MULTI-VALUED DECISION DIAGRAMS

by

Timothy Y.K. Kam and Robert K. Brayton

Memorandum No. UCB/ERL M90/125

21 December 1990

MULTI-VALUED DECISION DIAGRAMS

by

Timothy Y.K. Kam and Robert K. Brayton

Memorandum No. UCB/ERL M90/125

21 December 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

MULTI-VALUED DECISION DIAGRAMS

by

Timothy Y.K. Kam and Robert K. Brayton

Memorandum No. UCB/ERL M90/125

21 December 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Multi-valued Decision Diagrams

Timothy Y.K. Kam and Robert K. Brayton

University of California
Berkeley, CA 94720

Department of Electrical Engineering
and Computer Sciences

Abstract

Many CAD problems are NP-complete or coNP-complete. They can usually be stated naturally as discrete functions defined on multi-valued variables which take values from discrete sets. Traditionally, heuristics are devised for these problems to find good solution(s). The technique of this report implicitly enumerates all possible solutions which are stored in a compact graph structure, called the multi-valued decision diagram (MDD). Thus, such an approach guarantees to find the exact, optimum solution(s) if they exist, or proves the non-existence of a solution. This research was inspired by its binary analog, the binary decision diagrams (BDD's), as presented by Bryant [Bry86].

In this report, the MDD is defined and its properties are analyzed. Algorithms for constructing and manipulating MDD's are provided. Various problems, ranging from graph coloring to routing to scheduling, are formulated and solved using MDD's. The first, direct implementation of the MDD structure was reported in an earlier paper [SKMB90]. With the advent of an efficient BDD package based on [BRB90], a new MDD package was developed in which MDD's are mapped onto BDD's according to an optimal encoding. The limits on the sizes of real problems which can be handled by the MDD package is explored. With good variable ordering and mapping heuristics, all the benchmark examples tried can be solved within a minute on a VAX 8800. Various efficient encoding, ordering and mapping techniques are discussed.

This work represents just a beginning for exploring the capabilities of MDD's and the results are encouraging. As a natural setting to approach discrete symbolic variable problems, MDD's should have many other applications. One exciting application involves using the MDD package for the formal verification of finite state machines. We are currently integrating this with the AT&T COSPAN verification system [HK88].

Acknowledgements

This work is funded in part by the Defense Advanced Research Projects Agency under contract N00039-87-C-0182, and grants from IBM, DEC, Intel, Motorola, AT&T, BNR, and the California state MICRO program. The authors gratefully acknowledge their support. In addition, we express thanks to many members of the Berkeley CAD research group, with special thanks to Professor Alberto Sangiovanni-Vincentelli, Arvind Srinivasan, Hervé Touati and Sharad Malik for their helpful advice and discussions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many CAD problems have been proved to be NP-hard or coNP-complete [GJ79], which means that the best algorithms for solving such problems will suffer exponential-time (and sometimes exponential-space) complexities in the worst case. In some cases, there exists algorithms that are not polynomial but we can find, most of the time, a solution of the problem quickly, even in linear time. An example of this behavior is the tautology algorithm described in [BHMSV84]. A key to devising such algorithms so that they are practical is to use a compact representation of the problem itself or its solutions. We also need a set of efficient algorithms to manipulate the data structure. The approach of this report uses a graph-based representation of all solutions. We give it the name multi-valued decision diagram (MDD).

1.1 Optimization Problems and Decision Problems

Many CAD problems are formulated as *combinatorial optimization problems.* Graph coloring is a well-known combinatorial optimization problem and is used here as an example for illustration. Given a graph G(V,E) and a set of colors, the vertices are to be colored such that no two vertices connected by an edge will have the same color. Optimization problems requires the computation of a certain value (e.g. the minimum number of colors needed to color G) or the construction of a certain solution object (e.g. the coloring for G such that the minimum number of colors is used). Optimization problems can usually be solved as a sequence of *decision problems* - problems for which the only solutions are "Yes" and "No". (e.g. is there a coloring of G using at most k colors?) For example, the minimum number

```
$min = 0$;
$max = | V |$;
while ($min \neq max$) {
      $k = \frac{min+max}{2}$;
      if there is a coloring of G using at most k colors (decision problem)
        then $max = k$;
        else $min = k$;
}
return ($k$);
```

Figure 1.1: Binary search of optimal solution by solving decision problems.

of colors (the answer to the optimization problem) can be found by solving the decision problem at most $log_2(| V |)$ times using a binary search as illustrated in Figure 1.1.

This report concentrates mainly on solving decision problems. Such problems can be stated naturally as discrete functions using multi-valued variables that can take values from a discrete set. For the graph coloring example, the color assignment of each vertex can be represented by a multi-valued variable, each of which can take one of the k colors. For the graph example shown in Figure 1.2, we have four 3-valued variables, c_1, c_2, c_3 and c_4. Each can be assigned a color from the color set $\{r, g, b\}$.

A solution to the graph coloring problem is called a satisfying color assignment and can be denoted by a 4-tuple, e.g. (r, g, b, r). Now a discrete function $\mathcal{F}$ can be defined as:

$$\mathcal{F}(c_1, c_2, c_3, c_4) = \begin{cases} 1 & \text{if } (c_1, c_2, c_3, c_4) \text{ is a satisfying assignment} \\ 0 & \text{otherwise} \end{cases}$$

With the definition of $\mathcal{F}$, the solution to the decision problem is a "Yes" if and only if there is a satisfying color assignment, i.e.

$$\exists (c_1, c_2, c_3, c_4) \ s.t. \ \mathcal{F}(c_1, c_2, c_3, c_4) = 1$$

Thus the graph coloring problem can be transformed into the discrete function

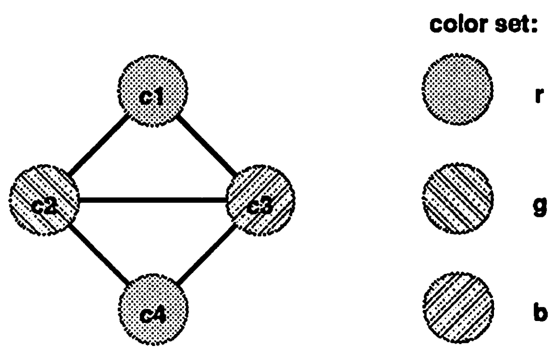


Figure 1.2: Graph coloring example.

$\mathcal{F}$. Other CAD problems can be similarly formulated using discrete functions, and will be described in Chapter 3. It is crucial to have a good representation for $\mathcal{F}$ such that

- satisfiability can be easily checked,
- if satisfiable, all/some satisfying assignments can be easily found.

The satisfiability test serves also as a proof of the existence or non-existence of solutions for the decision problem.

As we shall see in Chapter 2, the multi-valued decision diagram (MDD) has such properties, and will be used to represent the discrete function $\mathcal{F}$.

1.2 Previous Work

The multi-valued decision diagram (MDD) is inspired by and intimately related to its binary analog, the binary decision diagram (BDD). BDD's were first proposed by Akers [Ake78] and popularized by Bryant in [Bry86]. BDD's are graph-based representations of Boolean functions. Bryant imposed restrictions on them and proposed the reduced ordered BDD which is a canonical form, i.e. there is a unique (up to isomorphism) representation for any given function. Consequently, testing for satisfiability of a function and equivalence of two functions become trivial tasks using BDD's. The notion of a strong canonical form was introduced by Karplus in [Kar87]. Brace *et al.* provided efficient techniques for manipulating BDD's which have been incorporated in the Berkeley BDD package. The size of the BDD of a function is sensitive to the ordering of the input variables. Friedman *et al.* in [FS87] found an $O(n^2 3^n)$ algorithm for finding the optimal variable ordering. Faster variable ordering heuristics for BDD's have been provided by Malik *et al.* in [MWBSV88] and Fujita *et al.* in [FFK88]. Devadas [Dev89] formulated the channel routing, partitioning and placement problems using Boolean satisfiability and solved them using BDD's. We also formulate a variety of CAD problems as satisfiability tests but we do not restrict ourselves in the Boolean domain. We extend the graph-based representation to the more natural multi-valued setting. Most CAD problems can be stated naturally in terms of multi-valued variables instead of binary-valued ones. Discrete functions defined on these variables can also take on multi-valued output. The notation and definitions used in this report closely follow these publications. The first direct implementation of the MDD structure was developed by Srinivasan and was jointly published in [SKMB90]. This report will concentrate more on the second- and third-generation MDD package which uses BDD's indirectly, by mapping MDD's into BDD's.

