Ordered Binary Decision Diagrams (OBDDs) represent Boolean functions as directed acyclic graphs. They form a canonical representation, making solving of functional problems 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 manipulation. First, the possible variations in system parameters and operating conditions are encoded with Boolean variables. Then in symbolic manipulation, Boolean functions are 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 manipulation. Additional keywords and phrases: Binary decision diagrams, branching programs, symbolic manipulation, symbolic manipulation, symbolic manipulation, symbolic manipulation, symbolic manipulation.
<table>
<thead>
<tr>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
</tr>
<tr>
<td>1. OBDD REPRESENTATION</td>
</tr>
<tr>
<td>1.1. Binary Decision Diagrams</td>
</tr>
<tr>
<td>1.2. Ordering and Reducing</td>
</tr>
<tr>
<td>1.3. Effect of Variable Ordering</td>
</tr>
<tr>
<td>1.4. Complexity Characteristics</td>
</tr>
<tr>
<td>1.5. Refinements and Variations</td>
</tr>
<tr>
<td>2. OPERATIONS</td>
</tr>
<tr>
<td>3. CONSTRUCTION AND MANIPULATION</td>
</tr>
<tr>
<td>3.1. The Operation</td>
</tr>
<tr>
<td>3.2. The Operation</td>
</tr>
<tr>
<td>3.3. The Operation</td>
</tr>
<tr>
<td>3.4. The Operation</td>
</tr>
<tr>
<td>3.5. Impedance Techniques</td>
</tr>
<tr>
<td>3.6. Performance Characteristics</td>
</tr>
<tr>
<td>3.7. Detailed Operation</td>
</tr>
<tr>
<td>3.8. The Function Operation</td>
</tr>
<tr>
<td>3.9. The Impedance Operation</td>
</tr>
<tr>
<td>3.10. The Impedance Operation</td>
</tr>
<tr>
<td>3.11. The Impedance Operation</td>
</tr>
<tr>
<td>4. REPRESENTING MATHEMATICAL SYSTEMS</td>
</tr>
<tr>
<td>4.1. Encoding of Finite Domains</td>
</tr>
<tr>
<td>4.2. Relations</td>
</tr>
<tr>
<td>4.3. Reductions</td>
</tr>
<tr>
<td>4.4. Sets</td>
</tr>
<tr>
<td>4.5. Encoding of Finite Domains</td>
</tr>
<tr>
<td>5. FORMAL SYSTEM ANALYSIS</td>
</tr>
<tr>
<td>6. OTHER APPLICATION AREAS</td>
</tr>
<tr>
<td>7. AREAS FOR IMPROVEMENT</td>
</tr>
<tr>
<td>8. REFERENCES</td>
</tr>
<tr>
<td>9. SUMMARY</td>
</tr>
</tbody>
</table>

CONTENTS

REFERENCES

SUMMARY

AREAS FOR IMPROVEMENT

OTHER APPLICATION AREAS

FINITE SYSTEM ANALYSIS

DIGITAL SYSTEM DESIGN APPLICATIONS

REPRESENTING MATHEMATICAL SYSTEMS

CONSTRUCTION AND MANIPULATION

OPERATIONS

ODD REPRESENTATION

INTRODUCTION

CONTENTS
INTRODUCTION

Many tasks in digital system design, combinatorial optimization, mathematical logic, and artificial intelligence can be framed as a problem in the domain of digital system design, where the distinction between these domains becomes blurred. By introducing a binary encoding of the elements in these domains, the problem can be further reduced to a binary encoding of the elements in these domains. This symbolic representation can be combined in terms of operations over small, finite domains. By introducing the symbolic representation, we can express a problem in a very general form. Solving this generalized problem via symbolic manipulation avoids the need to express a problem in a very general form. Solving this generalized problem via symbolic manipulation avoids the need to express a problem in a very general form.

Two forms of symbolic Boolean functions are BDDs (Binary Decision Diagrams) and OBDDs (Ordered Binary Decision Diagrams).

BDDs are defined by imposing restrictions on the BDD representation that ensure a unique form is canonized. OBDDs are defined by imposing restrictions on the BDD representation that ensure a unique form is canonized. OBDDs have been used to solve a number of problems in digital system design, combinatorial optimization, and artificial intelligence. OBDDs have been used to solve a number of problems in digital system design, combinatorial optimization, 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 OBDDs. The following section describes some basic techniques for representing and manipulating OBDDs. The following section describes some basic techniques for representing and manipulating OBDDs. The following section describes some basic techniques for representing and manipulating OBDDs.
symbolic manipulation. This issue is discussed in the next section.

concerning for any algorithm. In practice, selecting a satisfactorily efficient algorithm is critical for the efficient

1.1. Binary Decision Diagrams

they provide a suitable data structure for a symbolic Boolean manipulation.

