Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams

Randal E. Bryant*
Fujitsu Laboratories, Ltd.
1015 Kamikodanaka, Nakahara-ku
Kawasaki 211, Japan

June, 1992

Ordered Binary Decision Diagrams (OBDDs) represent Boolean functions as directed acyclic graphs. They form a canonical representation, making testing of functional properties such as satisfiability and equivalence straightforward. A number of operations on Boolean functions can be implemented as graph algorithms on OBDD data structures. Using OBDDs, a wide variety of problems can be solved through *symbolic analysis*. First, the possible variations in system parameters and operating conditions are encoded with Boolean variables. Then the system is evaluated for all variations by a sequence of OBDD operations. Researchers have thus solved a number of problems in digital system design, finite state system analysis, artificial intelligence, and mathematical logic. This paper describes the OBDD data structure, and surveys a number of applications that have been solved by OBDD-based symbolic analysis.

Categories and Subject Descriptors: I.1 [**Algebraic Manipulation**]: Expressions and Their Representations, Algorithms; B.6 [**Logic Design**]: Reliability and Testing, Design Aids; F.1.1 [**Models of Computation**]: Automata; I.2.3 [**Deduction and Theorem Proving**]: Algorithms

General Terms: Algorithms, Verification

Additional Key Words and Phrases: Binary decision diagrams, branching programs, symbolic manipulation, symbolic analysis, Boolean functions, Boolean algebra

*Current address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

CONTENTS

CONTENTS **2**
INTRODUCTION **3**
1. OBDD REPRESENTATION **4**
1.1. Binary Decision Diagrams 4
1.2. Ordering and Reducing 4
1.3. Effect of Variable Ordering 6
1.4. Complexity Characteristics 7
1.5. Refinements and Variations 10
2. OPERATIONS **10**
3. CONSTRUCTION AND MANIPULATION **11**
3.1. The APPLY Operation 11
3.2. The RESTRICT Operation 14
3.3. Derived Operations 15
3.4. Performance Characteristics 15
3.5. Implementation Techniques 15
4. REPRESENTING MATHEMATICAL SYSTEMS **16**
4.1. Encoding of Finite Domains 17
4.2. Sets 18
4.3. Relations 19
5. DIGITAL SYSTEM DESIGN APPLICATIONS **20**
5.1. Verification 20
5.2. Design Error Correction 20
5.3. Sensitivity Analysis 21
5.4. Probabilistic Analysis 22
6. FINITE STATE SYSTEM ANALYSIS **26**
7. OTHER APPLICATION AREAS **29**
8. AREAS FOR IMPROVEMENT **29**
9. SUMMARY **30**
REFERENCES **30**

INTRODUCTION

Many tasks in digital system design, combinatorial optimization, mathematical logic, and artificial intelligence can be formulated in terms of operations over small, finite domains. By introducing a binary encoding of the elements in these domains, these problems can be further reduced to operations over Boolean values. Using a symbolic representation of Boolean functions, we can express a problem in a very general form. Solving this generalized problem via symbolic Boolean function manipulation then provides the solutions for a large number of specific problem instances. Thus, an efficient method for representing and manipulating Boolean functions symbolically can lead to the solution of a large class of complex problems.

Ordered Binary Decision Diagrams (OBDDs) [Bryant 1986] provide one such representation. This representation is defined by imposing restrictions on the the Binary Decision Diagram (BDD) representation introduced by Lee[1] [Lee 1959] and Akers [Akers 1978], such that the resulting form is canonical. These restrictions and the resulting canonicity were first recognized by Fortune, Hopcroft, and Schmidt [Fortune et al 1978]. Functions are represented as directed acyclic graphs, with internal vertices corresponding to the variables over which the function is defined and terminal vertices labeled by the function values 0 and 1. Although the OBDD representation of a function may have size exponential in the number of variables, many useful functions have more compact representations.

Operations on Boolean functions can be implemented as graph algorithms operating on OBDDs. Tasks in many problem domains can be expressed entirely in terms of operations on OBDDs, such that a full enumeration of the problem space (e.g., a truth table, state transition graph, or search tree) need never be constructed. Researchers have solved problems using OBDDs that would not be possible by more traditional techniques such as case analysis or combinatorial search.

To date, most applications of OBDDs have been in the areas of digital system design, verification, and testing. More recently, interest has spread into other areas such as concurrent system design, mathematical logic, and artificial intelligence.

This paper provides a combined tutorial and survey on symbolic Boolean manipulation with OBDDs. The next three sections describe the OBDD representation and the algorithms used to construct and manipulate them. The following section describes several basic techniques for representing and operating on a number of mathematical structures, including functions, sets, and relations, by symbolic Boolean manipulation. We illustrate these techniques by describing some of the applications for which OBDDs have been used to date and conclude by describing further areas for research. Although most of the application examples involve problems in digital system design, we believe that similar methods can be applied to a variety of application domains. For background, we assume only that the reader has a basic knowledge of Boolean functions, digital logic design, and finite automata.

---

[1] Lee represented Boolean functions as Binary Decision *Programs*, a form of straight-line program. Such a program can be viewed as a linear ordering of the vertices in a directed acyclic graph, and hence the distinction between these two forms is minor.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

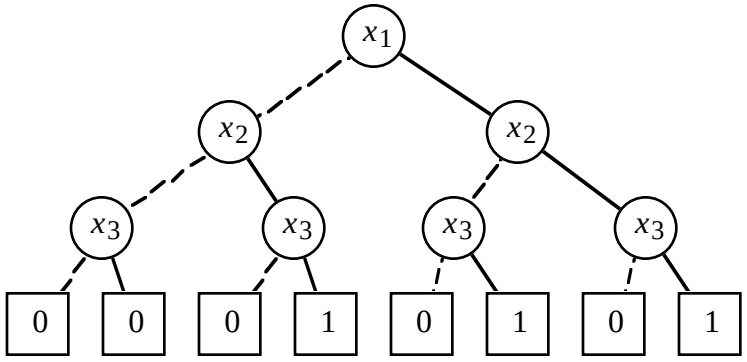

Figure 1: **Truth Table and Decision Tree Representations of a Boolean Function.** A dashed (solid) tree branch denotes the case where the decision variable is 0 (1).

1. OBDD REPRESENTATION

Binary decision diagrams have been recognized as abstract representations of Boolean functions for many years. Under the name "branching programs" they have been studied extensively by complexity theorists [Wegener 1988; Meinel 1990]. The key idea of OBDDs is that by restricting the representation, Boolean manipulation becomes much simpler computationally. Consequently, they provide a suitable data structure for a symbolic Boolean manipulator.

1.1. Binary Decision Diagrams

A binary decision diagram represents a Boolean function as a rooted, directed acyclic graph. As an example, Figure 1 illustrates a representation of the function $f(x_1, x_2, x_3)$ defined by the truth table given on the left, for the special case where the graph is actually a tree. Each nonterminal vertex v is labeled by a variable *var*(v) and has arcs directed toward two children: *lo*(v) (shown as a dashed line) corresponding to the case where the variable is assigned 0, and *hi*(v) (shown as a solid line) corresponding to the case where the variable is assigned 1. Each terminal vertex is labeled 0 or 1. For a given assignment to the variables, the value yielded by the function is determined by tracing a path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The function value is then given by the terminal vertex label. Due to the way the branches are ordered in this figure, the values of the terminal vertices, read from left to right, match those in the truth table, read from top to bottom.

1.2. Ordering and Reducing

For an *Ordered* BDD (OBDD), we impose a total ordering $<$ over the set of variables and require that for any vertex u, and either nonterminal child v, their respective variables must be ordered $var(u) < var(v)$. In the decision tree of Figure 1, for example, the variables are ordered $x_1 < x_2 < x_3$. In principle, the variable ordering can be selected arbitrarily—the algorithms will operate correctly for any ordering. In practice, selecting a satisfactory ordering is critical for the efficient symbolic manipulation. This issue is discussed in the next section.

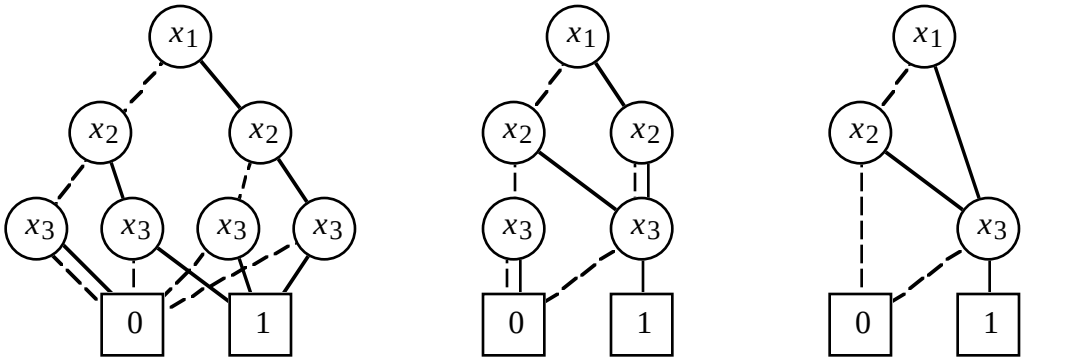

Figure 2: **Reduction of Decision Tree to OBDD.** Applying the three reduction rules to the tree of Figure 1 yields the canonical representation of the function as an OBDD.

We define three transformation rules over these graphs that do not alter the function represented:

Remove Duplicate Terminals: Eliminate all but one terminal vertex with a given label and redirect all arcs into the eliminated vertices to the remaining one.

Remove Duplicate Nonterminals: If nonterminal vertices u and v have $\mathit{var}(u) = \mathit{var}(v)$, $\mathit{lo}(u) = \mathit{lo}(v)$, and $\mathit{hi}(u) = \mathit{hi}(v)$, then eliminate one of the two vertices and redirect all incoming arcs to the other vertex.

Remove Redundant Tests: If nonterminal vertex v has $\mathit{lo}(v) = \mathit{hi}(v)$, then eliminate v and redirect all incoming arcs to $\mathit{lo}(v)$.

Starting with any BDD satisfying the ordering property, we can reduce its size by repeatedly applying the transformation rules. We use the term "OBDD" to refer to a maximally reduced graph that obeys some ordering. For example, Figure 2 illustrates the reduction of the decision tree shown in Figure 1 to an OBDD. Applying the first transformation rule (A) reduces the eight terminal vertices to two. Applying the second transformation rule (B) eliminates two of the vertices having variable x_3 and arcs to terminal vertices with labels 0 (lo) and 1 (hi). Applying the third transformation rule (C) eliminates two vertices: one with variable x_3 and one with variable x_2. In general, the transformation rules must be applied repeatedly, since each transformation can expose new possibilities for further ones.

The OBDD representation of a function is canonical—for a given ordering, two OBDDs for a function are isomorphic. This property has several important consequences. Functional equivalence can easily be tested. A function is satisfiable if and only if its OBDD representation does not correspond to the single terminal vertex labeled 0. Any tautological function must have the terminal vertex labeled 1 as its OBDD representation. If a function is independent of variable x, then its OBDD representation cannot contain any vertices labeled by x. Thus, once OBDD representations of functions have been generated, many functional properties become easily testable.

