Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification*

Randal E. Bryant
Carnegie Mellon University
Pittsburgh, PA 15213
Randy.Bryant@cs.cmu.edu
http://www.cs.cmu/~bryant

Abstract

Ordered Binary Decision Diagrams (OBDDs) have found widespread use in CAD applications such as formal verification, logic synthesis, and test generation. OBDDs represent Boolean functions in a form that is both canonical and compact for many practical cases. They can be generated and manipulated by efficient graph algorithms. Researchers have found that many tasks can be expressed as series of operations on Boolean functions, making them candidates for OBDD-based methods.

The success of OBDDs has inspired efforts to improve their efficiency and to expand their range of applicability. Techniques have been discovered to make the representation more compact and to represent other classes of functions. This has led to improved performance on existing OBDD applications, as well as enabled new classes of problems to be solved.

This paper provides an overview of the state of the art in graph-based function representations. We focus on several recent advances of particular importance for formal verification and other CAD applications.

1. Introduction

Although the idea of representing Boolean functions as decision graphs has a long heritage, e.g., Akers' seminal paper [1], their widespread use as a data structure for symbolic Boolean manipulation only started with the 1986 formulation of a set of algorithms for constructing and operating on these data structures [6]. One key to this algorithmic formulation was the imposition of an ordering requirement, i.e., that the variables along every path from the root to a leaf occur in a fixed order.

Since 1986, activity involving BDDs has been widespread [8]. Numerous applications, especially in CAD area have been found. Refinements to the data structure and the manipulation algorithms have yielded improved time and memory performance. The basic ideas of BDDs have been extended to allow efficient representation of other classes of functions. An unfortunate byproduct of the dynamic and diverse research community in this area has been a proliferation of names and terminology, leading to an "alphabet soup" of acronyms. The following is only a partial list: OBDD, FBDD, FDD, OKFDD, EVBDD, MTBDD, BMD, *BMD, ZBDD, ABDD, HDD, TDD, and OPDD. Keeping track of the developments and how they relate to one another is a challenging task.

In this paper, we present recent advances in the area of graph-based function representations. Since it would be impractical to chronicle all of the developments, we instead highlight a few that have had particularly strong impact, especially in the area of formal verification.

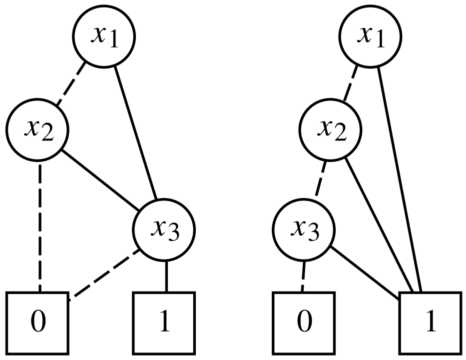

Figure 1: **Example OBDDs.** A dashed (solid) line indicates the branch when the decision variable is 0 (1).

2. Background

A BDD represents a function as a graph, with each nonterminal node labeled by a function variable. Figure 1 illustrates BDDs representing of the functions $(x_1 \vee x_2) \wedge x_3$ (left) and $x_1 \vee x_2 \vee x_3$ (right), where we use symbols $\vee$, $\wedge$, and overline to indicate Boolean OR, AND, and NOT, respectively. Each node has two outgoing edges, corresponding to the cases where the variable evaluates to 0 (shown as a dashed line) or to 1 (shown as a solid line). The terminal nodes (shown as boxes) are labeled with 0 or 1, corresponding to the possible function values. For any assignment to the variables, the function value is determined by tracing a path from the root to a terminal node following the appropriate branch from each node.

The graphs of Figure 1 are examples *Ordered* BDDs (OBDDs). That is, if we consider the variables to be ordered $x_1 < x_2 < x_3$, then every path from the root to a leaf encounters variables in ascending order. By applying a set of reduction rules [6], it is possible to reduce an OBDD to a canonical form, i.e., to a unique

*This paper will appear in *International Conference on Computer-Aided Design* (ICCAD), November, 1995. This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330.

representation for that particular variable ordering.

Many operations on Boolean functions can be implemented by simple graph algorithms that operate on their OBDD representations [6, 8]. Examples include: determining whether two functions are equivalent, generating the function corresponding to the AND, OR, or NOT of other functions, or determining the size of the on-set for a function. These algorithms all have time and space complexities that are polynomial in the sizes of their operand graphs. They also have the property that all BDDs generated obey a single variable ordering and are in canonical form, making them suitable as arguments for subsequent operations.

To apply BDDs to a problem domain, the data to be represented must first be encoded as Boolean functions. The task is then expressed as a sequence of steps, each involving an operation on one or more BDDs. For example, consider the task of generating BDD representations for the output functions of a combinational circuit. Variables are defined for the primary inputs and BDDs for the functions denoted by the variables (each having a single nonterminal node) are generated. Then the circuit is evaluated symbolically, generating the BDD for each gate output by applying the gate operation to the BDDs representing the gate inputs. This evaluation proceeds according to some topological order of the network until the BDDs for all primary outputs have been generated.

