_set for every bound set whose size is less than or equal to k which is useful for the technology mapping of k-input look-up table field programmable gate arrays [58].

A naive way to compute the *cut_set* for every bound set is to move bound variables to the top of the EVBDD. Compared to this approach, the above approach has the following advantages. First, it is well known that the size of OBDD (and EVBDD) is very sensitive to the variable ordering (at least in many practical applications [66]). Moving bound variables to the top will change the variable ordering and hence may cause storage problems. The EVBDD-based method will not change the variable ordering. Second, the number of variables in the direct variable exchange approach is never reduced. In contrast, in EVBDD-based method, after the *include* and *exclude* operations, the number of variables will be decreased by 1.

6.5 Multiple-Output Decomposition

Given a vector of Boolean functions $f_0, \ldots, f_{m-1}$ on n variables, we can not extend the concept of *cut_set* by taking the set union of the *cut_sets* for the individual functions. To see this, consider the two extreme cases $\bigcup_{i=0}^{m-1} cut\_set(\mathbf{v}_i, 0)$ (bound set size 1) and $\bigcup_{i=0}^{m-1} cut\_set(\mathbf{v}_i, n-1)$ (bound set size n). The cardinality of the former might be greater than 2. This implies that more than two distinct functions must be implemented by a single variable which is not possible. On the other hand, the *cut_set* of the latter is always $\{0, 1\}$. This implies that m Boolean functions could be implemented by only one output line which is again not possible. This is because even when an OBDD node is shared by i^{th} and j^{th} functions, it should be treated differently.

While a problem with finite domain can be solved by conversion to Boolean functions, a problem related to multiple-output Boolean functions can also be solved by interpreting them as the bit representation of an integer function. For example, a multiple-output Boolean function $\langle f_0, \ldots, f_{m-1} \rangle$ can be transformed to an integer function F by $F = 2^{m-1} f_0 + \ldots + 2^0 f_{m-1}$. Based on this formulation, we show how to perform multiple-output decomposition using EVBDDs. We first extend the definitions of function decomposition and cut_sets to EVBDD representation. We then develop an EVBDD-based disjunctive decomposition algorithm.

Definition 6.5.1 A pseudo Boolean function $f(x_0, \ldots, x_{n-1})$ is said to be *decomposable* under *bound set* $\{x_0, \ldots, x_{i-1}\}$ and *free set* $\{x_i, \ldots x_{n-1}\}$, $0 < i < n$, if f can be transformed to $f'(g_0(x_0, \ldots, x_{i-1}), \ldots, g_j(x_0, \ldots, x_{i-1}), x_i, \ldots, x_{n-1})$ such that the number of inputs to f' is smaller than that of f. If j equals 1, then it is *simply decomposable.*

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

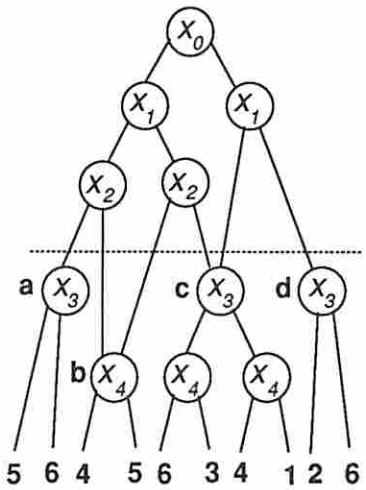


Figure 6.14: An example of cut_set in EVBDD.

Definition 6.5.2 Given an EVBDD $\langle c, \mathbf{v}\rangle$ representing $f(x_0, \ldots, x_{n-1})$ with variable ordering $x_0 < \ldots < x_{n-1}$ and bound set $B = \{x_0, \ldots, x_i\}$, we define

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

For readability, the flattened form of EVBDDs is used in this section.