As Figures 1 and 2 illustrate, we can construct the OBDD representation of a function given its truth table by constructing and reducing a decision tree. This approach is practical, however, only for

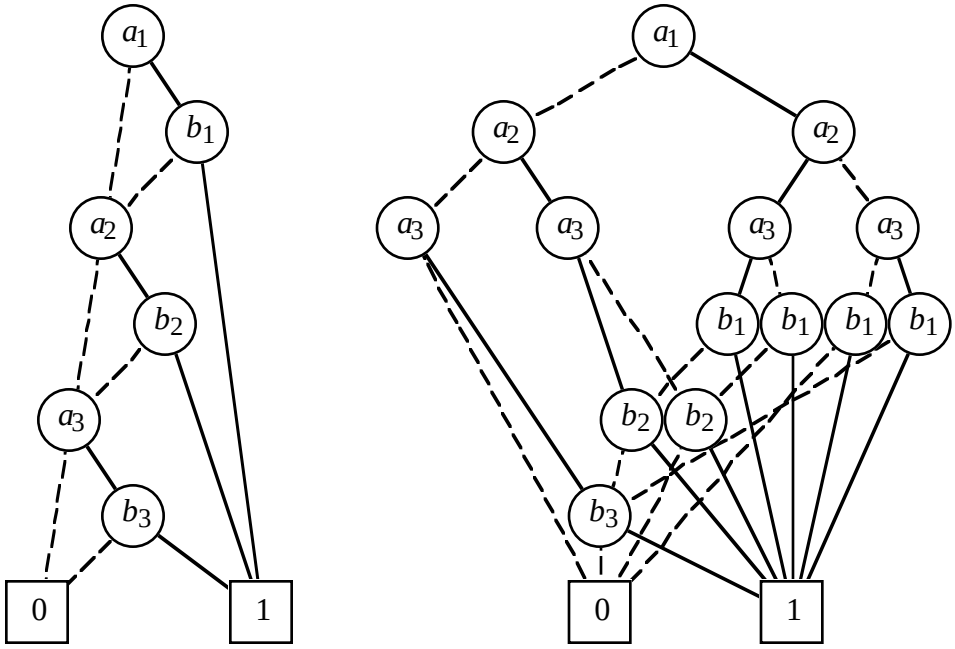

Figure 3: **OBDD Representations of a Single Function for Two Different Variable Orderings**.

functions of a small number of variables, since both the truth table and the decision tree have size exponential in the number of variables. Instead, OBDDs are generally constructed by "symbolically evaluating" a logic expression or logic gate network using the APPLY operation described in Section 3.

1.3. Effect of Variable Ordering

The form and size of the OBDD representing a function depends on the variable ordering. For example, Figure 3 shows two OBDD representations of the function denoted by the Boolean expression $a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$, where $\cdot$ denotes the AND operation and $+$ denotes the OR operation. For the case on the left, the variables are ordered $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$, while for the case on the right they are ordered $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$.

We can generalize this function to one over variables $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ given by the expression:

$$a_1 \cdot b_1 + a_2 \cdot b_2 + \cdots + a_n \cdot b_n$$

Generalizing the first variable ordering to $a_1 < b_1 < \cdots < a_n < b_n$ yields an OBDD with $2n$ nonterminal vertices—one for each variable. Generalizing the second variable ordering to $a_1 < \cdots < a_n < b_1 < \cdots < b_n$, on the other hand, yields an OBDD with $2(2^n - 1)$ nonterminal vertices. For large values of n, the difference between the linear growth of the first ordering versus the exponential growth of the second has a dramatic effect on the storage requirements and the efficiency of the manipulation algorithms.

Examining the structure of the two graphs of Figure 3, we can see that in the first case the variables are paired according to their occurrences in the Boolean expression $a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$. Thus,

Function Class	Complexity	
	Best	Worst
Symmetric	linear	quadratic
Integer Addition (any bit)	linear	exponential
Integer Multiplication (middle bits)	exponential	exponential

Table 1: **OBDD complexity for common function classes.**

from every second level in the graph, only two branch destinations are required: one to the terminal vertex labeled 1 for the case where the corresponding product yields 1, and one to the next level for the case where every product up to this point yields 0. On the other hand, the first 3 levels in the second case form a complete binary tree encoding all possible assignments to the a variables. In general, for each assignment to the a variables, the function value depends in a unique way on the assignment to the b variables. As we generalize this function and ordering to one over $2n$ variables, the first n levels in the OBDD form a complete binary tree.

Most applications using OBDDs choose some ordering of the variables at the outset and construct all graphs according to this ordering. This ordering is chosen either manually, or by a heuristic analysis of the particular system to be represented. For example, several heuristic methods have been devised that, given a logic gate network, generally derive a good ordering for variables representing the primary inputs. [Fujita et al 1988; Malik et al 1988]. Others have been developed for sequential system analysis [Jeong et al 1991]. Note that these heuristics do not need to find the best possible ordering—the ordering chosen has no effect on the correctness of the results. As long as an ordering can be found that avoids exponential growth, operations on OBDDs remain reasonably efficient.

1.4. Complexity Characteristics

OBDDs provide a practical approach to symbolic Boolean manipulation only when the graph sizes remain well below the worst case of being exponential in the number of variables. As the previous examples show, some functions are sensitive to the variable ordering but remain quite compact as long as a good ordering is chosen. Furthermore, there has been ample empirical evidence indicating that many functions encountered in real applications can be represented efficiently as OBDDs. One way to more fully understand the strengths and limitations of OBDDs is to derive lower and upper bounds for important classes of Boolean functions.

Table 1 summarizes the asymptotic growth rate for several classes of Boolean functions, and their sensitivity to the variable ordering. Symmetric functions, where the function value depends only the number of arguments equal to 1, are insensitive to the variable ordering. Except for the trivial case of constant functions, these functions have graphs ranging between linear (e.g., parity) and quadratic (e.g., at least half the inputs equal 1).

We can consider each output of an n-bit adder as a Boolean function over variables $a_0, a_1, \ldots, a_{n-1}$, representing one operand, and $b_0, b_1, \ldots, b_{n-1}$, representing the other operand. The function for any bit has OBDD representations of linear complexity for the ordering $a_0 < b_0 < a_1 < b_1 < \cdots < a_{n-1} < b_{n-1}$, and exponential complexity for the ordering $a_0 < \cdots < a_{n-1} < b_0 < \cdots < b_{n-1}$. In

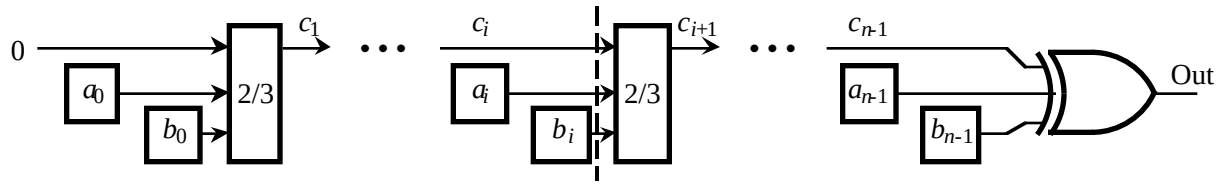

Figure 4: **Linear Arrangement of Circuit Computing Most Significant Bit of Integer Addition**

fact, these functions have representations similar to those for the function shown in Figure 3.

The Boolean functions representing integer multiplication, on the other hand, form a particularly difficult case for OBDDs. Regardless of the ordering, the Boolean function representing either of the middle two outputs of an n-bit multiplier have exponential OBDD representations. [Bryant 1991].

Upper bounds for other classes of Boolean functions can be derived based on the structural properties of their logic network realizations. Berman [Berman 1989] and more recently McMillan [McMillan 1992] have derived useful bounds for several classes of "bounded width" networks. Consider a network with n primary inputs and one primary output consisting of m "logic blocks." Each block may have multiple inputs and outputs. Primary inputs are represented by "source" blocks with no input and one output. As an example, Figure 4 shows a network having as output the most significant sum bit of an n-bit adder. This network consists of a carry chain computing the carry input c_{n-1} into the final stage. Blocks labeled "2/3" compute the MAJORITY function having 1 as output when at least two inputs are 1. The output is computed as the EXCLUSIVE-OR of the most significant bits of the inputs and c_{n-1}.

Define a *linear arrangement* of the network as a numbering of the blocks from 1 to m such that the block producing the primary output is numbered last. Define the *forward cross section* at block i as the total number of wires from an output of a block j such that $j < i$ to an input of a block k such that $i \leq k$. Define the forward cross section w_f of the circuit (with respect to an arrangement) as the maximum forward cross section for all of the blocks. As the dashed line in Figure 4 shows, our adder circuit has a forward cross section of 3. Similarly, define the *reverse cross section* at block i as the total number of wires from an output of a block j such that $j > i$ to an input of a block k such that $i \geq k$. In arrangements where the blocks are ordered *topologically* (the only case considered by Berman), such as the one shown in Figure 4, the reverse cross section is 0. Define the reverse cross section w_r of the circuit (with respect to an arrangement) as the maximum reverse cross section for all of the blocks. Given these measures, it can be shown that there is an OBDD representing the circuit function with at most $n2^{w_f 2^{w_r}}$ vertices. Furthermore, finding an arrangement with low cross section leads to a good ordering of the function variables—namely the reverse of the ordering of the corresponding source blocks in the arrangement.

This bound based on network realizations leads to useful bounds for a variety of Boolean functions. For example, functions having realizations with constant forward cross section and zero reverse cross section, such as the adder circuit of Figure 4, have linear OBDD representations. A symmetric function of n variables can be realized by a circuit having forward cross section $2+\log n$ and reverse cross section 0. This circuit consists of a series of stages to compute the total number of inputs

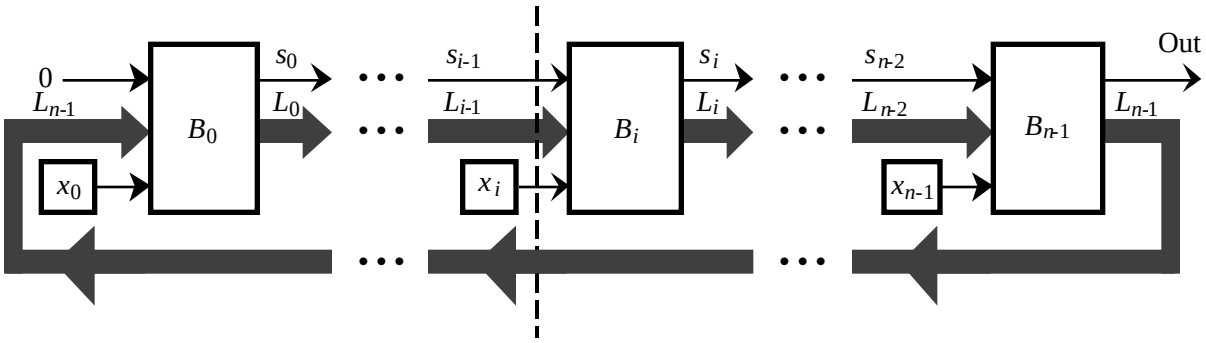

Figure 5: **Linear Arrangement of Within-K Ring Circuit**. As shown by the dashed line, the circuit has forward cross section $2 + \lceil \log_2 K \rceil$ and reverse cross section $\lceil \log_2 K \rceil$.

having value 1, encoding the total as a $\lceil \log_2 n \rceil$-bit binary number. This realization implies the quadratic upper bound in OBDD size stated in Table 1.

Figure 5 shows an application of this result for a circuit with non-zero reverse cross section. This circuit shows a general realization of the *Within-K* function, where K is some constant such that $0 < K < n$. For inputs $x_0, x_1, \ldots, x_{n-1}$, this function yields 1 if there are two inputs x_i and $x_{i'}$ equal to 1 such that i' equals $i + j$ mod n for some value j such that $0 < j < K$. As Figure 5 illustrates, this function can be computed by a series of blocks arranged in a ring, where each block B_i has as outputs a 1-bit value s_i and a k-bit integer value L_i, where $k = \lceil \log_2 K \rceil$:

$$s_i = \begin{cases} 1, & x_i = 1 \text{ and } L_{i-1} \neq 0 \\ s_{i-1}, & \text{otherwise} \end{cases}$$

$$L_i = \begin{cases} K-1, & x_i = 1 \\ L_{i-1} - 1, & x_i = 0 \text{ and } L_{i-1} > 0 \\ 0, & \text{otherwise.} \end{cases}$$

In this realization, each L_i signal encodes the number of remaining positions with which the most recent input value of 1 can be paired, while each s_i signal indicates whether a pair of 1 input values within distance K has occurred so far. To realize the modular proximity measure, output L_{n-1} of the final stage is routed back to the initial stage. Note that although this circuit has a cyclic structure, its output is uniquely defined by the input values. As the dashed line indicates, this ring structure can be "flattened" into a linear arrangement having forward cross section $k + 2$ and reverse cross section k. This construction yields an upper bound of $(8K4^K)n$ on the OBDD size. For constant values of K, the OBDD is of linear size, although the constant factor grows rapidly with K.

