MULTI-VALUED DECISION DIAGRAMS FOR LOGIC SYNTHESIS AND VERIFICATION

by

Timothy Kam, Tiziano Villa, Robert K. Brayton,
and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M96/75

3 December 1996

MULTI-VALUED DECISION DIAGRAMS FOR LOGIC SYNTHESIS AND VERIFICATION

by

Timothy Kam, Tiziano Villa, Robert K. Brayton,
and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M96/75

3 December 1996

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Multi-valued Decision Diagrams for Logic Synthesis and Verification

Timothy Kam[1] Tiziano Villa[2] Robert K. Brayton[2]
Alberto L. Sangiovanni-Vincentelli[2]

[1]Intel Development Labs
Intel Corporation
Hillsboro, Oregon 97124-6497

[2]Department of EECS
University of California at Berkeley
Berkeley, CA 94720

Abstract

Many problems can be stated naturally using variables that have multiple values (i.e., take their values from a discrete set). Functions defined on these variables can also take on values from a discrete set. Examples of such problems range from combinatorial optimization such as routing and resource scheduling, to logic simulation and formal verification, and to logic synthesis such as state minimization and state assignment. In many cases these problems are NP-Complete or coNP-Complete. Compact representation and efficient manipulation of such multi-valued functions are key to the design of efficient algorithms that advance the frontier of the problems that can be solved exactly. Binary decision diagrams (BDDs) are such a compact representation for problems involving binary variables.

In this paper, we define the multi-valued decision diagram (MDD), which is a canonical representation of a multi-valued function as a directed acyclic graph. We analyze its properties, and provide algorithms for constructing and manipulating MDDs. With our MDD package, an MDD is mapped into a BDD using either a logarithmic encoding or a 1-hot encoding, each suitable for a different class of applications. We have applied both kinds of MDD to many different applications and this paper serves as a summary of the work done so far. Furthermore, general problem solving techniques, such as binate table covering and other graph algorithms have been formulated using MDDs.

1 Introduction

In this paper, we shall introduce techniques for efficient representation and manipulation of objects. We say that a representation is **explicit** if the objects are listed internally one by one. Objects are manipulated explicitly, if they are processed one after another. An **implicit** representation means a shared representation of the objects, such that the size of the representation is not linearly proportional to the number of objects in it. In an implicit manipulation many objects are processed simultaneously in one step.

The objects we need to represent include functions, relations, sets, and sets of sets. In Section 2, we introduce a new data structure called the multi-valued decision diagram (MDD) which can represent multi-valued input multi-valued output functions. An MDD is a generalization of a binary decision diagram

(BDD) [6]. BDDs represent binary input binary output functions, and they will be reviewed in Section 3. Relations and sets can be expressed in terms of characteristic functions [7] as shown in Section 3. The remaining of the paper describes efficient representations of multi-valued input binary output functions. Section 4 shows that an MDD can be mapped to an equivalent BDD after choosing an encoding for each multi-valued variable. In Section 5, the logarithmically encoded MDD is described. And in Section 6, the 1-hot encoded MDD is introduced. The latter is particularly useful for an efficient representation of set of sets, so an extensive suite of operators will be introduced for efficient manipulations of sets of sets. Issues of MDD variable ordering will be discussed in Section 7. Applications of logarithmic encoded MDDs are discussed in Section 8 and of 1-hot encoded MDDs in Section 9. Section 10 closes with final remarks.

First we give definitions pertaining to MDDs.

Definition 1.1 *Let $\mathcal{F}$ be a multi-valued input multi-valued output function of n variables - $x_1, x_2, \ldots, x_n$:*

$$\mathcal{F} : P_1 \times P_2 \times \ldots \times P_n \rightarrow Y$$

Each variable, x_i, may take any one of the p_i values from a finite set $P_i = \{0, 1, \ldots, p_i - 1\}$. The output of $\mathcal{F}$ may take m values from the set $Y = \{0, 1, \ldots, m-1\}$. Without loss of generality, we may assume that the domain and range of $\mathcal{F}$ are integers. In particular, $\mathcal{F}$ is a binary-valued output function if $m = 2$, and $\mathcal{F}$ is a binary-valued input function if $p_i = 2$ for every $i : 1 \leq i \leq n$.

Definition 1.2 *Let T_i be a subset of P_i. $x_i^{T_i}$ represents a* **literal** *of variable x_i which is defined as the Boolean function:*

$$x_i^{T_i} = \begin{cases} 0 & \text{if } x_i \notin T_i \\ 1 & \text{if } x_i \in T_i \end{cases}$$

Definition 1.3 *The* **cofactor** *of $\mathcal{F}$ with respect to a variable x_i taking a constant* **value** *j is denoted by $\mathcal{F}_{x_i^j}$ and is the function resulting when x_i is replaced by j:*

$$\mathcal{F}_{x_i^j}(x_1, \ldots, x_n) = \mathcal{F}(x_1, \ldots, x_{i-1}, j, x_{i+1}, \ldots, x_n)$$

Definition 1.4 *The* **cofactor** *of $\mathcal{F}$ with respect to a* **literal** *$x_i^{T_i}$ is denoted by $\mathcal{F}_{x_i^{T_i}}$ and is the union of the cofactors of $\mathcal{F}$ with respect to each value the literal represents:*

$$\mathcal{F}_{x_i^{T_i}} = \bigcup_{j \in T_i} \mathcal{F}_{x_i^j}$$

The cofactor of $\mathcal{F}$ is a simpler function than $\mathcal{F}$ itself because the cofactor no longer depends on the variable x_i.

Definition 1.5 *The* **Shannon decomposition** *of a function $\mathcal{F}$ with respect to a variable x_i is:*

$$\mathcal{F} = \sum_{j=0}^{p_i - 1} x_i^j \cdot \mathcal{F}_{x_i^j}$$

The Shannon decomposition expresses function $\mathcal{F}$ as a sum of simpler functions, i.e., its cofactors $\mathcal{F}_{x_i^j}$. This allows us to construct a function by recursive decomposition.

2 Multi-valued Decision Diagrams

This section describes a new data structure - the multi-valued decision diagram [35, 19] that is used to solve discrete variable problems [19, 21, 41, 22, 36, 2, 4, 27, 31, 30]. Our definition of multi-valued decision diagrams closely follows that of Bryant, [6], with two exceptions: we do not restrict ourselves to the Boolean domain, and the range of our functions is multi-valued.

Definition 2.1 ***A multi-valued decision diagram*** *(MDD) is a rooted, directed acyclic graph. Each nonterminal vertex v is labeled by a multi-valued variable $var(v)$ which can take values from a range $range(v)$. Vertex v has arcs directed towards $|range(v)|$ children vertices, denoted by $child_k(v)$ for each $k \in range(v)$. Each terminal vertex u is labeled a value $value(u) \in Y = \{0, 1, \ldots, m-1\}$.*

Each vertex in an MDD represents a multi-valued input multi-valued output function and all used-visible vertices are roots. The terminal vertex v represent the constant (function) $value(u)$. For each nonterminal vertex v representing a function F, its child vertex $child_k(v)$ represents the function F_{v^k} for each $k \in range(v)$. Therefore $F = \sum_{k \in range(v)} v^k \cdot F_{v^k}$.

For a given assignment to the variables, the value yielded by the function is determined by tracing a decision 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.

Example 2.1 *The MDD in Figure 1 represents the discrete function $F = max(0, x - y)$ where x and y are 3-valued variables.*

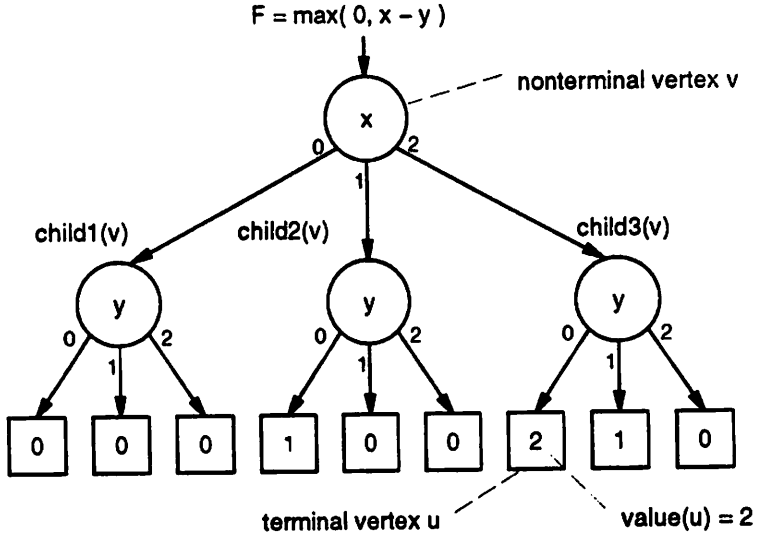

Figure 1: Example of an MDD for a discrete function.

A multi-terminal binary decision diagram (MTBDD) [9] is an MDD, with nonterminal vertices restricted to binary variables and terminals taking values from a discrete set. A type of MTBDD called algebraic decision diagram (ADD) [33] has been implemented at the University of Colorado at Boulder and used to produce implicit algorithms for manipulating matrices. From Section 3 onwards, MDDs will be used only for binary-valued output functions, however the theory is valid for the more general multi-valued functions.

2.1 Reduced Ordered MDDs

Definition 2.2 *An MDD is* **ordered** *if there is a total order $\prec$ over the set of variables such that for every nonterminal vertex v, $var(v) \prec var(child_k(v))$ if $child_k(v)$ is also nonterminal.*

Definition 2.3 *An MDD is* **reduced** *if*

1. *it contains no vertex v such that all outgoing arcs from v point to a same vertex, and*
2. *it does not contain two distinct vertices v and v' such that the subgraphs rooted at v and v' are isomorphic.*

Definition 2.4 *A* **reduced ordered multi-valued decision diagram** *(ROMDD) is an MDD which is both reduced and ordered.*

Henceforth, we consider only ROMDDs and the name MDD will be used to mean ROMDD.

Variable ordering must be decided before the construction of any MDD. We assume that this has been decided and that the naming of input variables has been permuted so that $x_i \prec x_{i+1}$. MDDs are guaranteed to be reduced at any time during the constructions and operations on MDDs. Each operation returns a resultant MDD in a reduced ordered form.

Example 2.2 *The ROMDD for the MDD in Figure 1 is shown in Figure 2. The variable ordering is $x \prec y$. Note that one redundant nonterminal vertex and six terminal vertices have been eliminated.*

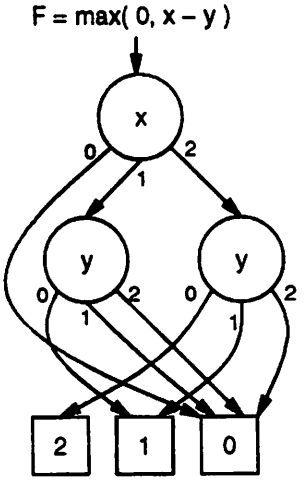

Figure 2: Reduced ordered MDD for the same function.

A very desirable property of an ROMDD is that it is a canonical representation.

Theorem 2.1 *For any multi-valued function $\mathcal{F}$, there is a unique reduced ordered (up to isomorphism) MDD denoting $\mathcal{F}$. Any other MDD denoting $\mathcal{F}$ contains more vertices.*

Proof. The complete proof has been given in [35]. □

Corollary 2.1 *Two functions are equivalent if and only if the ROMDDs for each function are isomorphic.*

2.2 CASE Operator

The CASE operator forms the basis for manipulating MDDs. All operations on discrete functions can be expressed in terms of the CASE operator on MDDs.

Definition 2.5 *The* **CASE operator** *selects and returns a function G_i according to the value of the function F:*

$$CASE(F, G_0, G_1, \ldots, G_{m-1}) = G_i \; if \; (F = i)$$

The operator is defined only if range(F) = {0, 1, ..., m-1}. The range of the function returned from the CASE operation is range(G_i). In particular, if the G_i are binary-valued, the resultant function will also be a binary-valued output function.

The input parameters to the CASE operator are, in general, multi-valued functions given in the form of MDDs. The task is to generate the resultant function $H = CASE(F, G_0, G_1, \ldots, G_{m-1})$. Since the selector F can be a function instead of a variable, we need a recursive algorithm to compute the CASE operator.

If the selector is a variable x, the following function returned by the CASE operator corresponds to a vertex with a top variable x and with children functions $G_0, G_1, \ldots, G_{p_i-1}$. The vertex is denoted by a $(p_i + 1)$-tuple on the right:

$$CASE(x, G_0, G_1, \ldots, G_{p_i-1}) = (x, G_0, G_1, \ldots, G_{p_i-1}) \tag{1}$$

Moreover, it holds:

$$CASE(F, 0, 1, \ldots, m-1) = F \tag{2}$$

Equation 1 and 2 will form the terminal cases for our recursive algorithm.