3. Dynamic Variable Ordering

When using OBDDs the user must select some total ordering of the function variables. In most applications, a single ordering is used for all functions represented, making it feasible to combine and compare different functions. Practitioners have found that finding a suitable variable ordering is critical—a good ordering will yield compact representations with reasonable memory requirements, while a poor one will exceed the physical memory limits of even the largest workstations. Beyond this point, the performance becomes unacceptable due to severe thrashing in the virtual memory system.

Although the 1986 paper discusses the variable ordering issue and presents several rules of thumb for manually choosing a good one, OBDDs only became popular with the development of heuristic methods for automatically generating an ordering based on information about the application. For example, given a gate-level combinational circuit, heuristic methods can generally find an ordering for variables representing the primary inputs such that the primary output functions have compact OBDD representations [13, 19]. Other applications, such as when analyzing sequential circuits, have proved less amenable to such an approach. Furthermore, this approach yields a static ordering to be used from application start to finish. In practice, the ideal ordering may change as the application moves through different phases of computation. Finally, every new application requires a new heuristic method to choose an appropriate variable ordering based on the problem description.

Rudell's dynamic variable ordering method [26] overcomes many of these limitations. As the application proceeds, a global, multi-rooted BDD is maintained representing all functions of interest with a single variable ordering. Periodically the program attempts to reorder the variables in this graph to reduce its memory requirement. This reordering can proceed in the background without any direct involvement by the application program. Thus, dynamic reordering can adapt to the changing functions being represented, and it works with any application.

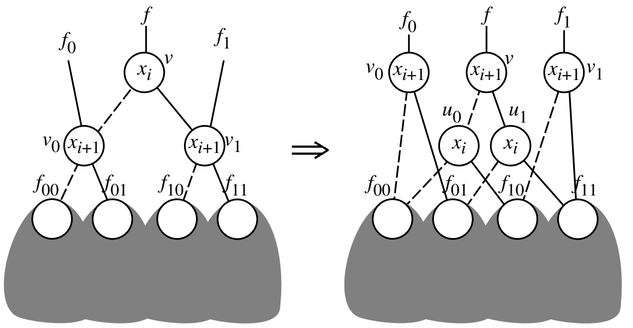

Figure 2: **Swapping Adjacent Variables in BDD.** The exchange can be implemented without altering any incoming pointers.

A key to the success of dynamic variable reordering is the ease with which it can be integrated into existing BDD packages. For example, with the package described in [5], functions are denoted by pointers to nodes within the global BDD. A reference count is maintained for each node to detect when all pointers to it have been eliminated, and hence the node storage can be reclaimed. This "garbage collection" is performed in the background periodically. Thus, at any given time it is possible to determine the number of pointers to a node, but not where these pointers are located.

Rudell's implementation performs the variable ordering in conjunction with garbage collection. That is, the program periodically attempts to reduce the storage requirement by rearranging the global BDD according to a new variable ordering and by reclaiming unused node storage. It reorders the variables by performing a series of swaps between adjacent variables. This swapping and reclaiming does not require altering any external pointers to the global BDD and hence can proceed with no effect on the application except for an occasional reduction in performance.

Figure 2 illustrates the procedure for swapping variables x_i and x_{i+1} in the global BDD. Suppose in the original ordering (left), function f is indicated by a pointer to a node v in the global BDD, where node v is labeled by variable x_i. In the usual case, node v will have branches to nodes v_0 and v_1, each labeled by variable x_{i+1}, and these nodes will have branches to the subgraphs indicated as f_{00}, f_{01}, f_{10}, and f_{11}. Following the swapping, function f should be indicated by a pointer to a node labeled by variable x_{i+1}; this node should have branches to nodes labeled by variable x_i; and these nodes should in turn have branches to the subgraphs f_{00}, f_{10}, f_{01}, and f_{11}. This result is indicated on the right. Note how function f is still indicated by a pointer to node v, and that any pointers to existing functions (shown as f_0 and f_1) remain undisturbed. Instead, we have introduced nodes u_0 and u_1. Thus, the swapping can take place without changing any of the external pointers to the global BDD. Although this figure shows an increase in the graph size, in practice node v_0 or v_1 can often be reclaimed, and nodes u_0 or u_1 may already exist. Somewhat different transformations are required when node v has branches to nodes labeled by variables other than x_{i+1}.

Dynamic variable reordering has proved especially useful when verifying and analyzing sequential circuits, e.g., to prove that two circuits realize equivalent state machines. These applications typically proceed through a series of phases, first constructing representations of the next state functions from the combinational logic, possibly converting to a transition relation, and

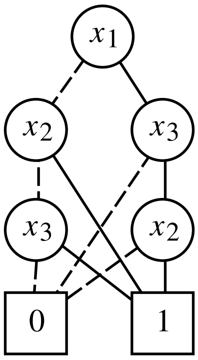

Figure 3: **Example of a Free BDD.** Variables may occur in any order, but only once along each path.

then performing an iterative computation of the reachable state set. Although heuristic methods based on traversing the gate-level network can find a suitable ordering for representing the next state functions, this ordering often does not work well for subsequent phases. Dynamic reordering allows the representation to adapt to the changing functions as the computation proceeds. Dynamic reordering can slow down an application by as much as a factor of 10, but users have found that it can make the difference between success and failure in completing an application.