McMillan has generalized this technique to *tree arrangements* in which the network is organized as a tree of logic blocks with branching factor b and with the primary output produced by the block at the root. In such an arrangement, forward (respectively, reverse) cross section refers to wires directed toward (respectively, away from) the root. Such an arrangement yields an upper bound on the OBDD size of $n \left[2^b n^{b-1}\right]^{w_f 2^{w_r}}$. The upper bound for the linear arrangement is given by this formula for $b = 1$. Observe that for constant values of b, w_f, and w_r, the OBDD size is polynomial in n.

These upper bound results give some insight into why many of the functions encountered in digital design applications have efficient OBDD representations. They also suggest strategies for finding good variable orderings by finding network realizations with low cross section. Results of this form for other representations of Boolean functions could prove useful in characterizing the potential of OBDDs for other application domains.

1.5. Refinements and Variations

In recent years, many refinements to the basic OBDD structure have been reported. These include using a single, multi-rooted graph to represent all of the functions required [Brace et al 1990; Karplus 1989; Minato et al 1990; Reeves and Irwin 1987], adding labels to the arcs to denote Boolean negation [Brace et al 1990; Karplus 1989; Minato et al 1990; Madre and Billon 1988] and generalizing the concept to other finite domains [Srinivasan et al 1990]. These refinements yield significant savings in the memory requirement—generally the most critical resource in determining the complexity of the problems that can be solved. Applications that require generating over 1 million OBDD vertices are now routinely performed on workstation computers.

2. OPERATIONS

Let us introduce some notation for describing operations on Boolean functions. We will use the standard operations of Boolean algebra: $+$ for OR, $\cdot$ for AND, $\oplus$ for EXCLUSIVE-OR and an overline for NOT. In addition, we will use the symbol $\overline{\oplus}$ to indicate the complement of the EXCLUSIVE-OR operation (sometimes referred to as EXCLUSIVE-NOR). We will also use summation ($\sum$) and product ($\prod$) notation in reference to Boolean sums (OR) and products (AND). Observe that these operations are defined over *functions* as well as over the Boolean values 0 and 1. For example, if f and g are functions over some set of variables, then $f + g$ is itself a function h over these variables. For some assignment $\vec{a}$ of values to the variables, $h(\vec{a})$ yields 1 if and only if either $f(\vec{a})$ or $g(\vec{a})$ yields 1. The constant functions, yielding either 1 or 0 for all variable assignments, are denoted **1** and **0**, respectively.

The function resulting when some argument x to a function f is assigned a constant value k (either 0 or 1) is called a *restriction* of f (other references call this a "cofactor" of f [Brayton et al 1984]) denoted $f|_{x \leftarrow k}$. Given the two restrictions of a function with respect to a variable, the function can be reconstructed as

$$f \quad = \quad \overline{x} \cdot f|_{x \leftarrow 0} \; + \; x \cdot f|_{x \leftarrow 1}$$

This identity is commonly referred as the *Shannon expansion* of f with respect to x, although it was originally recognized by Boole [Brown 1990].

A variety of other useful operations can be defined in terms of the algebraic operations plus the restriction operation. The *composition* operation, where a function g is substituted for variable x of function f is given by the identity

$$f|_{x \leftarrow g} \quad = \quad \overline{g} \cdot f|_{x \leftarrow 0} + g \cdot f|_{x \leftarrow 1} \cdot$$

The *variable quantification* operation, where some variable x to function f is existentially or

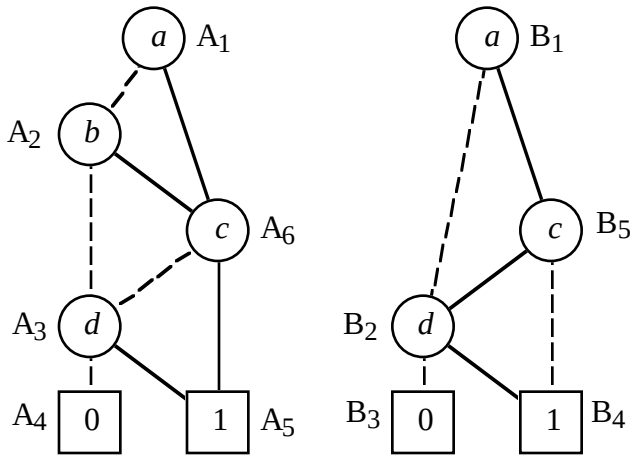

Figure 6: **Example Arguments to** APPLY **operation**. Vertices are labeled for identification during the execution trace.

universally quantified is given by the identities

$$\begin{aligned} \exists x\, f &= f|_{x \leftarrow 0} + f|_{x \leftarrow 1} \\ \forall x\, f &= f|_{x \leftarrow 0} \cdot f|_{x \leftarrow 1} \end{aligned}$$

Some researchers prefer to call these operations *smoothing* (existential) and *consensus* (universal) to emphasize that they are operations on Boolean functions, rather than on truth values [Lin et al 1990]

3. CONSTRUCTION AND MANIPULATION

A number of symbolic operations on Boolean functions can be implemented as graph algorithms applied to the OBDDs. These algorithms obey an important closure property—given that the arguments are OBDDs obeying some ordering, the result will be an OBDD obeying the same ordering. Thus we can implement a complex manipulation with a sequence of simpler manipulations, always operating on OBDDs under a common ordering. Users can view a library of BDD manipulation routines as an implementation of a Boolean function abstract data type. Except for the selection of a variable ordering, all of the operations are implemented in a purely mechanical way. The user need not be concerned with the details of the representation or the implementation.

3.1. The APPLY Operation

The APPLY operation generates Boolean functions by applying algebraic operations to other functions. Given argument functions f and g, plus binary Boolean operator $\langle op \rangle$, (e.g., AND or OR) APPLY returns the function $f\ \langle op \rangle\ g$. This operation is central to a symbolic Boolean manipulator. With it we can complement a function f by computing $f \oplus \mathbf{1}$. Given functions f and g, and "don't care" conditions expressed by the function d (i.e., $d(\vec{x})$ yields 1 for those variable assignments $\vec{x}$

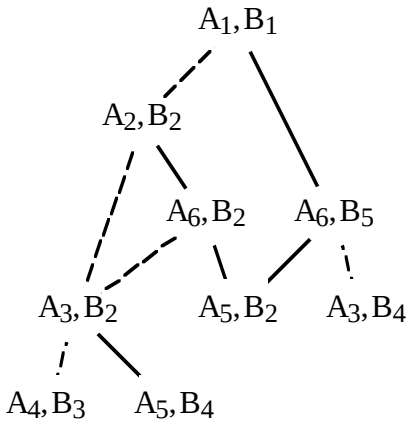

Figure 7: **Execution Trace for** APPLY **operation with operation** $+$. Each evaluation step has as operands a vertex from each argument graph.

for which the function values are unimportant,) we can test whether f and g are equivalent for all "care" conditions by computing $(f \,\overline{\oplus}\, g) + d$ and testing whether the result is the function **1**. We can also construct the OBDD representations of the output functions of a combinational logic gate network by "symbolically interpreting" the network. That is, we start by representing the function at each primary input as an OBDD consisting of a test of a single variable. Then, proceeding in order through the network, we use the APPLY operation to construct an OBDD representation of each gate output according to the gate operation and the OBDDs computed for its inputs.

The APPLY algorithm operates by traversing the argument graphs depth-first, while maintaining two hash tables: one to improve the efficiency of the computation, and one to assist in producing a maximally reduced graph. Note that whereas earlier presentations treated the reduction to canonical form as a separate step [Bryant 1986], the following algorithm produces a reduced form directly. To illustrate this operation, we will use the example of applying the $+$ operation to the functions $f(a,b,c) = (a+b)\cdot c + d$ and $g(a,b,c) = (a\cdot\overline{c}) + d$, having the OBDD representations shown in Figure 6.

The implementation of the APPLY operation relies on the fact that algebraic operations "commute" with the Shannon expansion for any variable x:

$$f\ \langle op\rangle\ g \quad = \quad \overline{x}\cdot\Big(f|_{x\leftarrow 0}\ \langle op\rangle\ g|_{x\leftarrow 0}\Big)\ +\ x\cdot\Big(f|_{x\leftarrow 1}\ \langle op\rangle\ g|_{x\leftarrow 1}\Big) \tag{1}$$

Observe that for a function f represented by an OBDD with root vertex r_f, the restriction with respect to a variable x such that $x \leq \mathit{var}(r_f)$ can be computed simply as:

$$f|_{x\leftarrow b} \quad \begin{cases} r_f, & x < \mathit{var}(r_f) \\ \mathit{lo}(r_f), & x = \mathit{var}(r_f) \text{ and } b = 0 \\ \mathit{hi}(r_f), & x = \mathit{var}(r_f) \text{ and } b = 1 \end{cases}$$

That is, the restriction is represented by the same graph, or one of the two subgraphs of the root.

Equation 1 forms the basis of a recursive procedure for computing the OBDD representation of $f\ \langle op\rangle\ g$. For our example, the recursive evaluation structure is illustrated in Figure 7. Note that

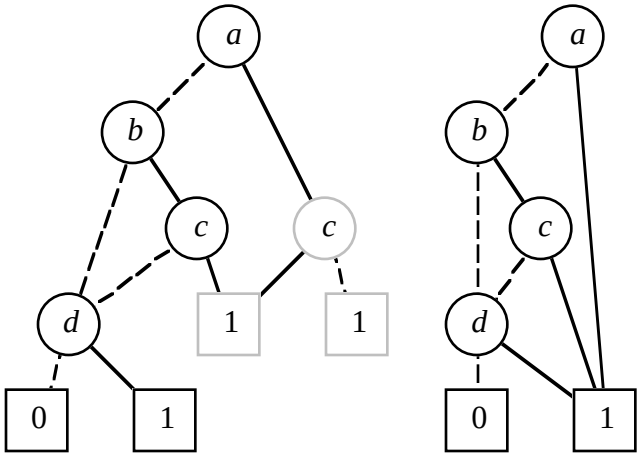

Figure 8: **Result Generation for** APPLY **operation**. The recursive calling structure naturally leads to an unreduced graph (left). By applying reduction rules at the end of each recursive call, the reduced graph is generated directly (right).

each evaluation step is identified by a vertex from each of the argument graphs. Suppose functions f and g are represented by OBDDs with root vertices r_f and r_g, respectively. For the case where both r_f and r_g are terminal vertices, the recursion terminates by returning an appropriately labeled terminal vertex. In our example, this occurs for the evaluations A_4, B_3, and A_5, B_4. Otherwise, let variable x be the *splitting* variable, defined as the minimum of variables *var*(r_f) and *var*(r_g). OBDDs for the functions $f|_{x \leftarrow 0}$ $\langle op \rangle$ $g|_{x \leftarrow 0}$ and $f|_{x \leftarrow 1}$ $\langle op \rangle$ $g|_{x \leftarrow 1}$ are computed by recursively evaluating the restrictions of f and g for value 0 (indicated in Figure 7 by the dashed lines) and for value 1 (indicated by solid lines). For our example, the initial evaluation with vertices A_1, B_1 causes recursive evaluations with vertices A_2, B_2 and A_6, B_5.