Notice that the Shannon decomposition of H with respect to x can be realized by:

$$\begin{aligned} H &= \sum_{j=0}^{p-1} x^j \cdot H_{x^j} \\ &= CASE(x, H_{x^0}, H_{x^1}, \ldots, H_{x^{p-1}}) \\ &= (x, H_{x^0}, H_{x^1}, \ldots, H_{x^{p-1}}) \end{aligned} \tag{3}$$

Recursion is based on the following reasoning. Remember that we can express a complex function in terms of its cofactors using Shannon decomposition. The cofactors of a function are simpler to compute than the original function. So to compute the CASE of complex functions, we first compute the CASE of their cofactors and then compose them together using Shannon decomposition. More rigorously,

$$\begin{aligned} CASE(F, G_0, G_1, \ldots, G_{m-1}) &= \sum_{i=0}^{p-1} x^i \cdot CASE(F, G_0, G_1, \ldots, G_{m-1})_{x^i} \\ &= \sum_{i=0}^{p-1} x^i \cdot (G_j \; if \; (F = j))_{x^i} \\ &= \sum_{i=0}^{p-1} x^i \cdot (G_{j\,x^i} \; if \; (F = j)_{x^i}) \\ &= \sum_{i=0}^{p-1} x^i \cdot (G_{j\,x^i} \; if \; (F_{x^i} = j)) \\ &= CASE(x, \end{aligned}$$

$$
\begin{aligned}
&CASE(F_{x^0}, G_{0\,x^0}, G_{1\,x^0}, \ldots, G_{m-1\,x^0}),\\
&CASE(F_{x^1}, G_{0\,x^1}, G_{1\,x^1}, \ldots, G_{m-1\,x^1}),\\
&\ldots.\\
&CASE(F_{x^{p-1}}, G_{0\,x^{p-1}}, G_{1\,x^{p-1}}, \ldots, G_{m-1\,x^{p-1}}))
\end{aligned}
\tag{4}
$$

```
CASE(F, G_0, ..., G_{m-1}) {
    if terminal case return result
    if CASE(F, G_0, ..., G_{m-1}) in computed-table return result
    let x be the top-variable of F, G_0, ..., G_{m-1}
    let p be the number of values x takes
    for j = 0 to (p - 1) do
        H_xj = CASE(F_xj, G_0xj, ..., G_{m-1 xj});
    result = (x, H_x0, ..., H_{xp-1})
    insert result in computed-table for CASE(F, G_0, ..., G_{m-1})
    return result
}
```

Figure 3: Pseudo-code for the $CASE$ algorithm.

The pseudo-code for the recursive $CASE$ algorithm is given in Figure 3. First, the algorithm checks for terminal cases. Then if the function needed has already been computed and stored in the unique table, it will be returned. If not, the cofactors H_{xj} of the function H are computed by calling $CASE$ recursively with the cofactors $F_{xj}, G_{0\,xj}, \ldots, G_{m-1\,xj}$ as its arguments. These are composed together using Shannon decomposition. By Equation 3, Shannon decomposition with respect to x is equivalent to the $(p+1)$-tuple $(x, H_{x^0}, \ldots, H_{x^{p-1}})$.

It is shown in [35] that the worst-case time complexity of the $CASE$ algorithm is $O(p_{max} \cdot \mid F \mid \cdot \mid G_0 \mid \ldots \mid G_{m-1} \mid)$.

3 Binary Decision Diagrams

The literal x_i denotes that variable x_i has the value 1 and the literal $\overline{x_i}$ denotes that variable x_i has the value 0. Cofactors with respect to literals are similar to the ones in the previous section, and are formally defined in Section 3.1.

Binary decision diagrams were first proposed by Akers in [1] and then canonicalized by Bryant in [6].

Definition 3.1 ***A binary decision diagram*** *(BDD) is a rooted, directed acyclic graph. Each nonterminal vertex v is labeled by a Boolean variable $var(v)$. Vertex v has two outgoing arcs, $child_0(v)$ and $child_1(v)$. Each terminal vertex u is labeled 0 or 1.*

Each vertex in a BDD represents a binary input binary output function and all used-visible vertices are roots. The terminal vertices represent the constants (functions) 0 and 1. For each nonterminal vertex v representing a function F, its child vertex $child_0(v)$ represents the function $F_{\overline{v}}$ and its other child vertex $child_1(v)$ represents the function F_v. i.e., $F = \overline{v} \cdot F_{\overline{v}} + v \cdot F_v$.

For a given assignment to the variables, the value yielded by the function is determined by tracing a decision 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.

Definition 3.2 *A BDD is* **ordered** *if there is a total order $\prec$ over the set of variables such that for every nonterminal vertex v, $var(v) \prec var(child_0(v))$ if $child_0(v)$ is nonterminal, and $var(v) \prec var(child_1(v))$ if $child_1(v)$ is nonterminal.*

Definition 3.3 *A BDD is* **reduced** *if*

1. *it contains no vertex v such that $child_0(v) = child_1(v)$, and*
2. *it does not contain two distinct vertices v and v' such that the subgraphs rooted at v and v' are isomorphic.*

Definition 3.4 *A* **reduced ordered binary decision diagram** *(ROBDD) is a BDD which is both reduced and ordered.*

Any **subset** S in a Boolean space B^n can be represented by a unique Boolean function $\chi_S : B^n \rightarrow B$, which is called its **characteristic function** [7], such that:

$$\chi_S(x) = 1 \text{ if and only if } x \text{ in } S$$

In the sequel, we will not distinguish the subset S from its characteristic function χ_S, and we will use S to denote both.

Any **relation** $\mathcal{R}$ between a pair of Boolean variables can also be represented by a characteristic function $\mathcal{R} : B^2 \rightarrow B$ as:

$$\mathcal{R}(x, y) = 1 \text{ if and only if } x \text{ is in relation } \mathcal{R} \text{ to } y$$

$\mathcal{R}$ can be a one-to-many relation over the two sets in B.

These definitions can be extended to any relation $\mathcal{R}$ between n Boolean variables, and can be represented by a characteristic function $\mathcal{R} : B^n \rightarrow B$ as:

$$\mathcal{R}(x_1, x_2, \ldots, x_n) = 1 \text{ if and only if the } n\text{-tuple } (x_1, x_2, \ldots, x_n) \text{ is in relation } \mathcal{R}$$

3.1 BDD Operators

A rich set of BDD operators has been developed and published in the literature [6, 3]. The following is the subset of operators useful in our work.

The ITE operator forms the basis for the construction and manipulation of BDDs. The use of the ITE operator also guarantees that the resulting BDD is in strong canonical form [3]. The CASE operator is the multi-valued analog of the ITE operator.

Definition 3.5 *The* **ITE operator** *returns function G_1 if function F evaluates true, else it returns function G_0:*

$$ITE(F, G_1, G_0) = \begin{cases} G_1 & \text{if } F = 1 \\ G_0 & \text{otherwise} \end{cases}$$

where range(F)={0,1}.

Definition 3.6 *The* **substitution** *in the function $\mathcal{F}$ of variable x_i with variable y_i is denoted by:*

$$[x_i \rightarrow y_i]\mathcal{F} = \mathcal{F}(x_1, \ldots, x_{i-1}, y_i, x_{i+1}, \ldots, x_n)$$

and the substitution in the function $\mathcal{F}$ of a set of variables $x = x_1 x_2 \ldots x_n$ with another set of variables $y = y_1 y_2 \ldots y_n$ is obtained simply by:

$$[x \rightarrow y]\mathcal{F} = [x_1 \rightarrow y_1][x_2 \rightarrow y_2] \ldots [x_n \rightarrow y_n]\mathcal{F}$$

In the description of subsequent computations, some obvious substitutions will be omitted for clarity in formulas.

Definition 3.7 *The* **cofactor** *of $\mathcal{F}$ with respect to the literal x_i ($\overline{x_i}$ respectively) is denoted by $\mathcal{F}_{x_i}$ ($\mathcal{F}_{\overline{x_i}}$ respectively) and is the function resulting when x_i is replaced by 1 (0 respectively):*

$$\mathcal{F}_{x_i}(x_1, \ldots, x_n) = \mathcal{F}(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$
$$\mathcal{F}_{\overline{x_i}}(x_1, \ldots, x_n) = \mathcal{F}(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

The cofactor of $\mathcal{F}$ is a simpler function than $\mathcal{F}$ itself because the cofactor no longer depends on the variable x_i.

Definition 3.8 *The* **existential quantification** *(also called* **smoothing**, *or* **forsome**) *of a function $\mathcal{F}$ over a variable x_i is denoted by $\exists x_i(\mathcal{F})$ and is defined as:*

$$\exists x_i\ (\mathcal{F}) = \mathcal{F}_{\overline{x_i}} + \mathcal{F}_{x_i}$$

and the existential quantification over a set of variables $x = x_1, x_2, \ldots, x_n$ is defined as:

$$\exists x\ (\mathcal{F}) = \exists x_1\ (\exists x_2\ (\ldots(\exists x_n\ (\mathcal{F}))))$$

Definition 3.9 *The* **universal quantification** *(also called* **consensus**, *or* **forall**) *of a function $\mathcal{F}$ over a variable x_i is denoted by $\forall x_i(\mathcal{F})$ and is defined as:*

$$\forall x_i\ (\mathcal{F}) = \mathcal{F}_{\overline{x_i}} \cdot \mathcal{F}_{x_i}$$

and the universal quantification over a set of variables $x = x_1, x_2, \ldots, x_n$ is defined as:

$$\forall x\ (\mathcal{F}) = \forall x_1\ (\forall x_2\ (\ldots(\forall x_n\ (\mathcal{F}))))$$

3.2 Unique Quantifier

Now we introduce a new BDD operator called the unique quantifier, which is in the same class as the existential and universal quantifiers.

Definition 3.10 *The* **unique quantification** *of a function $\mathcal{F}$ over a variable x_i is denoted by $!x_i\ (\mathcal{F})$ and is defined as:*

$$!x_i\ (\mathcal{F}) = \mathcal{F}_{\overline{x_i}} \oplus \mathcal{F}_{x_i}$$

and the unique quantification over a set of variables $x = x_1, x_2, \ldots, x_n$ is defined as:

$$!x\ (\mathcal{F}) = !x_1\ (!x_2\ (\ldots(!x_n\ (\mathcal{F}))))$$

Suppose $\mathcal{F}$ is a relation on x, y and z. $!x\ \mathcal{F}(x,y,z) = \{(y,z) | \text{a pair } (y,z) \text{ is related to a unique } x\}$.

Some properties of the unique quantifier will be presented:

Lemma 3.1 $!x\ !y\ F \Leftrightarrow\ !y\ !x\ F$

Proof. $!x\ !y\ F \Leftrightarrow (F_{\overline{x}\,\overline{y}} \oplus F_{\overline{x}y} \oplus F_{x\overline{y}} \oplus F_{xy}) \Leftrightarrow\ !y\ !x\ F$
using the distributive property of cofactor over XOR, $(F \oplus G)_x \Leftrightarrow F_x \oplus G_x$. □

It is well known that $\exists x\ \forall y\ F \Rightarrow \forall y\ \exists x\ F$. Let us investigate if similar properties hold for the unique quantifier.

Lemma 3.2 $!x\ \exists y\ F \Rightarrow \exists y\ !x\ F$

Proof.

$$
\begin{aligned}
!x\ \exists y\ F &\Leftrightarrow (F_{\overline{y}} + F_y)_{\overline{x}} \doteq (F_{\overline{y}} + F_y)_x \\
&\Leftrightarrow F_{\overline{x}\,\overline{y}} \cdot \overline{F_{x\overline{y}}} \cdot \overline{F_{xy}} + F_{\overline{x}y} \cdot \overline{F_{x\overline{y}}} \cdot \overline{F_{xy}} + F_{x\overline{y}} \cdot \overline{F_{\overline{x}\,\overline{y}}} \cdot \overline{F_{\overline{x}y}} + F_{xy} \cdot \overline{F_{\overline{x}\,\overline{y}}} \cdot \overline{F_{\overline{x}y}} \\
\exists y\ !x\ F &\Leftrightarrow (F_{\overline{x}} \oplus F_x)_{\overline{y}} + (F_{\overline{x}} \oplus F_x)_y \\
&\Leftrightarrow F_{\overline{x}\,\overline{y}} \cdot \overline{F_{x\overline{y}}} + F_{\overline{x}y} \cdot \overline{F_{xy}} + F_{x\overline{y}} \cdot \overline{F_{xy}} + F_{xy} \cdot \overline{F_{\overline{x}y}} \\
!x\ \exists y\ F &\Rightarrow \exists\ \cdot\ F
\end{aligned}
$$

□

The converse, $\exists y\ !x\ F = !x\ \exists y\ F$, is not true in general. Consider the relation $F(x,y) = \{(0,0),(0,1),(1,1)\}$, $\exists y\ !x\ F$ is true but $!x\ \exists y\ F$ is false. For the same F, $!x\ \forall y\ F$ is true but $\forall y\ !x\ F$ is not; therefore $!x\ \forall y\ F \Rightarrow \forall y\ !x\ F$ is not true in general. Also, $!x\ \forall y\ F \Leftarrow \forall y\ !x\ F$ is not true in general because of the counter example $F(x,y) = \{(0,0),(1,1)\}$, where $\forall y\ !x\ F$ is true but $!x\ \forall y\ F$ is false.