1.3 Overview

The focus of this report is on solving the decision version of CAD problems. On the other hand, most schedulers, routers, etc, tackle the optimization problem. Although MDD's can be used to solve optimization problems, as described in Section 1.1 and 3.1, the strength of the MDD approach is in a slightly different domain. Thus the MDD approach should be viewed as complementing traditional approaches.

Traditional heuristic approaches do not guarantee to find a solution if it exists. Moreover, if it gives a solution, there is no guarantee on its optimality. Thus, the MDD approach is valuable for problems which have the following characteristics:

- A very tight set of constraints.
- A tight range of values for the multi-valued variables.
- Heuristic methods fail to generate a solution, or an acceptable solution.
- We want to know if an acceptable solution exists.

Interestingly enough, MDD's are very good at solving such highly constrained problems which we know have no or only a few solutions. The more constrained the problem is, the less solutions it will have, the smaller the intermediate and final MDD's will be and the faster the algorithm will run. Some examples of such applications will be given in Section 3.6

Chapter 2 is concerned with definitions and notation for MDD's. The theory behind MDD's is described. To emphasize its practicality, a variety of CAD problems are formulated using the MDD approach in Chapter 3. The remaining two-thirds of the report is devoted to the implementation of an efficient MDD package using mapped-BDD's. Chapter 4 describes the process of mapping MDD's into BDD's. Variable ordering is crucial to limit exponential complexities and some ordering heuristics are presented in Chapter 5. The user interface to the MDD package implementation is described in Chapter 6. Some benchmark results are presented in Chapter 7. Finally, Chapter 8 offers some conclusions and future directions of research.

Chapter 2

MDD Theory

2.1 Notation

2.1.1 Multi-valued Function

Definition 2.1 *Let $\mathcal{F}$ be a multiple-valued input, multiple-valued output function of n variables - $x_1, x_2, \ldots, x_n$.*

$$\mathcal{F} : P_1 \times P_2 \times \ldots \times P_n \rightarrow Y \tag{2.1}$$

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 $p_i = 2 \; \forall \; i \; \leq \; n$.

2.1.2 Literal

Definition 2.2 *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:*

$$x_i^{T_i} = \begin{cases} 0 & \textit{if } x_i \notin T_i \\ 1 & \textit{if } x_i \in T_i \end{cases} \tag{2.2}$$

2.1.3 Cofactor

Definition 2.3 *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:

$$\mathcal{F}_{x_i^j}(x_1,\ldots,x_n) = \mathcal{F}(x_1,\ldots,x_{i-1},j,x_{i+1},\ldots,x_n) \tag{2.3}$$

Definition 2.4 *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:*

$$\mathcal{F}_{x_i^{T_i}} = \bigcup_{j\in T_i} \mathcal{F}_{x^j} \tag{2.4}$$

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

2.1.4 Shannon Decomposition

Definition 2.5 *The* **Shannon decomposition** *of a function $\mathcal{F}$ with respect to a variable x_i is:*

$$\mathcal{F} = \sum_{j=0}^{p_i-1} x_i^j \cdot \mathcal{F}_{x_i^j} \tag{2.5}$$

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.

2.1.5 Smoothing Operator

Definition 2.6 *The* **smoothing** *of a function $\mathcal{F}$ by a variable x_i is denoted by $S_{x_i}\mathcal{F}$ and is defined as:*

$$S_{x_i}\mathcal{F} = \sum_{j=0}^{p_i-1} \mathcal{F}_{x_i^j} \tag{2.6}$$

2.1.6 Support

Definition 2.7 *The* **support** *of $\mathcal{F}$, denoted by $D_{\mathcal{F}}$, is the set of variables that $\mathcal{F}$ depends upon:*

$$D_{\mathcal{F}} = \{x_i \mid \exists j,k \ \ s.t.\ \ \mathcal{F}_{x_i^j} \neq \mathcal{F}_{x_i^k}\} \tag{2.7}$$

2.2 Multi-valued Decision Diagrams

This section describes a new data structure - the **multi-valued decision diagram** that is used to solve discrete variable problems. Our definition of multi-valued decision diagrams closely follows that of Bryant, [Bry86], with two exceptions: we do not restrict ourselves to the Boolean domain, and the range of our functions is multi-valued. From Chapter 3 on, we use only binary-valued output functions, however the theory is valid for the more general multi-valued functions.

Definition 2.8 *An* **multi-valued decision diagram** *(MDD) is a rooted, directed acyclic graph with a vertex set V containing two types of vertices. Each* **nonterminal** *vertex v_j is labeled with a multi-valued variable x_i. It has as attributes an argument* **index** *$index(v_j)$, which designates a variable associated with that vertex, where $1 \leq index(v_j) \leq n$, a* **range** *$range(v_j)$ where $range(v_j) = P_{index(v_j)}$, and $\mid P_{index(v_j)} \mid$* **children**, *$child_k(v_j) \in V$, $\forall\ k \in P_{index(v_j)}$. A* **terminal** *vertex u_m has as attribute a* **value** *$value(u_m) \in Y = \{0,\ 1,\ \ldots,\ m-1\}$.*

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

2.3 Reduced Ordered MDD's

Definition 2.9 *An MDD is* **ordered** *if $index(v) < index(child_k(v))$ for any nonterminal vertex v such that $child_k(v)$ is also nonterminal.*

Definition 2.10 *An MDD is* **reduced** *if (1) it contains no vertex v such that for all children, $child_j(v) = child_k(v)$, and (2) it does not contain two distinct vertices v and v' such that the subgraphs rooted at v and v' are isomorphic.*

Definition 2.11 *A* **reduced ordered multi-valued decision diagram** *(ROMDD) is an MDD which is both reduced and ordered.*

Henceforth, we consider only ROMDD's 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 input variables have been permuted so that

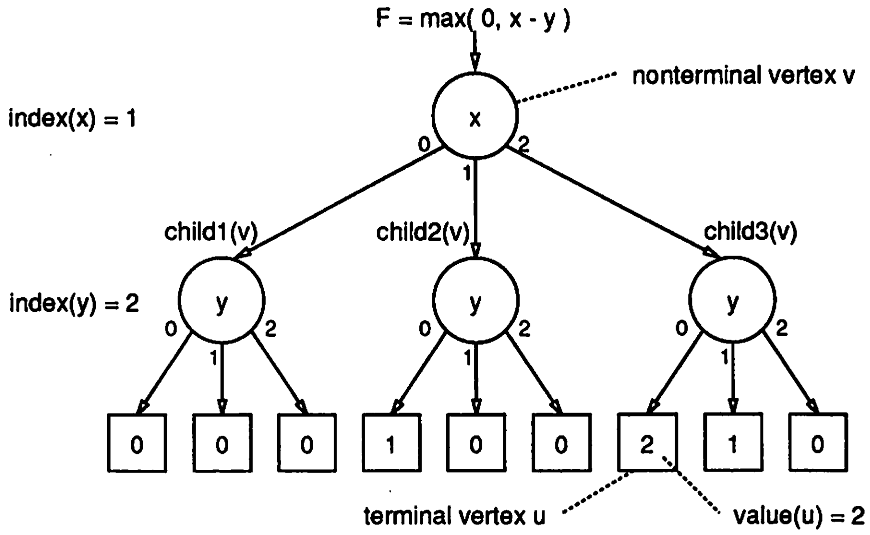


Figure 2.1: Example of an MDD for a discrete function.

$index(x_i) < index(x_{i+1})$. MDD's are guaranteed to be reduced at any time during the constructions and operations on MDD's discussed in Chapter 4. Each operation returns a resultant MDD in a reduced ordered form.

Example The ROMDD for the MDD in Figure 2.1 is shown in Figure 2.2. The variable ordering is $x \prec y$. i.e. $index(x) = 1$ and $index(y) = 2$. Note that one redundant nonterminal vertex and six terminal vertices have been eliminated.

A very desirable property of a 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 proof is given in [SKMB90]. □

Corollary 2.2 *Two functions are equivalent if and only if the ROMDD's for each function are isomorphic.*

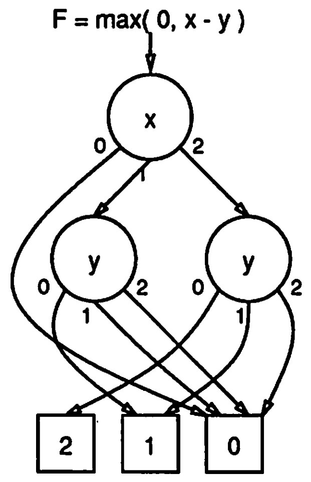


Figure 2.2: Reduced ordered MDD for the same function.