Example 6.5.1 Given a function f as shown in Fig. 6.14 with bound set $B = \{x_0, x_1, x_2\}$, $cut\_set(f, B) = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$. □

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

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

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

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

4. Construct $\mathbf{v_{f'}}$ to represent function f':

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

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

Lemma 6.5.1 Given an EVBDD $\mathbf{v_f}$ with variable ordering $x_0 < \ldots < x_{n-1}$ representing $f(x_0, \ldots, x_{n-1})$, a bound set $B = \{x_0, \ldots, x_{i-1}\}$ and $cut\_set\_ev(\mathbf{v_f}, B) = \{\mathbf{u_0}, \ldots, \mathbf{u_{k-1}}\}$, if *decomp_ev* returns EVBDDs $\mathbf{v_{f'}}, \mathbf{v_{g_0}}, \ldots, \mathbf{v_{g_{j-1}}}$, then

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

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

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

Example 6.5.2 Fig. 6.15 shows an example of disjunctive decomposition by using EVBDDs. The evaluation of the input pattern $x_0 = 1$, $x_1 = 0$, and $x_2 = 1$ in function F will end at the leftmost x_4-node which has encoding 10. The evaluation of the same input pattern in functions g_0 and g_1 would produce function values 1 and

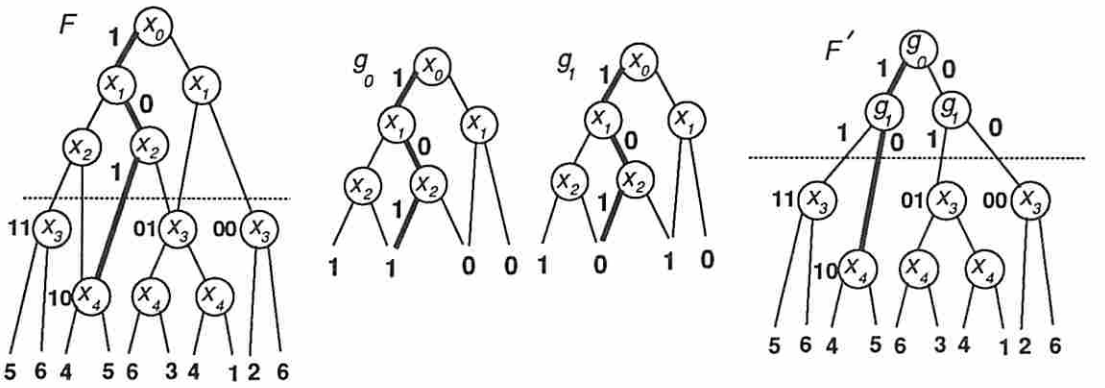


Figure 6.15: An example of disjunctive decomposition in EVBDD.

0. Then, with g_0 being 1 and g_1 being 0 in function F', it would also end at the leftmost x_4-node. □

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

Example 6.5.3 A 3-output Boolean function as shown in Fig 6.16 (a) can be converted into an integer function as shown in Fig. 6.16 (b) through $f = 4f_0 + 2f_1 + f_2$. The application of *decomp_ev* on F is the one shown in the previous example. After applying the synthesis paradigm described in Sect. 3.1 on F', we can convert f back to a 3-output Boolean function f'_0, f'_1, and f'_2. □

6.6 Incompletely Specified Functions

When functions are incompletely specified, the detection of decomposability becomes very complicated. For example, the compatibility in the Roth-Karp method is no longer an equivalence relation. The determination of k compatible classes

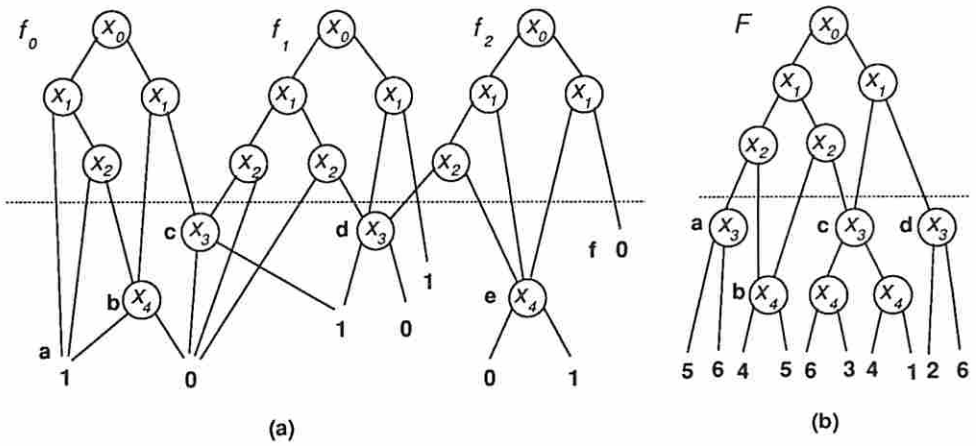


Figure 6.16: Representation of multiple-output functions.

then requires solving the minimum clique covering problem for the compatibility graph which is NP-hard [42].