1.2. Ordering and Reducing

(1) (1)

1. OBDD REPRESENTATION

(solid) line branches denote the case where the decision variable is 0. (1)
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 be tested easily by checking isomorphism. 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 0 as its OBDD representation. If a function is independent of variable \( x_i \), then its OBDD representation does not contain any vertices labeled \( x_i \). Thus, once OBDD representations of functions have been generated, many functional properties become easily testable.

The OBDD representation of a function is independent of the order of the variables. Ordering the variables in some systematic way can simplify the OBDD representation. However, ordering the variables in any arbitrary way yields an OBDD representation of the function.

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. For a decision tree, this approach is practical. The OBDD representation of a function has been defined, along with some properties of OBDDs. The OBDD representation is a unique canonical representation of a function that can be used to efficiently test functional properties of the function.

We define three transformation rules over the OBDD representation of the function as an OBDD.

### Transformation Rules

**Remove Redundant Nonterminals:**
If nonterminal vertices \( a \) and \( b \) have incoming arcs \( (a)\varphi = (b)\varphi \) and \( (a)\varphi \) and \( (b)\varphi \) are both variables, then eliminate \( b \) and replace all incoming arcs to \( b \) into \( a \).

**Remove Redundant Terminals:**
If nonterminal vertex \( a \) has incoming arcs \( (a)\varphi = (b)\varphi \) and \( (a)\varphi \) and \( (b)\varphi \) are both variables, then eliminate \( b \) and replace all incoming arcs to \( b \) into \( a \).

**Remove Redundant Tests:**
If nonterminal vertex \( a \) has incoming arcs \( (a)\varphi = (b)\varphi \) and \( (a)\varphi \) and \( (b)\varphi \) are both variables, then eliminate \( b \) and replace all incoming arcs to \( b \) into \( a \).
are paired according to their occurrences in the Boolean expression. Thus, Examining the structure of the two graphs of Figure 3, we can see that in the first case the variables are paired according to the variable ordering. In the second case, the variables are paired according to the opposite variable ordering. The form and size of the OBDD representation of a function depends on the variable ordering. For example, Figure 3 shows two OBDD representations of the function denoted by the Boolean expression: $1 \land (1 \lor 2) \land (1 \lor 2)$. In general, OBDDs are exponentially smaller than the corresponding decision trees when the number of variables is large. This is because the OBDD representation is more efficient in terms of storage and manipulation algorithms. For large values of $n$, the difference between the linear growth of the first ordering versus the exponential growth of the second ordering is dramatic. In practice, we can often reduce the size of the OBDD by carefully selecting the variable ordering. This is done by heuristics that take into account the structure of the Boolean expression. Examining the structure of the two graphs of Figure 3, we can see that in the first case the variables are paired according to the variable ordering.
We can consider each output of an n-bit adder as a Boolean function over variables \( \{0, 1\} \), and the same for an n-bit multiplier. As we generalize this function and consider it as over \( \{0, 1\} \) variables, for each assignment to the variables, the function value depends on the logical values of the \( n \) variables. In second case form a complete binary tree containing all \( n! \) leaves. If we care about the logical values corresponding to the leaf variables, for the case where the corresponding product yields 1, and one in the next level for vertex labelled 1.

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

<table>
<thead>
<tr>
<th>Function Class</th>
<th>Best Complexity</th>
<th>Worst Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric</td>
<td>Integer Addition (any bit)</td>
<td>Integer Multiplication (middle bits)</td>
</tr>
<tr>
<td>Quadratic</td>
<td>Linear</td>
<td>Quadratic</td>
</tr>
<tr>
<td>Linear</td>
<td>Exponential</td>
<td>Exponential</td>
</tr>
</tbody>
</table>

Table 1 summarizes the asymptotic growth rate for several classes of Boolean functions, and their

### 1.4. Complexity Characteristics

OBDDs provide a practical approach to symbolic Boolean manipulation only when the graph sizes remain reasonably efficient. However, as we discussed previously, the OBDD representation can be exponential in the number of variables. The previous results from every second level in the graph, only two branch destinations are required: one to the terminal

...
This bound based on network realizations lead to useful bounds for a variety of Boolean functions.

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

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

The corresponding source blocks in the arrangement of the circuit can be good approximations of the function values. Furthermore, finding an arrangement with the minimum OBDD representation is a difficult case for OBDDs. Regardless of the ordering, the OBDD function representing integer multiplication is a particularly interesting function, having representations similar to those for the function shown in Figure 3.

[1992] have developed useful bounds for several classes of Boolean functions. The bounds are expressed by the structural properties of the function and the resulting OBDDs. These bounds are useful for determining the complexity of a Boolean function.
McMillan has generalized this technique to tree arrangements in which the network is organized as a tree of logic blocks with branching factor $p$ and with the primary output produced by the block signal that has the most recent input value of 1. To realize the modular proximity measure $x \equiv y \mod p$ for some value of $p$, each signal encodes the number of remaining positions within which the most recent input value of 1 has occurred. In this realization, each signal encodes the number of remaining positions within which the most recent input value of 1 has occurred. This realization implies the formula for $\log_2 y$. This function can be computed by a series of blocks arranged in a tree, where each block signal encodes the number of remaining positions within which the most recent input value of 1 has occurred. This realization shows a general realization of theWithin-Y function, where $Y$ is some constant such that $Y > 0$. Figure 5 shows an application of this realization to a circuit with non-zero reverse cross section.