To implement the APPLY operation efficiently, we add two more refinements to the procedure described above. First, if we ever reach a condition where one of the arguments is a terminal vertex representing the "dominant" value for operation $\langle op \rangle$ (e.g., 1 for OR and 0 for AND), then we can stop the recursion and return an appropriately labeled terminal vertex. This occurs in our example for the evaluations A_5, B_2 and A_3, B_4. Second, we avoid ever making multiple recursive calls on the same pair of arguments by maintaining a hash table where each entry has as key a pair of vertices from the two arguments and as datum a vertex in the generated graph. At the start of an evaluation for arguments u and v, we check for an entry with key $\langle u, v \rangle$ in this table. If such an entry is found, we return the datum for this entry, thereby avoiding any further recursion. If no entry is found, then we follow the steps described above, creating a new entry in the table before returning the result. In our example, this refinement avoids multiple evaluations of the arguments A_3, B_2 and A_5, B_2. Observe that with this refinement, the evaluation structure is represented by a directed acyclic graph, rather than the more familiar tree structure for recursive routines.

Each evaluation step returns as result a vertex in the generated graph. The recursive evaluation structure naturally defines an unreduced graph, where each evaluation step yields a vertex labeled by the splitting variable and having as children the results of the recursive calls. For our example, this graph is illustrated on the left hand side of Figure 8. To generate a reduced graph directly,

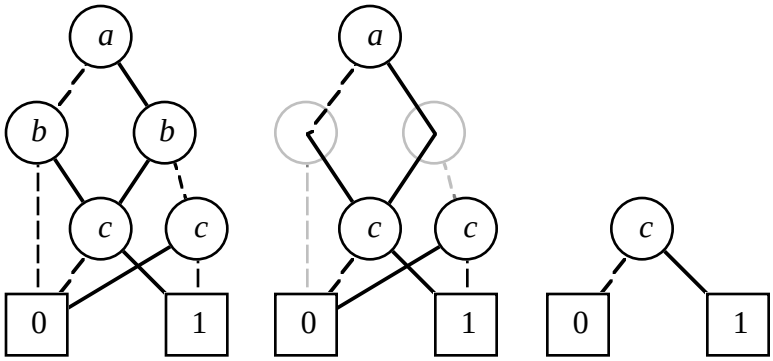

Figure 9: **Example of** RESTRICT **operation**. Restricting variable b of the argument (left) to value 1 involves bypassing vertices labeled by b (center), and reducing the graph (right).

each evaluation step attempts to avoid creating a new vertex by applying tests corresponding to the transformation rules described in Section 1.2. Suppose an evaluation step has splitting variable x, and the recursive evaluations return vertices v_0 and v_1. First we test whether $v_0 = v_1$, and if so return this vertex as the procedure result. Second, we test whether the generated graph already contains some vertex v having *var*$(v) = x$, *lo*$(v) = v_0$, and *hi*$(v) = v_1$. This test is assisted by maintaining a second hash table containing an entry for each nonterminal vertex v in the generated graph with key $\langle$*var*(v), *hi*(v), *lo*$(v)\rangle$. If the desired vertex is found, it is returned as the procedure result. Otherwise, a vertex is added to the graph, its entry is added to the hash table, and the vertex is returned as the procedure result. Similarly, terminal vertices are entered in the hash table having their labels as keys. A new terminal vertex is generated only if one with the desired label is not already present. For our example, this process avoids creating the shaded vertices shown on the left hand side of Figure 8. Instead, the graph on the right hand side is generated directly. Observe that this graph represents the function $a \ + \ b{\cdot}c \ + d$, which is indeed the result of applying the OR operation to the two argument functions.

The use of a table to avoid multiple evaluations of a given pair of vertices bounds the complexity of the APPLY operation and also yields a bound on the size of the result. That is, suppose functions f and g are represented by OBDDs having m_f and m_g vertices, respectively. Then, there can be at most $m_f\, m_g$ unique evaluation arguments, and each evaluation adds at most one vertex to the generated result. Given a good implementation of the hash tables, each evaluation step can be performed, on average, in constant time. Thus, both the complexity of the algorithm and the size of the generated result must be $O(m_f\, m_g)$.

3.2. The RESTRICT Operation

Computing a restriction to a function represented by any kind of BDD is straightforward. To restrict variable x to value k, we can simply redirect any arc into a vertex v having *var*$(v) = x$ to point either to *lo*(v) for $k = 0$, or to *hi*(v) for $k = 1$. Figure 9 illustrates the restriction of variable b in the function $b{\cdot}c \ + \ a{\cdot}\overline{b}{\cdot}\overline{c}$ to the value 1. With the original function given by the OBDD on the left, redirecting the arcs has the effect of bypassing any vertex labeled by b, as illustrated in the center.

As this example shows, a direct implementation of this technique may yield an unreduced graph. Instead, the operation is implemented by traversing the original graph depth-first. Each recursive call has as argument a vertex in the original graph and returns as result a vertex in the generated graph. To ensure that the generated graph is reduced, the procedure maintains a hash table with an entry for each vertex in the generated graph, applying the same reduction rules as those described for the APPLY operation. For our example, the result would be an OBDD representation of the function c as shown on the right hand side of the Figure 9.

Computing the restriction of a function f having an OBDD representation of m_f vertices involves at most m_f recursive calls, each generating at most one vertex in the result graph. Using a good hash table implementation, each recursive step requires constant time on average. Thus, both the complexity of the algorithm and the size of the generated result must be $O(m_f)$.

3.3. Derived Operations

As was described in Section 2, a variety of operations on Boolean functions can be expressed in terms of algebraic and restriction operations. The APPLY and the RESTRICT algorithms therefore provide a way to implement these other operations. Furthermore, for each of these operations, both the complexity and the size of the generated graph are bounded by some polynomial function of the argument functions. For function f, let m_f denote the size of its OBDD representation. Given two functions f and g, and "don't care" conditions expressed by a function d, we can compute the equivalence of f and g for the "care" conditions in time $O(m_f\, m_g\, m_d)$. We can compute the composition of functions f and g with two restrictions and three calls to APPLY. This approach would have time complexity $O(m_f^2\, m_g^2)$. By implementing the entire computation with one traversal, this complexity can be reduced to $O(m_f\, m_g^2)$ [Bryant 1986]. Finally, we can compute the quantification of a variable in a function f in time $O(m_f^2)$.

3.4. Performance Characteristics

A problem is solved using OBDDs by expressing the task as a series of operations on Boolean functions such as those discussed above. As we have seen, all of these operations can be implemented by algorithms having complexity polynomial in the sizes of the OBDDs representing the arguments. As a result, OBDD-based symbolic Boolean manipulation has two advantages over other common approaches. First, as long as the graphs remain of reasonable size, the total computation remains tractable. Second, although the graph sizes can grow with each successive operation, any single operation has reasonable worst case performance. In contrast, most other representations of Boolean functions lack this "graceful degradation" property. For example, even if a function has a reasonably compact sum of products representation, its complement may be of exponential size [Brayton et al 1984].

3.5. Implementation Techniques

From the standpoint of implementation, OBDD-based symbolic manipulation has very different characteristics from many other computational tasks. During the course of a computation, thousands of graphs, each containing thousands of vertices, are constructed and discarded. Information is

Class	Typical Operations	Typical Tests
Logic	$\wedge$, $\vee$, $\neg$, $\forall$, $\exists$	satisfiability, implication
Finite domains	*domain dependent*	equivalence
Functions	application, composition	equivalence
Sets	$\cup$, $\cap$, $-$	subset
Relations	composition, closure	symmetry, transitivity

Table 2: **Example Systems that can be Represented with Boolean Functions.**

represented in an OBDD more by its overall structure rather than in the associated data values, and hence very little computational effort is expended on any given vertex. Thus, the computation has a highly dynamic character, with no predictable patterns of memory access. To date, the most successful implementations have been on workstation computers with large physical memories, where careful attention has been given to programming the memory management routines [Brace et al 1990].

To extract maximum performance, it would be desirable to exploit the potential of pipelined and parallel computers. In symbolic analysis tasks, parallelism could exist at the *macro* level where many operations are performed simultaneously, and at the *micro* level where many vertices within a given OBDD are operated on simultaneously. Compared to other tasks that have been successfully mapped onto vector and parallel computers, OBDD manipulation requires considerably more communication and synchronization among the computing elements, and considerably less local computation. Thus, this task provides a challenging problem for the design of parallel computer architectures, programming models, and languages. Nonetheless, some of the early attempts have proved promising. Researchers have successfully exploited vector processing [Ochi et al 1991] and have shown good results executing on shared memory multiprocessors [Kimura and Clarke 1990]. Both of these implementations exploit micro parallelism by implementing the APPLY operation by a breadth-first traversal of the argument graphs, in contrast to the depth-first traversal of conventional implementations.

4. REPRESENTING MATHEMATICAL SYSTEMS

Some applications, most notably in digital logic design, call for the direct representation and manipulation of Boolean functions. In general, however, the power of symbolic Boolean manipulation comes more from the ability of binary values and Boolean operations to represent and implement a wide range of different mathematical domains. This basic principle is so well ingrained that we seldom even think about it. For example, few people would define the ADD operation of a computer as a set of 32 Boolean functions over a set of 64 arguments. Table 2 lists examples of several areas of mathematics where objects can be represented, operated on, and analyzed using symbolic Boolean manipulation, as long as the underlying domains are finite.

By providing a unified framework for a number of mathematical systems, symbolic Boolean manipulation can solve not just problems in the individual areas, but also ones involving multiple areas simultaneously. For example, recent programs to analyze the sequential behavior of digital circuits (see Section 6), involve operating in all of the areas listed in Table 2. The desired properties of

$\cdot_t$	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

$+_t$	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

a	$\overline{a}^t$
0	1
1	0
X	X

Table 3: **Ternary Extensions of** AND**,** OR**, and** NOT**.** The third value X indicates an unknown or potentially nondigital voltage.

the system are expressed as formulas in a logic. The system behavior is given by the next-state functions of the circuit. The analyzer computes sets of states having some particular properties. The transition structure of the finite state system is represented as a relation. During execution, the analyzer can readily shift between these representations, using only OBDDs as the underlying data structures. Furthermore, the canonical property of OBDDs makes it easy to detect conditions such as convergence, or whether any solutions exist to a problem.

The key to exploiting the power of symbolic Boolean manipulation is to express a problem in a form where all of the objects are represented as Boolean functions. In the remainder of this section we describe some standard techniques that have been developed along this line. With experience and practice, a surprisingly wide range of problems can be expressed in this manner. The mathematical concepts underlying these techniques have long been understood. None of the techniques rely specifically on the OBDD representation—they could be implemented using any of a number of representations. OBDDs have simply extended the range of problems that can be solved practically. In doing so, however, the motivation to express problems in terms of symbolic Boolean operations has increased.

4.1. Encoding of Finite Domains

Consider a finite set of elements A where $|A| = N$. We can encode an element of A as a vector of n binary values, where $n = \lceil \log_2 N \rceil$. This encoding is denoted by a function $\sigma\colon A \rightarrow \{0,1\}^n$ mapping each element of A to a distinct n-bit binary vector. Let $\sigma_i(a)$ denote the ith element in this encoding. A function mapping elements in A to elements in A, $f\colon A \rightarrow A$ is represented as a vector of n Boolean functions $\vec{f}$, where each $f_i\colon \{0,1\}^n \rightarrow \{0,1\}$ is defined as:

$$f_i(\sigma(a)) \;\; = \;\; \sigma_i(f(a))$$

In many applications, the domains have a "natural" encoding, e.g., the binary encoding of finite integers, while in others it is constructed artificially.