4. Relaxing the Ordering Requirement

Although selecting a good initial variable ordering and changing it dynamically as the application proceeds can minimize the amount of memory used by OBDDs, there are times when the total ordering requirement of OBDDs becomes an unacceptable limitation. For example, there are some classes of functions that have OBDDs of size exponential in the number of variables, regardless of the ordering chosen [7]. This exponential blowup severely limits the problems sizes that can be handled. One example of a class of intractable functions are those representing the outputs of a combinational multiplier. Ochi *et al* [22] have successfully built the OBDDs for a 15-bit multiplier, requiring over 12 million nodes (around 0.26 Gigabytes). Increasing the word size by one bit causes the number of nodes to increase by a factor of approximately 2.7, and hence even more powerful computers will not be able to get much beyond this point.

One obvious strategy then is to eliminate or relax the variable ordering requirement. For example, Ashar *et al* go to the opposite extreme of removing all restrictions on how variables occur in the graph [2]. A variable may occur at any point along a path and even multiple times. At this extreme, however, most of the desirable algorithmic properties found in OBDDs are lost. There is no canonical form, and the algorithms for many operations have exponential worst case complexity.

A more prudent approach is to relax the ordering requirement enough to improve the compactness of the representation, but without eliminating the desirable algorithmic properties of OBDDs. One such relaxation is to allow variables to occur in any order but at most once along any path from the root to a terminal node. Such BDDs are known as either "Free" BDDs or 1-time branching programs, an example of which is shown in Figure 3. Both Gergov and Meinel [14], as well as Sieling and Wegener [25] have developed efficient algorithms based on this representation. In both cases, they further require that all functions to be represented obey a common ordering. That is, for any given variable assignment, the resulting paths in all graphs would contain the variables in the same order. The difference with OBDDs is simply that the variable ordering may differ from one variable assignment to another.

Recently, Bern *et al* have shown that such free BDDs can be implemented by simple extensions to an existing OBDD package [4]. In their implementation, the reordering of variables according to different variable assignments is viewed as a transformation of the function input space. Many of the operations performed on functions are independent of such a transformation, as long as all functions are transformed in a consistent way. They were able to show that some functions which were previously intractable could now be handled easily, and that modest performance gains could be obtained on a number of benchmark circuits. For the case of multipliers, unfortunately, only modest gains can be expected: it has recently been shown that any free BDD representation of these functions grows exponentially with the number of variables [23].

It is too early to determine the full impact of this relaxation in the ordering constraint. It allows BDDs to be applied to classes of circuits that previously experienced exponential blowup, but it is not clear how significant these classes are. In its favor, the approach is backward compatible with OBDDs, and the overhead caused by this generalization is relatively low.

5. Changing Function Decompositions

Another approach to obtaining a more compact representation for Boolean functions is to change the interpretation of the nodes. BDDs are based on a decomposition of Boolean functions commonly called the "Shannon expansion," but which was, in fact, formulated by Boole. A function f can be decomposed in terms of a variable x as:

$$f \quad = \quad \overline{x} \wedge f_{\overline{x}} \;\vee\; x \wedge f_x \tag{1}$$

In this equation f_x is the "positive cofactor" of f with respect to x, i.e., the result of replacing variable x by the value 1. Similarly, $f_{\overline{x}}$ is the "negative cofactor," i.e., the result of replacing variable x by the value 0. Observe that at least one of the terms in the equation must evaluate to 0. In other words, this decomposition splits the function into two separate cases, according to whether the variable x evaluates to 1 or to 0. Each node and its descendants in a BDD represents a Boolean function f, where for node label x, one outgoing edge is directed to the subgraph representing f_x, and the other to $f_{\overline{x}}$. In following a path from the root to a terminal node, we are simply taking successive cofactors of the function until it reduces to a constant value.

Alternative function decompositions can be expressed in terms of the XOR (exclusive-or) operation:

$$\begin{aligned} f &= f_{\overline{x}} \;\oplus\; x \wedge f_{\delta x} & (2)\\ &= f_x \;\oplus\; \overline{x} \wedge f_{\delta x} & (3)\end{aligned}$$

where $f_{\delta x}$ denotes the *Boolean difference* of function f with respect to variable x, i.e., $f_{\delta x} = f_x \oplus f_{\overline{x}}$. Equation 2 is commonly referred to as either the "Reed-Muller" or "Negative Davio" expansion, while Equation 3 is referred to as the "Positive Davio" expansion. These decompositions are somewhat analogous to the Taylor expansion of a continuous function—they describe the function in terms of its value for a fixed value of x (either 0 or 1), combined with how the function varies as x changes (given

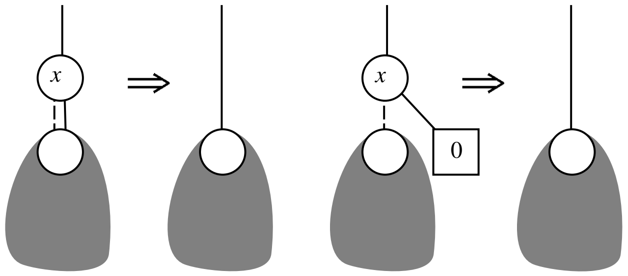

Figure 4: **Reduction Rules** Different rules apply for BDDs (left), and for FDDs or ZBDDs (right).

by $f_{\delta x}$). Since the variable x only takes on two values, only two terms in the expansion are required.