This realization uses a formula for $\log_2 y$, encoding the total number of remaining positions with which the most recent input value of 1 has occurred. This realization implies the formula for $\log_2 y$. This function can be computed by a series of blocks arranged in a tree, where each block signal encodes the number of remaining positions within which the most recent input value of 1 has occurred. This realization shows a general realization of theWithin-Y function, where $Y$ is some constant such that $Y > 0$. Figure 5 shows an application of this realization to a circuit with non-zero reverse cross section.

This realization uses a formula for $\log_2 y$, encoding the total number of remaining positions with which the most recent input value of 1 has occurred. This realization implies the formula for $\log_2 y$. This function can be computed by a series of blocks arranged in a tree, where each block signal encodes the number of remaining positions within which the most recent input value of 1 has occurred. This realization shows a general realization of theWithin-Y function, where $Y$ is some constant such that $Y > 0$. Figure 5 shows an application of this realization to a circuit with non-zero reverse cross section.

This realization uses a formula for $\log_2 y$, encoding the total number of remaining positions with which the most recent input value of 1 has occurred. This realization implies the formula for $\log_2 y$. This function can be computed by a series of blocks arranged in a tree, where each block signal encodes the number of remaining positions within which the most recent input value of 1 has occurred. This realization shows a general realization of theWithin-Y function, where $Y$ is some constant such that $Y > 0$. Figure 5 shows an application of this realization to a circuit with non-zero reverse cross section.

This realization uses a formula for $\log_2 y$, encoding the total number of remaining positions with which the most recent input value of 1 has occurred. This realization implies the formula for $\log_2 y$. This function can be computed by a series of blocks arranged in a tree, where each block signal encodes the number of remaining positions within which the most recent input value of 1 has occurred. This realization shows a general realization of theWithin-Y function, where $Y$ is some constant such that $Y > 0$. Figure 5 shows an application of this realization to a circuit with non-zero reverse cross section.
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 were originally recognized by Brodu [Brown 1990].

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 the functions defined on a set of variables [Karloff et al 1989; Maeder et al 1989; Staff et al 1989].
- Adding labels to the graph to indicate the function [Karloff et al 1989; Maeder et al 1989; Staff et al 1989].
- Adding additional edges to the graph to represent other functions [Karloff et al 1989; Maeder et al 1989; Staff et al 1989].
- Adding additional nodes to the graph to represent other functions [Karloff et al 1989; Maeder et al 1989; Staff et al 1989].
- Adding additional attributes to the graph to represent other functions [Karloff et al 1989; Maeder et al 1989; Staff et al 1989].
- Adding additional connections to the graph to represent other functions [Karloff et al 1989; Maeder et al 1989; Staff et al 1989].
- Adding additional colors to the graph to represent other functions [Karloff et al 1989; Maeder et al 1989; Staff et al 1989].
- Adding additional textures to the graph to represent other functions [Karloff et al 1989; Maeder et al 1989; Staff et al 1989].
- Adding additional heights to the graph to represent other functions [Karloff et al 1989; Maeder et al 1989; Staff et al 1989].

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 were originally recognized by Brodu [Brown 1990].
3.1. The APPLY Operation

The APPLY operation generates Boolean functions by applying algebraic operations to other functions. Given functions \( f \) and \( g \), and binary Boolean operator \( \oplus \) (e.g., \( \& \) or \( \lor \)), the APPLY operation returns the function \( f \oplus g \). This operation is central to a symbolic Boolean manipulation.

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. To avoid confusion, let us call them APPLY and APPLY UNIVERSE.

The APPLY operation is defined as follows:

\[
1 \rightarrow x f, 0 \rightarrow x f = f \lor
\]

\[
1 \rightarrow x f, 0 \rightarrow x f = f \land
\]

A number of symbolic operations on Boolean functions can be implemented as graph algorithms applied to OBDDs. Thus we can implement a complex manipulation with a sequence of simpler manipulations, and all of the operations are implemented in a purely mechanical way. The user need not be concerned with the details of the representation of the manipulation.