A similar task has to be performed on OBDDs. We first extend OBDDs to include a third terminal node **dc** to represent the constant function *dc*. Next, we compute the *cut_set* as before. Since each node in *cut_set* may represent an incompletely specified function, we need to compute the compatibility between any two nodes in the *cut_set* so that a minimal k can be found. The determination of compatibility between two OBDD nodes, or the compatibility between their corresponding functions, can be carried out by algorithm *is_compatible*. After this step, the construction of compatibility graph and the computation of minimum clique cover is the same as in the Roth-Karp algorithm.

```
is_compatible(f, g)        /* f and g are OBDDs with dc */
{
1   if (f == dc || g == dc || f == g)
      return 1;
2   if (f == 0 && g == 1 || f == 1 && g == 0)
      return 0;
3   g_l = l_child(g, min(index(f), index(g)));
4   g_r = r_child(g, min(index(f), index(g)));
5   f_l = l_child(f, min(index(f), index(g)));
6   f_r = r_child(f, min(index(f), index(g)));
7   }
8   else { f_l = f_r = f; }
9   return(is_compatible(f_l, g_l) && is_compatible(f_r, g_r));
}
```

6.7 Experimental Results

The algorithm *cut_set_all* has been implemented and compared with the Roth-Karp decomposition algorithm implemented in SIS. In particular, the following command is used on a number of mcnc91 benchmark sets:

"xl_k_decomp[1] -n 4 -e -d -f 100" which for every node in the Boolean network, finds the best bound set of size ≤ 4 that reduces the node's variable support after decomposition, and then decomposes the node, and modifies the network to reflect the change.

An equivalent EVBDD-based command was implemented.

[1] *xl_k_decomp* does not process circuits with ≥ 32 inputs.

To assign a unique encoding for each EVBDD node, we need integers with 2^i bits where i is the number of variables considered so far. This is clearly very expensive. One way to overcome this difficulty is to relax the uniqueness condition (e.g, use 2 instead of 2^{2^i}). Then, two different EVBDD nodes representing different functions may be assigned the same encoding. As a result, the size of the cut_set for a given bound set may be underestimated. This scheme may be used as a filter. For example, to find the bound set which has the smallest cut_set, we first perform *cut_set_all* to find the best ones, and then check for the real cut_sets by moving the bound variables to the top of the EVBDD.

Results shown in Table 6.1 used 2 as the weight in *exclude* operation. The processes were stopped when they took more than 5000 cpu seconds on a Sun Sparc-Station II with 64 MB of memory. The EVBDD-based approach obtains significant speed-ups (by an average factor of 35.4).

Circuit	karp	EVBDD	Speed-up
5xp1	31.1	2.9	10.7
9sym	513.6	3.7	138.8
apex4	2544.8	49.8	51.1
misex3	> 5000	381.9	13.1
misex3c	> 5000	131.8	37.9
Z5xp1	31.4	4.3	7.3
misex2	416.8	87.7	4.8
sao2	337.9	23.6	14.3
xor5	2.0	0.3	6.7
b12	74.3	6.4	11.6
ex1010	> 5000	48.5	> 103.1
squar5	4.3	0.9	4.8
Z9sym	508.2	4.2	121.0
t481	> 5000	62.1	> 80.5
alu4	> 5000	124.5	> 40.2
table3	> 5000	318.6	> 15.7
table5	> 5000	1225.8	> 4.1
cordic	> 5000	1331.7	> 3.8
vg2	> 5000	2051.3	> 2.4
seq	NA	> 5000	—
apex1	NA	> 5000	—
apex2	NA	> 5000	—
apex3	NA	> 5000	—
e64	NA	> 5000	—
Average			35.4

Table 6.1: Finding all decomposable forms with bound set size ≤ 4.