Kebschull *et al* have proposed using the Reed-Muller expansion as the basis for a graph representation of Boolean functions, yielding a representation they term "Ordered Functional Decision Diagrams" (OFDDs) [16]. This representation is analogous to OBDDs, except that the two outgoing arcs from a node represent the negative cofactor and the Boolean difference of the function with respect to the node variable. As examples, the graphs of Figure 1 can be interpreted as FDDs, with the solid edge from each node indicating the Boolean difference. Under this interpretation, the example graphs denote the functions $(x_1 \oplus x_2) \wedge x_3$ (left) and $x_1 \oplus x_2 \oplus x_3$ (right).

OFDDs have many properties in common with OBDDs: the representation is canonical and many operations can be implemented with polynomial complexity graph algorithms. Several differences are worth highlighting. First, a different reduction rule is applied to make the graph canonical. The rules for the two classes of graphs are illustrated in Figure 4. Both express conditions under which a node can be eliminated from the graph with all incoming pointers directed to a child. In both cases we want such a transformation to occur when the function is independent of the node variable. If function f is independent of variable x, then its cofactors will satisfy $f_x = f_{\overline{x}} = f$, and its difference will satisfy $f_{\delta x} = 0$. Thus, we want the transformation to occur under the conditions illustrated on the left hand side of the figure for BDDs, and under those of the right hand side for FDDs.

Second, the evaluation of a function given its OFDD involves more than tracing a single path. In particular, for the case of a node variable with value x we must evaluate both subgraphs and take their XOR. In other words, we compute the positive cofactor of function f as $f_x = f_{\overline{x}} \oplus f_{\delta x}$. Nonetheless, evaluation can be performed in time linear in the number of nodes by a postorder traversal of the graph.

For some classes of functions, FDDs are exponentially more compact than OBDDs, but the reverse can also hold. To obtain the advantages of each, Drechsler *et al* have proposed a hybrid form they call "Ordered Kronecker Functional Decision Diagrams" (OKFDDs) [12]. In their representation, each variable has an associated decomposition, which can be any one of the three given by Equations 1–3. All functions to be represented must follow a common variable ordering and every occurrence of a given variable must use the same decomposition. They implement decomposition in conjunction with dynamic variable reordering. That is, at the same time variables are being reordered, the program tries different decompositions for the variables, attempting to reduce the global graph size. Their experiments show an average reduction in the graph size of 35% over OBDDs for a set of benchmark circuits.

It is still to early to judge the impact of FDDs and OKFDDs on practical applications. It is not clear how much reduction can be obtained for real-life examples. However, the ideas of changing the interpretation of nodes and of setting the decomposition type independently for each variable have had major impact in verifying arithmetic circuits, as will be discussed later.

6. Zero-Suppressed BDDs

Minato has developed a variant of BDDs for solving combinatorial problems. His "Zero-Suppressed" BDDs (ZBDDs) are suitable for applications that can be solved by representing and manipulating sparse sets of bit vectors [20]. That is, suppose the data for a problem are encoded as bit vectors of length n. Then any subset of the vectors can be represented by a Boolean function over n variables yielding 1 when the vector corresponding to the variable assignment is in the set.

For example, consider problems involving sum-of-products forms of logic functions. Such forms are commonly represented as sets of "cubes," each denoted by a string containing symbols `0`, `1`, and `-`. For example, the function $(\overline{x_1} \wedge x_2) \vee (\overline{x_2} \oplus x_3)$ can be represented by the cube set $\{$`01-`, `-11`, `-00`$\}$. To encode cubes with bit vectors, we introduce a pair of bits for each cube position, using the encodings 10, 01 and 00 to denote symbols `1`, `0`, and `-`, respectively. We would then represent the cube set shown with bit vectors $\{011000, 001010, 000101\}$.

Observe in our example that the set of bit vectors is "sparse" in two ways. First, the set contains much fewer than the 2^n possible bit vectors. Second, the bit vectors themselves have many elements equal to zero. In fact, our particular encoding of cube symbols was chosen for this purpose. This motivates choosing a representation that exploits both forms of sparseness. Zero-suppressed BDDs are much like OBDDs, except that they use the reduction rule shown on the right hand side of Figure 4, rather than that on the left. That is, a node can be omitted if setting the node variable to 1 causes the function to yield 0. When representing sets of bit vectors, this condition occurs when having a 1 at some bit position implies that the vector is not in the set. For sparse sets, this condition arises frequently, and hence many node eliminations are possible.