The APPLY operation is central to a symbolic Boolean manipulation. Given a function \( f \) and \( g \), the APPLY operation returns \( f \oplus g \). This operation is defined as follows:

\[
1 \rightarrow x f, 0 \rightarrow x f = f \lor
\]

\[
1 \rightarrow x f, 0 \rightarrow x f = f \land
\]

The APPLY operation is defined as follows:

\[
1 \rightarrow x f, 0 \rightarrow x f = f \lor
\]

\[
1 \rightarrow x f, 0 \rightarrow x f = f \land
\]

The APPLY operation is defined as follows:

\[
1 \rightarrow x f, 0 \rightarrow x f = f \lor
\]

\[
1 \rightarrow x f, 0 \rightarrow x f = f \land
\]
Figure 7: Execution Trace for $A \lor B$ operation with operation $\oplus$.

Each $A \lor B$ operation yields a subset of the same graph of one of the two subgraphs of the root.

\begin{align*}
I &= q \land (f_A)_I = x, \quad (f_A)_I = x \\
0 &= q \land (f_A)_I = x, \quad (f_A)_I > x \\
q \rightarrow x & \mid f
\end{align*}

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

\[ I \rightarrow x \mid \delta(\text{do}) \quad I \rightarrow x \mid f \cdot x + (0 \rightarrow x \mid \delta(\text{do}) \quad 0 \rightarrow x \mid f) \cdot x = \delta(\text{do}) \mid f \]

The implementation of the $A \lor B$ operation relies on the fact that algebraic operations "commute" with the Shannon expansion for any variable $x$:

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ (1) \]

Equation (1) forms the basis of a recursive procedure for computing the OBDD representation of each $A \lor B$ operation represented by an OBDD with root vertex $q$.

Observe that for a function represented by an OBDD with root vertex $q$.

The implementation of the $A \lor B$ operation relies on the fact that algebraic operations "commute" with the Shannon expansion for any variable $x$:

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ (1) \]

Equation (1) forms the basis of a recursive procedure for computing the OBDD representation of each $A \lor B$ operation represented by an OBDD with root vertex $q$.

Observe that for a function represented by an OBDD with root vertex $q$.

The implementation of the $A \lor B$ operation relies on the fact that algebraic operations "commute" with the Shannon expansion for any variable $x$:

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ (1) \]

Equation (1) forms the basis of a recursive procedure for computing the OBDD representation of each $A \lor B$ operation represented by an OBDD with root vertex $q$.

Observe that for a function represented by an OBDD with root vertex $q$.

The implementation of the $A \lor B$ operation relies on the fact that algebraic operations "commute" with the Shannon expansion for any variable $x$:

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ (1) \]

Equation (1) forms the basis of a recursive procedure for computing the OBDD representation of each $A \lor B$ operation represented by an OBDD with root vertex $q$.

Observe that for a function represented by an OBDD with root vertex $q$.

The implementation of the $A \lor B$ operation relies on the fact that algebraic operations "commute" with the Shannon expansion for any variable $x$:

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ (1) \]

Equation (1) forms the basis of a recursive procedure for computing the OBDD representation of each $A \lor B$ operation represented by an OBDD with root vertex $q$.

Observe that for a function represented by an OBDD with root vertex $q$.

The implementation of the $A \lor B$ operation relies on the fact that algebraic operations "commute" with the Shannon expansion for any variable $x$:

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ (1) \]

Equation (1) forms the basis of a recursive procedure for computing the OBDD representation of each $A \lor B$ operation represented by an OBDD with root vertex $q$.

Observe that for a function represented by an OBDD with root vertex $q$.

The implementation of the $A \lor B$ operation relies on the fact that algebraic operations "commute" with the Shannon expansion for any variable $x$:

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ (1) \]

Equation (1) forms the basis of a recursive procedure for computing the OBDD representation of each $A \lor B$ operation represented by an OBDD with root vertex $q$.

Observe that for a function represented by an OBDD with root vertex $q$.

The implementation of the $A \lor B$ operation relies on the fact that algebraic operations "commute" with the Shannon expansion for any variable $x$:

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ (1) \]

Equation (1) forms the basis of a recursive procedure for computing the OBDD representation of each $A \lor B$ operation represented by an OBDD with root vertex $q$.

Observe that for a function represented by an OBDD with root vertex $q$.

The implementation of the $A \lor B$ operation relies on the fact that algebraic operations "commute" with the Shannon expansion for any variable $x$:

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ (1) \]

Equation (1) forms the basis of a recursive procedure for computing the OBDD representation of each $A \lor B$ operation represented by an OBDD with root vertex $q$.

Observe that for a function represented by an OBDD with root vertex $q$.

The implementation of the $A \lor B$ operation relies on the fact that algebraic operations "commute" with the Shannon expansion for any variable $x$:

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ (1) \]

Equation (1) forms the basis of a recursive procedure for computing the OBDD representation of each $A \lor B$ operation represented by an OBDD with root vertex $q$.

Observe that for a function represented by an OBDD with root vertex $q$.