2.4 Strong Canonical Form

The notion of a strong canonical form was first introduced by Karplus in [Kar87] and was used for BDD's in [BRB90]. Equivalent functions are represented by the same address pointer in memory. So each function is represented exactly once in memory.

Each MDD vertex v_j, with its subgraph rooted from it, represents an multi-valued function F. Each function F has a top-variable x_i, which is the variable labeled at the vertex v_j which represents F. Each MDD vertex can be denoted by a $(p_i + 1)$-tuple $(x_i,\ child_0(v_j),\ child_1(v_j),\ \ldots,\ child_{p_i-1}(v_j))$ where $p_i = |\ range(x_i)\ |$.

In the first generation MDD package, unique identifiers are associated to each $(p_i + 1)$-tuple. The strong canonical form is maintained by the use of a unique-table which maps identifiers to MDD vertices. When a new vertex is needed, the unique table is first checked. If the corresponding $(p_i + 1)$-tuple already exists, its address pointer is reused, otherwise a new vertex (table entry) for the new $(p_i + 1)$-tuple is created. With a strong canonical representation, equivalence between functions can be checked very easily by:

Corollary 2.3 *Two functions are equivalent if and only if they are mapped into the same unique-table entry.*

2.5 CASE Operator

In providing an MDD package, it is desirable to disallow the user to modify the MDD data structure directly. Rather, we manipulate MDD's indirectly by performing a set of operations on the MDD function. The CASE operator forms the basis for constructing and manipulating MDD's. Most operations on discrete functions can be expressed in terms of the CASE operator on MDD's. It is the singly most important operator in the MDD package. The rest of this chapter and the next chapter is devoted to the CASE operator.

Definition 2.12 *The CASE operator selects and returns a function G_i according to the value of the function F:*

$$CASE(F, G_0, G_1, \ldots, G_{m-1}) \;=\; G_i \; if \; (F = i) \tag{2.8}$$

The operator is defined only if range(F) = {0, 1, ..., m-1}. The function returned from the CASE operation has a range of 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 MDD's. 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.

In Section 2.4, we described how we represent MDD vertices by $(p_i + 1)$-tuples. The $(p_i + 1)$-tuple for a vertex v_j with top-variable x is actually:

$$(x, G_0, G_1, \ldots, G_{p_i-1}) = CASE(x, G_0, G_1, \ldots, G_{p_i-1}) \tag{2.9}$$

and also we know that

$$CASE(F, 0, 1, \ldots, m-1) = F \tag{2.10}$$

Equation 2.9 and 2.10 will form the terminal cases for our recursive algorithm.

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

$$\begin{aligned} H &= \sum_{j=0}^{p-1} x^j \cdot H_{x^j} \\ &= CASE(x, H_{x^0}, H_{x^1}, \ldots, H_{x^{p-1}}) \\ &= (x, H_{x^0}, H_{x^1}, \ldots, H_{x^{p-1}}) \end{aligned} \tag{2.11}$$

Recursion is based on the following reasoning. Remember in Section 2.1 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,

$$\begin{aligned}
CASE(F, G_0, G_1, \ldots, G_{m-1}) &= \sum_{i=0}^{p-1} x^i \cdot CASE(F, G_0, G_1, \ldots, G_{m-1})_{x^i} \\
&= \sum_{i=0}^{p-1} x^i \cdot (G_j \ if \ (F = j))_{x^i} \\
&= \sum_{i=0}^{p-1} x^i \cdot (G_{j\ x^i} \ if \ (F = j)_{x^i}) \\
&= \sum_{i=0}^{p-1} x^i \cdot (G_{j\ x^i} \ if \ (F_{x^i} = j)) \\
&= CASE(x, \\
&\quad CASE(F_{x^0}, G_{0\ x^0}, G_{1\ x^0}, \ldots, G_{m-1\ x^0}), \\
&\quad CASE(F_{x^1}, G_{0\ x^1}, G_{1\ x^1}, \ldots, G_{m-1\ x^1}), \\
&\quad \ldots, \qquad (2.12) \\
&\quad CASE(F_{x^{p-1}}, G_{0\ x^{p-1}}, G_{1\ x^{p-1}}, \ldots, G_{m-1\ x^{p-1}}))
\end{aligned}$$

The pseudo-code for the recursive CASE algorithm is given in Figure 2.3. 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. It not, the cofactors H_{x^j} of the function H are computed by calling CASE recursively with the cofactors $F_{x^j}, G_{0\,x^j}, \ldots, G_{m-1\,x^j}$ as its arguments. These are composed together using Shannon decomposition. By Equation 2.11, 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 [SKMB90] that the worst-case time complexity of the CASE algorithm is $O(p_{max} \cdot \mid F \mid \cdot \mid G_0 \mid \ldots \mid G_{m-1} \mid)$.

2.6 Other Operators using CASE

All operators on discrete functions can be expressed in terms of the CASE operator only.

```
function CASE($F, G_0, \ldots, G_{m-1}$)
begin
        if terminal case, return result;
        if table has entry for $CASE(F, G_0, \ldots, G_{m-1})$, return table entry;
        let $x$ be the top-variable of $F, G_0, \ldots, G_{m-1}$
        let $p$ be the number of values $x$ takes
        for j = 0 to $(p-1)$
            $H_{x^j}$ = CASE($F_{x^j}, G_{0\,x^j}, \ldots, G_{m-1\,x^j}$);
        add $(x, H_{x^0}, \ldots, H_{x^{p-1}})$ to table;
        return $(x, H_{x^0}, \ldots, H_{x^{p-1}})$;
end;
```

Figure 2.3: Pseudo-code for the CASE algorithm.

2.6.1 If-then-else

The if-then-else construct can be realized simply by:

$$\textbf{if } \textit{condition} \textbf{ then } \textit{result}_1 \textbf{ else } \textit{result}_0 \;\Leftrightarrow\; \text{CASE}(\textit{condition},\ \textit{result}_0,\ \textit{result}_1) \quad (2.13)$$

2.6.2 Multi-valued Input, binary-valued Output Operators

$$F = G \;\Leftrightarrow\; \text{CASE}(F, (G=0), (G=1), \ldots, (G=m-1)) \quad (2.14)$$

$$F \neq G \;\Leftrightarrow\; \text{CASE}(F, (G\neq 0), (G\neq 1), \ldots, (G\neq m-1)) \quad (2.15)$$

$$F > G \;\Leftrightarrow\; \text{CASE}(F, 0, (G<1), \ldots, (G<m-1)) \quad (2.16)$$

$$F \geq G \;\Leftrightarrow\; \text{CASE}(F, (G=0), (G\leq 1), \ldots, (G\leq m-1)) \quad (2.17)$$

$$F > i \;\Leftrightarrow\; \text{CASE}(F, 0, 0, \ldots, 0, 1, 1, \ldots, 1) \quad (2.18)$$

with i zeros where i is some constant

Similarly, other operators such as $F < G + i, \ldots,$ can be expressed using CASE.

2.6.3 Binary-valued Input, binary-valued Output Operators

$$NOT(F) \Leftrightarrow \mathbf{CASE}(F, 1, 0) \tag{2.19}$$

$$AND(F, G) \Leftrightarrow \mathbf{CASE}(F, 0, G) \tag{2.20}$$

$$NAND(F, G) \Leftrightarrow \mathbf{CASE}(F, 1, NOT(G)) \tag{2.21}$$

$$OR(F, G) \Leftrightarrow \mathbf{CASE}(F, G, 1) \tag{2.22}$$

$$NOR(F, G) \Leftrightarrow \mathbf{CASE}(F, NOT(G), 0) \tag{2.23}$$

$$XOR(F, G) \Leftrightarrow \mathbf{CASE}(F, G, NOT(G)) \tag{2.24}$$

$$XNOR(F, G) \Leftrightarrow \mathbf{CASE}(F, NOT(G), G) \tag{2.25}$$

Similarly, other binary operators can be expressed using the CASE operator.