The pseudo-code in Figure 4 outlines the BDD *unique* algorithm. The BDD function for $!x\ F$ is returned by calling $unique(F, x, 1)$ where $x = \{x_1, x_2, \ldots, x_n\}$.

First, the algorithm checks for terminal cases, and checks if the result has already been computed before. Otherwise if the top variable v of F is the same as x_i then the following recursive formula is applied:

$$!x_i, x_{i+1}, \ldots, x_n\ (F) = !x_{i+1}, x_{i+2}, \ldots, x_n\ (F_{\overline{x_i}}) \doteq !x_{i+1}, x_{i+2}, \ldots, x_n\ (F_{x_i})$$

If v is below x_i, F is independent of variable x_i and therefore the result from the next recursion can be simply returned. If v is above x_i, we need to compute $!x\ F_{\overline{v}}$ and $!x\ F_v$ and merge the results by the ITE operator. Finally the result is stored in the computed-table, and returned.

4 Mapping MDDs into BDDs

The first-generation MDD package is a direct implementation of the theory presented in Section 2. For efficiency, our current MDD package uses BDDs as its internal representation. By hiding the mapped-BDD and its encoded variables from the users, it lets users construct functions, manipulate them and output results in terms of multi-valued variables only. But only Boolean output (multi-valued input) functions can be represented by mapped-BDDs.

The mapping of MDDs into BDDs involves two distinct steps: *variable encoding* and *variable ordering*. **Variable encoding** is the process of associating a number of binary-valued variables to each multi-valued variable, and assigning codes to represent the values that the multi-valued variables can take. **Variable**

```
unique(F, x, i) {
    if (i > |x|) or (top_index(F) > bottom_index(x)) return F
    if unique(F, x, i) in computed-table return result
    let v be the top variable of F
    if (index(v) = index(x_i)) {
        T = unique(F_{x_i}, x, i + 1)
        E = unique(F_{x̄_i}, x, i + 1)
        result = ITE(T, Ē, E)
    } else if (top_index(F) > index(x_i)) {
        result = unique(F, x, i + 1)
    } else {
        T = unique(F_v, x, i)
        E = unique(F_v̄, x, i)
        result = ITE(v, T, E)
    }
    insert result in computed-table for unique(F, x, i)
    return result
}
```

Figure 4: Pseudo-code for the *unique* quantifier.

ordering is the process of finding an ordering of the encoded binary-valued variables such that the size of the final BDD is minimized. Section 7 will be devoted to variable ordering techniques used in our work, while we shall first describe the variable encoding process here. To encode each m-valued MDD vertex, we must decide on:

1. the number of binary variables used: n encoding variables result in 2^n code points, and each code corresponds to a decision path in the full BDD subgraph of these variables.

2. the assignments of codes to values: At least one distinct code point must be assigned to each value. Therefore $2^n \geq m$ must be true.

3. the treatment of unused code points: If $2^n > m$, there is one or more unused code points. Each unused code can either be left unassigned, or be associated to a value to which another code has already been assigned. For the former case, its corresponding path always points to the terminal vertex 0. For the latter case, the corresponding path can point to the same place as another path in the BDD subgraph.

In the next two sections, we shall investigate two encoding schemes. Logarithmic encoding in Section 5 offers a compact representation of functions, e.g., characteristic functions, whereas 1-hot encoding in Section 6 is useful for set representation. Once an encoding and an ordering are chosen, there is a unique way to map each MDD vertex into a BDD subgraph.

Example 4.1 *Figure 5a shows an MDD representing the following function:*

$$F = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{otherwise} \end{cases}$$

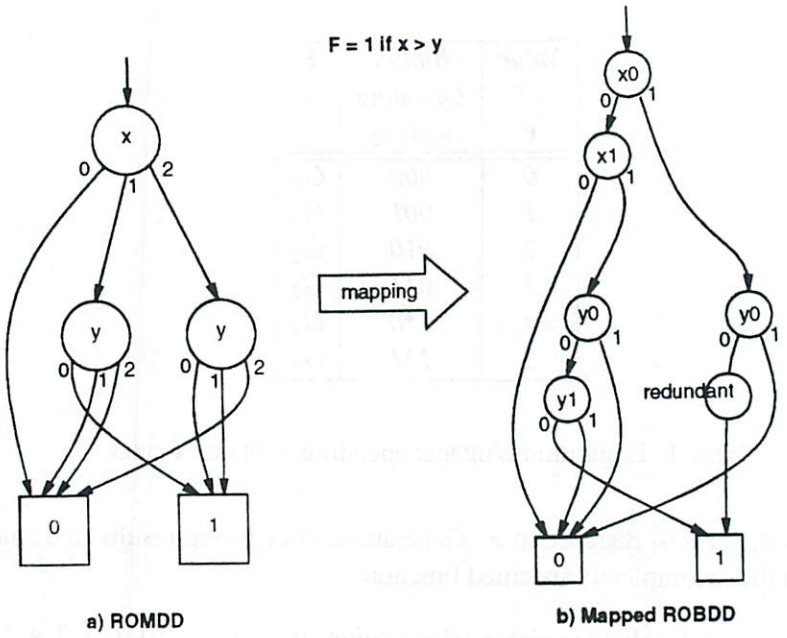

Figure 5: MDD and mapped-BDD representing the relation $x > y$.

x and y are 3-valued variables which can take values from $P_x = P_y = \{0, 1, 2\}$. To represent the MDD using BDDs, each MDD nonterminal vertex must be mapped into a number of BDD vertices interconnected in a subgraph. For example in Figure 5, the MDD vertex labeled by variable x is mapped into the BDD vertices labeled by x_0 and x_1. In addition, different indices have to be assigned to these binary variables. In this case, since $x \prec y$ for the MDD, this ordering is respected for the associated binary variables: $x_0 \prec x_1 \prec y_0 \prec y_1$. The mapping process dictates the encoding used. The same encoding, as well as ordering, must be used consistently throughout all function manipulations.

For space's sake in this paper we do not deal with Zero-suppressed BDDs [29] that are a variant of BDDs suited to represent sparse combinatorial sets. In matter of principle it is possible to map MDDs to ZBDDs instead than to BDDs, and experimentally it turns out to be a better choice for some applications [30].

5 Logarithmically Encoded MDDs

In this section, logarithmic encoding (i.e., integer encoding) is used to map MDDs into BDDs. Our prime concern here is to use the least number of variables and BDD vertices. An m-valued MDD vertex is represented with $\lceil \lg_2 m \rceil$ Boolean variables, and is mapped to a BDD subgraph of $m - 1$ vertices. These numbers are provably minimum in graph theory.

Each value is assigned an integer code. As discrete variables in CAD problems usually take values from the ordered set of integers and the operations between them are sometimes integer-arithmetic in nature, integer encoding results in an efficient representation and manipulation.

Of course, not all 2^n code points will be used since typically $m < 2^n$. On the other hand the mapped-BDD will have a path for each binary code point. The decision path for each unused code point is chosen so as to minimize the BDD size. In fact, an unused point is assigned to the same path of the used point whose encoding is closest to the unused code point. This mapping is related to the *generalized cofactor* operator in [37] which was initially proposed in [10] as the *constraint* operator. Given a function f and a care set c, the generalized cofactor of f with respect to c is the projection of f that maps a don't care point x to the care

Value of v	*Binary Encoding* $u_0u_1u_2$	F =
0	*000*	G_0
1	*001*	G_1
2	*010*	G_2
3	*011*	G_3
4	*1*0*	G_4
5	*1*1*	G_5

Table 1: Logarithmic/integer encoding with don't cares.

point $y \in c$ which has the closest distance to x. Generalized cofactoring results in a small and canonical BDD representation of the incompletely specified function.

Example 5.1 *Suppose v is a 6-valued variable taking values from $P_v = \{0, 1, 2, 3, 4, 5\}$. Three binary-valued variables u_0, u_1 and u_2 can be assigned to encode variable v as shown in Table 1. The last column is used in the example in Section 5.1.*

Note that if the value range is not a power of 2, some codes will not be used, e.g., 110 and 111. These encodings are used as don't cares since the values will never occur. In this case these don't cares are mapped into the same nodes as 100 and 101 respectively. The notation $1 * 0$ is used to represent both encodings 100 and 110 as we "don't care" about the variable u_1.

5.1 Relationships between CASE and ITE Operators

As the CASE and ITE operators form the basis for manipulation of MDDs and BDDs respectively, mapping can be conveniently performed by replacing each CASE operation by a set of ITE operations. The recursion step in Equation 4 is our starting point. It gives an outer CASE operator in terms of a top-variable v, and enables conversion to a hierarchy of ITE operators. The conversion can be summarized by the following recursive formulas:

$$\begin{aligned}
\text{if } p \text{ is even: } & CASE(v, \underbrace{G'_0, G'_1}, \underbrace{G'_2, G'_3}, \ldots, \underbrace{G'_{p-2}, G'_{p-1}}) \\
&= CASE(v', ITE(u, G'_1, G'_0), ITE(u, G'_3, G'_2), \\
&\quad \ldots . ITE(u, G'_{p-1}, G'_{p-2})) \\
\text{if } p \text{ is odd: } & CASE(v, \underbrace{G'_0, G'_1}, \underbrace{G'_2, G'_3}, \ldots, \underbrace{G'_{p-3}, G'_{p-2}}, G'_{p-1}) \\
&= CASE(v', ITE(u, G'_1, G'_0), ITE(u, G'_3, G'_2), \\
&\quad \ldots, ITE(u, G'_{p-2}, G'_{p-3}), G'_{p-1})
\end{aligned}$$

This recursion terminates when there are only two child-functions remaining in the outer CASE operator:

$$CASE(v, G'_0, G'_1) = ITE(v, G'_1, G'_0).$$

While pairing up child-functions with the ITE operator, these formulas replace the big MDD vertex labeled with variable v with a smaller one, labeled with a new multi-valued variable v', and a number of BDD vertices labeled with a new binary variable u. This mapping process is best explained by an example.

Example 5.2 *Suppose v is a 6-valued MDD vertex, and $G'_0, \ldots, G'_5$ are the six child-functions connected to it, the CASE to ITE mapping proceeds as follows:*

$$\begin{aligned} & CASE(v, \underbrace{G'_0, G'_1}, \underbrace{G'_2, G'_3}, \underbrace{G'_4, G'_5}) \\ &= CASE(v', \underbrace{ITE(u_2, G'_1, G'_0), ITE(u_2, G'_3, G'_2)}, ITE(u_2, G'_5, G'_4)) \\ &= CASE(v'', \underbrace{ITE(u_1, ITE(u_2, G'_3, G'_2), ITE(u_2, G'_1, G'_0))}, ITE(u_2, G'_5, G'_4)) \\ &= ITE(u_0, ITE(u_2, G'_5, G'_4), ITE(u_1, ITE(u_2, G'_3, G'_2), ITE(u_2, G'_1, G'_0))) \end{aligned}$$

Note that while pairing up child-functions for ITE operations in the first step, we effectively replace the original 6-valued MDD vertex with a smaller 3-valued MDD vertex. During the assignment of BDD variables, the ordering $u_0 \prec u_1 \prec u_2$ is used. Figure 6 shows the bottom-up recursive mapping process. Note that the original MDD node labeled v has been mapped into a BDD subgraph with 5 internal nodes.

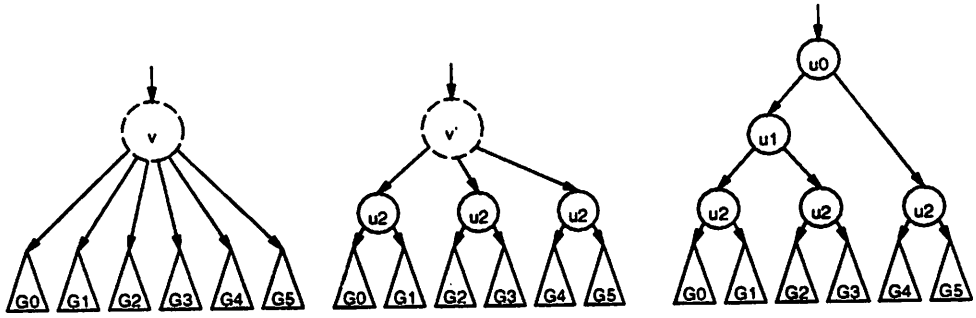

Figure 6: Recursive mapping from an MDD vertex to a mapped-BDD subgraph.