The implementation of the $A \lor B$ operation relies on the fact that algebraic operations "commute" with the Shannon expansion for any variable $x$:

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ e \text{op}_x f \cdot a \lor b = e \text{op}_x f \cdot a \lor b \]

\[ (1) \]
Recall that OBDDs for the functions $\varphi_a$, $\varphi_b$, and $\varphi_c$ are computed by the dashed lines, and the simple directed acyclic graph (DAG) leads to the reduced graph, which is the final graph produced by the algorithm. The recursive algorithm naturally defines an inductive graph where each evaluation step yields a vertex labeled $\varphi$. Each evaluation step returns a result as a vertex in the generated graph. The recursive evaluation structure naturally defines a directed acyclic graph with vertices labeled $\varphi$.

Figure 8: Result Generation for \textit{App} operation. The recursive evaluation structure naturally defines a directed acyclic graph with vertices labeled $\varphi$. Each evaluation step returns a result as a vertex in the generated graph.
The restriction operation

Computing a restriction to a function represented by any kind of BDD is straightforward. To restrict the function the arcs have the effect of bypassing any vertex labeled by q, as illustrated in the center.

Figure 9: Example of restriction operation. Restricting variable \( y \) to value 1. With the original function given by the OBDD on the left, the function \( p \cdot q \cdot x + q \cdot x \) to value 1.

Figure 9 illustrates the restriction of variable \( x \) to value 1. For \( y \) to value 0, or to \( y \) having value \( y \), we can simply redirect any arc into a vertex \( y \) having value \( y \), to point variable \( x \) to value 1. We can simply redirect any arc into a vertex \( y \) having value \( y \), to point variable \( x \) to value 1.

Restricting a restriction to a function represented by any kind of BDD is straightforward. To restrict the function the arcs have the effect of bypassing any vertex labeled by q, as illustrated in the center.

3.2. The Restrict() Operation

Computing a restriction to a function represented by any kind of BDD is straightforward. To restrict variable \( y \) to value 1, we can simply redirect any arc into a vertex \( y \) having value \( y \), to point variable \( x \) to value 1.
As this example shows, a direct implementation of this technique may yield an unReduced graph.

3.3 Implementation Techniques

The main advantage of OBDDs is their succinct representation of the structure of Boolean functions. This means that a large function can be compactly represented by a small OBDD, even though the number of vertices in the graph may be exponentially large. This property is particularly useful for operations such as restriction, which can be performed efficiently on OBDDs.

3.4 Performance Characteristics

OBDDs provide a way to express the last as a series of operations on Boolean functions.

3.5 Derived Operations

The OBDD representation of a function can be used to express the result of a variety of operations on Boolean functions. For example, the composition of two functions can be expressed as a single OBDD, which can then be used to compute the result of the composition.

As was described in Section 2, a variety of operations on Boolean functions can be expressed in terms of OBDDs. The performance characteristics of these operations are highly dependent on the size of the OBDD representation of the function. In general, the complexity of these operations is polynomial in the size of the OBDD representation of the function.

3.6 Implementation Techniques

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

From this exampleshows,adirectimplementationofthistechniquemayyieldanunreducedgraph.

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 an OBDD representing the result. For each vertex in the original graph, the OBDD is reduced, the procedure is repeated, and so on, until the desired result is obtained.
symbolic Boolean manipulation, as long as the underlying domains are finite.

4. REPRESENTING MATHEMATICAL SYSTEMS

Some applications, most notably in digital logic design, call for the direct representation and manipulation of mathematical systems. In general, however, the power of symbolic Boolean systems is more fully realized in a finite context, where the underlying domains are finite.

Table 2: Examples of Symbolic Systems

<table>
<thead>
<tr>
<th>Class</th>
<th>Typical Operations</th>
<th>Typical Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic</td>
<td>Domain, composition</td>
<td>In, –, \lor, \land</td>
</tr>
<tr>
<td>Sets</td>
<td>Composition, closure</td>
<td>\forall, \exists, \subseteq</td>
</tr>
<tr>
<td>Functions</td>
<td>Application, composition</td>
<td>Domain, range</td>
</tr>
<tr>
<td>Finite domains</td>
<td>Logic</td>
<td>Class</td>
</tr>
</tbody>
</table>
In many applications, the domains have a "natural" encoding, e.g., the binary encoding of finite integers, while in others it is constructed synthetically.

\[(v)f \] 

where each of \( v \) is defined as: 

\[
\begin{array}{c|c}
0 & 0 \\
1 & 0 \\
\end{array}
\]

The remainder of this section we describes some standard techniques that have been developed along this line. With experience and background, these techniques can be expressed in this manner. The mathematical concepts underlying these techniques have long been understood. None of the techniques rely on the OBDD representation—nothing could be implemented using any of a number of other representations. OBDDs have simply extended the range of problems that can be solved practically.