As an example, the COSMOS symbolic simulator [Cho and Bryant 1989] uses OBDDs to compute the behavior of a transistor circuit symbolically. Such a simulator can be used to automatically generate tests for faults in a circuit and to formally verify that the circuit has some desired behavior. The circuit model represents node voltages with a three-valued signal set, where values 0 and 1 represent low and high voltages, and the third value X indicates an unknown or potentially

nondigital voltage. During symbolic simulation, the node states must be computed as three-valued functions over a set of Boolean variables introduced by the user to represent values of the primary inputs or initial state. COSMOS represents the state of a node by a pair of OBDDs. That is, it encodes each of the $N = 3$ elements of the signal set as a vector of $n = 2$ binary values according to the encoding $\sigma(\ \mathsf{0}) = [0, 1]$, $\sigma(\ \mathsf{1}) = [1, 0]$, and $\sigma(\ \mathsf{X}) = [1, 1]$.

The "next-state functions" computed by the simulator are defined entirely according to this Boolean encoding, allowing Boolean functions to accurately describe the three-valued circuit behavior. For example, Table 3 shows the three-valued extensions of the logic operations AND, OR, and NOT. Observe that the operations yield X in every case where an unknown argument would cause an uncertainty in the function value. Letting $[a_1, a_0]$ denote the encoding of a three-valued signal a, the three-valued operation can be expressed entirely in terms of Boolean operations:

$$\begin{aligned} [a_1, a_0] \ \cdot_t \ [b_1, b_0] &= [a_1 \cdot b_1 \ , \ a_0 + b_0] \\ [a_1, a_0] \ +_t \ [b_1, b_0] &= [a_1 + b_1 \ , \ a_0 \cdot b_0] \\ \overline{[a_1, a_0]}^t &= [a_0, a_1] \end{aligned}$$

During operation, the simulator operates much like a conventional event-driven logic simulator. It begins with each internal node initialized to state $[\mathbf{1}, \mathbf{1}]$ indicating the node value is unknown under all conditions. During simulation, node states are updated by evaluating the Boolean representation of the next-state function with the APPLY operation. Each time the state of a node is recomputed, the old state is compared with the new state, and if not equivalent, an event is created for each fanout of the node. This process continues until the event list becomes empty, indicating that the network is in a stable state. This method of processing events relies heavily on having an efficient test for equivalence.

4.2. Sets

Given an encoding of a set A, we can represent and manipulate its subsets using "characteristic functions" [Cerny and Marin 1977]. A set $S \subseteq A$ is denoted by the Boolean function $\chi_S : \{0, 1\}^n \rightarrow \{0, 1\}$, where

$$\chi_S(\vec{x}) \ = \ \sum_{a \in S} \prod_{1 \leq i \leq n} x_i \, \overline{\oplus} \, \sigma_i(a),$$

where $\overline{\oplus}$ represents the complement of the EXCLUSIVE-OR operation. Operations on sets can then be implemented by Boolean operations on their characteristic functions, e.g.,

$$\begin{aligned} \chi_\emptyset &= \mathbf{0} \\ \chi_{S \cup T} &= \chi_S + \chi_T \\ \chi_{S \cap T} &= \chi_S \cdot \chi_T \\ \chi_{S - T} &= \chi_S \cdot \overline{\chi_T} \end{aligned}$$

Set S is a subset of T if and only if $\chi_S \cdot \overline{\chi_T} = \mathbf{0}$. In many applications of OBDDs, sets are constructed and manipulated in this manner without ever explicitly enumerating their elements.

Alternatively, a (nonempty) set can be represented as the set of possible outputs of a function vector [Coudert et al 1990]. That is, we consider $\vec{f}$ to denote the set

$$\left\{ a \;\middle|\; \sigma(a) = \vec{f}(\vec{b}), \text{ for some } \vec{b} \in \{0,1\}^n \right\}$$

This representation can be convenient in applications where the system being analyzed is represented as a function vector. By modifying these functions, we can also represent subsets of the system states.

4.3. Relations

A k-ary relation can be defined as a set of ordered k-tuples. Thus, we can also represent and manipulate relations using characteristic functions. For example, consider a binary relation $R \subseteq A \times A$. This relation is denoted by the Boolean function χ_R defined as:

$$\chi_R(\vec{x}, \vec{y}) \;=\; \sum_{a \in A} \sum_{\substack{b \in A \\ aRb}} \left[\prod_{1 \le i \le n} x_i \,\overline{\oplus}\, \sigma_i(a) \right] \cdot \left[\prod_{1 \le i \le n} y_i \,\overline{\oplus}\, \sigma_i(b) \right]$$

With this representation, we can perform operations such as intersection, union, and difference on relations by applying Boolean operations to their characteristic functions.

Using a combination of functional composition and variable quantification, we can also compose relations. That is:

$$\chi_{R \circ S} = \exists \vec{z} \, [\chi_R(\vec{x}, \vec{z}) \cdot \chi_S(\vec{z}, \vec{y})]$$

where $R \circ S$ denotes the composition of relations R and S. Quantification over a variable vector involves quantifying over each of the vector elements in any order.

Taking this further, we can compute the transitive closure of a relation using fixed-point techniques [Burch et al 1990a]. The function χ_{R^*} is computed as the limit of a sequence of functions χ_{R_i}, each defining a relation:

$$\begin{aligned} R_0 &= I \\ R_{i+1} &= I \cup R \circ R_i \end{aligned}$$

where I denotes the identity relation. The computation converges when it reaches an iteration i such that $\chi_{R_i} = \chi_{R_{i-1}}$, again making use of efficient equivalence testing. If we think of R as representing a graph, with a vertex for each element in A, and an edge for each element in R, then the relation R_i denotes those pairs reachable by a path with at most i edges. Thus, the computation must converge in at most $N - 1$ iterations, where $N = |A|$. A technique known as "iterative squaring" [Burch et al 1990a] reduces the maximum number of iterations to $n = \lceil \log_2 N \rceil$. Each iteration computes a relation $R_{(i)}$ denoting those pairs reachable by a path with at most 2^i edges:

$$\begin{aligned} R_{(0)} &= I \cup R \\ R_{(i+1)} &= R_{(i)} \circ R_{(i)} \end{aligned}$$

Many applications of OBDDs involve manipulating relations over very large sets, and hence the reduction from N iterations (e.g., 10^9) down to n (e.g., 30) can be dramatic.

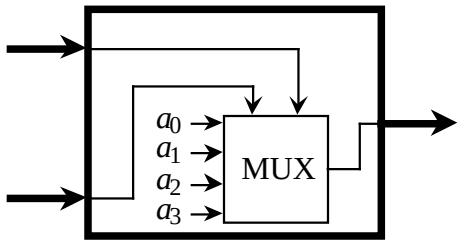

Figure 10: **Universal function block**. By assigning different values to the variables $\vec{a}$, an arbitrary 2-input operation can be realized.

5. DIGITAL SYSTEM DESIGN APPLICATIONS

The use of OBDDs in digital system design, verification, and testing has gained widespread acceptance. In this section, we describe a few of the areas and methods of application.

5.1. Verification

OBDDs can be applied directly to the task of testing the equivalence of two combinational logic circuits. This problem arises when comparing a circuit to a network derived from the system specification [Bryant 1986], or when verifying that a logic optimizer has not altered the circuit functionality. Using the APPLY operation, functional representations for both networks are derived and tested for equivalence. By this method, two sequential systems can also be compared, as long as they use the same state encoding [Madre and Billon 1988]. That is, the two systems must have identical output and next-state functions.

5.2. Design Error Correction

Not content to simply detect the existence of errors in a logic design, researchers have developed techniques to automatically correct a defective design. This involves considering some relatively small class of potential design errors, such as a single incorrect logic gate, and determining if any variant of the given network could meet the required functionality [Madre et al 1989]. This analysis demonstrates the power of the quantification operations for computing projections, in this case projecting out the primary input values by universal quantification.

Such an analysis can be performed symbolically by encoding the possible gate functions with Boolean variables, as illustrated in Figure 10. As this example shows, an arbitrary k-input gate can be emulated by a 2^k-input multiplexor, where the gate operation is determined by the multiplexor data inputs $\vec{a}$ [Mead and Conway 1990]. Consider a single output circuit N, where one of the gates is replaced by such a block, giving a resulting network functionality of $N(\vec{x}, \vec{a})$, where $\vec{x}$ represents the set of primary inputs. Suppose that the desired functionality is $S(\vec{x})$. Our task is to determine whether (and if so, how) the two functions can be made identical for all primary input values by "programming" the gate appropriately. This involves computing the function $C(\vec{a})$, defined as

$$C(\vec{a}) \;=\; \forall \vec{x}\,[N(\vec{x}, \vec{a}) \,\overline{\oplus}\, S(\vec{x})]$$

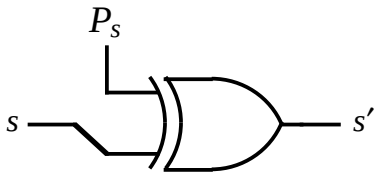

Figure 11: **Signal line modifier**. A nonzero value of P_s alters the value carried by the line.

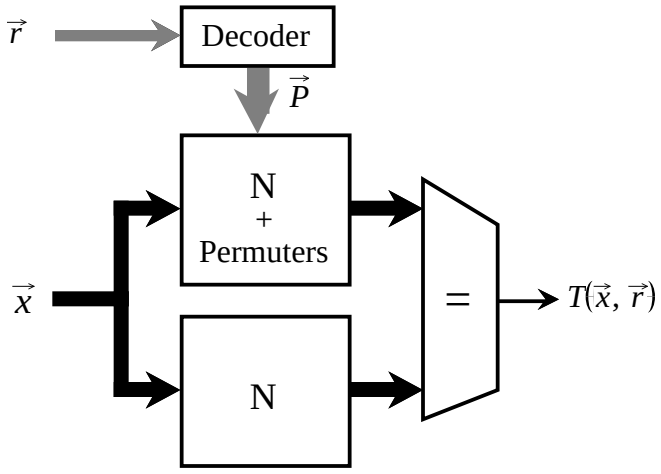

Figure 12: **Computing sensitivities to single line modifications**. Each assignment to the variables in $\vec{r}$ causes the value on just one line to be modified.

Any assignment to $\vec{a}$ for which C yields 1 is then a satisfactory solution.

Although major design errors cannot be corrected in this manner, it eliminates the tedious task of debugging circuits with common errors such as misplaced inverters, or the use of an incorrect gate type. This task is also useful in logic synthesis, where designers want to alter a circuit to meet a revised specification [Fujita et al 1991].

5.3. Sensitivity Analysis

A second class of applications involves characterizing the effects of altering the signal values on different lines within a combinational circuit. That is, for each signal line s, we want to compute the Boolean difference for every primary output with respect to s [Sellers et al 1968]. This analysis can be performed symbolically by introducing "signal line modifiers" into the network. as illustrated in Figure 11. That is, for each line that would normally carry a signal line s, we selectively alter the value to be s' under the control of a Boolean value P_s by computing $s' = s \oplus P_s$. We can determine the conditions under which some output of the circuit is sensitive to the value on a signal line by comparing the outputs of the original and altered circuits, as illustrated in Figure 12. As this figure illustrates, we can even compute the effect of every single-line modification in a circuit in one symbolic evaluation [Cho and Bryant 1989]. That is, number every signal line from 0 to $m-1$, and

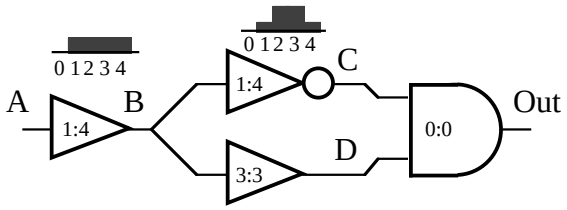

Figure 13: **Circuit with uncertain delays**. Gates labeled by min/max delays. Inverters have distribution of delays.

introduce a set of $\lceil \log m \rceil$ "permutation variables" $\vec{r}$. Each permutation signal P_s is then defined to be the function that yields 1 when the permutation variables are the binary representation of the number assigned signal s. In logic design terms, this is equivalent to generating the permutation signals with a decoder having $\vec{r}$ as input. The resulting function $T(\vec{x}, \vec{r})$ yields 1 if the original network and the network permuted by $\vec{r}$ produce the same output values for input $\vec{x}$.