6 1-hot Encoded MDDs

In this section we describe how MDDs can be used to represent and manipulate implicitly sets of objects. This theory is especially useful for applications where sets of sets of objects need to be constructed and manipulated, as it is often the case in logic synthesis and combinatorial optimization. In many applications such as FSM minimization, encoding and partitioning, the number of objects (number of states in these cases) to be handled is usually not large. But their exact optimization algorithms require exploration of many different subsets of such objects. For instance exact state minimization requires selection of a minimum closed cover out of a huge number of candidate sets of state sets [20]. Therefore our prime concern here is to have a compact representation for set of sets.

Suppose the elements corresponds to n distinct objects. With 1-hot encoding, a Boolean variable is associated with each object so n Boolean variables are used. Each singleton element is assigned a distinct 1-hot code. Obviously with this 1-hot encoding scheme, there are a lot of unused code points. Unlike the logarithmic encoding where unused code points are reassigned to values, these code points are used for a purpose other than representing elements or values, but to represent sets other than singletons.

6.1 Positional-set Notation

Given that there are 2^n possible distinct sets of objects, in order to represent collections of them it is not possible to encode the objects using $log_2\, n$ Boolean variables. Instead, each subset of objects is represented in **positional-set** or positional-cube notation form, using a set of n Boolean variables, $x = x_1 x_2 \ldots x_n$.

The presence of an element s_k in the set is denoted by the fact that variable x_k takes the value 1 in the positional-set, whereas x_k takes the value 0 if element s_k is not a member of the set. One Boolean variable is needed for each element because the element can either be present or absent in the set [1].

In the above example, $n = 6$, and the set with a single element s_4 is represented by 000100 while the set $s_2 s_3 s_5$ is represented by 011010. The elements s_1, s_4, s_6 which are not present correspond to 0s in the positional-set.

A set of sets of objects is represented as a set S of positional-sets, by a characteristic function $\chi_S : B^n \rightarrow B$ defined as:

$\chi_S(x) = 1$ if and only if the set represented by the positional-set x is in the set S of sets.

A 1-hot encoded MDD representing $\chi_S(x)$ will contain minterms, each corresponding to a set in S. Operators for manipulating positional-sets and characteristic functions will be introduced in the next two subsections.

A 1-hot encoded MDD can be represented as a mapped-BDD where each Boolean variable corresponds to a BDD variable. From now on, we use BDD to refer to an 1-hot encoded MDD where there is no ambiguity.

6.2 Operations on Positional-sets

With our previous definitions of relations and positional-set notation for representing set of objects, useful relational operators on sets can be derived. We propose a unified notational framework for set manipulation which extends the notation used in [23]. In this section, each operators Op acts on two sets of variables $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_n$ and returns a relation $(x\ Op\ y)$ (as a characteristic function) of pairs of positional-sets. Alternatively, they can also be viewed as constraints imposed on the possible pairs out of two sets of objects, x and y. For example, given two sets of sets X and Y, the set pairs (x, y) where x contains y are given by the product of X and Y and the containment constraint, $X(x) \cdot Y(y) \cdot (x \supseteq y)$.

Lemma 6.1 *The* **equality relation** *evaluates true if the two sets of objects represented by positional-sets x and y are identical, and can be computed as:*

$$(x = y) = \prod_{k=1}^{n} x_k \Leftrightarrow y_k$$

where $x_k \Leftrightarrow y_k = x_k \cdot y_k + \neg x_k \cdot \neg y_k$ designates the Boolean XNOR *operation and $\neg$ designates the Boolean* NOT *operation.*

Proof. $\prod_{k=1}^{n} x_k \Leftrightarrow y_k$ requires that for every element k, either both positional-sets x and y contain it, or it is absent from both. Therefore, x and y contains exactly the same set of elements and thus are equal. □

Lemma 6.2 *The* **containment relation** *evaluates true if the set of objects represented by x contains the set of objects represented by y, and can be computed as:*

$$(x \supseteq y) = \prod_{k=1}^{n} y_k \Rightarrow x_k$$

where $x_k \Rightarrow y_k = \neg x_k + y_k$ designates the Boolean implication operation.

[1] The representation of primes proposed by Coudert *et al.* [11] needs 3 values per variable to distinguish if the present literal is in positive or negative phase or in both phases.

Proof. $\prod_{k=1}^{n} y_k \Rightarrow x_k$ requires that for all objects, if an object k is present in y (i.e., $y_k = 1$), it must also be present in x ($x_k = 1$). Therefore set x contains all the objects in y. □

Lemma 6.3 *The* **strict containment relation** *evaluates true if the set of objects represented by x strictly contains the set of objects represented by y, and can be computed as:*

$$(x \supset y) = (x \supseteq y) \cdot \neg(x = y) \tag{5}$$

Alternatively, $(x \supset y)$ can be computed by:

$$(x \supset y) = \prod_{k=1}^{n} [y_k \Rightarrow x_k] \cdot \sum_{k=1}^{n} [x_k \cdot \neg y_k] \tag{6}$$

Proof. Equation 5 follows directly from the two previous theorems. For Equation 6, the first term is simply the containment constraint, while the second term $\sum_{k=1}^{n} [x_k \cdot \neg y_k]$ requires that for at least one object k, it is present in x ($x_k = 1$) but it is absent from y ($y_k = 0$), i.e., x and y are not the same. So it is an alternative way of computing $(x \supset y)$. □

Lemma 6.4 *The* **equal-union relation** *evaluates true if the set of objects represented by x is the union of the two sets of objects represented by y and z, and can be computed as:*

$$(x = y \cup z) = \prod_{k=1}^{n} x_k \Leftrightarrow (y_k + z_k)$$

Proof. For each position k, x_k is set to the value of the OR between y_k and z_k. Effectively, $\prod_{k=1}^{n} x_k \Leftrightarrow (y_k + z_k)$ performs a bitwise OR on y and z to form a single positional-set x, which represents the union of the two individual sets. □

Lemma 6.5 *The* **equal-intersection relation** *evaluates true if the set of objects represented by x is the intersection of the two sets of objects represented by y and z, and can be computed as:*

$$(x = y \cap z) = \prod_{k=1}^{n} x_k \Leftrightarrow (y_k \cdot z_k)$$

Proof. For each position k, x_k is set to the value of the AND between y_k and z_k. Effectively, $\prod_{k=1}^{n} x_k \Leftrightarrow (y_k \cdot z_k)$ performs a bitwise AND on y and z to form a single positional-set x, which represents the intersection of the two individual sets. □

Lemma 6.6 *The* **contain-union relation** *evaluates true if the set of objects represented by x contains the union of the two sets of objects represented by y and z, and can be computed as:*

$$(x \supseteq y \cup z) = \prod_{k=1}^{n} (y_k + z_k) \Rightarrow x_k$$

Proof. Note the similarity in the computations of $(x \supseteq y \cup z)$ and $(x = y \cup z)$. $(x \supseteq y \cup z)$ performs bitwise OR on singletons y and z. If either of their k-th bits is 1, the corresponding k-th bit of x, i.e., x_k, is constrained to 1. Otherwise, x_k can take any values (i.e., don't care). The outer product $\prod_{k=1}^{n}$ requires that the above is true for each k. □

Lemma 6.7 *The* **contain-intersection relation** *evaluates true if the set of objects represented by x contains the intersection of the two sets of objects represented by y and z, and can be computed as:*

$$(x \supseteq y \cap z) = \prod_{k=1}^{n} (y_k \cdot z_k) \Rightarrow x_k$$

Proof. Note the similarity in the computations of $(x \supseteq y \cap z)$ and $(x = y \cap z)$. $(x \supseteq y \cap z)$ performs bitwise AND on singletons y and z. If either of their k-th bits is 1, the corresponding k-th bit of x, i.e., x_k, is constrained to 1. Otherwise, x_k can take any values (i.e., don't care). The outer product $\prod_{k=1}^{n}$ requires that the above is true for each k. □

6.3 Operations on Sets of Positional-sets

The first three lemmas in this section introduce operators that return a set of positional-sets as the result of some implicit set operations on one or two sets of positional-sets.

Lemma 6.8 *Given the characteristic functions χ_A and χ_B representing the sets A and B, set operations on them such as* **union**, **intersection**, **sharp**, *and* **complementation** *can be performed as logical operations on their characteristic functions, as follows:*

$$\begin{aligned} \chi_{A \cup B} &= \chi_A + \chi_B \\ \chi_{A \cap B} &= \chi_A \cdot \chi_B \\ \chi_{A-B} &= \chi_A \cdot \neg \chi_B \\ \chi_{\overline{A}} &= \neg \chi_A \end{aligned}$$

Lemma 6.9 *The* **maximal** *of a set χ of subsets is the set containing subsets in χ not strictly contained by any other subset in χ, and can be computed as:*

$$Maximal_x(\chi) = \chi(x) \cdot \nexists y\, [(y \supset x) \cdot \chi(y)]$$

Proof. The term $\exists y\, [(y \supset x) \cdot \chi(y)]$ is true if and only if there is a positional-set y in χ such that $x \subset y$. In such a case, x cannot be in the maximal set by definition, and can be subtracted out. What remains is exactly the maximal set of subsets in $\chi(x)$. □

Lemma 6.10 *Given a set of positional-sets $\chi(x)$ and an array of the Boolean variables x, the maximal of positional-sets in χ with respect to x can be computed by the recursive BDD operator $Maximal(\chi, 0, x)$:*

```
Maximal(χ, k, x) {
    if (χ = 0) return 0
    if (χ = 1) return ∏_{i=k}^{n} x_i
    M0 = Maximal(χ_{x̄_k}, k + 1)
    M1 = Maximal(χ_{x_k}, k + 1)
    return ITE(x_k, M1, M0 · ¬M1)
}
```

Proof. The operator starts at the top of the BDD and recurses down until a terminal node is reached. At each recursive call, the operator returns the maximal set of positional-sets within χ made up of elements from k to n. If terminal **0** is reached, there is no positional-set within χ so **0** (i.e., nothing) is returned. If terminal **1** is reached, χ contains all possible position-sets with elements from k to n, and the maximum one is $\prod_{i=k}^{n} x_i$. At any intermediate BDD node, we find the maximal positional-sets M_0 on the *else* branch of χ, the maximal positional-sets M_1 on the *then* branch of χ. The resultant maximal set of sets contains (1) positional-sets in M_1 each with element x_k added to it as they cannot be contained by any set in M_1 which has $x_k = 0$, and (2) positional-sets that are in M_0 but not in M_1, because if a set is present in both it is already accounted for in (1). Thus the ITE operation returns the required maximal set after each call. □

To guarantee that each node of the BDD χ is processed exactly once, intermediate results should be cached in a computed-table.

Lemma 6.11 *The* **minimal** *of a set χ of subsets is the set containing subsets in χ not strictly containing any other subset in χ, and can be computed as:*

$$Minimal_x(\chi) = \chi(x) \cdot \nexists y\,[(x \supset y) \cdot \chi(y)]$$

Proof. The term $\exists y\,[(x \supset y) \cdot \chi(y)]$ is true if and only if there is a positional-set y in χ such that $x \supset y$. In such a case, x cannot be in the minimal set by definition, and can be subtracted out. What remains is exactly the minimal set of subsets in $\chi(x)$. □

A recursive BDD operator $Minimal(\chi, k, x)$ can be similarly defined.

The next three operators check set equality, containment and strict containment between two sets of sets, whereas Lemmas 6.1, 6.2 and 6.3 check on a pair of sets only. These following operators return tautology if the tests are passed.

Lemma 6.12 *Given the characteristic functions $\chi_A(x)$ and $\chi_B(x)$ representing two sets A and B (of positional-sets), the* **set equality test** *is true if and only if sets A and B are identical, and can be computed by:*

$$Equal_x(\chi_A, \chi_B) = \forall x\,[\chi_A(x) \Leftrightarrow \chi_B(x)]$$

Alternatively, Equal can be found by checking if their corresponding ROBDDs are the same by bdd_equal(χ_A, χ_B).

Proof. $\chi_A(x)$ and $\chi_B(x)$ represents the same set if and only if for every x, either $x \in A$ and $x \in B$, or $x \notin A$ and $x \notin B$. As the characteristic function representing a set in positional-set notation is unique, two characteristic functions will represent the same set if and only if their ROBDDs are the same. □

Lemma 6.13 *Given the characteristic functions $\chi_A(x)$ and $\chi_B(x)$ representing two sets A and B (of positional-sets), the* **set containment test** *is true if and only if set A contains set B, and can be computed by:*

$$Contain_x(\chi_A, \chi_B) = \forall x\,[\chi_B(x) \Rightarrow \chi_A(x)]$$

Lemma 6.14 *Given the characteristic functions χ_A and χ_B representing two sets A and B (of positional-sets), the* **set strict containment test** *is true if and only if set A strictly contains set B, and can be computed by:*

$$Strict\_Contain_x(\chi_A, \chi_B) = Contain_x(\chi_A, \chi_B) \cdot \neg Equal_x(\chi_A, \chi_B)$$

Proof. The proof follows directly from previous two theorems. □

Beside operating on sets of sets, the above operators can also be used on relations of sets. The effect is best illustrated by an example. Suppose A and B are binary relations on sets. $Contain_x(\chi_A(x, y), \chi_B(x, z))$ will return another relation on pairs (y, z) of sets. Position sets y and z are in the resultant relation if and only if the set of positional-sets x associated with y in relation A *contains* the set of positional-sets x associated with z in B.