As examples, the graphs of Figure 1 can both be interpreted as ZBDDs, denoting the sets $\{101, 011\}$ (left) and $\{100, 010, 001\}$ (right). Observe that the function denoted by some subgraph in a ZBDD depends on its context. For example, the right hand example of Figure 1 has three edges leading to the terminal node labeled 1, each having a different interpretation. The rightmost one corresponds to the case where $x_2 = x_3 = 0$, the middle to the case where $x_3 = 0$, and the bottom to the case where no further constraints are placed on the variables. In a sense, the reduction rule defines what the "default" condition is when a variable does not occur along some path in a graph. For BDDs and FDDs, the default is that the function value is independent of the variable, while with ZBDDs the default is that this variable must be 0.

Minato has shown that a number of combinatorial problems can be solved efficiently using a ZBDD representation [21]. These include classical problems in two-level logic minimization, as well as techniques used in multi-level minimization such as weak

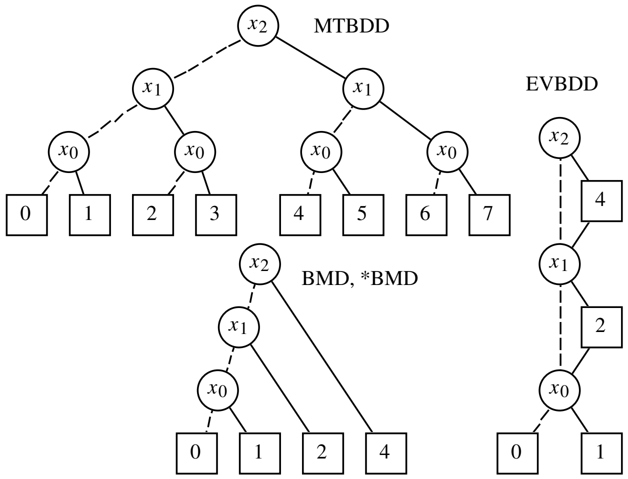

Figure 5: **Example Numeric Function Representations.** Each represents the function $x_0 + 2x_1 + 4x_2$.

division. It can be shown that ZBDDs reduce the size of the representation of a set of n-bit vectors over OBDDs by at most a factor of n [24]. In practice, the reduction is large enough to have significant impact.

7. Representing Numeric-Valued Functions

Building on the success of BDDs, there have been several efforts to extend the concept to represent functions over Boolean variables, but having non-Boolean ranges, such as integers or real numbers. These functions arise in such applications as computing the state probabilities in a sequential circuit [3], using spectral methods for technology mapping [10], integer linear programming [18], and for verifying arithmetic circuits [9, 17]. Keeping the variables Boolean allows the use of a branching structure similar to BDDs. The challenge becomes finding a compact way to encode the numeric function values.

7.1. MTBDDs

One straightforward way to represent numeric-valued functions is use a decision graph like a BDD, but to allow arbitrary values on the terminal nodes. We will call such a representation a "Multi-Terminal" BDD (MTBDD) [10], although they have also been called "Arithmetic Decision Diagrams" (ADDs) [3]. Figure 5 illustrates the MTBDD representation of the function $x_0 + 2x_1 + 4x_2$, i.e., the result of interpreting three bits as an unsigned binary number. Observe that evaluating an MTBDD for a given variable assignment is similar to evaluation in a BDD: we simply trace the unique path from the root to a terminal node determined by the variable values and return the terminal node value as the function value.

As this example illustrates, MTBDDs are very inefficient for representing functions yielding values over a large range. For the case of unsigned binary numbers of length n, there are 2^n possible values and hence the MTBDD representation must have an exponential number of terminal nodes. For some applications, the number of possible values is small enough that this shortcoming is acceptable. With such applications, the simplicity of the representation and its similarity to BDDs make MTBDDs viable candidates.

7.2. EVBDDs

For applications where the number of possible function values is too high for MTBDDs, alternative representations have been devised to try to obtain a more compact form. Lai *et al* developed "Edge-Valued" BDDs (EVBDDs) incorporating numeric weights on the edges in order to allow greater sharing of subgraphs [17, 18]. As an example, the EVBDD representation of the function $x_0 + 2x_1 + 4x_2$ is shown in Figure 5. In our drawings of EVBDDs, edge weights are shown in square boxes, where an edge without a box has weight 0. Evaluating a function represented by an EVBDD involves tracing the path determined by the variable assignment, summing the edge weights and the terminal node value. As can be seen in this example, a sum of weighted bits is represented by having each bit weight on the edge leading out of the node labeled with the corresponding variable. This representation grows linearly with the number of bits, a major improvement over MTBDDs.

Various schemes can be used for "normalizing" edge weights so that the resulting graph provides a canonical form for the function. For example, the standard formulation of EVBDDs requires the edge corresponding to variable value 0 from each node to have weight 0. EVBDDs can represent Boolean functions as well, requiring the same number of nonterminal nodes as the corresponding OBDD. However, the overhead of storing and manipulating edge weights makes them somewhat less efficient for this task. Similarly, EVBDDs always improve on MTBDDs in terms of the number of nonterminal nodes, but with somewhat higher overhead per node.