Chapter 3

Applications

3.1 General Paradigm

MDD can be used to solve a wide variety of problems, from graph coloring to routing, to scheduling, to verification of FSM's. Before describing each application in detail, we first discuss the general paradigm used for solving such problems.

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 FSM's. With such information, the problem can be mapped into a number of multi-valued variables and a set of constraints between these variables. In our current implementation, such a mapping is done manually and all the constraint relationships are put into an input file provided by the user. The constraint input format to the MDD package is described in Section 6.1. From the inherited graph structure, we also derive a good ordering of the multi-valued variables, which is discussed in Chapter 5.

The input constraint file is first scanned and an MDD is built for each constraint. To minimize memory usage for storing the intermediate MDD's, they are ANDed together as soon as they are created, as illustrated on the right of Figure 3.1. This is actually a tradeoff between speed and memory requirements. We decided to AND the MDD's together as soon as possible because it is observed that with moderate-sized benchmarks, the MDD approach always gives a solution in reasonable time unless it runs out of memory. This observation is generally true for other graph-based representation such as the BDD's.

The final MDD contains implicitly all solutions of the problem. Satisfiability can

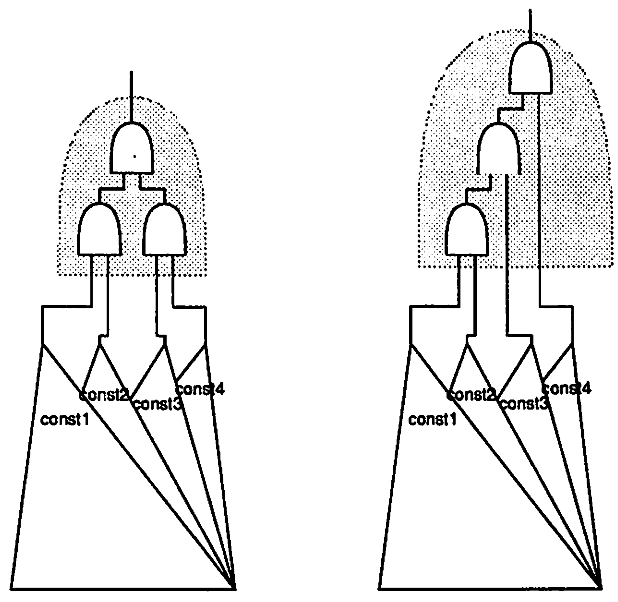


Figure 3.1: Two ways of ANDing together MDD constraints.

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. As pointed out in Chapter 1, we use MDD's to solve the decision problem, instead of its corresponding optimization problem. We've also investigated some possible ways to perform optimization which will be discussed in Section 3.6.

In this report, we focus on some restricted versions of the following CAD applications to illustrate the power of MDD's. Extensions to the general cases are fairly straightforward.

3.2 Hardware Resource Scheduling

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

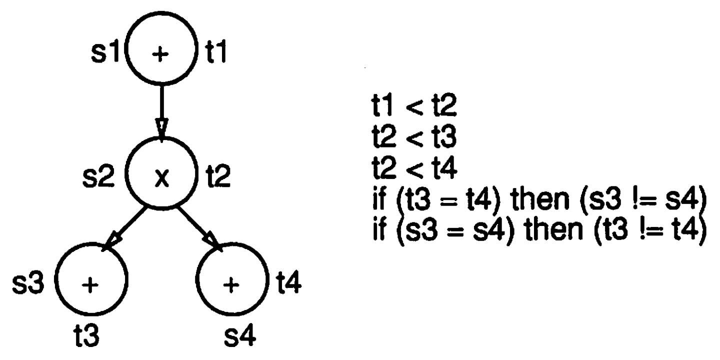


Figure 3.2: Hardware resource scheduling example.

functional units $\sigma_1, \ldots, \sigma_k$ and a table specifying the *single* machine type that each operation can be performed on, 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:

$$if\ (t_i = t_j)\ then\ (s_i \neq s_j) \tag{3.1}$$

$$if\ (s_i = s_j)\ then\ (t_i \neq t_j) \tag{3.2}$$

Note that conditions 3.1 and 3.2 are logical equivalent because of the following logical property:

$$if\ A\ then\ B \iff \overline{A}\ or\ B \tag{3.3}$$

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.

3.3 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) [YK82] 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:

$$y_i \neq y_j \tag{3.4}$$

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

$$y_i > y_j \tag{3.5}$$

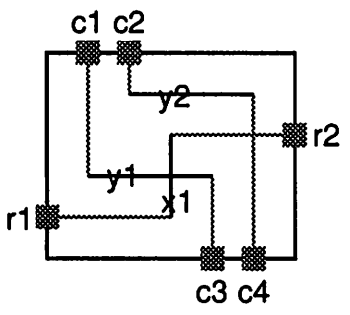
To determine if a route exists for the channel that uses t tracks, we let each variable y_i take on t values in the ordered set $\{0, \ldots, t-1\}$ and construct the MDD that is the conjunction of the above conditions (3.4) and (3.5). If the resulting MDD is not the terminal vertex with value 0, a solution 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 [Dev89].

3.4 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:

$$y_i \neq y_j \tag{3.6}$$

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 3.3.



```
y1 != y2
if (x1 >= c1) then (y1 != r1)
if (x1 <= c2) then (y1 != r2)
if (y1 >= r1) then (x1 != c3)
if (y1 <= r2) then (x1 != c1)
......
```

Figure 3.3: Switchbox routing example.

In Figure 3.3, 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 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.

3.5 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. y_i is allowed to assume k values. For each pair of adjacent nodes in G, generate the following constraint:

$$y_i \neq y_j \tag{3.7}$$

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.

3.6 Finding Alternate Solutions

We've investigated some possible ways to perform optimization. With binary search algorithms similar to the one in Figure 1.1, we can solve an optimization problem as a sequence of decision problems. This however involves the solution of a similar decision problem over and over again. For each iteration, we need to generate a new MDD. When a decision problem is satisfiable, the final MDD represents the set of all satisfying solutions. Hence, any optimum solution is guaranteed to be within this set. So instead of generating another MDD from scratch again, we can constrain the MDD available by cofactoring. Two examples for finding alternate solutions are given below. To find the absolute optimum solution, a binary search can still be used.

Example The color assignment on Figure 1.2 shows that the graph is colorable in 3 colors. Suppose we want to see if it is colorable in 2 colors only. The answer (to the satisfiability test) can be easily obtained if we cofactor the original MDD with the literals ${c_1}^P$, ${c_2}^P$, ${c_3}^P$ and ${c_4}^P$ where now $P = \{r, g\}$. The result of cofactoring gives the '0' vertex. Obviously in this case, this is not satisfiable because the graph is not colorable with only two colors.

Example In Figure 3.3, we see that the switchbox has been routed in 5 columns and 4 rows. Suppose in our chip layout, we only have room for a switchbox of 4 columns by 4 rows. This is a more constrained problem. Suppose that the heuristic router fails to come up with a solution. We want to first make sure that the switchbox *cannot* be routed in such an area before we are willing to modify the whole chip layout. Using the MDD approach, satisfiability (the answer to the above query) can be trivially check by cofactoring the original MDD with respect to the literals ${x_i}^T$ where x_i is constrained to the value set T = {0,1,2,3}.

Chapter 4

Mapping MDD's into BDD's

4.1 Introduction

So far, we have discussed the theory behind MDD's and some applications. The first-generation MDD package is a direct implementation of these concepts. Around the same time, the Berkeley BDD package became available, which incorporates the efficient BDD techniques published in [BRB90]. Instead of incorporating these techniques into the original MDD package, we decided to develop a new MDD package which makes use of the efficient BDD package. Later, we discover other advantages of using such a *mapped-BDD* approach instead of the direct MDD approach.

The implementation of MDD's using mapped-BDD's involves three distinct steps: *encoding*, *variable ordering* and *mapping*.

Definition 4.1 Encoding *is the process of associating a number of binary-valued variables to each multi-valued variable, and assigning codes to represent the values the multi-valued variables can take.*

An integer encoding is used on all multi-valued variables. For non-integer variables, the code points are assigned using the algorithm described in Section 4.2.

Definition 4.2 Variable Ordering *is the process of finding an ordering of the encoded binary-valued variables such that the size of the final BDD is minimized.*

Chapter 5 discusses two kinds of variable ordering techniques: *natural ordering* and *interleaved ordering*. The MDD package allows also the use of a mixture of both ordering techniques in a single mapped-BDD such that some variables can be specified to be interleaved while other variables just follow a natural ordering.