The remaining operators in this section take a set of sets and a set of variables as parameters, and return a singleton positional-set on those variables.

Lemma 6.15 *Given a characteristic function $\chi_A(x)$ representing a set A of positional-sets, the* **set union relation** *tests if positional-set y represents the union of all sets in A, and can be computed by:*

$$Union_{x \to y}(\chi_A) = \prod_{k=1}^{n} y_k \Leftrightarrow \exists x\,[\chi_A(x) \cdot x_k]$$

Proof. For each position k, the right hand expression sets y_k to 1 if and only if there exists an x in χ_A such that its k-th bit is a 1 ($\exists x\,[\chi_A(x) \cdot x_k]$). This implies that the positional-set y will contain the k-th element if and only if there exists a positional-set x in A such that k is a member of x. Effectively, the right hand expression performs a multiple bitwise OR on all positional-sets of χ_A to form a single positional-set y which represents the union of all such positional-sets. □

Alternatively, we implemented the set $Union$ operation as a recursive BDD operator. Bitwise OR is performed at the BDD DAG level, by traversing the BDD and performing OR on BDD vertices with the variables of interest.

Lemma 6.16 *Given a set of positional-sets $\chi(x)$ and an array of the Boolean variables x, the union of positional-sets in χ with respect to x can be computed by the BDD operator $Bitwise\_Or(\chi, 0, x)$, assuming that the variables in x are ordered last:*

```
Bitwise_Or(χ, k, x) {
    if (k ≥ |x|) return χ
    t = top_var(χ)
    if (t < x_k) {
        T = Bitwise_Or(χ_t, k, x)
        E = Bitwise_Or(χ_t̄, k, x)
        return ITE(t, T, E)
    } else {
        if (χ_{x_k} = 0) return x̄_k · Bitwise_Or(χ_{x̄_k}, k + 1, x)
        else return x_k · Bitwise_Or(χ_{x_k} + χ_{x̄_k}, k + 1, x)
    }
}
```

Proof. x_k denotes the k-th variable in the array x. Assuming that the variables in x are ordered last, the above recursion terminates after all of them have been processed ($k \geq |x|$, and a **0** or a **1** is returned as χ). At a BDD vertex where $t < x_k$, the recursion has not reached a variable of interest yet, and we simply recurse down its right and left children and merge the *Bitwise_Or* results by creating a new vertex $ITE(t, T, E)$. If $t \geq x_k$, we have to perform the bitwise OR operation on variable v. If $\chi_{x_k} = \mathbf{0}$, variable x_k never takes a value 1 in any satisfying assignments of χ, so it is set to 0 by $\overline{x_k}$. The bitwise OR of the remaining variables is given by $Bitwise\_Or(\chi_{\overline{x_k}}, k+1, x)$. Otherwise if $\chi_{x_k} \neq \mathbf{0}$, there exists a satisfying assignment of χ in which $x_k = 1$. So x_k is set to 1, while a bitwise OR is performed over all remaining satisfying assignments of χ, i.e., $\chi_{x_k} + \chi_{\overline{x_k}}$. □

This recursive BDD operator is very fast, but unfortunately, its operation is valid only if the variables to be bitwise ORed are at the bottom of the BDD DAG. So to execute this BDD operator, we need to perform variable substitutions before and after the operation. Experimentally, these substitution steps are too slow to be practical and sometimes cause an exponential explosion in the BDD size. As a result, we use the computation in Lemma 6.15 instead.

Lemma 6.17 *Given a characteristic function $\chi_A(x)$ representing a set A of positional-sets, the* **set intersection relation** *tests if positional-set y represents the intersection of all sets in A, and can be computed by:*

$$Intersect_{x \to y}(\chi_A) = \prod_{k=1}^{n} y_k \Leftrightarrow \forall x \, [\chi_A(x) \cdot x_k]$$

Proof. For each position k, the right hand expression sets y_k to 1 if and only if the k-th bit of all x in χ_A is a 1. This implies that the positional-set y will contain the k-th element if and only if all positional-sets x in χ_A have k as a member. Effectively, the right hand expression performs a multiple bitwise AND on all positional-sets of χ_A to form a single positional-set y which represents the intersection of all such positional-sets. □

6.4 k-out-of-n Positional-sets

Let the number of objects be n. In subsequent computations, we will use extensively a suite of sets of sets of objects, $Tuple_{n,k}(x)$, which contains all positional-sets x with exactly k elements in them (i.e., $|x| = k$).

In particular, the set of singleton elements $Tuple_{n,1}(x)$, the set of pairs $Tuple_{n,2}(x)$, the universal set of all objects $Tuple_{n,n}(x)$, and the set of empty set $Tuple_{n,0}(x)$ [2] are common ones. When n is clear from the context we will write $Tuple_k(x)$ instead of $Tuple_{n,k}(x)$. An efficient way of constructing and storing such collections of k-tuple sets using BDDs will be given next. Figure 7 represents a reduced ordered BDD of $Tuple_{5,2}(x)$:

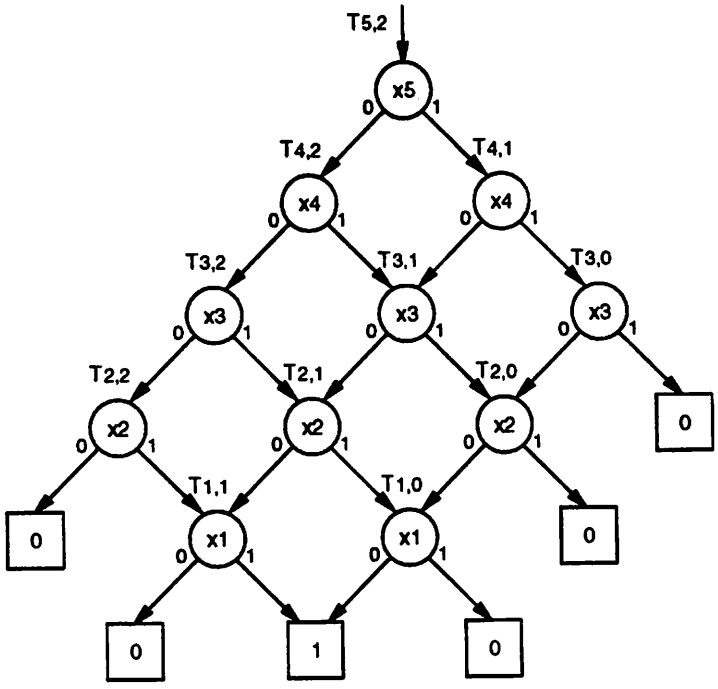

Figure 7: BDD representing $Tuple_{5,2}(x)$.

The root of the BDD represents the set $Tuple_{5,2}(x)$, while the internal nodes represent the sets $Tuple_{i,j}(x)$ $(i < 5, j < 2)$. For ease of illustration, the variable ordering is chosen such that the top variable corresponding to $Tuple_{i,j}(x)$ is x_i. At that node, if we choose element i to be in the positional-set, x_i takes the value 1 and we follow the right outgoing arc. In doing so, we still have $i-1$ elements/variables left to be processed. As we have put element i in the positional-set, we still have to add exactly $j-1$ elements into the positional-set. That is why the right child of $Tuple_{i,j}(x)$ should be $Tuple_{i-1,j-1}(x)$. Similarly, the left child is $Tuple_{i-1,j}(x)$ because element i has not been put in the positional-set and we have $j-1$ elements/variables left. Thus, the BDD for $Tuple_{i,j}$ can be constructed by the algorithm shown in Figure 8.

The total number of nonterminal vertices in the BDD of $Tuple_{n,k}$ is $(n-k+1)\cdot(k+1)-1 = nk-k^2+n = O(nk)$. With the use of the computed table [3], the time complexity of the above algorithm is also $O(nk)$ as the BDD is built from bottom up and each vertex is built once and then re-used. Given any n, the BDD for $Tuple_{n,k}$ is largest when $k = n/2$.

7 Variable Ordering

We frequently suffer from exponential time and/or space complexities if we neglect the issue of variable ordering. As with most variants of BDDs, the space and time complexities for constructing an MDD for

[2] $Tuple_{n,0}(x)$ will be denoted by $\emptyset(x)$.

```
Tuple(i, j) {
    if (j < 0) or (i < j) return 0
    if (i = j) and (i = 0) return 1
    if Tuple(i, j) in computed-table return result
    T = Tuple(i − 1, j − 1)
    E = Tuple(i − 1, j)
    F = ITE(x_i, T, E)
    insert F in computed-table for Tuple(i, j)
    return F
}
```

Figure 8: Pseudo-code for the $Tuple$ operator.

any discrete function is in the worst case exponential in the number of variables of the function. Luckily in real life, many discrete functions of interest have reasonable representations provided that a good variable ordering is chosen. Friedman *et al.* in [14] found an $O(n^2 3^n)$ algorithm for finding the optimal variable ordering where n is the number of Boolean variables. Faster variable ordering heuristics for BDDs have been provided by Malik *et al.* in [25] and Fujita *et al.* in [15]. Rudell [32] proposed an effective dynamic variable reordering heuristic which offers a tradeoff of runtime for compactness of BDD representation.

The goal in this section is to find a good variable ordering so as to minimize the total number of vertices used. With a mapped-BDD representation of an MDD, the ordering process consists of two steps: order the BDD variables within each MDD variable, and then merge these orderings into a single BDD variable ordering.

Two well-known rule-of-thumbs suggested in [25] and [15] can be used for the ordering of BDD variables within each individual MDD variable:

1. Variables that are closely related should be ordered close to each other.
2. Variables that "control" more the function should be ordered at the top.

There are two ways of merging these individual orderings. **Cluster ordering** places BDD variables, which correspond to the same multi-valued variable, in consecutive positions in the final ordering. Within each cluster, the binary variable which corresponds to the most significant bit (MSB) is ordered first (i.e., highest). Then the next significant encoding variable is ordered next, and so on. For state minimization, cluster ordering is used to merge different sets of input and output variables; they are ordered before (i.e., on top of) the state variables because the transition relation depends heavily on inputs and outputs.

Example 7.1 *In Figure 9, the relation $(x = y)$ is represented as an MDD on the left and a mapped-BDD (i.e., logarithmic encoded MDD) by cluster ordering in the middle. Variables x and y each can take four values. Note that the multi-valued variable x is encoded into two binary variables x_0 (MSB) and x_1 (LSB) on the right. The circled subgraph before reduction has the same number of outgoing arcs as the MDD vertex and the two representations are equivalent. With the mapped-BDD representation, vertices with equivalent subgraphs can be merged as shown by the two lowest nonterminal vertices.*

The problem of using cluster ordering for variables with large value ranges is illustrated by Figure 9. Consider the FSM named *squares* in the MCNC benchmark which has 371 states. Using 1-hot encoding on

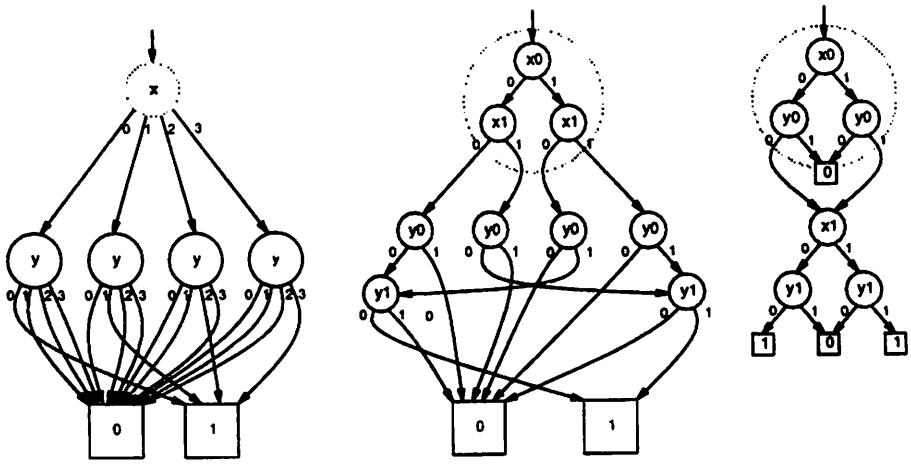

Figure 9: Comparison between cluster ordering and interleaved ordering.

its states, multi-valued variables x and y require each 371 BDD variables. As mentioned before, there are 2^{371}, that is about 5×10^{111} outgoing edges from the circled BDD subgraph cluster. In the worst case, it would have 5×10^{111} subgraphs below, each rooted at such an edge. Thus the size of the MDD will grow exponentially in the number of binary encoded variables.