Although EVBDDs improve on MTBDDs in many cases, there are still important classes of functions for which they have unacceptable complexity. For example, one important application of numeric-valued functions is in formally verifying arithmetic circuits, e.g., those that compute functions such as addition, multiplication, division, and radix conversion. In this application, we want to express the numeric values encoded by operations on words of data. As an example, consider two n-bit unsigned integers, represented by bit vectors $\vec{x} = x_{n-1}, \ldots, x_0$ and $\vec{y} = y_{n-1}, \ldots, y_0$. We can view these as encoding numeric values $X = \sum_{i=0}^{n-1} 2^i x_i$, and similarly for Y. As we have seen, EVBDDs can represent these "encoding functions" for X and Y with complexity linearly in n. The result of applying addition or subtraction to these words, i.e., $X + Y$ and $X - Y$ can also be represented by linear-sized EVBDDs. However, EVBDD representations of multiplication $X \cdot Y$ or exponentiation 2^X grow exponentially with n. Thus, for verifying arithmetic circuits, EVBDDs are limited to relatively simple units such as adders, ALUs, and comparators.

7.3. BMDs

An alternative approach for representing numeric functions, especially those encountered in arithmetic circuit verification, involves changing the function decomposition with respect to its variables in a manner analogous to FDDs. The resulting representation is known as a "Binary Moment Diagram" (BMD) [9].

For expressing functions having numeric range, the Boole-Shannon expansion of Equation 1 can be generalized as:

$$f \quad = \quad (1 - x) \cdot f_{\overline{x}} \; + \; x \cdot f_x \qquad (4)$$

where $\cdot$, $+$, and $-$ denote multiplication, addition, and subtraction, respectively. Note that this expansion relies on the assumption that variable x is Boolean, i.e., it evaluates to either 0 or 1. Both

MTBDDs and EVBDDs are based on such a pointwise decomposition.

The moment decomposition of a function is obtained by rearranging the terms of Equation 4:

$$\begin{aligned} f &= f_{\overline{x}} + x \cdot (f_x - f_{\overline{x}}) \\ &= f_{\overline{x}} + x \cdot f_{\partial x} \end{aligned} \quad (5)$$

where $f_{\partial x} = f_x - f_{\overline{x}}$ is called the *linear moment* of f with respect to x. This terminology arises by viewing f as being a linear function with respect to its variables, and thus $f_{\partial x}$ is the partial derivative of f with respect to x. Since we are interested in the value of the function for only two values of x, we can always extend it to a linear form. The negative cofactor is called the *constant moment*, i.e., it denotes the portion of function f that remains constant with respect to x. Observe the similarity between the moment decomposition and the Reed-Muller decomposition of Equation 2.

Each node of a BMD describes a function in terms of its moment decomposition with respect to the variable labeling the node. As an example, the BMD representation of the unsigned binary encoding function $x_0 + 2x_1 + 4x_2$ is illustrated in Figure 5. The two outgoing arcs from each node denote the constant (dashed) and linear moments (solid) of the function with respect to the variable. Observe that the linear moment of the example function with respect to any variable x_i is simply 2^i: the function decomposes according to the bit weights. This property makes BMDs particularly well suited for representing arithmetic functions.

An extension of BMDs is to incorporate weights on the edges [9], yielding a representation called "Multiplicative" BMDs (*BMDs) [9]. These edge weights combine multiplicatively, rather than additively as with EVBDDs. With *BMDs, such word-level functions as $X + Y$, $X - Y$, $X \cdot Y$, 2^X all have linear-sized representations.

Both BMDs and *BMDs offer considerable advantages over all other known representations for representing arithmetic functions, but this gain comes at a cost. Many of the operations one would like to perform on the functions have worst case complexity that grows exponentially in the operand graph sizes. Although, in practice the algorithms are often quite efficient, our experience has been that the exponential cases do arise, mandating various tactics to work around them. In addition, BMDs can represent Boolean functions, but generally not as efficiently as BDDs. In fact, there are cases where the BMD representation of a Boolean function can be exponentially larger than the BDD representation, and we have encountered such cases in practice. Furthermore, the overhead of representing and manipulating edge weights is high. Since we encounter very large weights (e.g., 2^{512} in verifying a 256-bit multiplier), the code must use an unbounded precision number representation.

Adapting the idea of OKFDDs, Clarke, *et al* have recently developed a hybrid between BMDs and MTBDDs that they term "Hybrid" Decision Diagrams (HDDs) [11]. In their representation, each variable can use one of four different decompositions, including both pointwise and moment decompositions.

7.4. Word-Level Verification of Arithmetic Circuits

A major success in formal verification has been in using numeric-valued functions for verifying arithmetic circuits. Figure 6 illustrates schematically an approach to circuit verification originally formulated by Lai and Vrudhula [17]. The overall goal is to prove a correspondence between a logic circuit, represented by a vector of Boolean functions $\vec{f}$, and the specification, represented by the word level function F. The Boolean functions can correspond to the outputs of a combinational circuit in terms of the primary inputs, or to the outputs of a sequential circuit operated for a fixed number of steps, in terms of the initial state and the input values at each step. Assume that the circuit inputs (and possibly initial state) are partitioned into vectors of binary signals $\vec{x}^1, \ldots, \vec{x}^k$ (in the figure $k = 2$). For each set of signals $\vec{x}^i$, we are given an encoding function ENC_i describing a word-level interpretation of the signals, e.g., as an unsigned binary number. This encoding yields a numeric value $X_i = \text{ENC}_i\left[\vec{x}^i\right]$ for each input word. The circuit implements a set of Boolean functions over the inputs, denoted by the vector of functions $\vec{f}(\vec{x}^1, \ldots, \vec{x}^k)$. Typically this circuit is given in the form of a network of logic gates. Furthermore, we are given an encoding function ENC_o defining a word-level interpretation of the output. Finally, we are given as specification a word-level function $F(X_1, \ldots, X_k)$ yielding a numeric value based on the numeric inputs. The task of verification is then to prove the equivalence:

$$\begin{aligned} &\text{ENC}_o\left[\vec{f}\left(\vec{x}^1, \ldots, \vec{x}^k\right)\right] \\ &\quad = F\left(\text{ENC}_1\left[\vec{x}^1\right], \ldots, \text{ENC}_k\left[\vec{x}^k\right]\right) \end{aligned} \quad (6)$$

That is, the circuit output, interpreted as a word should match the specification when applied to word interpretations of the circuit inputs.

Both EVBDDs and (*)BMDs provide suitable data structures for this form of verification, because they can represent both bit-level and word-level functions efficiently. As discussed above, however, EVBDDs are limited to relatively simple circuit functions. By contrast, BDDs can only represent bit-level functions, and hence the specification must be expanded into bit-level form. While this can be done readily for standard functions such as binary addition, a more complex function such as binary to BCD conversion would be difficult to specify at the bit level.

A straightforward approach to verifying such circuits with EVBDDs or (*)BMDs is to construct function representations and test the equivalence given by Equation 6. That is, we construct representations of the Boolean circuit functions $\vec{f}$, as we would using BDDs, and apply the encoding function ENC_o to obtain a word-level representation of the circuit output. We would then derive a representation of the word-level specification F by applying the operations of the specification to the word-level input

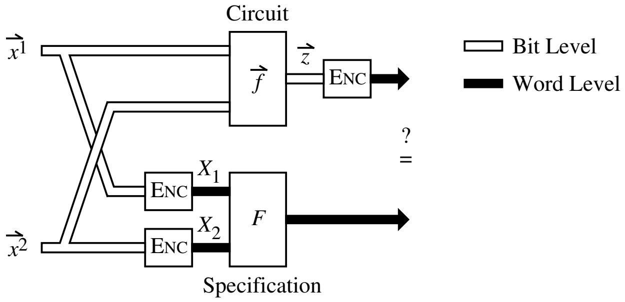

Figure 6: **Formulation of Verification Problem.** The goal of verification is to prove a correspondence between a bit-level circuit and a word-level specification

representations. Testing these two functions for equivalence then completes the verification.

This straightforward approach works well for circuits where the Boolean circuit functions $\vec{f}$ can be represented efficiently. For complex circuits, such as multipliers, dividers, and radix converters, however, these functions become intractable. Two methods have been developed to overcome this limitation. The first requires the user to partition the circuit into blocks, each having its own word-level specification [9]. Typical blocks include adders, Booth step units, etc. Each block is verified according to the method outlined above, but the overall word-level circuit function is derived by composing the block specifications at the word-level. Using this technique, multipliers with word sizes up to 256 bits have been verified successfully, far larger than could be achieved by other approaches. This approach has the limitation that it can only be applied to circuits having a relatively clean hierarchical structure that is well understood by the user.

More recently, Hamaguchi *et al* have developed an approach to verifying multipliers using *BMDs that does not require a hierarchical partitioning [15]. Instead, they directly construct a representation of the word-level circuit output $\textsc{Enc}_O\left[\vec{f}\left(\vec{x}^1,\ldots,\vec{x}^k\right)\right]$ from a flat, gate-level representation of the circuit. To avoid the complexity of representing the Boolean circuit functions, they work backwards through the circuit, starting with a *BMD representation of the function $\textsc{Enc}_O\left[\vec{z}\right]$, where $\vec{z}$ represents the circuit output signals. The program maintains a *BMD representation of the circuit output in terms of a set of variables corresponding to intermediate signal points in the circuit. At each step, it replaces one of these variables with a function representing the logic gate driving the signal. This process continues until all variables correspond to primary inputs.

8. Conclusions

Although we have only covered a small fraction of the recent developments in graph-based function representations, the material covered demonstrates the types of advances that have been made. Ideas such as dynamic variable reordering focus on the actual implementation of the algorithms. A key to its success is the ease with which it could be integrated into existing packages with minimal rewriting of application code. Ideas such as relaxing the variable ordering or changing the variable decomposition attempt to improve the compactness of the representation for Boolean functions. Again, the greatest successes have been those methods that could be integrated into existing packages.

Zero-suppressed BDDs show the utility of tuning the representation to a particular class of applications. The success of ZBDDs depends on two factors: that useful combinatorial problems can be expressed in terms of operations on sparse bit vector sets, and that such sets can be represented efficiently using a form of BDD using a nonstandard reduction rule. Perhaps there are other application domains for which the BDD representation could be adapted and tuned.