One application of this sensitivity analysis is to automatic test generation. The sensitivity function describes the set of all tests for each single fault. Suppose a signal line numbered in binary as $\vec{b}$ has function $s(\vec{x})$ in the normal circuit. Then an input pattern $\vec{a}$ will detect a stuck-at-1 fault on the line if and only if $\overline{T(\vec{a}, \vec{b})} \cdot \overline{s(\vec{a})} = 1$. Similarly, $\vec{a}$ will detect a stuck-at-0 fault if and only if $\overline{T(\vec{a}, \vec{b})} \cdot s(\vec{a}) = 1$. This method can also be generalized to sequential circuits and to circuits represented at the switch-level [Cho and Bryant 1989].

A second application is in the area of combinational logic optimization. For a signal line numbered in binary as $\vec{b}$, the function $T(\vec{x}, \vec{b})$ represents the "don't care set" for each line of the circuit, i.e., those cases where the circuit outputs are independent of the signal value on this line. Using this information as guidance, the circuit optimizer can apply transformations such as eliminating a signal line, or moving a line to a different gate output. One drawback of this approach, however, is that the sensitivity function must be recomputed every time the optimizer modifies the circuit. An alternative approach yields a more restricted, but "compatible" set of don't care functions, where the don't care sets remain valid even as the circuit structure is altered [Sato et al 1990].

5.4. Probabilistic Analysis

Recently, researchers have devised a method for statistically analyzing the effects of varying circuit delays in a digital circuit [Deguchi et al 1991]. This application of OBDDs is particularly intriguing, since conventional wisdom would hold that such an analysis requires evaluation of real-valued parametric variations, and hence could not be encoded with Boolean variables.

Consider a logic gate network in which each gate has a delay given by some probability distribution. This circuit may exhibit a range of behaviors, some of which are classified as undesirable. The "yield" is then defined as the probability that these behaviors do not occur. As an example, Figure 13 shows a simple circuit where two of the logic gates have a variable distribution of delays, and we wish to evaluate the probability of a glitch occurring on node Out as the input signal A makes a transition for 0 to 1. Figure 14 shows an analysis when signal A changes to 1 at time 0. Signals

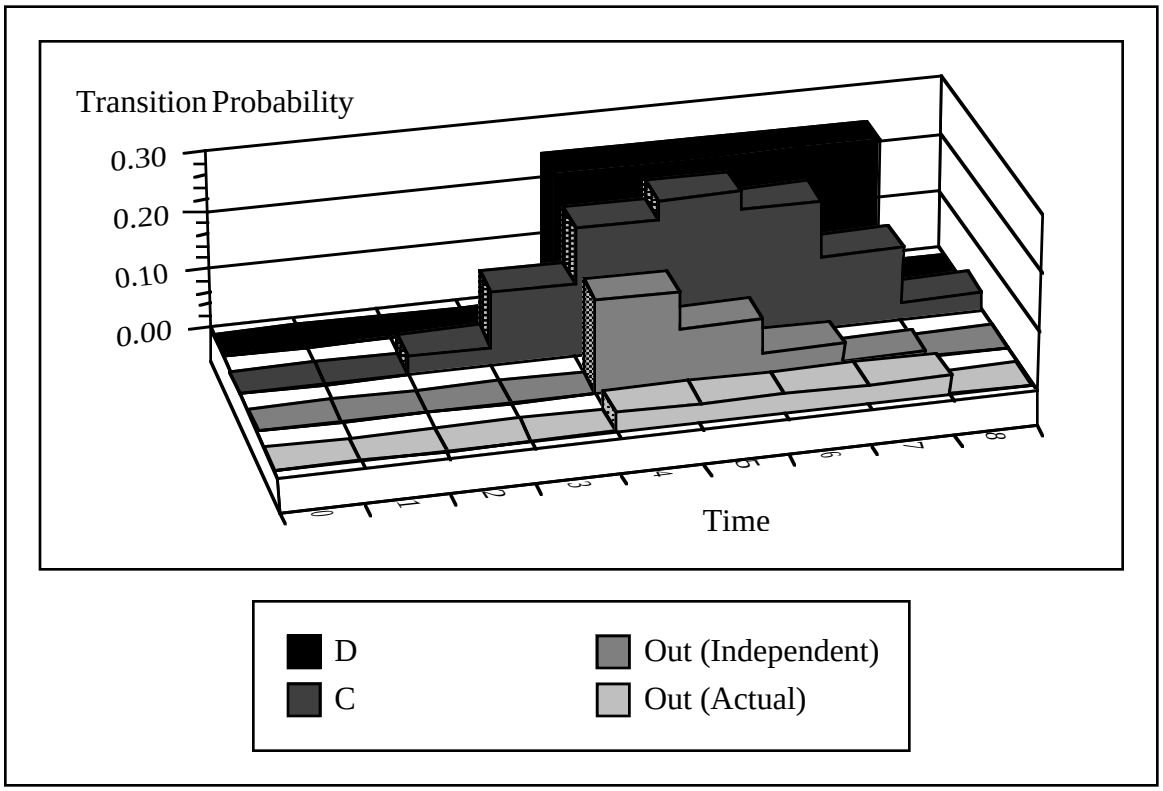

Figure 14: **Effect of uncertain delays**. Signal A switches from 0 to 1 at time 0. Ignoring signal correlations causes overestimate of transition probability.

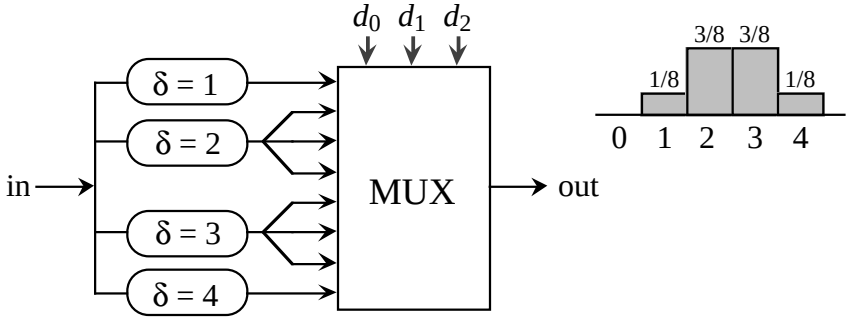

Figure 15: **Modeling uncertain delays**. Boolean variables control delay selection. Signals are replicated according to delay distribution.

C and D will make transitions, where the transition times have probability distributions shown. One simple analysis would be to treat the waveform probabilities for all signals as if they were independently distributed. Then we can easily compute the behavior of each gate output according to the gate function and input waveforms. For example, if we treat signals C and D as independent, then we could compute the probability of a rising transition on node Out at time t as the product of the probability that C makes a transition at t and the probability that no transition on D occurs at time $\leq t$. This would lead to the transition probability distribution labeled as "Out (Independent)." The net probability of a transition occurring (i.e., a glitch) would then be computed as 30%. In reality, of course, the transition times of signals C and D are highly correlated— both are affected by the delay through the initial buffer gate. Hence, a more careful analysis would yield the transition time probability distribution labeled as "Out (Actual)," having a net probability of occurrence of 12.5%. Thus, the simplified analysis underestimates the circuit yield. In other cases, a simplified analysis will overestimate the yield [Deguchi et al 1991].

To solve this problem through symbolic Boolean analysis, we must make two restrictions. First, all circuit delays must be integer-valued (for an appropriately chosen time unit), and hence transitions occur only at discrete time points. Second, the delay probabilities for a gate must be multiples of a value $1/k$, where k is a power of 2. For example both variable gates in Figure 13 have delays ranging from 1 to 4. One has uniformly distributed delays $[1/4,\ 1/4,\ 1/4,\ 1/4]$, while the other has delays that more nearly approximate a normal distribution $[1/8,\ 3/8,\ 3/8,\ 1/8]$. The delay value for a gate can then be encoded by a set of $\log k$ Boolean variables, as shown in Figure 15. That is, we model the circuit element with a k-input multiplexor, where a delay value having probability c/k is fed to c of the inputs. The circuit is then evaluated using a symbolic extension of a conventional logic gate simulation algorithm. The signal value on a node N at each time t is then a Boolean function $N(t)$ of the delay variables.

For the example of Figure 15 suppose that variables $[e_1, e_0]$ encode the delay between A and B, while variables $[d_2, d_1, d_0]$ encode the delay between B and C, as shown in Table 4. For times $t < 0$, the node functions are given as: $A(t) = B(t) = D(t) = Out(t) =$ **0** and $C(t) =$ **1**. For times $t \geq 0$, node A has function $A(t) =$ **1**, while the others would be computed as:

$$\begin{aligned} B(t) &= \overline{e_1}\cdot\overline{e_0}\cdot A(t-1) + \overline{e_1}\cdot e_0\cdot A(t-2) + e_1\cdot\overline{e_0}\cdot A(t-3) + e_1\cdot e_0\cdot A(t-4) \\ C(t) &= \overline{d_2}\cdot\overline{d_1}\cdot\overline{d_0}\cdot\overline{B(t-1)} + \overline{d_2}\cdot(d_1+d_0)\cdot\overline{B(t-2)} + \end{aligned}$$

A → B

Delay	Condition
1	$\overline{e_1}\cdot\overline{e_0}$
2	$\overline{e_1}\cdot e_0$
3	$e_1\cdot\overline{e_0}$
4	$e_1\cdot e_0$

B → C

Delay	Condition
1	$\overline{d_2}\cdot\overline{d_1}\cdot\overline{d_0}$
2	$\overline{d_2}\cdot(d_1+d_0)$
3	$d_2\cdot(\overline{d_1}+\overline{d_0})$
4	$d_2\cdot d_1\cdot d_0$

Table 4: **Delay Conditions for Example Circuit.**

$$
\begin{aligned}
 & d_2\cdot(\overline{d_1}+\overline{d_0})\cdot\overline{B(t-3)} + d_2\cdot d_1\cdot d_0\cdot\overline{B(t-4)} \\
D(t) &= B(t-3) \\
Out(t) &= C(t)\cdot D(t)
\end{aligned}
$$

From these equations, the output signal would be computed as $Out(t) = \mathbf{0}$ for $t \leq 3$ and $t \geq 8$, and for other times as:

$$
\begin{aligned}
Out(4) &= d_2\cdot d_1\cdot d_0\cdot\overline{e_1}\cdot\overline{e_0} \\
Out(5) &= d_2\cdot d_1\cdot d_0\cdot\overline{e_1}\cdot e_0 \\
Out(6) &= d_2\cdot d_1\cdot d_0\cdot e_1\cdot\overline{e_0} \\
Out(7) &= d_2\cdot d_1\cdot d_0\cdot e_1\cdot e_0
\end{aligned}
$$

We can compute a Boolean function indicating the delay conditions under which some undesirable behavior arises. For example, we could compute the probability of a glitch occurring on node Out as $G = \sum Out(t)$. In this case, we would compute $G = d_2\cdot d_1\cdot d_0$, i.e., a glitch occurs if and only if the delay between B and C equals 4.