Chapter 7

Conclusions

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

Because of the compactness and canonical properties, EVBDDs were shown to be effective for handling verification problems. An extension of EVBDDs, denoted by SEVBDDs, were successfully used to model conditional statements and array data structure. A future extension is to model loop statements and recursive functions which can then be used for verifying sequential machines. Because of the additive property, EVBDDs are also useful for solving integer linear programming problems. A possible improvement is to incorporate linear programming techniques (e.g., simplex method) in order to obtain better bounds and hence improve the runtime of the procedure.

An OBDD-based Boolean matching algorithm using a level-first search strategy was presented. Unlike depth-first and breadth-first strategies, a level-first strategy permits significant pruning of the search space. In addition, a set of filters which further improve the the performance of the matching algorithm was presented

Future research directions include the matching of arithmetic functions and finite state machines. Again, the key issue is to identify properties which can work as filters for pruning the search spaces.

Because OBDDs are a compact and canonical representation of Boolean functions, the function decomposition algorithms presented here are more efficient than the cube-based function decomposition implementation. However, the OBDD-based extraction of common sublogic to minimize the literal cost of the circuits remains an open problem to be studied in future.

Reference List

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

[2] E. Balas, "An Additive Algorithm for Solving Linear Programs with Zero-One Variables," *Operations Research*, 13 (4), pp. 517-546, 1965.

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

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

[5] J. C. Barros and B. W. Johnson, "Equivalence of the Arbiter, the Synchronizer, the Latch and the Inertial Delay," *IEEE Trans. on Computers*, C-32(7), July 1983.

[6] H. G. Barrow, "VERIFY: A Program for Proving Correctness of Digital Hardware Designs," *Artificial Intelligence*, 24:437-491, 1984.

[7] M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere and O. Vincent, "Experiments in Mixed-Integer Linear Programming," *Math. Programming 1*, pp. 76-94, 1971.

[8] M. Beardslee, B. Lin, and A. Sangiovanni-Vincentelli, "Communication Based Logic Partitioning," *Proc. of the European Design Automation Conf.*, pp. 32-37, 1992.

[9] G. V. Bochmann, "Hardware Specification with Temporal Logic: An Example," *IEEE Trans. on Computers*, 31(3):223-231, March 1982.

[10] R. T. Boute, "Representational and Denotational Semantics of Digital Systems," *IEEE Trans. on Computers*, 38(7):986-999, July 1989.

[11] V. J. Bowman, Jr. and J. H. Starr, "Partial Orderings in Implicit Enumeration," *Annals of Discrete Mathematics*, 1, North-Holland, pp. 99-116, 1977.

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

[13] R. Brayton and C. McMullen, "The Decomposition and Factorization of Boolean Expressions," *Proc. Int. Sym. Circ. Syst. (ISCAS-82)*, pp. 49-54, May 1982.

[14] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization ALgorithms for VLSI Synthesis*, Boston: Kluwer Academic Publishers, 1984.

[15] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: Multiple-Level Interactive Logic Optimization System," *IEEE Trans. Comput. Aided Design*, vol. CAD-6, no. 6, pp. 1062-1081, Nov. 1987.

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

[17] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *Computing Surveys*, Vol. 24, No. 3, pp. 293-318, Sept. 1992.

[18] J. R. Burch and D. E. Long, "Efficient Boolean Function Matching," *Int. Conf. Computer-Aided Design*, pp. 408-411, 1992.

[19] A. Camilleri, M. Gordon, and T. Melham, "Hardware Verification Using Higher-Order Logic," *Technical Report 91*, Computer Laboratory, University of Cambridge, Cambridge CB2 3QG, U.K., Sept. 1986.

[20] S-C. Chang and M. Marek-Sadowska, "Technology Mapping via Transformations of Function Graph," *Proc. International Conf. on Computer Design*, pp. 159-162, 1992.

[21] M. J. Ciesielski and S. Yang, "PLADE: A Two-Stage PLA Decomposition," *IEEE Trans. on Computer-Aided Design*, Vol. 11, No. 8, pp. 943-954, Aug. 1992.