To avoid such exponential growth, we use an **interleaved ordering** for the BDD variables instead, as shown on the right of Figure 9. The encoded variables x_0 and x_1 for x are interleaved with y_0 and y_1 for y. The more significant bits are compared before the less significant ones as the mapped-BDD is traversed from top to bottom. With interleaving, the width of the mapped-BDD can be kept slim. As a comparison for our example, the mapped-BDD using cluster ordering has 9 nonterminal vertices while interleaved ordering results in 6 nonterminal vertices. For the FSM *squares*, the mapped-BDD for $(x = y)$ has only $3 \times 371 = 1113$ nonterminal vertices using interleaved ordering. The example in Figure 9 is instructive in comparing the size of pure MDD representations vs. the size of BDD representations, because the pure MDD representation has $2^k + 3$ nodes, while a mapped BDD representation (with interleaved ordering) has $3.k + 1$ nodes; this supports the case for mapping MDDs to BDDs.

For state minimization, our implicit algorithms need to operate on multiple sets of state variables, and each such variable set can represent a positional-set. Interleave ordering must be used for these sets of variables for the reason described above. To avoid exponential complexities, a common wisdom is to use as few BDD variables as possible. For instance in the application reported in [20], we allocate only 4 sets of state BDD variables although a total of 10 state vector names are used in the equations. This is possible because we never have to operate on more than 4 sets of state variables simultaneously within a single BDD operation. The actual BDD variables are reused for different purposes, by binding at different times more than one set of variables from the equations to a single set of BDD variables.

8 Applications of Logarithmic Encoded MDDs

8.1 General Paradigm

Many CAD problems can be naturally formulated in a multi-valued setting. Often, we inherit a graph structure from the problem. For example, the constraint graphs for routing, the flow graphs for scheduling and the state transition graphs for FSMs. With such information, the problem can be mapped into a number of multi-valued variables and a set of constraints between these variables. From the inherited graph structure,

a good ordering of the multi-valued variables can be derived.

The input constraint file is first scanned and an MDD is built for each constraint. They are ANDed together as soon as they are created. The final MDD contains implicitly all solutions of the problem. Satisfiability can be checked trivially, as the final MDD will consist of a single terminal vertex '0' if, and only if, it is not satisfiable.

If the problem is satisfiable, we can enumerate some, or all solutions and print them out. We use MDDs to solve the decision problem, instead of its corresponding optimization problem. The latter can be solved by binary search of an optimal solution by solving multiple decision problems.

8.2 Hardware Resource Scheduling

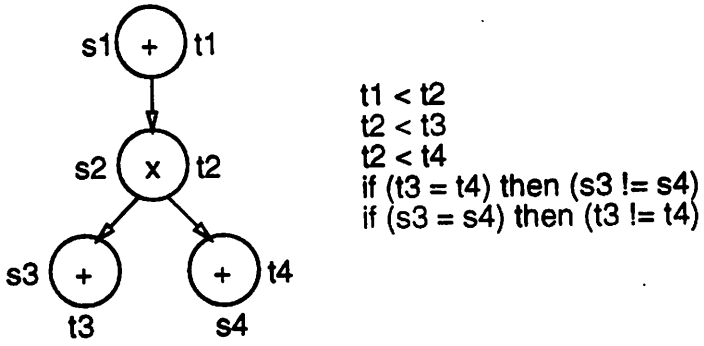

Figure 10: Hardware resource scheduling example.

The resource scheduling problem arises frequently in synthesis of VLSI layouts from high level descriptions of digital systems. We chose a general formulation as follows: given a flow graph specifying temporal and spatial relationships between operations $o_1, \ldots, o_n \in \Theta$ that can be performed at discrete time intervals on machine types or functional units $\sigma_1, \ldots, \sigma_k$ and a table specifying the *single* machine type on which each operation can be performed, determine an optimal *schedule* for the operations, based on some user-specified optimality criteria. Some criteria may be: (1) Minimum total time to perform all operations, given an allocation of θ_j machines of each type σ_j, (2) Minimum number of machines of type σ_i, (3) Minimum total cost of machines, given that all operations are completed in time τ.

With each operation o_i, we associate two integer variables, t_i and s_i, where t_i denotes the time slot in which o_i is performed and s_i denotes the "space" variable or the machine on which o_i is performed. If we would like to construct the MDD for all solutions with τ time slots, and θ_j machines of type σ_j, t_i can take on τ values and s_i can assume θ_j values, where σ_j is the machine type on which o_i can be performed.

Given the flow graph, for each pair of operations o_i and o_j, if there is an directed edge from o_i to o_j, we write: $t_i < t_j$. For each pair o_i and o_j that can be performed on the same machine type, if there is no path between i and j in the flow graph, we write:

$$if\ (t_i = t_j)\ then\ (s_i \neq s_j)$$
$$if\ (s_i = s_j)\ then\ (t_i \neq t_j)$$

Note that the previous conditions are logical equivalent (because $a \rightarrow b \leftrightarrow \overline{b} \rightarrow \overline{a}$ is a tautology). So they are redundant and only one is used in order to speed up the construction of the final MDD.

The final MDD that is the conjunction of these constraints will test for the existence of a solution with τ time slots and θ_j machines of type σ_j. Cofactoring may be used to test for alternate solutions.

8.3 Channel Routing

We make the assumption that each routing layer runs in one direction. Given N nets to be routed in a channel, the objective is to minimize the number of tracks used to route them. The horizontal interval of net i is defined as: $I(i) = r(i) - l(i)$, where $r(i)$ is the column number in which the rightmost pin of net i lies and $l(i)$ is the leftmost column occupied by net i. Two nets with intersecting intervals cannot be placed on the same track. The Vertical Constraint Graph (VCG) [42] restricts the relative positions of nets in the channel. If there is a path from net i to net j in the VCG, then the track of net i must lie above the track of net j in the channel.

We first construct the VCG for the channel. All directed edges in the VCG that can be implied by other edges are removed, i.e., the VCG is made irredundant. Let y_i denote the track occupied by net i. Then, for each net pair i, j if $I(i) \cap I(j) \neq \phi$ and there is no path from i to j in the VCG, we write the following condition:

$$y_i \neq y_j.$$

For each directed edge from i to j in the irredundant VCG, we write:

$$y_i > y_j.$$

To determine if a route exists for the channel that uses t tracks, we let each variable y_i take on t values in the ordered set $\{0, \ldots, t-1\}$ and construct the MDD that is the conjunction of the above conditions. If the resulting MDD is not the terminal vertex with value 0, a solution using t tracks exists. We can then test for solutions using fewer tracks by cofactoring each of the variables y_i with respect to the literal y_i^S where $S = \{0, 1, \ldots, s\}$ and $s < t - 1$. If however, the MDD is a terminal vertex with zero value, we must increase t and reconstruct the graph. For extensions to doglegging and multiple layers, the reader is referred to [12].

8.4 Switchbox Routing

We consider a restricted form of switchbox routing to illustrate the use of *if...then...* conditions. Extensions to more general cases are possible. The restriction we place is that nets must connect from the top of the switchbox to the bottom or from left to right, and that all nets have been decomposed into two-terminal nets. Also, we only consider one-bend patterns that connect such nets. We form two graphs - the Vertical Constraint Graph (VCG) and the Horizontal Constraint Graph (HCG). Each graph gives rise to constraints similar to the channel routing problem. Let y_i denote the y position of a vertical net, i.e., a net that connects from top to bottom. Let x_j denote the x position of a horizontal net. For each net pair i and j in the VCG, if there is no path between them in the VCG, we write:

$$y_i \neq y_j.$$

Similarly, we write constraints for nets in the HCG. The interaction between nets in the VCG and nets in the HCG generates *cross-constraints*. Two such sets of cross-constraints are illustrated in Figure 11.

In Figure 11, c_1 and c_3 are the columns in which pins of horizontal net 1 are located and c_2 and c_4 are for horizontal net 2. r_1 and r_2 are the rows in which pins of vertical net 2 lie. Suppose that the switchbox uses R rows and C columns. We let the variables y_i take on R values and the variables x_j take on C values. We then build the canonical MDD for the binary-valued function that is the conjunction of the above constraints to test for the existence of a solution that uses one-bend patterns.

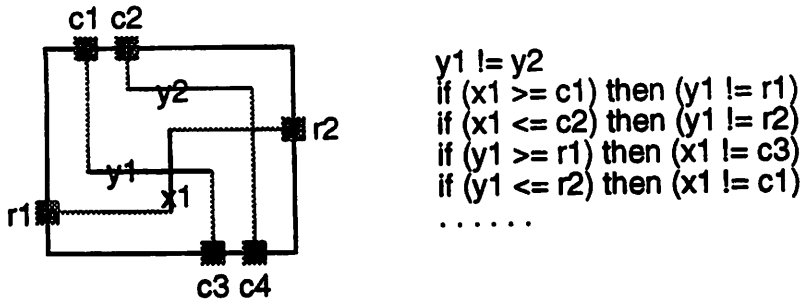

Figure 11: Switchbox routing example.

8.5 Graph Coloring

The objective of the graph coloring problem is to find the minimum number of colors that suffice to color a given graph G. Starting with a reasonable estimate k of the number of colors, let y_i denote the color of node i in G. The variable y_i is allowed to assume k values. For each pair of adjacent nodes in G, generate the following constraint:

$$y_i \neq y_j.$$

The final MDD for the conjunction of these constraints will test for the existence of a graph coloring with k colors. The problem can be simplified slightly if a maximal clique in the graph is preassigned a unique color for each node of the clique.

8.6 Cycle-based Logic Simulation

Logic simulation is a critical but time consuming step in the design cycle. Cycle-based logic simulation computes signal values for outputs and latches only. The core operation is discrete function evaluation of combinational logic blocks in between latch boundaries. In [27], a new approach of discrete function evaluation is proposed using MDDs. Each MDD variable corresponds to a group of inputs to the combinational block, and each output function is represented as a MDD. The MDD of a logic function is translated into a table on which function evaluation is performed by a sequence of address lookups. The value of a function for a given input assignment is obtained with at most one lookup per input. This represents a significant improvement over traditional simulation, because evaluation time becomes independent of the complexity of the logic network. Theoretically, MDD-based function evaluation offers orders of magnitude potential speedup over traditional logic simulation. In practices, memory bandwidth becomes the dominant consideration on large designs.

8.7 Formal Verification

Finite state machine (FSM) is a common representation for sequential designs on which many logic synthesis and formal verification programs are based. An FSM can be viewed as a 5-tuple $M = \langle S, I, O, T, R \rangle$ where S represents the finite state space, I represents the finite input space and O represents the finite output space. T is the transition relation defined by its characteristic function $T : I \times S \times S \times O \rightarrow B$. On input i, the FSM can transit from present state p to a next state n and output o if and only if $T(i, p, n, o) = 1$ (i.e., (i, p, n, o) is a transition). There exists one or more transitions for each combination of present state p and input i. $R \subseteq S$ represents the set of reset states.

A state transition graph (STG) is commonly used as the internal representation of FSMs in sequential synthesis systems, such as SIS [34]. Many algorithms for sequential synthesis have been developed to apply to STGs. However, large FSMs cannot be stored and manipulated without memory usage and CPU time

becoming prohibitively large. A limitation of STGs is the fact that they are a two-level form of representation where state transitions are stored explicitly, one by one. This may degrade the performance of conventional graph algorithms.

Alternatively, FSMs can be compactly represented by decision diagrams such as MDDs. A state s can be symbolically represented by a multi-valued variable, whose value can range over the discrete state space. Similarly, input i and output o are represented as multi-valued variables. As a result, the transition relation can be represented as a characteristic function (four multi-valued inputs, binary output MDD function). The set of reset states is represented by the MDD of its characteristic function.

Though various synthesis and verification applications use the above MDD representation, each algorithm is different in the way it manipulates intermediate MDD objects to produce the desired results. Here we will outline a few applications and the reader is referred to the literature for details.

Language containment and CTL model checking are two common verification methodologies. The first implicit algorithm for language containment [36] using MDDs was implemented within COSPAN [18]. HSIS [2] is a hierarchical formal verification system from UCB and its language containment and model checking algorithms are based on MDDs. This is true also for VIS [4], a package of verification interacting with synthesis recently developed at UCB.

At the core of language containment and CTL model checking is some form of state space exploration, e.g., state reachability. Implicit methods manipulate sets of states at a time. There are many examples of large state spaces that can be explored with implicit techniques but not with explicit ones. A straightforward translation of an explicit algorithm is not necessarily the best for MDDs. The language containment check is translated to a language emptiness check and this fails if there is an accepting run in the automaton. A fair state is one that is involved in some cycle satisfying all fairness constraints and thus a reachable fair state means a failed language containment check.

During and after logic synthesis, combinational logic verification is used to certify that the resultant circuit description is functionally equivalent to the initial description. BDDs have been used very successfully to compare Boolean logic networks [25] where a BDD function (representing a Shannon decomposition of the network functionality) is built for each Boolean network. Each BDD is a canonical form of the Boolean function of binary-valued variables. Hence verifying that two Boolean functions are identical reduces to verifying that their BDDs are identical. Verification of multi-valued networks is a straightforward extension, as multi-valued functions are identical if and only if their MDDs are identical. This verification step is carried out after MIS-MV [21] optimizes a multi-level logic with multi-valued inputs. A lot of efforts have been invested in the verification of sequential networks with BDDs [37, 17], but more work is needed to fill the gap between what can be verified currently (circuits with at most a few hundred latches) and industrial-strength designs (circuits with thousands of latches).