Definition 4.3 Mapping *is the process of constructing a BDD subgraph to represent an MDD. Mapping is performed after the encoding and variable ordering steps.*

This chapter describes two kinds of mapping techniques. *General mapping*, suitable for mapping any MDD into a BDD, is described in Section 4.3. Following from the direct MDD approach, the idea is to map each MDD vertex into a subgraph of BDD vertices in an optimal way. As we have seen in Section 2.5, the CASE operator forms the basis for the construction and manipulation of MDD's. The ITE operator is the binary analog of CASE and forms the basis for BDD manipulations. This *general mapping* can be conveniently performed by replacing each CASE operation by a set of ITE operations. This will be discussed in Section 4.3.3.

For MDD's representing arithmetic constraint relations mentioned in Chapter 3, we can use a *special mapping* which is described in Section 4.4. By taking advantage of the arithmetic properties of such functions, we can construct the mapped-BDD with much less ITE operations than by the *general mapping* method. However, other functions such as logical operations cannot take advantage of this *special mapping.*

Variable ordering and mapping are two separate processes. The choice of the kind of mapping and ordering used can be made independent of one another. Thus any combination of these techniques can be used. *Variable ordering* affects the size of the final mapped-BDD while *mapping* affects the speed of construction of the final mapped-BDD. Fortunately we can refine the mapping and ordering techniques independently.

In this chapter, we start by describing the encoding process. Then we describe how the MDD for any arbitrary function can be constructed. We then investigate the relationship between the CASE operator and the ITE operator. The way the CASE operator is converted to ITE operators dictates the encoding used, and are discussed in Section 4.3.4. Special mapping techniques for arithmetic/comparison operations are described in Section 4.4. Lastly, we discuss some methods to represent multi-valued output functions using mapped-BDD's in Section 4.5.

Although the internal representation of an MDD is in the form of a BDD, all the bookkeeping required between the MDD and the mapped-BDD is hidden from the users of the package. Users can input functions, manipulate them and output results in terms of multi-valued variables only, without worrying about the implicit variable encoding and ordering. The user-interface of the MDD package is described in Chapter 6.

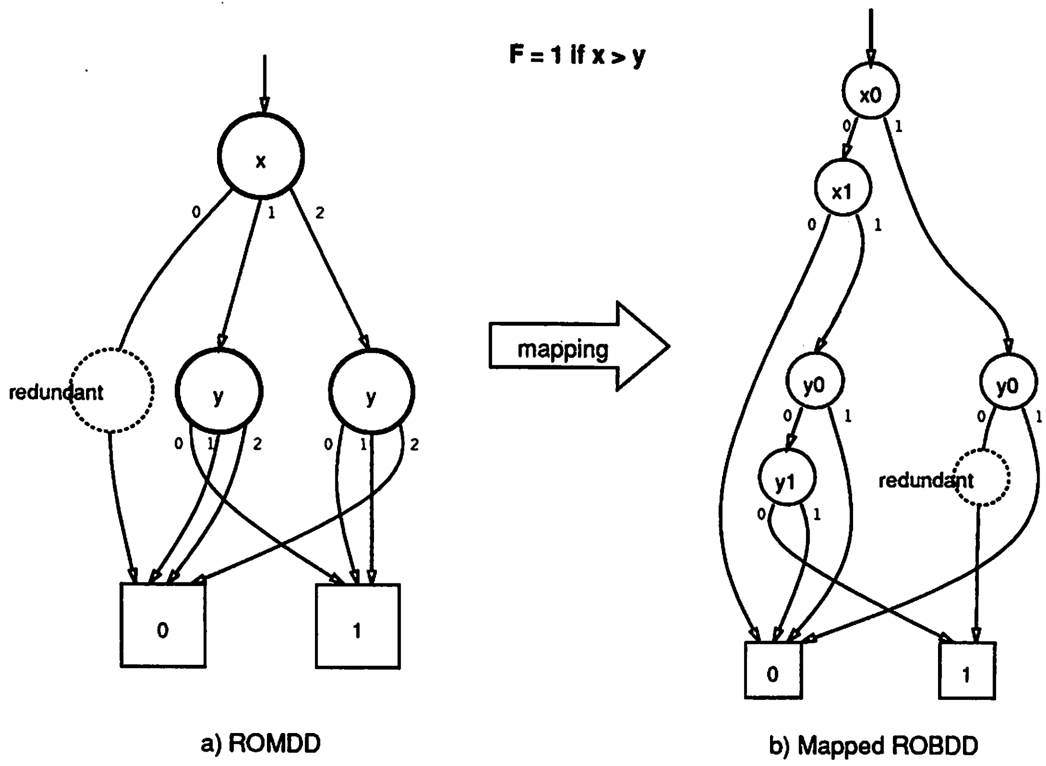


Figure 4.1: MDD and mapped-BDD representing the relation $x > y$.

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

$$F = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

x and y are 3-valued variables which can take values from $P_x = P_y = \{0, 1, 2\}$. To represent the MDD using BDD's, each MDD nonterminal vertex must be mapped into a number of BDD vertices interconnected in a subgraph. For example in Figure 4.1, 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. The issue of variable ordering is discussed in Chapter 5.

4.2 Encoding

Clearly, at least k unique binary variables are required to encode a m-valued MDD vertex where $2^{k-1} < m \leq 2^k$.

All variables are encoded using the minimum number of binary variables. In addition, many multi-valued variables usually take values from the set of integers (an ordered set) and the operations between them are sometimes integer-arithmetic in nature. Thus, integer encoding is used on these. This enables fast construction and compact representation of the mapped-BDD, as shown in Section 4.4 and 5.2.2.

Note that many encodings use the same number of variables. But different ones may result in smaller or larger BDD's. Thus different multi-valued variables because of the way they are used in the functions to be represent should have different encoding requirements. However, techniques for encoding each variable separately have not been thoroughly investigated yet.

4.2.1 Use of Don't Cares

Of course, not all 2^k code points will be used since typically $m < 2^k$. On the other hand the mapped-BDD will have a path for such binary code point. Which path is assigned to an unused code point is only relevant to the size of the BDD. Thus these don't care should be used to decrease the BDD sizes, similar to their use in the *constraint* operation [CBM89].

Example 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 4.1. The last column is used in the example in Section 4.3.3.

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 don't cares since the values will never occur. In this case these don't care are mapped into the same nodes as 100 and 101 respectively. The notation

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 4.1: Integer encoding with don't cares.

$1 * 0$ is used to represent both encodings 100 and 110 as we don't care about the variable u_1.

4.3 General Mapping

4.3.1 Constructing Functions using CASE Operator

Before going into the details of the mapping process as shown in Figure 4.1, we first discuss how to obtain an MDD for any arbitrary function of two multi-valued variables. Although we are not building the MDD structure directly, this extra level of abstraction (MDD) will help us construct the function via the CASE operator indirectly. As seen from Equation 2.9, we can view each MDD vertex as the result of a CASE operator with the variable associated with the vertex being its first argument while its child functions being the remaining arguments. As long as we know how to construct an MDD for any arbitrary function, its mapped-BDD can be derived relatively easily by converting each of these CASE operators into a number of ITE operators as detailed in Section 4.3.3.

The MDD for any arbitrary function of two multi-valued variables, such as the one in Figure 2.1, can be constructed using a table lookup approach. These lookup tables for multi-valued functions are analogous to the truth tables for completely-specified Boolean functions. For simplicity in implementation, we need not worry about building a reduced MDD because the MDD package will take care of that automatically by returning an ROMDD after each MDD manipulation. As shown in Figure 2.1, the full unreduced MDD consists of a fixed subgraph of MDD nonterminal vertices which are connected at the bottom to an array of terminal vertices. The interconnections between the terminal vertices are fixed and represent a Shannon decomposition on the two multi-valued variables involved, in this case x and y. The connections between the terminal and nonterminal vertices can be stored as a lookup table for each function we need. So to realize a function, we first lookup for the nonterminal-to-terminal connections at the bottom, then connect them together with MDD vertices by calling the routine *mdd_case*, which will be described in Section 4.3.4. This builds up a subgraph of nonterminal vertices recursively in a bottom-up fashion.