[22] E. M. Clarke, M. Fujita, P. C. McGeer, K. L. McMillan, and J. C.-Y. Yang, "Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation," *International Workshop on Logic Synthesis*, pp. 6a:1-15, May 1993.

[23] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. C.-Y. Yang, "Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping," *Proc. of the 30th Design Automation Conference*, pp. 54-60, 1993.

[24] A. Cohn, "A Proof of Correctness of the VIPER Microprocessor: the First Level," Technical Report 104, Computer Laboratory, University of Cambridge, Cambridge CB2 3QG, U.K., Jan. 1987.

[25] O. Coudert, C. Berthet, J. C. Madre, "Verification of Synchronous Sequential Machines based on Symbolic Execution," *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989.

[26] O. Coudert and J. C. Madre, "Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions," *Proc. of the 29th Design Automation Conference*, pp.36-39, 1992.

[27] H. A. Curtis, "A New Approach to the Design of Switching Circuits," Princeton, N.J., Van Nostrand, 1962.

[28] Y-T. Lai and S. Sastry, "Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification," *Proc. of 29th Design Automation Conf.*, pp. 608-613, 1992.

[29] Y-T. Lai, M. Pedram and S. Sastry, "BDD Based Decomposition of Logic Functions with Application to FPGA Synthesis," *Proc. of 30th Design Automation Conf*, pp. 642-647, 1993.

[30] W.-M. Dai, M. Sato, and E. S. Kuh, "A Dynamic and Efficient Representation of Building-Block Layout," *Proc. 24th Design Automation Conf.*, pp. 376-384, 1987.

[31] R. J. Dakin, "A Tree-Search Algorithm for Mixed Integer Programming Problems," *Comput. J. 9*, pp. 250-255, 1965.

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

[33] J. A. Darringer, "The Application of Program Verification Techniques to Hardware Verification," *Proc. of 16th Design Automation Conference*, pp. 375-381, June 1979.

[34] G. De Micheli, "Symbolic Design of Combinational and Sequential Logic Circuits Implemented by Two-Level Macros," *IEEE Trans. on Computer-Aided Designs*, Vol. 5, No. 9, pp. 597-616, Sept. 1986.

[35] G. De Micheli, D. Ku, F. Mailhot, and T. Truong, "The Olympus Synthesis System," *IEEE Design and Test of Computer*, 7(5), pp. 37-53, 1990.

[36] S. Devadas, H.-K. Ma, and A. R. Newton, "MUSTANG: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations," *IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 12, pp. 1290-1300, Dec. 1988.

[37] N. J. Driebeck, "An Algorithm of the Solution of Mixed Integer Programming Problems," *Management Science 12*, pp. 576-587, 1966.

[38] C. R. Edwards, "The Application of the Rademacher-Walsh Transform to Boolean Function Classification and Threshold-Logic Synthesis," *IEEE Transactions on Computers*, C-24, pp. 48-62, 1975.

[39] H. Eveking and S. Höreth, "Optimization and Resynthesis of Complex Data-Paths," *Proc. of 30th Design Automation Conf*, pp. 637-641, June 1993.

[40] B. J. Falkowski, I. Schafer and M. A. Perkowski, "Effective Computer Methods for the Calculation of Rademacher-Walsh Spectrum for Completely and Incompletely Specified Boolean Functions," *IEEE Transaction on Computer-Aided Design*, Vol. 11, No. 10, pp. 1207-1226, Oct. 1992.

[41] M. L. Fisher, "The Lagrangian Relaxation Method for Solving Integer Programming Problems," *Management Science*, pp. 1-18, 1981.

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

[43] R. E. Gomory, "Outline of an Algorithm for Integer Solutions to Linear Program," *Bulletin of the American Mathematical Society 64*, pp. 275-278, 1958.

[44] R. E. Gomory, "Solving Linear Programming Problems in Integers," in *Combinatorial Analysis*, R.E. Bellman and M. Hall, Jr., eds., American Mathematical Society, pp. 211-216, 1960.

[45] p. Gutberlet, H. Krämer, and W. Rosenstiel, "CASCH - a Scheduling Algorithm for High Level Synthesis," *Proc. of European Design Automation Conference*, pp. 311-315, Feb. 1991.