Given a Boolean function representing the conditions under which some event occurs, we can compute the event probability by computing the *density* of the function, i.e., the fraction of variable assignments for which the function yields 1. With the aid of the Shannon expansion, the density $\rho(f)$ of a function f can be shown to satisfy the recursive formulation:

$$
\begin{aligned}
\rho(\mathbf{1}) &= 1 \\
\rho(\mathbf{0}) &= 0 \\
\rho(f) &= \frac{1}{2}\left[\rho(f|_{x\leftarrow 0}) + \rho(f|_{x\leftarrow 1})\right]
\end{aligned}
$$

Thus, given an OBDD representation of f, we can compute the density in linear time by traversing the graph depth-first, labeling each vertex by the density of the function denoted by its subgraph. This computation is shown in Figure 16 for the OBDD representing the conditions under which node C in Figure 15 has a rising transition at time 6, indicating that this event has probability $7/32$.

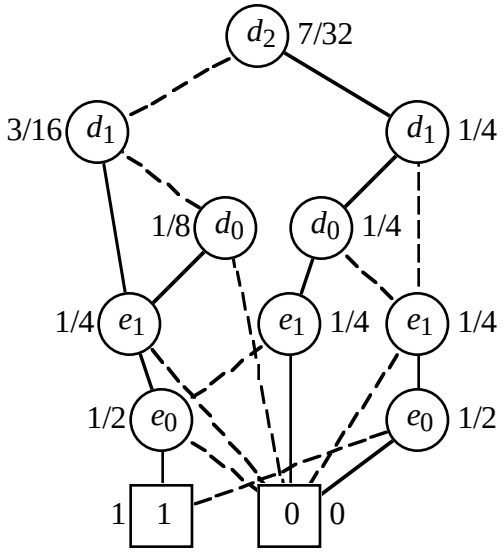

Figure 16: **Computation of Function Density**. Each vertex is labeled by the fraction of variable assignments yielding 1.

As this application shows, OBDD-based symbolic analysis can be applied to systems with complex parametric variations. Although this requires simplifying the problem to consider only discrete variations, useful results can still be obtained. The key advantage this approach has over other simplified methods of probabilistic analysis (e.g., controllability/observability measures [Brglez et al 1984]) is that it accurately considers the effects of correlations among stochastic values.

6. FINITE STATE SYSTEM ANALYSIS

Many problems in digital system verification, protocol validation, and sequential system optimization require a detailed characterization of a finite state system over a sequence of state transitions. Classic algorithms for this task construct an explicit representation of the state graph and then analyze its path and cycle structure [Clarke et al 1986]. These techniques become impractical, however, as the number of states grows large. Unfortunately, even relatively small digital systems can have very large state spaces. For example, a single 32-bit register can have over 4×10^9 states.

More recently, researchers have developed "symbolic" state graph methods, in which the state transition structure is represented as a Boolean function [Burch et al 1990a; Coudert et al 1990].[2] This involves first selecting binary encodings of the system states and input alphabet. The next-state behavior is described as a relation given by a characteristic function $\delta(\vec{x}, \vec{o}, \vec{n})$ yielding 1 when input $\vec{x}$ can cause a transition from state $\vec{o}$ to state $\vec{n}$. As an example, Figure 18 illustrates an OBDD representation of the nondeterministic automaton having the state graph illustrated in Figure 17. This example represents the three possible states using two binary values by the encoding

[2]Apparently, McMillan [McMillan 1992] implemented the first symbolic model checker in 1987, but he did not publish this work.

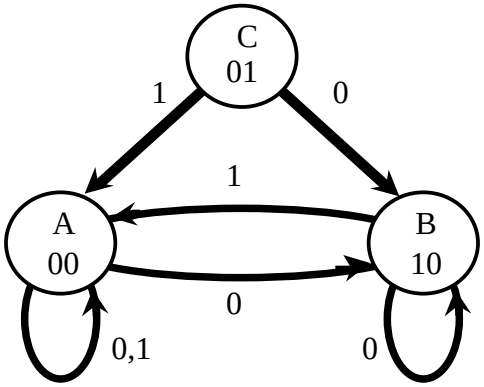

Figure 17: **Explicit representation of non-deterministic finite state machine**. The size of the representation grows linearly with the number of states.

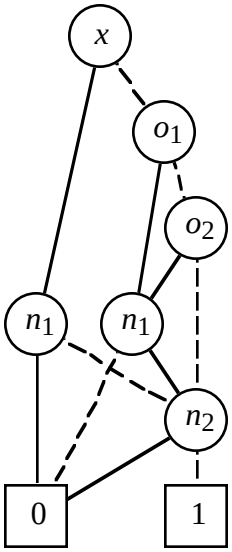

Figure 18: **Symbolic representation of non-deterministic finite state machine**. The number of variables grows logarithmically with the number of states.

$\sigma(\mathrm{A}) = [0,0]$, $\sigma(\mathrm{B}) = [1,0]$, and $\sigma(\mathrm{C}) = [0,1]$. Observe that the unused code value $[1,1]$ can be treated as a "don't care" value for the arguments $\vec{o}$ and $\vec{n}$ in the function δ. In the OBDD of Figure 18, this combination is treated as an alternate code for state C to simplify the OBDD representation.

For such a small automaton, the OBDD representation does not improve on the explicit representation. For more complex systems, on the other hand, the OBDD representation can be considerably smaller. Based on the upper bounds derived for bounded width networks discussed in Subsection 1.4, McMillan [McMillan 1992] has characterized some conditions under which the OBDD representing the transition relation for a system grows only linearly with the number of system components, whereas the number of states grows exponentially. In particular, this property holds when both (1) the system components are connected in a linear or tree structure, and (2) each component maintains only a bounded amount of information about the state of the other components. As the example of Figure 5 illustrated, this bound holds for ring-connected systems, as well, since a ring can be "flattened" into a linear chain of bidirectional links. McMillan has identified a variety of systems satisfying these conditions, including a hierarchical distributed cache in a shared memory multiprocessor, and a ring-based distributed mutual exclusion circuit.

Given the OBDD representation, properties of a finite state system can then be expressed by fixed point equations over the transition function, and these equations can be solved using iterative methods, similar to those described to compute the transitive closure of a relation. For example, consider the task of determining the set of states reachable from an initial state having binary coding $\vec{q}$ by some sequence of transitions. Define the relation S to indicate the conditions under which for some input $\vec{x}$, there can be a transition from state $\vec{o}$ to state $\vec{n}$. This relation has a characteristic function

$$\chi_S(\vec{o},\vec{n}) \quad = \quad \exists \vec{x}\,[\delta(\vec{x},\vec{o},\vec{n})]$$

Then set of states reachable from state $\vec{q}$ has characteristic function:

$$\chi_R(\vec{s}) \quad = \quad \chi_{S^*}(\vec{q},\vec{s})$$

Systems with over 10^{20} states have been analyzed by this method [Burch et al 1990b], far larger than could ever be analyzed using explicit state graph methods. A number of refinements have been proposed to speed convergence [Burch et al 1990a; Filkorn 1991] and to reduce the size of the intermediate OBDDs [Coudert et al 1990].

Unfortunately, the system characteristics that guarantee an efficient OBDD representation of the transition relation do not provide useful upper bounds on the results generated by symbolic state machine analysis. For example, one can devise a system having a linear interconnection structure for which the characteristic function of the set of reachable states requires an exponentially-sized OBDD [McMillan 1992]. On the other hand, researchers have shown that a number of real-life systems can be analyzed by these methods.

One application of finite state system analysis is in verifying the correctness of a sequential digital circuit. For example, one can prove that a state machine derived from the system specification is equivalent to one derived from the circuit even though they use different state encodings. For this application, more specialized techniques have also been developed that exploit characteristics of the system to be verified, e.g., that the circuit is synchronous and deterministic, and that the specification requires analyzing only a bounded number of clock cycles [Bose and Fisher 1989; Beatty et al 1991]. For example, we have verified pipelined data paths containing over 1000 bits of register state. Such a system, having over 10^{300} states, exceeds the capacity of current symbolic state graph methods.

7. OTHER APPLICATION AREAS

Historically, OBDDs have been applied mostly to tasks in digital system design, verification, and testing. More recently, however, their use has spread into other application domains. For example, the fixed point techniques used in symbolic state machine analysis can be used to solve a number of problems in mathematical logic and formal languages, as long as the domains are finite [Burch et al 1990a; Touati et al 1991]. Researchers have also shown that problems from many application areas can be formulated as a sets of equations over Boolean algebras which are then solved by a form of unification [Büttner and Simonis 1987].

In the area of artificial intelligence, researchers have developed a truth maintenance system based on OBDDs [Madre and Coudert 1991]. They use an OBDD to represent the "database," i.e., the known relations among the elements. They have found that by encoding the database in this form, the system can make inferences more readily than with the traditional approach of simply maintaining an unorganized list of "known facts." For example, determining whether a new fact is consistent with or follows from the set of existing facts involves a simple test for implication.

8. AREAS FOR IMPROVEMENT

Although a variety of problems have been solved successfully using OBDD-based symbolic manipulation, there are still many cases where improved methods are required. Of course, most of the problems to be solved are NP-hard, and in some cases even PSPACE-hard [Garey and Johnson 1979]. Hence, it is unlikely that any method with polynomial worst case behavior can be found. At best, we can develop methods that yield acceptable performance for most tasks of interest.

One possibility is to improve on the representation itself. For working with digital systems containing multipliers and other functions involving a complex relation between the control and data signals, OBDDs quickly become impractically large. Several methods have been proposed that follow the same general principles of OBDD-based symbolic manipulation, but with fewer restrictions on the data structure. For example, Karplus has proposed a variant termed "If-Then-Else DAGs," where the test condition for each vertex can be a more complex function than a simple variable test [Karplus 1989]. Researchers at CMU have experimented with "Free BDDs," in which the variable ordering restriction of OBDDs is relaxed to the extent that the variables can appear in any order, but no path from the root to a terminal vertex can test a variable more than once [Brace 1988]. Such graphs, known as "1-time branching programs" in the theoretical community [Wegener 1988], have many of the desirable properties of OBDDs, including an efficient (although probabilistic) method for testing equivalence [Blum and Chandra 1980]. Recently, techniques based on this representation have been developed that maintain several of the desirable characteristics of OBDDs, including a canonical form and a polynomial time APPLY operation [Gergov and Meinel 1992]. Other researchers have removed all restrictions on variable occurrence, allowing paths with multiple tests of a single variable [Ashar et al 1991; Burch 1991]. In each of these extensions, we see a trade-off between the compactness of the representation and the difficulty of constructing them or testing their properties.

In many combinatorial optimization problems, symbolic methods using OBDDs have not performed as well as more traditional methods. In these problems, we are typically interested in finding only

one solution that satisfies some optimality criterion. Most approaches using OBDDs on the other hand, derive all possible solutions and then select the best from among these. Unfortunately, many problems have too many solutions to encode symbolically. More traditional search methods such as branch-and-bound techniques often prove more efficient and able to solve larger problems. For example, our test generation program determines all possible tests for each fault [Cho and Bryant 1989], whereas more traditional methods stop their search as soon as a single test is found. One possibility would be apply the idea of "lazy" or "delayed" evaluation [Abelson et al 1985] to OBDD-based manipulation. That is, rather than eagerly creating a full representation of every function during a sequence of operations, the program would attempt to construct only as much of the OBDDs as is required to derive the final information desired. Recent test generation programs have some of this character, using a hybrid of combinatorial search and functional evaluation [Giraldi and Bushnell 1990].

9. SUMMARY

As researchers explore new application areas and formulate problems symbolically, they find they can exploit several key features of Boolean functions and OBDDs:

- By encoding the elements of a finite domain in binary, operations over these domains can be represented by vectors of Boolean functions.
- Symbolic Boolean manipulation provides a unified framework for representing a number of different mathematical systems.
- For many problems, a variable ordering can be found such that the OBDD sizes remain reasonable.
- The ability to quickly test equivalence and satisfiability makes techniques such as iterative methods and sensitivity analysis feasible.
- The APPLY and RESTRICT operations provide a powerful basis for many more complex operations.