Example The function in Figure 4.1, $x > y$, is used as another example. We are expressing the following multi-valued operators, described in Section 2.6.2, using the CASE operator as follows:

$$x > y \Leftrightarrow CASE(x, (y < 0), (y < 1), \ldots, (y < m - 1)) \quad (4.2)$$

$$y < i \Leftrightarrow CASE(y, 1, 1, \ldots, 1, 0, 0, \ldots, 0) \quad (4.3)$$

$$\textit{with } i \textit{ ones where } i \textit{ is some constant}$$

Equation 4.2 corresponds to the top MDD vertex labeled with variable x in Figure 4.1a, while Equation 4.3 corresponds to the set of MDD vertices labeled y. The lookup table for this function is shown in Table 4.2. This function is implemented in the MDD package as the function call, *mdd_lt_g*. It's C code is outlined in Figure 4.2. Note that the lookup table is implicitly embedded into the code. Also note that the first MDD node that is built from this example is the one for $y < 0$. This case is taken from the first 3 rows of Table 4.2. This produces a redundant node which is reduced to the 0 terminal node. The next node built is the one from the case $y < 1$, then $y < 2$, using the second three and last three rows of Table 4.2 respectively. Finally, these three are taken as the children of the x node which is constructed by calling $CASE(x, (y < 0), (y < 1), (y < 2))$.

The MDD package uses the Berkeley dynamic array package whose function calls have prefix *array_*. Since the C codes listed in this report calls the *array* package, part of its documentation is included in Appendix A. The function call interface to the MDD package is described in Section 6.2. The role of the *mdd_manager*, and the use of the list of multi-valued and binary-valued variables, *mvar_list* and *bvar_list*, are discussed there also. *mdd_lt_g* takes, as its parameter, an *mdd_manager* and two multi-valued variables, x and y, and returns the MDD (in the form of a mapped-BDD) which represents the relation $x < y$.

x value	y value	terminal vertex
0	0	0
0	1	0
0	2	0
1	0	0
1	1	0
1	2	1
2	0	0
2	1	1
2	2	1

Table 4.2: Lookup table for $x > y$.

The nested *for* loops enumerate the proper connections to the nonterminal vertices. They are packed into arrays of child functions, *child_list_y* (and then *child_list_x*), which together with the top-variable y (x), are passed as parameters to the routine *mdd_case*. *mdd_case* maps the CASE operator into ITE operators so that the MDD is ultimately created out of BDD nodes. This is discussed in the Section 4.3.3. Note that *mdd_case* is performed in a bottom-up fashion, first with the top-variable y and then with x.

4.3.2 Properties of General Mapping

Definition 4.4 *A general mapping is* **valid** *if each mapped-BDD subgraph has the same number of child-edges as the corresponding MDD vertex, and the mapped-BDD follows an encoding and an ordering consistently.*

Lemma 4.1 *Any connected BDD subgraph of* m-1 *BDD nonterminal vertices is a* valid mapping *for an* m*-valued MDD vertex if an ordering is followed.*

Proof The proof is by induction on m. For [1] m=*2*, a binary-valued MDD nonterminal vertex can be represented as a single BDD nonterminal vertex. The trivial encoding for the two values of the MDD vertex is 0 and 1.

[1]Lemma 4.1 also holds for the case m=*1*; i.e. no BDD vertex is needed to represent a single-valued MDD vertex. Actually in the encoding algorithm described in Section 4.3.4, such an MDD vertex is redundant and will be eliminated and hence not encoded.

```
mdd_t *
mdd_lt_g(mgr, mvar1, mvar2)
mdd_manager *mgr;
int mvar1, mvar2;
{
    mvar_type x, y;
    array_t *child_list_x, *child_list_y;
    int i, j;
    mdd_t *tx1, *ty1;
    array_t *mvar_list;

    mvar_list = ((mdd_hook_type *)mgr->hooks.mdd)->mvar_list;
    x = array_fetch(mvar_type, mvar_list, mvar1);
    y = array_fetch(mvar_type, mvar_list, mvar2);
    child_list_x = array_alloc(mdd_t *, 0);
    for (i=0; i<x.values; i++) {
        child_list_y = array_alloc(mdd_t *, 0);
        for (j=0; j<y.values; j++) {
            if (i < j) array_insert_last(mdd_t *, child_list_y, one);
            else array_insert_last(mdd_t *, child_list_y, zero);
        }
        ty1 = mdd_case(mgr, mvar2, child_list_y);
        array_insert_last(mdd_t *, child_list_x, ty1);
        array_free(child_list_y);
    }
    tx1 = mdd_case(mgr, mvar1, child_list_x);
    array_free(child_list_x);
    return tx1;
}
```

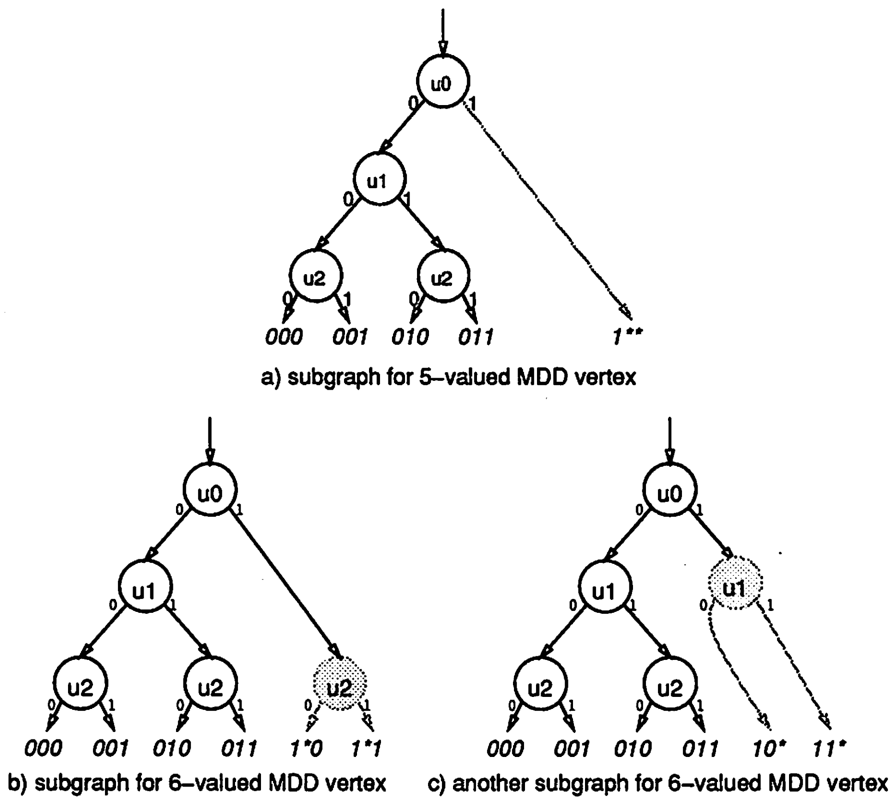
Figure 4.2: Simplified C code for *mdd_lt_g*.

Assume that a k-valued MDD nonterminal vertex can be mapped into a BDD subgraph T with *k-1* BDD nonterminal vertices. A valid mapping of *k+1*-valued MDD vertex, by Definition 4.4, requires a BDD subgraph T' which has *k+1* child-edges. T' can be obtained by connecting an extra BDD nonterminal vertex v under any leaf vertex u of T. As each BDD nonterminal vertex has one parent-edge and two child-edges, the overall number of child-edges out of T' is exactly *k+1*, one more than that of T. To respect the variable ordering, $index(v) < index(u)$. The encoding for T' is inherited from T except for the child-edges of the new BDD vertex v. The codes assigned to them can be found by tracing their paths from the root, as illustrated in the example below. □

Example The mapped-BDD subgraph for a 5-valued MDD vertex is shown in Figure 4.3a. The code assigned to a child-edge depends on its decision path from the root. e.g. the fourth child-edge is reached from the root via the path $u_0 = 0$, $u_1 = 1$, $u_2 = 1$, thus the code $u_0u_1u_2 = 011$ is assigned. A BDD vertex can be added to this subgraph to represent a 6-valued MDD vertex. Depending on where the vertex is added and what index is assigned to the vertex, different encodings will result. A BDD vertex is added under vertex u_0 in Figure 4.3b and 4.3c. The vertex is assigned an index of 3 in Figure 4.3b and an index of 2 in Figure 4.3c, so they result in slightly different encodings.