Indoing so, however, the motivation to express problems in terms of symbolic Boolean operations has increased. The key to exploiting the power of symbolic Boolean manipulation is to express a problem in a form

\[
\text{Table 3: Ternary Extensions of AND, OR, and NOT. The third value } X \text{ indicates an unknown or potentially non-digital voltage.}
\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1+</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
During simulation, the simulator operates much like a conventional event-driven logic simulator. It begins with each internal node initialized to state \([1, 1]\) 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 \(\oplus\) 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 fails heuristically rather than having an efficient network of nodes that increasingly becomes empty. An event is encountered only if the next-state function with the \(\oplus\) operation indicates that the state of a node is recomputed. Operations on sets can then be implemented by Boolean operations on their characteristic functions. For example, Table 4 shows the three-valued expressions of the logical operations \(\&\), \(\oplus\), and \(\sim\).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Boolean Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>([1, 0], [0, 1])</td>
<td>([0, 1])</td>
<td>([0, 1] + [0, 1])</td>
</tr>
<tr>
<td>([0, 1], [1, 0], [0, 1] + [0, 1])</td>
<td>([0, 1])</td>
<td>([0, 1] + [0, 1])</td>
</tr>
</tbody>
</table>

During operation, the simulator operates much like a conventional event-driven logic simulator. It begins with each internal node initialized to state \([1, 1]\) 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 \(\oplus\) 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 fails heuristically rather than having an efficient network of nodes that increasingly becomes empty. An event is encountered only if the next-state function with the \(\oplus\) operation indicates that the state of a node is recomputed. Operations on sets can then be implemented by Boolean operations on their characteristic functions. For example, Table 4 shows the three-valued expressions of the logical operations \(\&\), \(\oplus\), and \(\sim\).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Boolean Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>([1, 0], [0, 1])</td>
<td>([0, 1])</td>
<td>([0, 1] + [0, 1])</td>
</tr>
<tr>
<td>([0, 1], [1, 0], [0, 1] + [0, 1])</td>
<td>([0, 1])</td>
<td>([0, 1] + [0, 1])</td>
</tr>
</tbody>
</table>

For example, Table 4 shows the three-valued expressions of the logical operations \(\&\), \(\oplus\), and \(\sim\).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Boolean Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>([1, 0], [0, 1])</td>
<td>([0, 1])</td>
<td>([0, 1] + [0, 1])</td>
</tr>
<tr>
<td>([0, 1], [1, 0], [0, 1] + [0, 1])</td>
<td>([0, 1])</td>
<td>([0, 1] + [0, 1])</td>
</tr>
</tbody>
</table>

During operation, the simulator operates much like a conventional event-driven logic simulator. It begins with each internal node initialized to state \([1, 1]\) 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 \(\oplus\) 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 fails heuristically rather than having an efficient network of nodes that increasingly becomes empty. An event is encountered only if the next-state function with the \(\oplus\) operation indicates that the state of a node is recomputed. Operations on sets can then be implemented by Boolean operations on their characteristic functions. For example, Table 4 shows the three-valued expressions of the logical operations \(\&\), \(\oplus\), and \(\sim\).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Boolean Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>([1, 0], [0, 1])</td>
<td>([0, 1])</td>
<td>([0, 1] + [0, 1])</td>
</tr>
<tr>
<td>([0, 1], [1, 0], [0, 1] + [0, 1])</td>
<td>([0, 1])</td>
<td>([0, 1] + [0, 1])</td>
</tr>
</tbody>
</table>

During operation, the simulator operates much like a conventional event-driven logic simulator. It begins with each internal node initialized to state \([1, 1]\) 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 \(\oplus\) 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 fails heuristically rather than having an efficient network of nodes that increasingly becomes empty. An event is encountered only if the next-state function with the \(\oplus\) operation indicates that the state of a node is recomputed. Operations on sets can then be implemented by Boolean operations on their characteristic functions. For example, Table 4 shows the three-valued expressions of the logical operations \(\&\), \(\oplus\), and \(\sim\).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Boolean Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>([1, 0], [0, 1])</td>
<td>([0, 1])</td>
<td>([0, 1] + [0, 1])</td>
</tr>
<tr>
<td>([0, 1], [1, 0], [0, 1] + [0, 1])</td>
<td>([0, 1])</td>
<td>([0, 1] + [0, 1])</td>
</tr>
</tbody>
</table>
Many applications of OBDDs involve manipulating relations over very large sets, and hence the

\[(f)R \circ (g)R = (f+g)R\]
\[H \cap I = (0)R\]

Reduction from \(N\) relations to \(n\) down to \(n = 0\) can be dramatic.

A relation complex contains a relation \(R\) denoting those pairs reachable by a path with at most 2 edges:

\[\prod_{[\mathcal{N} \cup \mathcal{X}]} = 0\] each defining a relation. BUlch et al. in 1996 show that the number of relations to \(N\) can be reduced to \(N - 1\), where \(\mathcal{X}\) is the number of \(S\)-intersections. When the composition

The relation \(\mathcal{X}\) denotes those pairs reachable by a path with at most 2 edges. Thus, the composition

This representation can be convenient in applications where the system being analyzed is represented

\[S \circ \mathcal{X} = \mathcal{X} \circ \mathcal{X}\] as a function vector. By modifying these functions, we can also represent subsets of the system.