8.8 Logic Synthesis

MDDs are used also in POLIS [8], a system for hardware-software co-design of embedded systems developed at UCB. They are utilized in representing the transition relation of a Co-design FSM (CFSM) [3] and the control-data flow graphs (S-graphs) for software synthesis.

MDDs have also been applied in combinational and sequential logic optimization. An example is the minimization of multi-valued relations in the program GYOCRO [41].

In GYOCRO a relation $R \subseteq D \times B^m$ is represented by its characteristic function $R : D \times B^m \rightarrow B$ such that $R(\mathbf{x}, \mathbf{y}) = 1$ if and only if $(\mathbf{x}, \mathbf{y}) \in R$. The characteristic functions are represented by MDDs.

[3] A CFSM is a globally asynchronous FSM with finite, non-zero, unbounded reaction time and point to point communication to model both hardware and software implementations.

Multi-valued relations arise in many situations [5]. An application is in the synthesis of completely specified FSMs. For a given initial state, a set of equivalent states can be computed as a function $E : S \times S \rightarrow B$ such that $E(n, \tilde{n}) = 1$ if and only if n and $\tilde{n}$ are equivalent. Since a state can be mapped to any of the equivalent states of the next state, we have the possibility of implementing a more compact machine using the equivalent states. Namely, the objective is to find a least cost machine compatible with the function $\tilde{T} : I \times S \times S \times O \rightarrow B$ such that $\tilde{T}(i, p, n, o) = 1$ if and only if either $T(i, p, n, o) = 1$ or there exists a state $\tilde{n}$ for which $T(i, p, \tilde{n}, o) = 1$ and $E(n, \tilde{n}) = 1$. $\tilde{T}$ can be easily computed using MDDs. $\tilde{T}$ provides the complete family of finite state machines equivalent to the original machine under the equivalent states. Similarly, $\tilde{T}$ can be extended to include invalid states, which are defined as a set of states not reachable from some initial set of states.

8.9 Constrained Finite State Machine Minimization

In [22] L. Lavagno has described the following variant of state minimization occurring in asynchronous sequential synthesis and he has provided an exact solution which uses MDDs. We mention briefly the problem in regard to the usage of MDDs, referring to the source for an in-depth presentation.

Definition 8.1 *An input variable x is* **enabled** *in a state s of an FSM if it has a different value in a pair of edge labels respectively entering and leaving s.*

For instance, given $(00, s_1 \rightarrow s)$ and $(10, s \rightarrow s_2)$ the first variable is enabled in s, the second one is not (unless there is another edge label leaving s where the second variable takes value 1).

Given an ISFSM F of Moore type, a set of incompatible pairs of states of F, and a cost for each input variable of F, the problem is to find a **closed partition** of the states of F and a set D of input variables of F such that:

1) no two incompatible states are assigned to the same block,

2) for every pair of adjacent states $s \rightarrow s'$ assigned to different blocks, the set of variables that are enabled in s' and not enabled in s is contained in D,

3) no state s has two fanout edges going to two different states s' and s'' such that s' and s'' belong to the *same* block, different from the block of s (this last condition may require to drastically increase the number of blocks with respect to the minimum in standard state minimization),

4) the cost of the variables in D is minimized.

The exact algorithm given in [22] for the derivation of an optimal partitioning set is divided into three steps:

1. Formulation, as a conjunction of logic expressions over a set of multi-valued variables, of the conditions for a set D of STG signals to be a partitioning set with respect to any closed partition π derived from a closed cover C.

2. Partial solution of the clauses, to find a partitioning set D of minimum cost.

3. Derivation of π from C and D.

A minimum cost partitioning set, given the clauses defining it, is found by extending to MDDs the approach described in [24] to solve the binate covering problem using BDDs. So, given an MDD representing a conjunction of the clauses, any path from the root to the leaf labeled with 1 corresponds to a partial

assignment of values to the variables that satisfies the clauses. Hence this partial assignment represents a family of partitioning sets and associated closed partitions. A weight is assigned to each edge in the MDD, according to the cost function.

Then a shortest path from the root to the leaf labeled with 1 corresponds to a minimum cost assignment that satisfies all the constraints. The proof was given for BDDs, but since every MDD can be translated into a BDD with an appropriate encoding and the weights assigned to the multi-valued variables are all zero, the result applies directly to this case as well.

The assignment corresponding to a shortest path gives also a compatible for each state, that unfortunately cannot be used as its partition block, because the resulting partition may not be closed. In principle, one could add further clauses expressing the closure conditions, and use again the shortest path formulation.

9 Applications of 1-hot Encoded MDDs

9.1 Implicit Compatible Generation for State Minimization of ISFSMs

An **incompletely specified FSM** (ISFSM) can be defined as a 6-tuple $M = \langle S, I, O, \mathcal{T}, \mathcal{O}, R \rangle$ where S, I, O and R represents the states, inputs, outputs and reset states. $\mathcal{T}$ is the next state relation defined by: $\mathcal{T}(i, p, n) = 1$ iff n is the specified next state of state p on input i. $\mathcal{O}$ is the output relation defined by: $\mathcal{O}(i, p, o) = 1$ iff o is a possible output of state p on input i. The following example will be used to illustrate the implicit computations.

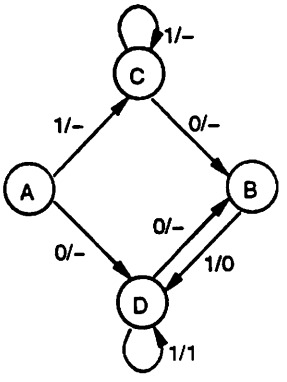

Figure 12: An ISFSM.

An exact algorithm for state minimization consists of two steps: generation of various sets of compatibles, and solution of a binate covering problem. The generation step involves identification of sets of states called compatibles which can potentially be merged into a single state in the minimized machine. For ISFSMs, the number of compatibles can be exponential in the number of states. Such state sets can be represented efficiently as positional-sets so that any set of state sets can be represented as a *1-hot encoded MDD*. Input and output MDD variables are left *logarithmic encoded.*

The covering step (described in Section 9.2) is to choose a minimum subset of compatibles satisfying covering and closure conditions, i.e., to find a minimum closed cover. In this section, we describe implicit computations to find sets of compatibles required for exact state minimization. First, incompatibility relations between pairs of states are derived from the output and transition relations.

Definition 9.1 *Two states are an* **output incompatible pair** *if, for some input, they cannot generate the*

same output. The set of output incompatible pairs, $\mathcal{OICP}(y,z)$, can be computed as:

$$\mathcal{OICP}(y,z) = Tuple_1(y) \cdot Tuple_1(z) \cdot \exists i \; \not\exists o \, [\mathcal{O}(i,y,o) \cdot \mathcal{O}(i,z,o)]$$

The last term is true for state B and D because on some input 1, they can both produce a same output pattern (i.e., output incompatible). The conditions $Tuple_1(y) \cdot Tuple_1(z)$ restrict y and z positional-sets to pairs of singleton states.

OICP(y, z)	
0100 B	0001 D
0001 D	0100 B

Definition 9.2 *Two states are an* **incompatible pair** *if (1) they are output incompatible, or (2) on some input, their next states are an incompatible pair. The set of incompatible pairs $\mathcal{ICP}$ can be obtained by the following fixed point computation:*

$$\begin{aligned} \mathcal{ICP}_0(y,z) &= \mathcal{OICP}(y,z) \\ \mathcal{ICP}_{k+1}(y,z) &= \mathcal{ICP}_k(y,z) + \exists i,u,v \, [\mathcal{T}(i,y,u) \cdot \mathcal{T}(i,z,v) \cdot \mathcal{ICP}_k(u,v)] \end{aligned}$$

The iteration can terminate when $\mathcal{ICP}_{k+1} = \mathcal{ICP}_k \; (= \mathcal{ICP})$.

The iteration starts with the set of output incompatible pairs $\mathcal{ICP}_0 = \{BD\}$. $\mathcal{ICP}_1$ contains all state pairs leading to pair(s) in $\mathcal{ICP}_0$ under some input, i.e., $\mathcal{ICP}_1 = \{BD, AC, AD\}$. And $\mathcal{ICP} = \mathcal{ICP}_1 = \mathcal{ICP}_2$.

ICP(y, z)	
1000 A	0010 C
1000 A	0001 D
0100 B	0001 D

ICP(y,z)(cont)	
0010 C	1000 A
0001 D	1000 A
0001 D	0100 B

So far we established relationships between pairs of states. By complementation, the following definition introduces *compatible* sets of states of arbitrary cardinalities.

Definition 9.3 *A set of states is a* **compatible** *if it is not an incompatible. An incompatible set of states contains at least one incompatible pair. The set of compatibles, $\mathcal{C}(c)$, can be computed as:*

$$\mathcal{C}(c) = \neg Tuple_0(c) \cdot \not\exists y,z \, [\mathcal{ICP}(y,z) \cdot Contain\_Union(c,y,z)]$$

$\mathcal{C}(c)$ simply contains all non-empty subsets of states (i.e., $\neg Tuple_0(c)$) which are *not* incompatibles. A state set c is an incompatible iff there is an incompatible state pair $\mathcal{ICP}(y,z)$ such that c contains y and z.

C(c)
1100 AB
0110 BC
0011 CD
1000 A
0100 B
0010 C
0001 D

The closure condition of a compatible c is captured by its class set d. The class set relation $CCS(c, d)$ evaluates to 1 iff the next state set d implied by c is in its class set. Its computation [20] will be omitted here.

```
        CCS(c,d)
----------------
1100 AB   0011 CD
0110 BC   0011 CD
```

To solve exactly the covering problem, it is sufficient to consider a subset of compatibles called prime compatibles. As proved in [16], at least one minimum closed cover consists entirely of prime compatibles.

Definition 9.4 *A compatible c'* **dominates** *a compatible c if (1) $c' \supset c$, and (2) class set of $c' \subseteq$ class set of c. The prime dominance relation is given by:*

$$Dominate(c', c) = (c' \supset c) \cdot Set\_Contain_d(CCS(c, d), CCS(c', d))$$

i.e., c' dominates c if c' covers all states covered by c, and the closure conditions of c' are a subset of the closure conditions of c. As a result, compatible c' expresses strictly less stringent conditions than compatible c, thus c can be excluded from further consideration.

Definition 9.5 *A* **prime compatible** *is a compatible not dominated by another compatible. The set of prime compatibles is given by:*

$$PC(c) = C(c) \cdot \not\exists c' [C(c') \cdot Dominate(c', c)]$$

prime compatible		class set
p_1	1100 AB	0011 CD
p_2	0110 BC	0011 CD
p_3	0011 CD	$\emptyset$
p_4	1000 A	$\emptyset$
p_5	0100 B	$\emptyset$

9.2 Implicit Binate Covering

Binate covering models a large class of optimization problems in logic synthesis. We refer to [38] for a detailed presentation of algorithms to solve it. In [20] we have contributed the first binate solver that represents and transforms the covering table using 1-hot encoded MDDs as the underlying data structure. Here we outline some key aspects that characterize the new algorithm. In the exposition we continue the solution of the instance of state minimization shown in Section 9.1.

A minimized machine is obtained by finding a minimum closed cover. The latter is determined by:

1. constructing a product-of-sums expression $\mathcal{E}$, with one variable per prime compatible, and as many clauses as there are covering and closure conditions;
2. finding a satisfying assignment which has the fewest variables assigned to 1, i.e., the fewest prime compatibles selected to form the reduced machine.

The clauses of the product-of-sums expression $\mathcal{E}$ are unate clauses for the covering conditions (each original state is covered by at least one selected prime compatible):

$$A : (p_1 + p_4), B : (p_1 + p_2 + p_5), C : (p_2 + p_3), D : (p_3),$$

and binate clauses for the closure conditions (if a prime compatible is selected also the prime compatibles that will be its next states in the reduced machine must be selected):

$$AB \Rightarrow CD : (\overline{p_1} + p_3), BC \Rightarrow CD : (\overline{p_2} + p_3).$$

So the final product-of-sums expression is:

$$\mathcal{E} = (p_1 + p_4)(p_1 + p_2 + p_5)(p_2 + p_3)p_3(\overline{p_1} + p_3)(\overline{p_2} + p_3).$$

A minimum satisfying assignment is $p_1 = p_3 = 1$ and $p_2 = p_4 = p_5 = 0$.

The same expression can be rewritten as a covering table, where each column is a prime compatible and each row is a clause:

	AB	BC	CD	A	B
A	1			1	
B	1	1			1
C		1	1		
D			1		
$AB \Rightarrow CD$	0		1		
$BC \Rightarrow CD$		0	1		

The binate table covering problem is to find a minimum subset of columns such that for each row, either there is a column in the subset intersecting the row at a 1, or there is a column *not* in the subset intersecting the row at a 0. Available explicit implementations represent the table as a matrix, using a sparse matrix package.