Lemma 4.1 and its proof indicate that there are many ways to map an MDD vertex into a BDD subgraph. One can construct an arbitrarily connected subgraph of BDD vertices, and assign indices to vertices such that $index(v) < index(child_k(v))$ for any nonterminal vertex v where $k \in P_{index(v)}$ and $child_k(v)$ is nonterminal. This mapping procedure results in a consistent variable ordering. Given that we use the minimum number of encoded variables, we know the mapping is as good as any other as it already uses the minimum number of BDD vertices as well as variables.

As mentioned in Section 4.2.1, the decision path to each don't care code point is chosen so as to minimize the BDD size. In fact, a don't care point is assigned to the same path as the care point whose encoding is closest to the don't care code point. In Figure 4.3b, the don't care point 110 shares the same path with the care point 100. This mapping is related to the *generalized cofactor* operator in [TSL+90] which was initially proposed in [CBM89] 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 point $y \in c$ which has the closest distance to x. *Generalized cofactor*



u0
u1
u2
0
1
000 001 010 011 1**
a) subgraph for 5–valued MDD vertex
000 001 010 011 1*0 1*1
b) subgraph for 6–valued MDD vertex
000 001 010 011 10* 11*
c) another subgraph for 6–valued MDD vertex

Figure 4.3: Encodings u_0-u_1u_2 for different mapped-BDD subgraphs.

operation results in a small and canonical BDD representation of the incompletely specified function.

4.3.3 Relationships between CASE and ITE Operators

The ITE operator forms the basis for the construction and manipulation of BDD's. The use of the ITE operator also guarantees that the resulting BDD is in strong canonical form [BRB90]. It is defined as follows:

$$ITE(F, G_1, G_0) = \begin{cases} G_1 & \text{if } F = 1 \\ G_0 & \text{otherwise} \end{cases} \tag{4.4}$$

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

The CASE operator is the multi-valued analog of the ITE operator. An understanding of the relationships between the CASE and ITE operators is important for converting the CASE operator into ITE operators for the second-generation MDD package.

The recursion step in Equation 2.12 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 formulae:

$$\begin{aligned} \text{if p is even: } & CASE(v, \underbrace{G'_0, G'_1}, \underbrace{G'_2, G'_3}, \ldots, \underbrace{G'_{p-2}, G'_{p-1}}) \\ = \ & CASE(v', ITE(u, G'_1, G'_0), \ldots, ITE(u, G'_{p-1}, G'_{p-2})) && (4.5) \\ \text{if p is odd: } & CASE(v, \underbrace{G'_0, G'_1}, \underbrace{G'_2, G'_3}, \ldots, \underbrace{G'_{p-3}, G'_{p-2}}, G'_{p-1}) \\ = \ & CASE(v', ITE(u, G'_1, G'_0), \ldots, ITE(u, G'_{p-2}, G'_{p-3}), G'_{p-1}) && (4.6) \end{aligned}$$

The recursive use of Equations 4.5 or 4.6 continues until this recursion terminates when there are only two child-functions remaining in the outer CASE operator:

$$CASE(v, G'_0, G'_1) = ITE(v, G'_1, G'_0) \tag{4.7}$$

While pairing up child-functions with the ITE operator, these formulae 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 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:

$$\begin{aligned}
& CASE(v, \underbrace{G'_0, G'_1}, \underbrace{G'_2, G'_3}, \underbrace{G'_4, G'_5}) && (4.8)\\
= \; & CASE(v', \underbrace{ITE(u_2, G'_1, G'_0), ITE(u_2, G'_3, G'_2)}, ITE(u_2, G'_5, G'_4)) && (4.9)\\
= \; & CASE(v'', \underbrace{ITE(u_1, ITE(u_2, G'_3, G'_2), ITE(u_2, G'_1, G'_0)), ITE(u_2, G'_5, G'_4)}) && (4.10)\\
= \; & ITE(u_0, ITE(u_2, G'_5, G'_4), ITE(u_1, ITE(u_2, G'_3, G'_2), ITE(u_2, G'_1, G'_0))) && (4.11)
\end{aligned}$$

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 4.4 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.

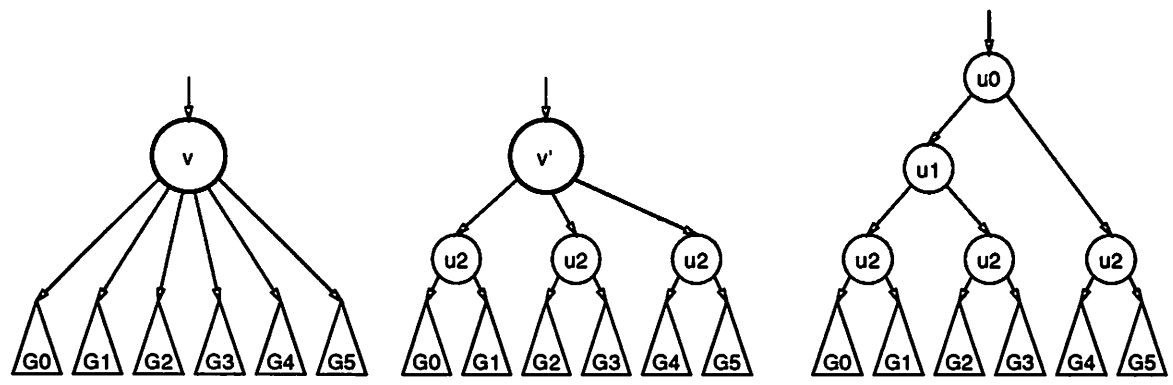


Figure 4.4: Recursive mapping from an MDD vertex to a mapped-BDD subgraph.

4.3.4 'mdd_case' Algorithm

Figure 4.5 outlines the C code for the function call *mdd_case*. It takes as parameter an *mdd_manager*, a multi-valued top-variable, and a list of child functions *child_list* in the form of MDD's (actually mapped-BDD). The actual construction of the mapped-BDD is performed by the efficient recursive routine *mdd_encode*.

Figure 4.6 outlines the C code for *mdd_encode*. When invoked the first time, *mdd_encode* takes as parameters an *mdd_manager*, an array of existing child-functions *child_list*, a pointer ($highest\_vertex$) to the current BDD variable in the *bvar_list*, and a parameter *stride*. The *bvar_list* is an ordered list of BDD variables stored with the *mdd_manager*. The parameter *stride* will be discussed in Section 5.2.2, but for the time being, we may assume it to be equal to 1. The terminal case of recursion is when the whole mapped-BDD has been built and *child_list* contains only a single child-function; then the original function is returned. *mdd_encode* pairs up child-functions and performs ITE's with the current variable f given by the variable pointed to by *highest_vertex*. Resulting ITE functions are appended to a *new_child_list*. The number of ITE operations can be calculated by modulo-dividing the size of *child_list* by 2. If the size is odd, the remaining function is appended also. Lastly, the *mdd_encode* is called recursively bottom-up, with the *new_child_list* and the decremented pointer in the ordering. The variable which is one earlier in the ordering than the current one, i.e. with next lowest index, will be used in the next recursion step.

As mentioned before, there are many valid mappings from MDD's into BDD's. A particular mapping can be characterized by answering the following two questions:

- which child-functions are grouped together,
- which variable (or index) is assigned to the BDD vertex thus formed.

The general mapping process described so far is the default one used in the project. It can be compactly defined by recursive Equations 4.5 and 4.6. In summary:

- child-functions are grouped consecutively in pairs,
- as many pairs as possible are assigned to a variable in each recursion step.

The BDD subgraph in Figure 4.4 corresponds to the following encoding as shown in Table 4.1. This corresponds to a natural integer encoding.