This representation can be convenient in applications where the system being analyzed is represented

\[\{0, 1\} \forall \mathcal{X} \frac{L}{R} \{0\} \forall f = (v)\mathcal{X} \frac{L}{R} \forall \mathcal{X}\]

Alternatively, a relation \(\mathcal{X}\) can be represented as the set of possible outcomes of a function vector.

Set \(S\) is a subset of \(T\) if and only if \(S \subseteq T\) in many applications of OBDDs, sets are constructed
5. DIGITAL SYSTEM DESIGN APPLICATIONS

Figure 10: Universal function block

By assigning different values to the variables \( x \) an arbitrary 2-input operation can be realized.

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 an \( \overline{\text{NOT}} \) network derived from the system specification [Bryant 1986], or when verifying that a desired function is determined. The universal quantification operations for computing projections, in this case project out the primary input values by universal quantification.

Not only can OBDDs be used to simply detect the existence of errors in a logic design, researchers have developed algorithms 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 [Mead and Conway 1989]. This analysis demonstrates the power of the quantification operations for computing projections, in this case project out the primary input values by universal quantification.

5.2. Design Error Correction

In this section, we describe a few of the areas and methods of application of OBDDs in digital system design, verification, and testing that have gained widespread acceptance.
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, we want to compute the difference between the original and altered circuit outputs.

Although major design errors cannot be corrected in this manner, this technique can be used to identify the region of influence of design errors.

Any assignment to a variable which yields 1 is then a satisfactory solution.

5.3. Sensitivity Analysis

Figure 11: Signalline modifier. A nonzero value of $s$ alters the value carried by the line.

Figure 12: Computing sensitivities to single line modifications. Each assignment to the variables in $I$ causes the value on just one line to be modified.
5.4 Probabilistic Analysis

Recently, researchers have developed a method for statistically analyzing the effects of random delays in a digital circuit. This application of ODBDD is particularly intriguing, since conventional wisdom would hold that such probabilistic evaluation of real-valued delay is impossible. However, the current approach can apply standard algorithms such as oblivious and adaptive, which do not require the circuit to be a digital circuit, but only that the probabilities are known. In this way, the function $T(x)$, representing the 'don't care' set of all input patterns for each line of the circuit, is computed by the network and the circuit is reconfigured to produce the same output value. The sensitivity function is then defined as the probability that the circuit output is different from the normal output. The sensitivity function is then computed by the network and the circuit is reconfigured to produce the same output value. The sensitivity function is then defined as the probability that the circuit output is different from the normal output.

One application of this sensitivity analysis is to automatic test generation. The sensitivity function is used to determine which test patterns are most effective in detecting faults. This method can also be generalized to sequential circuits and to circuits represented in a switched-resistor model. The sensitivity function is then defined as the probability that the circuit output is different from the normal output. The sensitivity function is then computed by the network and the circuit is reconfigured to produce the same output value. The sensitivity function is then defined as the probability that the circuit output is different from the normal output.

Figure 13: Circuit with uncertain delays. Gates labeled by minimum delays. Inverters have distribution of delays.
Figure 14: Effect of uncertain delays. Signal 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.
Thus, given an \textit{OBDD} representation of \( f \), we can compute the density in linear time by iteration:

\[
\left[ (I \rightarrow x | f) d + (0 \rightarrow x | f) d \right] \frac{\tau}{I} = (f) d \\
0 = (0) d \\
1 = (1) d
\]

From these equations, the output signal would be computed as

\[
\left( i \right) \bar{A} \cdot \left( \bar{I} \right) C = \left( i \right) m O \\
\left( \bar{E} - i \right) \bar{G} = \left( i \right) G \\
\left( \bar{V} - i \right) \bar{D} \cdot \left( \bar{D} \right) \bar{P} + \left( \bar{E} - i \right) \bar{D} \cdot \left( \bar{D} + 1 \right) \bar{P} \bar{P}
\]

Table 4: Delay Conditions for Example Circuit.
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.

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. This involves a detailed characterization of the state graph, which is represented as a Boolean function. This function describes the next-state behavior given by the characteristic function $\delta(x')$ yielding 1 when input $x'$ can cause a transition from state $x$ to state $x'$. An example of this application is shown in Figure 17. This example illustrates binary encodings of the system states and input alphabet. The next-state function is represented as a Boolean function. The classical algorithms for this task construct an explicit representation of the state graph and then analyze its path and cycle structure. These techniques become impractical 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 $4 \times 10^9$ states. However, for the number of states grows large, this technique becomes impractical, and the state graph structure becomes impractical. As this application shows, OBDD-based symbolic analysis is a powerful method of probabilistic analysis (e.g., controllability/observability measures) that can handle large numbers of states. Although this requires simplifying the problem to consider only discrete parametric variations, simplified methods of probabilistic analysis can still be applied. The key to applying this approach is over other probabilistic analysis. Although this requires simplifying the problem to consider only discrete parametric variations, simplified methods of probabilistic analysis can still be applied.
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.
A system, having over 10^{60} states, exceeds the capacity of current symbolic state-graph methods. For example, we have verified properties of clock cycles [Bose and Fisher 1998; Beatty et al. 1999], and real-time or other systems requiring only a bounded number of clock cycles [Bose and Fisher 1998; Beatty et al. 1999].