We avoid an explicit representation of each entry, row and column in the table. Instead we represent the table implicitly by the following encoding scheme:

- p - a column label (a positional set), (c, d) - a row label (2 positional sets)
- $C(p)$ - a set of columns (a BDD), $R(c, d)$ - a set of rows (a BDD)

Each column label is a prime compatible:

$$C(p) = \mathcal{PC}(p)$$

Each row label represents a unate or binate clause:

$$\begin{aligned} R_{unate}(c, d) &= \emptyset(c) \cdot \mathcal{S}(d) \\ R_{binate}(c, d) &= \mathcal{PC}(c) \cdot \mathcal{CCS}(c, d) \\ R(c, d) &= R_{unate}(c, d) + R_{binate}(c, d) \end{aligned}$$

At the intersection of column $p \in C$ and row $(c, d) \in R$, the following rules hold:

1. table entry is a 1 iff $(p \supseteq d)$,
2. table entry is a 0 iff $(p = c)$.

In our example the rows are encoded as follows. For each state d, a **unate clause** $(p_1 + p_2 + \cdots + p_j)$ has to be satisfied, where p_k is a prime compatible containing state d. So the unate clauses are:

$$R_{unate}(c, d) = \emptyset(c) \cdot \mathcal{S}(d)$$

```
   Ru(c, d)
---------------
0000     1000 A
0000     0100 B
0000     0010 C
0000     0001 D
```

For each prime compatible p and each of its class sets d, a **binate clause** $(\overline{p} + p_1 + p_2 + \cdots + p_i)$ has to be satisfied, where p_k is a prime compatible containing the class set d. So the binate clauses are:

$$R_{binate}(c,d) = \mathcal{PC}(c) \cdot \mathcal{CCS}(c,d)$$

```
   Rb(c, d)
---------------
1100 AB 0011 CD
0110 BC 0011 CD
```

As a summary, we annotate the previous table with the labels of the rows and columns. Notice that we do not need anymore to represent the entries of the table. With the given rules, we can always determine what entry exists at the intersection of a given row and column, by checking their labels.

		AB 1100	BC 0110	**CD 0011**	A 1000	B 0100
A	0000 1000	1			1	
B	0000 0100	1	1			1
C	0000 0010		1	**1**		
D	**0000 0001**			**1**		
$BC \Rightarrow CD$	0110 0011		0	**1**		
$AB \Rightarrow CD$	1100 0011	0		**1**		

We solve exactly the binate covering problem with a branch-and-bound algorithm, that differs from a standard one due to: (1) implicit representation of the covering table, (2) implicit computation of a reduced covering table, and (3) implicit computation of branching column, maximal independent set and table partitioning.

A covering table is reduced by applying to it a sequence of operations that remove rows and columns. and still preserve at least one minimum solution. We demonstrate two such operations: detection of essential columns and column dominance.

A column p is **essential** iff there is a row having a 1 in column p and no other entry. In our table there is one such column, labeled by **0011** as highlighted in the above table which is essential to cover row **0000 0001**.

The essential columns are computed by:

$$ess\_col(p) = C(p) \cdot \exists c,d\,[R(c,d) \cdot \emptyset(c) \cdot (p \supseteq d) \cdot \nexists p'\,(C(p') \cdot (p' \supseteq d) \cdot (p' \neq p))]$$

Since the essential columns must be in the solution, they are deleted from the table together with all rows intersecting them in a 1. The computations to add the essential columns to the solution and update the table are:

$$\begin{aligned} solution(p) &= solution(p) + ess\_col(p) \\ C(p) &= C(p) \cdot \neg ess\_col(p) \\ R(c,d) &= R(c,d) \cdot \nexists p\,(ess\_col(p) \cdot (p \supseteq d)) \end{aligned}$$

The resulting table is:

		AB **1100**	BC 0110	A 1000	B 0100
A	0000 1000	**1**		1	
B	0000 0100	**1**	1		1

A column p' **dominates** another column p iff p' has all the 1s of p, and p' contains no 0. In our table above, the column labeled **1100** dominates all the other columns.

The dominated columns are computed by:

$$dominated(p) = \exists p' \{C(p') \cdot (p' \neq p) \cdot \nexists c, d\,[R(c,d) \cdot (p \supseteq d) \cdot (p' \not\supseteq d)] \cdot \nexists d\, R(p',d)\}$$

If p' column dominates p, there is at least one minimum cost solution with column p eliminated ($p = 0$), together with all the rows in which it has 0s. Therefore dominated columns are deleted, along with the rows intersecting them in a 0:

$$\begin{aligned} C(p) &= C(p) \cdot \neg dominated(p) \\ R(c,d) &= R(c,d) \cdot \neg dominated(c) \end{aligned}$$

and the table is reduced to:

		AB **1100**
A	0000 1000	**1**
B	0000 0100	**1**

By one more search of essential columns one finds that p_1 is essential and so we found a minimum cost $solution = \{p_1, p_3\} = \{AB, CD\}$!

9.3 Implicit Minimization of Generalized Prime Implicants

The problem of state assignment for optimal two-level implementations has a long history of research efforts [28, 39]. An exact algorithm was proposed in [13]. It extends to the multi-valued input and output domains the two main features of exact standard two-level minimization: generation of a set of product-terms sufficient to find at least a minimum cover, i.e., the prime implicants, and computation of a minimum cover as solution of a set covering problem, represented as a table covering problem [26]. More precisely, in [13] the notion of prime implicants is extended to the notion of **generalized prime implicants** (GPIs) and the set covering problem is extended to a constrained set covering problem, because it is not sufficient to find a minimum cover of GPIs, but it is necessary to find a minimum encodeable cover, i.e., a minimum cover of GPIs whose associated encoding constraints are satisfiable so that it can be mapped into an equivalent encoded cover. This is the problem of exact minimization of GPIs. Since GPIs are a superset of prime implicants and moreover subsets of GPIs must be checked for encodeability, GPI minimization is absolutely intractable from the point-of-view of explicit enumerative techniques.

In [40] it is described an implicit procedure to compute minimum or minimal encodeable covers of GPIs. It uses 1-hot encoded MDDs to check encodeability of encoding constraints, and it relies on the implicit table solver described in Section 9.2 to select covers of GPIs. The procedure is quite intricate and we refer the interested reader to the original documentation.

10 Conclusions

We have presented the multi-valued decision diagram data structure and along with a suite of operators for its manipulation. As a natural setting to model problems with discrete variables, MDDs have been successfully applied to a very wide variety of problems.

We have shown that logarithmic encoded MDDs are particularly useful to represent compactly multi-valued functions, sets, relations and graphs. They have been applied to combinatorial optimization problems such as graph coloring, channel and switchbox routing, and hardware resource scheduling. Also, logarithmic encoded MDDs are capable of representing huge transition relations, and formal verification systems developed at UCB and at Bell Labs are based on them. Finally, the MDD was the main idea for fast function evaluation behind a new breed of cycle-based logic simulators.

Positional-sets have been introduced to represent sets of objects so that a set of such sets (e.g., a set of compatibles) can be represented by a single 1-hot encoded MDD. This idea has been tested in sequential logic synthesis, enabling to find an exact solution of some hard state minimization and GPI minimization instances. While developing such implicit algorithms, we have developed a fully implicit solver of binate covering. Our solver can be applied to many problems in computer-aided design and combinatorial optimization. Also implicit compatible generation has other applications in logic synthesis [31].

An efficient, general purpose MDD package is available and is distributed with U.C. Berkeley's VIS [4] and HSIS [2] software.

References

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, vol. 27:509–516, 1978.

[2] A. Aziz, F. Balarin, S. Cheng, R. Hojati, T. Kam, S. Krishnan, R. Ranjan, T. Shiple, V. Singhal, S. Tasiran, H. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. HSIS: A BDD-based environment for formal verification. In *The Proceedings of the Design Automation Conference*, pages 454–459, June 1994.

[3] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *The Proceedings of the Design Automation Conference*, pages 40–45, June 1990.

[4] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. In R. Alur and T. Henzinger, editors, *Proc. of the Conf. on Computer-Aided Verification*, volume 1102 of *LNCS*, pages 332–334. Springer Verlag, August 1996.

[5] R. Brayton and F. Somenzi. An exact minimizer for Boolean relations. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 316–319, November 1989.

[6] R. Bryant. Graph based algorithm for Boolean function manipulation. In *IEEE Transactions on Computers*, pages C–35(8):667–691, 1986.

[7] E. Cerny. Characteristic functions in multivalued logic systems. *Digital Processes*, vol. 6:167–174, June 1980.

[8] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis of software programs from CFSM specifications. In *The Proceedings of the Design Automation Conference*, June 1995.

[9] E. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large Boolean functions with application to technology mapping. In *The Proceedings of the Design Automation Conference*, pages 54–60, June 1993.

[10] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems, vol. 407 of Lecture Notes in Computer Science*, pages 365–373, June 1989.

[11] O. Coudert and J.C. Madre. Implicit and incremental computation of prime and essential prime implicants of Boolean functions. In *The Proceedings of the Design Automation Conference*, pages 36–39, June 1992.

[12] S. Devadas. Optimal layout via boolean satisfiability. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 294–297, November 1989.

[13] S. Devadas and R. Newton. Exact algorithms for output encoding, state assignment and four-level Boolean minimization. *IEEE Transactions on Computer-Aided Design*, 10(1):13–27, January 1991.

[14] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computer-Aided Design*, vol. 39(no. 5):710–713, May 1990.

[15] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 2–5, November 1988.

[16] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IRE Transactions on Electronic Computers*, EC-14(3):350–359, June 1965.

[17] G. Hachtel and F. Somenzi. *Logic synthesis and verification algorithms*. Kluwer Academic, 1996.

[18] Z. Har'El and R.P. Kurshan. Software for analysis of coordination. *Proc. Int. Conf. Syst. Sci. Eng.*, pages 382–385, 1988.

[19] T. Kam and R.K. Brayton. Multi-valued decision diagrams. *Tech. Report No. UCB/ERL M90/125*, December 1990.

[20] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. In *The Proceedings of the Design Automation Conference*, pages 684–690, June 1994.

[21] L. Lavagno, S. Malik, R. Brayton, and A. Sangiovanni-Vincentelli. MIS-MV: Optimization of multi-level logic with multiple valued inputs. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 560–563, November 1990.

[22] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits*. Kluwer Academic, 1993.

[23] B. Lin, O. Coudert, and J.C. Madre. Symbolic prime generation for multiple-valued functions. In *The Proceedings of the Design Automation Conference*, pages 40–44, June 1992.

[24] B. Lin and F. Somenzi. Minimization of symbolic relations. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1990.

[25] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 6–9, November 1988.

[26] E. McCluskey. Minimization of Boolean functions. *Bell Laboratories Technical Journal*, 35:1417–1444, November 1956.

[27] P. C. McGeer, K. L. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, and P. Scaglia. Fast discrete function evaluation using decision diagrams. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 402–407, November 1995.

[28] G. De Micheli, R. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design*, pages 269–285, July 1985.

[29] S. Minato. *Binary decision diagrams and applications for VLSI CAD*. Kluwer Academic, 1996.

[30] A. Oliveira. Implicit minimization of loop free finite state machines using Zero-suppressed BDDs. *INESC Internal Report, Lisbon, Portugal*, October 1996.

[31] A. Oliveira, L. Carloni, T. Villa, and A. Sangiovanni-Vincentelli. Exact minimization of binary decision diagrams using implicit techniques. *Tech. Report No. UCB/ERL M96/16*, April 1996.

[32] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, November 1993.

[33] T. Sasao and M. Fujita. *Representations of discrete functions*. Kluwer Academic, 1996.

[34] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *The Proceedings of the International Conference on Computer Design*, pages 328–333, October 1992.

[35] A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 92–95, November 1990.

[36] H. Touati, R. Brayton, and R. Kurshan. Testing language containment of ω-automata using BDDs. *Information and Computation*, 118(1):101–109, April 1995.

[37] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. *The Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, November 1990.

[38] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli. Explicit and implicit algorithms for binate covering problems. *Tech. Report No. UCB/ERL M95/108*, December 1995.

[39] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment for optimal two-level logic implementations. *IEEE Transactions on Computer-Aided Design*, 9(9):905–924, September 1990.

[40] Tiziano Villa. *Encoding Problems in Logic Synthesis*. PhD thesis, University of California, Berkeley, Electronics Research Laboratory, May 1995. Memorandum No. UCB/ERL M95/41.

[41] Y. Watanabe and R. Brayton. Heuristic minimization of multi-valued relations. *IEEE Transactions on Computer-Aided Design*, vol. 12(no. 10):1458–1472, October 1993.

[42] T. Yoshimura and E.S. Kuh. Efficient algorithms for channel routing. *IEEE Transactions on Computer-Aided Design*, pages 25–35, January 1982.