[46] G. D. Hachtel and F. Somenzi, "A Symbolic Algorithm for Maximum Flow in 0-1 Networks," *International Workshop on Logic Synthesis*, pp. 6b:1-6, May 1993.

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

[48] P. L. Hammer and B. Simeone, "Order Relations of Variables in 0-1 Programming," *Annals of Discrete Mathematics*, 31, North-Holland, pp. 83-112, 1987.

[49] F. K. Hanna and N. Daeche, "Specification and Verification Using Higher-Order Logic," in C. J. Koomen and T. Moto-Oka, eds, *7th Int'l Symp. Computer Hardware Description Languages and their Applications*, pp.418-433, IFIP, North-Holland, Aug. 1985.

[50] S. He and M. Torkelson, "Disjoint Decomposition with Partial Vertex Chart," *International Workshop on Logic Synthesis*, pp. p2a:1-5, 1993.

[51] Yung-Te Lai and Sarma Sastry, "HINTS: A Hardware Interpretation System," International Workshop on Formal Methods in VLSI Design 1991.

[52] W-J. Hsu and W-Z. Shen, "Coalgebraic Division for Multilevel Logic Synthesis," *Proc. of 29th Design Automation Conf.*, pp. 438-442, June 1992.

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

[54] T-T. Hwang, R. M. Owens, and M. J. Irwin, "Efficiently Computing Communication Complexity for Multilevel Logic Synthesis," *IEEE Trans. on CAD*, Vol. 11, No. 5, pp. 545-554, May 1992.

[55] C-T. Hwang and Y-C. Hsu, "Zone Scheduling," *IEEE Trans. on Computer-Aided Design*, Vol. 12, No. 7, pp. 926-934, July 1993.

[56] T. Ibaraki, "The Power of Dominance Relations in Branch and Bound Algorithm," *J. Assoc. Comput. Mach. 24*, pp. 264-279, 1977.

[57] Y-T. Lai, S. Sastry and M. Pedram, "Boolean Matching Using Binary Decision Diagrams with Applications to Logic Synthesis and Verification," *Proc. International Conf. on Computer Design*, pp.452-458, 1992.

[58] Y-T. Lai, K-R. Pan, M. Pedram, and S. Vrudhula, "FGMap: A Technology Mapping Algorithm for Look-Up Table Type FPGAs based on Function Graphs", *International Workshop on Logic Synthesis*, 1993.

[59] S-W. Jeong and F. Somenzi, "A New Algorithm for 0-1 Programming based on Binary Decision Diagrams," Logic Synthesis Workshop, in Japan, pp. 177-184, 1992.

[60] N. Karmarkar, "A New Polynomial-Time Algorithm for Linear Programming," *Combinatorica*, 4:373-395, 1984.

[61] R. M. Karp, "Functional Decomposition and Switching Circuit Design," *J. Soc. Indust. Appl. Math.,* Vol. 11, No. 2, pp. 291-335, June, 1963.

[62] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, 49(2), pp. 291-307, 1970.

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

[64] A. H. Land and A. G. Doig, "An Automatic Method for Solving Discrete Programming Problems," *Econometrica* 28, pp. 497-520, 1960.

[65] A. Land and S. Powell, "Computer Codes for Problems of Integer Programming," *Annals of Discrete Mathematics* 5, North-Holland, pp. 221-269, 1979.

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

[67] R. N. Mayo, M. H. Arnold, W. S. Scott, D. Stark, and G. T. Hamachi, "1990 DECWRL/Livermore Magic Release," *Western Research Laboratory*, WRL Research Report 90/7, Sept. 1990.

[68] F. Mailhot and G. De Micheli, "Technology Mapping Using Boolean Matching," *Eruopean Design Automation Conf.*, pp. 180-185, 1990

[69] F. Mailhot and G. De Micheli, "Algorithms for Technology Mapping Based on Binary Decision Diagrams and on Boolean Operations," *IEEE Trans. on Computer-Aided Design*, Vol. 12, No. 5, pp. 599-620, May 1993.

[70] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment," *Proc. Int. Conf. Computer-Aided Design*, pp. 6-9, 1988.