The recent interest in numeric-valued functions illustrates how a successful approach for one class of application can "spill over" into other applications. Although the straightforward use of BDDs may not prove practical, suitable tuning and refinements to the representation can lead to major breakthroughs. It is interesting to speculate how many other applications could be solved by this strategy.

One interesting phenomenon noted by this paper is the possibility of combining different extensions and refinements. Examples include incorporating other compaction methods along with dynamic reordering, and setting the decomposition type for different variables independently. An area for further exploration would be to find other combinations. For example, it may prove desirable to form a hybrid between ZBDDS and BDDs, selecting different reduction rules for different variables. Allowing data-dependent variable orderings, as is done with Free BDDs, could also be done with most of the other representations.

The greatest area of success for BDDs and related structures has been in formal circuit verification. This application requires exact results: we want to show whether or not the circuit will operate correctly under all possible operating conditions. Hence the approximate methods used in applications such as placement and routing are not acceptable. Furthermore, whereas problems such as automatic test generation require finding only one solution, and hence can be solved efficiently by combinatorial search, formal verification must establish correctness for all cases. In a sense, we must show that there are no solutions to the problem of finding some condition where the circuit fails. By providing a complete representation of the circuit behavior in a compact and manageable form, BDDs provide the most powerful approach to formal verification known.

Acknowledgements

Thanks to Alok Jain for proofreading and commenting on this manuscript.

References

[1] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, Vol. C-27, No. 6 (June, 1978), pp. 509–516.

[2] P. Ashar, S. Devadas, and A. Ghosh, "Boolean satisfiability and equivalence checking using general binary decision diagrams," *International Conference on Computer Design*, IEEE, 1991, pp. 259–264.

[3] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F .Somenzi, "Algebraic decision diagrams and their applications," *International Conference on Computer-Aided Design*, November, 1993, pp. 188–191.

[4] J. Bern, C. Meinel, and A. Slobodová, "Efficient OBDD-based Boolean manipulation in CAD beyond current limits," *32nd Design Automation Conference*, June, 1995, pp. 408–413.

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

[6] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 8 (August, 1986), pp. 677–691.

[7] R. E. Bryant, "On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication," *IEEE Transactions on Computers*, Vol. 40, No. 2 (February, 1991), pp. 205–213.

[8] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary decision diagrams," *ACM Computing Surveys*, Vol. 24, No. 3 (September, 1992), pp. 293–318.

[9] R. E. Bryant, and Y.-A. Chen, "Verification of arithmetic circuits with binary moment diagrams," *32nd Design Automation Conference*, June, 1995, pp. 535–541.

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

[11] E. Clarke, M. Fujita, and X. Zhao, "Hybrid Decision Diagrams: Overcoming the Limitations of MTBDDs and BMDs," *International Conference on Computer-Aided Design*, November, 1995.

[12] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski, "Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams," *31st Design Automation Conference*, June, 1994, pp. 415–419.

[13] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluations and improvements of a Boolean comparison program based on binary decision diagrams," *International Conference on Computer-Aided Design*, November, 1988, pp. 2–5.

[14] J. Gergov, C. Meinel, "Efficient Boolean Manipulation with OBDDs can be extended to FBDDs," *IEEE Transactions on Computers*, Vol. 43, No. 10 (October, 1994), pp. 1197–1209.

[15] K. Hamaguchi, A. Morita, and S. Yajima, "Efficient construction of binary moment diagrams for verifying arithmetic circuits," *International Conference on Computer-Aided Design*, November, 1995.

[16] U. Kebschull, E. Schubert, and W. Rosentiel, "Multilevel logic based on functional decision diagrams," *European Design Automation Conference*, 1992, pp. 43–47.

[17] Y.-T. Lai, and S. Sastry, "Edge-valued binary decision diagrams for multi-level hierarchical verification," *29th Design Automation Conference*, June, 1992, pp. 608–613.

[18] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula, "EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 8 (August, 1994), pp. 959–975.

[19] S. Malik, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," *International Conference on Computer-Aided Design*, November, 1988, pp. 6–9.

[20] S.-i. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," *30th Design Automation Conference*, June, 1993, pp. 272–277.

[21] S.-i. Minato, *Binary Decision Diagrams and Applications for VLSI CAD*, Kluwer, 1995.

[22] H. Ochi, K. Yasuoka, and S. Yajima, "Breadth-first manipulation of very large binary-decision diagrams," *International Conference on Computer-Aided Design*, November, 1993, pp. 48–55.

[23] S. Ponzio, "A lower bound for integer multiplication with read-once only branching programs," *Symposium on Theory of Computing*, ACM, 1995.

[24] O. Schröer, and I. Wegener, "The theory of zero-suppressed BDDs and the number of knights tours," Technical report 552/1994, University of Dortmund, Fachbereich Informatik, 1994.

[25] D. Sieling, and I. Wegener, "Graph-driven OBDDs—a new data structure for Boolean functions," *Theoretical Computer Science*, 1995.

[26] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *International Conference on Computer-Aided Design*, November, 1993, pp. 42–47.