Discovering new application areas, and improving the performance of symbolic methods (OBDD or otherwise) for existing areas will provide a fruitful area of research for many years to come.

REFERENCES

[Abelson et al 1985] Abelson, H., Sussman, G. J., and Sussman, J. 1985. *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, pp. 261–264.

[Akers 1978] Akers, S. B. 1978. Binary decision diagrams. *IEEE Transactions on Computers C-27*, 6 (Aug.), pp. 509–516.

[Ashar et al 1991] Ashar, P., Devadas, S., and Ghosh, A. 1991. Boolean satisfiability and equivalence checking using general binary decision diagrams. *International Conference on Computer Design*, (Cambridge, Oct.). IEEE, New York, pp. 259–264.

[Beatty et al 1991] Beatty, D. L., Bryant, R. E., and Seger, C.-J. H. 1991. Formal hardware verification by symbolic trajectory evaluation. *Proceedings of the 28th ACM/IEEE Design Automation Conference* (San Francisco, June), ACM, New York, pp. 397–402.

[Berman 1989] Berman, C. L. 1989. Ordered binary decision diagrams and circuit structure. *International Conference on Computer Design* (Cambridge, October), IEEE, New York, pp. 392–395.

[Blum and Chandra 1980] Blum, M. W., and Chandra, A. K. 1980. Equivalence of free Boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters 10* (March 18), pp. 80–82.

[Bose and Fisher 1989] Bose, S., and Fisher, A. L. 1989. Verifying pipelined hardware using symbolic logic simulation. *International Conference on Computer Design* (Boston, Oct.). IEEE, New York.

[Brace 1988] Brace, K. S. 1988. private communication, Carnegie Mellon University, (Pittsburgh, PA).

[Brace et al 1990] Brace, K. S., Bryant, R. E., and Rudell, R. L. 1990. Efficient implementation of a BDD package. *Proceedings of the 27th ACM/IEEE Design Automation Conference* (Orlando, June), ACM, New York, pp. 40–45.

[Brayton et al 1984] Brayton, R. K., Hachtel, G. D., McMullen, C. T., and Sangiovanni-Vincentelli, A. L.. 1984. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, Boston.

[Brglez et al 1984] Brglez, F., Pownall, P., and Hum, P. 1984. Applications of testability analysis: From ATPG to critical path tracing. *International Test Conference* (Philadelphia, Oct.), IEEE, New York, pp. 705–712.

[Brown 1990] Brown, F. M. 1990. *Boolean Reasoning.* Kluwer Academic Publishers, Boston.

[Bryant 1986] Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers C-35*, 6 (Aug.), pp. 677–691.

[Bryant 1991] Bryant, R. E. 1991. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers 40* 2 (Feb.), pp. 205–213.

[Burch et al 1990a] Burch, J. R., Clarke, E. M., and McMillan, K. 1990. Symbolic model checking: 10^{20} states and beyond. *Fifth Annual IEEE Symposium on Logic in Computer Science* (Philadelphia, June), IEEE, New York, pp. 428–439.

[Burch et al 1990b] Burch, J. R., Clarke E. M., Dill, D. L., and McMillan, K. 1990. Sequential circuit verification using symbolic model checking. *Proceedings of the 27th ACM/IEEE Design Automation Conference* (Orlando, June) ACM, New York, pp. 46–51.

[Burch 1991] Burch, J. R. 1991. Using BDDs to verify multipliers. *Proceedings of the 28th ACM/IEEE Design Automation Conference*, (San Francisco, June) ACM, New York, pp. 408-412.

[Büttner and Simonis 1987] Büttner, W. and Simonis, H. 1987. "Embedding Boolean expressions into logic programming. *Journal of Symbolic Computation 4* pp. 191–205.

[Cerny and Marin 1977] Cerny, E. and Marin, M. A. 1977. An approach to unified methodology of combinational switching circuits. *IEEE Transactions on Computers C-26*, 8 (Aug.), pp. 745–756.

[Cho and Bryant 1989] Cho, K., and Bryant, R. E., 1989. Test pattern generation for sequential MOS circuits by symbolic fault simulation. *Proceedings of the 26th ACM/IEEE Design Automation Conference* (Las Vegas, June), ACM, New York, pp. 418–423.

[Clarke et al 1986] Clarke, E. M. , Emerson, E. A., and Sistla, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages 8* 2 (April), pp. 244–263.

[Coudert et al 1990] Coudert, O. Madre, J.-C., and Berthet, C. 1990. Verifying temporal properties of sequential machines without building their state diagrams. *Computer-Aided Verification '90*, E. M. Clarke, and R. P. Kurshan, *eds*. (Rutgers, June), American Mathematical Society, pp. 75–84.

[Deguchi et al 1991] Deguchi, Y., Ishiura, N., and Yajima, S. 1991. Probabilistic CTSS: Analysis of timing error probability in asynchronous logic circuits. *Proceedings of the 28th ACM/IEEE Design Automation Conference*, (San Francisco, June) ACM, New York, pp. 650–655.

[Filkorn 1991] Filkorn, T. 1991. A method for symbolic verification of synchronous circuits. *Computer Hardware Description Languages* (Marseilles, April), IFIP, pp. 229–239.

[Fortune et al 1978] Fortune, S., Hopcroft, J., and Schmidt, E. M. 1978. The complexity of equivalence and containment for free single variable program schemes. *Automata, Languages and Programming, Lecture Notes in Computer Science, Vol. 62*, G. Goos, J. Hartmanis, Ausiello, and Boehm, *eds*. Springer-Verlag, Berlin, pp. 227-240.

[Fujita et al 1988] Fujita, M., Fujisawa, H. and Kawato, N. 1988. Evaluations and improvements of a Boolean comparison program based on binary decision diagrams. *International Conference on Computer-Aided Design* (Santa Clara, Nov.), IEEE, New York, pp. 2–5.

[Fujita et al 1991] Fujita, M., Kakuda, T., and Matsunaga, Y. 1991. Redesign and automatic error correction of combinational circuits. *Logic and Architecture Synthesis: Proceedings of the IFIP TC10/WG10.5 Workshop on Logic and Architecture Synthesis,* P. Michel, and G. Saucier, *eds.* Elsevier, Amsterdam, pp. 253–262.

[Garey and Johnson 1979] Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability*, W. H. Freeman and Company, New York.

[Gergov and Meinel 1992] Gergov, J., and Meinel, C. 1992. Efficient analysis and manipulation of OBDDs can be extended to read-once-only branching programs. Technical Report 92-10, Universität Trier, Fachbereich IV—Mathematik/Informatik, Trier, Germany.

[Giraldi and Bushnell 1990] Giraldi, J., and Bushnell, M. L. 1990. EST: The new frontier in automatic test-pattern generation. *Proceedings of the 27th ACM/IEEE Design Automation Conference* (Orlando, June), ACM, New York, pp. 667–672.

[Jeong et al 1991] Jeong, S.-W., Plessier, B., Hachtel, G. D., and Somenzi, F. 1991. Variable ordering and Selection for FSM traversal. *International Conference on Computer-Aided Design* (Santa Clara, Nov.), IEEE, New York, pp. 476–479.

[Karplus 1989] Karplus, K. 1989. Using if-then-else DAGs for multi-level logic minimization. In *Advanced Research in VLSI*, C. Seitz, *ed.*, MIT Press, Cambridge, pp. 101–118.

[Kimura and Clarke 1990] Kimura, S., and Clarke, E. M. 1990. A parallel algorithm for constructing binary decision diagrams. *International Conference on Computer Design* (Boston, Oct.), IEEE, New York, pp. 220–223.

[Lee 1959] Lee, C. Y. 1959. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal 38*, pp. 985–999.

[Lin et al 1990] Lin, B., Touati, H. J., and Newton, A. R. 1990. Don't care minimization of multi-level sequential logic networks. *International Conference on Computer-Aided Design* (Santa Clara, Nov.), IEEE, New York, pp. 414–417.

[Madre and Billon 1988] Madre, J. C., and Billon, J. P. 1988. Proving circuit correctness using formal comparison between expected and extracted behaviour. *Proceedings of the 25th ACM/IEEE Design Automation Conference*, (Anaheim, June), ACM, New York, pp. 205–210.

[Madre et al 1989] Madre, J.-C., Coudert, O., and Billon, J. P. 1989. Automating the diagnosis and rectification of design errors with PRIAM. *International Conference on Computer-Aided Design* (Santa Clara, Nov.), IEEE, New York, pp. 30–33.

[Madre and Coudert 1991] Madre, J.-C., and Coudert, O. 1991. A logically complete reasoning maintenance system based on a logical constraint solver. *12th International Joint Conference on Artificial Intelligence* (Sydney, Aug.), pp. 294–299.

[Malik et al 1988] Malik, S., Wang, A., Brayton, R. K., and Sangiovanni-Vincentelli, A. 1988. Logic verification using binary decision diagrams in a logic synthesis environment. *International Conference on Computer-Aided Design* (Santa Clara, Nov.), IEEE, New York, pp. 6–9.

[McMillan 1992] McMillan, K. L. 1992. Symbolic model checking: an approach to the state explosion problem. PhD thesis, School of Computer Science, Carnegie Mellon University.

[Mead and Conway 1990] Mead, C. A., and Conway, L. 1980. *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA.

[Meinel 1990] Meinel, C. 1990 *Modified branching programs and their computational power, Lecture Notes in Computer Science Vol. 370*, G. Goos, and J. Hartmanis, *eds*. Springer-Verlag, Berlin.

[Minato et al 1990] Minato, S., Ishiura, N., and Yajima, S. 1990. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. *Proceedings of the 27th ACM/IEEE Design Automation Conference* (Orlando, June), ACM, New York, pp. 52–57.

[Ochi et al 1991] Ochi, H., Ishiura, N., and Yajima, S. 1991. Breadth-first manipulation of SBDD of function for vector processing. *Proceedings of the 28th ACM/IEEE Design Automation Conference*, (San Francisco, June) ACM, New York, pp. 413–416.

[Reeves and Irwin 1987] Reeves, D. S., and Irwin, M. J. 1987. Fast methods for switch-level verification of MOS circuits. *IEEE Transactions on CAD/IC CAD-6* 5 (Sept.), pp. 766–779.

[Sato et al 1990] Sato, H., Yasue, Y., Matsunaga, Y., and Fujita, M. 1990. Boolean resubstitution with permissible functions and binary decision diagrams. *Proceedings of the 27th ACM/IEEE Design Automation Conference* (Orlando, June), ACM, New York, pp. 284–289.

[Sellers et al 1968] Sellers, F. F., Hsiao, M. Y., and Bearnson, C. L. 1968. Analyzing errors with the Boolean difference. *IEEE Transactions on Computers C-17*, pp. 676–683.

[Srinivasan et al 1990] Srinivasan, A., Kam, T., Malik, S., and Brayton, R. K. 1990. Algorithms for discrete function manipulation. *International Conference on Computer-Aided Design* (Santa Clara, Nov.), IEEE, New York, pp. 92–95.

[Touati et al 1991] Touati, H. J., Brayton, R. K., and Kurshan, R. P. 1991. Testing language containment for ω-automata using BDD's. *Formal Methods in VLSI Design* (Miami, Jan.), ACM, New York.

[Watanabe and Brayton 1991] Watanabe, Y., and Brayton, R. K. 1991. Heuristic minimization of multiple-valued relations. *International Conference on Computer-Aided Design* (Santa Clara, Nov.), IEEE, New York, pp. 126–129.

[Wegener 1988] Wegener, I. 1988. On the complexity of branching programs and decision trees for clique functions. *J. ACM 35* 2 (April), pp. 461–471.