[71] M. C. McFarland and A. C. Parker, "An Abstract Model of Behavior for Hardware Descriptions," *IEEE Trans. on Computers*, C-32(7):621-637, July 1983.

[72] P. Michel, U. Lauther, and P. Duzy, *The Synthesis Approach to Digital System Design*, Kluwer Academic Publishers, 1992.

[73] G. J. Milne, "CIRCAL and the Representation of Communication, Concurrency, and Time," *ACM Trans. on Programming Languages and Systems*, 7(2):270-298, April 1985.

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

[75] G. Mitra, "Investigation of Some Branch and Bound Strategies for the Solution of Mixed Integer Linear Programs," *Math. Programming 4*, pp. 155-170, 1973.

[76] L. G. Mitten, "Branch-and-Bound Methods: General Formulation and Properties," *Operations Research*, 18, pp. 24-34, 1970.

[77] M. Mlinar, A. C. Parker, and J. T. Pizarro, "MAHA: A Program for Data Path Synthesis," *Proc. of 23rd Design Automation Conference*, pp. 235-240, June 1986.

[78] J. Mohnke and S. Malik, "Permutation and Phase Independent Boolean Comparison," *EDAC*, pp. 86-92, 1993.

[79] B. Moszkowski, "A Temporal Logic for Multilevel Reasoning about Hardware," *IEEE Computer*, pp. 10-19, Feb. 1985.

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

[81] P. G. Paulin, "High-Level Synthesis of Digital Circuits Using Global Scheduling and Binding Algorithms," Ph.D. thesis, Carleton University, 1988.

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

[83] T. Sasao, "FPGA Design by Generalized Functional Decomposition," in *Logic Synthesis and Optimization*, Sasao ed., Kluwer Academic Publisher, pp. 233-258, 1993.

[84] G. Saucier, C. Duff, and F. Poirot, "State Assigment of Controllers for Optimal Area Implementation," *Proc. of 1st European Design Automation Conference*, pp. 546-551, 1990.

[85] H. Savoj and H. Y. Wang, "Improved Scripts in MIS-II for Logic Minimization of Combinational Circuits," *Proc. of Int. Workshop on Logic Synthesis*, May 1991.

[86] U. Schlichtmann, F. Brglez, and M. Hermann, "Characterization of Boolean Functions for Rapid Matching in EPGA Technology Mapping," *Proc. of 29th Design Automation Conf*, pp. 374-379, 1992.

[87] U. Schlichtmann, F. Brglez, and P. Schneider, "Efficient Boolean Matching Based on Unique Variable Ordering," *International Workshop on Logic Synthesis*, May 1993.

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

[89] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," *IEEE Journal of Solid-State Circuits*, SC-20(2):510-522, 1985.

[90] J. F. Shapiro, "A Survey of Lagrangian Techniques for Discrete Optimization," *Annals of Discrete Mathematics 5*, North-Holland, pp. 113-138, 1979.

[91] K. Spielberg, "Enumerative Methods in Integer Programming," *Annals of Discrete Mathematics 5*, North-Holland, pp. 139-183, 1979.

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

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

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

[95] J. A. Tomlin, "Branch and Bound Methods for Integer and Non-convex Programming," in J. Abadie, ed., *Integer and non-linear programming*, North-Holland, Amsterdam, 1970.

[96] D. Varma and E. A. Trachtenberg, "Design Automation Tools for Efficient Implementation of Logic Functions by Decomposition," *IEEE Transactions of Computer-Aided Design*, Vol. 8, No. 8, Aug. 1989, pp. 901-916.

[97] J. Vasudevamurthy and J. Rajski, "A Method for Concurrent Decomposition and Factorization of Boolean Expressions," *Proc. of Int. Conf. on CAD*, pp. 510-513, 1990.

[98] T. Villa and A. Sangiovanni-Vincentelli, "NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementation," *IEEE Trans. on Computer-Aided Designs*, Vol. 9, No. 9, pp. 905-924, Sept. 1990.

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

[100] T. Yoshimura and E. S. Kuh, "Efficient Algorithms for Channel Routing," *IEEE Trans. on Computer-Aided Designs*, 1(1), pp. 25-35, 1982.