Moreover, our technique may be extended to systems with a single unused code value z_{11}. For example, one can prove that a single machine derived from the system specification is equivalent to one derived from the circuit even though they use different state encodings. For this equivalence, one can prove that a single machine derived from the system specification is equivalent to one derived from the circuit even though they use different state encodings. For this equivalence, one can prove that a single machine derived from the system specification is equivalent to one derived from the circuit even though they use different state encodings.

One application of these symbolic methods is the OBDD. In [McMillan 1992], we showed that, for example, one can derive a symbolic machine having a bounded-width OBDD representation from a symbolic machine having a bounded-width OBDD representation.

Systems with over 10^{20} states have been analyzed by this method [Burch et al. 1990], far larger than could ever be analyzed explicitly. A number of refinements have been proposed to speed convergence [Burch et al. 1990; Finkbeiner et al. 1999] and to reduce the size of the intermediate OBDD (Cooper et al. 1999). The OBDD representation can be used to analyze the symbolic representation that guarantees an efficient OBDD representation of the system.

Given the OBDD representation, properties of a finite-state system can 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 by some sequence of transitions. This relation has a characteristic function, and the set of states reachable from state \( s \) has characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]

Then set of states reachable from state \( s \) has characteristic function:

\[
\left[ \left( u \sim \cdot \right)^{n} x \right] E = \left( u \sim \cdot \right)^{n} x
\]

For some symbol \( \gamma \), there can be a transition from state \( s \) to state \( t \). This relation has a characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]

For some symbol \( \gamma \), there can be a transition from state \( s \) to state \( t \). This relation has a characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]

For some symbol \( \gamma \), there can be a transition from state \( s \) to state \( t \). This relation has a characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]

For some symbol \( \gamma \), there can be a transition from state \( s \) to state \( t \). This relation has a characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]

For some symbol \( \gamma \), there can be a transition from state \( s \) to state \( t \). This relation has a characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]

For some symbol \( \gamma \), there can be a transition from state \( s \) to state \( t \). This relation has a characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]

For some symbol \( \gamma \), there can be a transition from state \( s \) to state \( t \). This relation has a characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]

For some symbol \( \gamma \), there can be a transition from state \( s \) to state \( t \). This relation has a characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]

For some symbol \( \gamma \), there can be a transition from state \( s \) to state \( t \). This relation has a characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]

For some symbol \( \gamma \), there can be a transition from state \( s \) to state \( t \). This relation has a characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]

For some symbol \( \gamma \), there can be a transition from state \( s \) to state \( t \). This relation has a characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]

For some symbol \( \gamma \), there can be a transition from state \( s \) to state \( t \). This relation has a characteristic function:

\[
\left( s \sim \cdot \right)^{n} x = (s)^{n} x
\]
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 sets of equations over Boolean algebra which are then solved by a form of unification [B{"u}ttner and Simonis 1987].

In the area of artificial intelligence, researchers have developed a truth maintenance system based on OBDDs [McDermott and Condor 1961]. They use an OBDD to represent the database. If a new clause affects the database, the OBDD is simplified. This is done repeatedly until the database is represented by a single OBDD. The system then can make inferences more readily than with the traditional approach of simplifying the entire database at once. The OBDDs are used to represent the database because they are small and the system can make inferences more readily than with the traditional approach of simplifying the entire database at once.

In many combinatorial optimization problems, symbolic methods using OBDDs have not performed well as well as more traditional methods. In these problems, we are specifically interested in finding only the solution(s) that minimize a function of the variables. The OBDDs are used to represent the database because they are small and the system can make inferences more readily than with the traditional approach of simplifying the entire database at once.

ARIES FOR IMPROVEMENT

Although a variety of problems have been solved successfully using OBDD-based symbolic manipulation in many combinatorial optimization problems, symbolic methods using OBDDs have not performed well as well as more traditional methods. In these problems, we are specifically interested in finding only the solution(s) that minimize a function of the variables. The OBDDs are used to represent the database because they are small and the system can make inferences more readily than with the traditional approach of simplifying the entire database at once.
Discovering new application areas and improving the performance of symbolic methods (OBDDs)

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

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

REFERENCES


9. SUMMARY

Discovering new application areas and improving the performance of symbolic methods (OBDDs)

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

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

REFERENCES


Meinel, C. 1990. Modified branching programs and their computational power.