Lemma 4.2 *The mapping algorithm,* mdd_encode, *uses the minimum number of BDD vertices and variables. The mapped-BDD obtained by the algorithm is in strong canonical form.*

```
mdd_t *
mdd_case(mgr, mvar, child_list)
mdd_manager *mgr;
int mvar;
array_t *child_list;
{
    mvar_type mv;
    mdd_t *mnode;
    array_t *mvar_list;

    mvar_list = ((mdd_hook_type *)mgr->hooks.mdd)->mvar_list;
    mv = array_fetch(mvar_type, mvar_list, mvar);
    if (mv.values != array_n(child_list))
        fail("mdd_case: mvar.values different from child_list\n");
    mnode = mdd_encode(mgr,
                       child_list,
                       mv.start_vertex + mv.stride*(no_bit_encode(mv.values)-1),
                       mv.stride);
    return mnode;
}
```

Figure 4.5: Simplified C code for *mdd_case.*

```
mdd_t *
mdd_encode(mgr, child_list, highest_vertex, stride)
mdd_manager *mgr;
array_t *child_list;
int highest_vertex, stride;
{
    array_t *new_child_list;
    int i, child_count = 0;
    mdd_t *f, *g, *h, *t;
    array_t *bvar_list;

    bvar_list = ((mdd_hook_type *)mgr->hooks.mdd)->bvar_list;
    if (array_n(child_list) == 1)
        return array_fetch(mdd_t *, child_list, 0);
    new_child_list = array_alloc(mdd_t *, 0);
    for (i=0; i<(array_n(child_list)/2); i++) {
        f = array_fetch(mdd_t *, bvar_list,  highest_vertex);
        h = array_fetch(mdd_t *, child_list, child_count++);
        g = array_fetch(mdd_t *, child_list, child_count++);
        /* if trivial cases, return mdd */
        t = mdd_ite(f, g, h, 1, 1, 1);
        array_insert_last(mdd_t *, new_child_list, t);
    }
    if (array_n(child_list) % 2) {
        t = array_fetch(mdd_t *, child_list, child_count);
        array_insert_last(mdd_t *, new_child_list, t);
    }
    f =  mdd_encode(mgr, new_child_list, highest_vertex-stride, stride);
    array_free(new_child_list);
    return f;
}
```

Figure 4.6: Simplified C code for *mdd_encode*.

Proof The first statement follows directly from the encoding process described in Section 4.2. In the package, the terminal vertices are generated only once by *bdd_one* and *bdd_zero*. To build the BDD, only the function *bdd_ite* from the BDD package is called. When processing constraints, the *bdd_and* and *bdd_or* are also used. As described in [BRB90], BDD's built with these primitive operations are in strong canonical form. □

Example In Figure 4.4, a minimum of 5 BDD vertices and 3 BDD variables are used to encode a 6-valued MDD vertex. The mapped-BDD on the right of Figure 4.4 is a canonical ROBDD.

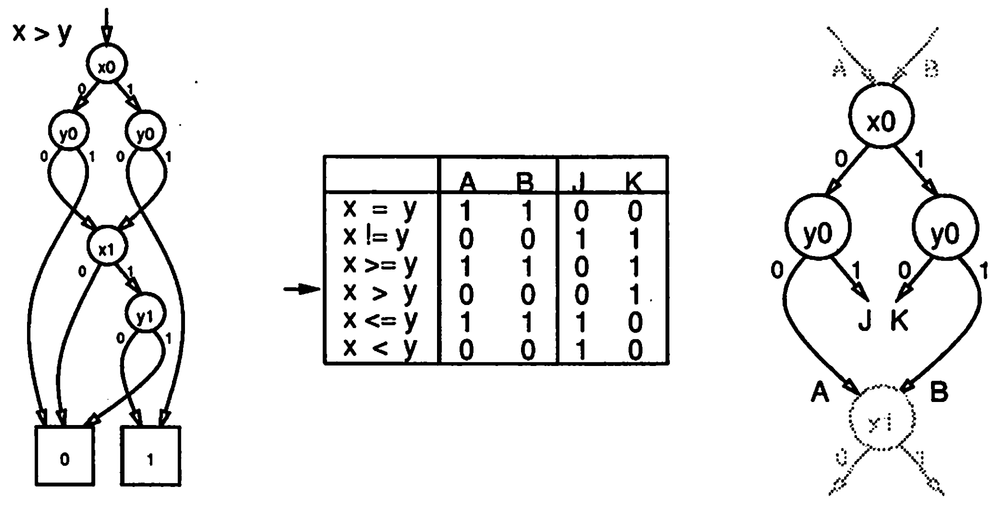
4.4 Special Mapping for Arithmetic Operations

The general mapping is a useful technique as it works for any arbitrary multi-valued function. But for a certain class of arithmetic/comparison functions, the corresponding MDD's can be constructed much faster if a special mapping is used. For most operators described in Section 2.6.2, we can take advantage of their arithmetic properties to devise a smaller set of ITE operations to realize such functions directly.

The idea of using special mapping for different functions can be extended further: Given a particular function and an encoding, we could represent the function as an multi-level Boolean network. Using a multi-level logic minimization program such as MIS [BRSVW87], the number of nodes in the network can be minimized. A special mapping can be carried out as suggested by the topology of the minimized network and this results in a minimal set of ITE operations to realize the function.

Example Figure 4.7a shows a mapped-BDD for the relation $x < y$ where x and y can each take four values. Again we could view each BDD vertex since an ITE operation whose arguments are its top-variable and its two child functions. With the special mapping, we shall not map the CASE operator to ITE operators. Instead, we construct a mapped-BDD for each type of constraint function directly using the ITE operator.

The special mapping is done as follows. To compare the values of two multi-valued variables, we start from the top of the MDD by comparing the most-significant-bit (MSB) encoded variables, x_0 and y_0. If $x_0 = 0$ and $y_0 = 1$, we know $x < y$ and can conclude that the function must evaluate to a 0 without looking at lesser significant bits. Similarly, if $x_0 = 1$ and $y_0 = 0$, we can conclude the result 1 since x is larger than y no matter what the lesser significant bits are. But if $x_0 = y_0$, we have to resort to comparisons of the next significant bits, x_1 and y_1. As shown in Figure 4.7, each pair of encoded variables corresponds to a 3-vertex-cluster.



	A	B	J	K
x = y	1	1	0	0
x != y	0	0	1	1
x >= y	1	1	0	1
x > y	0	0	0	1
x <= y	1	1	1	0
x < y	0	0	1	0

a) Special Mapping Example

b) Special Mapping Techniques

Figure 4.7: Construction of mapped-BDD using special mapping.

The interconnections between these clusters, and the terminal vertices can be summarized by the table in the middle of Figure 4.7. We have four outgoing edges for each 3-vertex-cluster which are labeled A, B, J and K. Edges A and B are taken if the values of x_0 and y_0 are the same. No conclusion can be drawn about the overall comparison so these edges point to the next significant bit x_1. The exception is when A and B are coming from the least-significant-bit (LSB) vertices and are pointing to terminal vertices. For this case, the values of all encoded variables have already been checked. So the second and third column of the table list the terminal vertices they should point to for each type of constraint function. If the values of x_0 and y_0 are different, we can jump to a conclusion directly, i.e. J and K can point directly to terminal vertices, which are listed in the fourth and fifth column of the table. We have implemented six commonly used constraint functions using this special mapping technique. These are listed in Section 6.2.1. Figure 4.8 outlines one of them, *mdd_lt_s*.

In general, to realize a comparison function of two multi-valued variables, each having p values, the general mapping requires $(p+1)(p-1)$ ITE operations, while the special mapping only needs $3(log_2(p))$ ITE operations. For example in Figure 4.7a, we need only 6 ITE operations to realize the function using the special mapping, instead of 15 using the general mapping. The arithmetic property of the comparison function helps us effectively 'fold' the MDD into a smaller number of ITE operations.

For the benchmark examples in Chapter 7, a 10% speed improvement is seen by using the special mapping as compared to the general mapping. The improvement is expected to be greater for larger problems with larger value ranges. We experimented with some simple functions with a few constraints whose variables have a range of over one hundred values. The final mapped-BDD can be constructed within a few seconds using the special mapping. But if the general mapping is used, the construction of the partial BDD is very slow and eventually the construction aborts due to memory constraints.

4.5 Representing Multi-valued Output Functions

We have been using discrete functions $\mathcal{F}$ which have a binary-valued range $Y = B = \{0,1\}$ since the beginning of Chapter 3. For a moment, consider the more general multi-valued function where the output can assume m values, $Y = \{0, 1, \ldots, m-1\}$. When mapping into a BDD, there are just two kinds of terminal vertices, 0 and 1. Therefore, we need to transform the function first into binary-valued output function(s).

Three ways to represent multi-valued outputs are shown in Figure 4.9. In this example, the discrete function $\mathcal{F}$ can output 4 values. The *one hot* method builds a separate BDD to represent each possible output value of the function. Only one of the binary-valued functions G_1, G_2, G_3, G_4 will be 1 at any time. Note that, as illustrated, sharing of common subgraphs between the BDD's is used whenever allowed.

The second and third method use a *characteristic function* G to represent the multi-valued function $\mathcal{F}$, which is defined as:

$$G : P_1 * P_2 * \ldots * P_n * Y \rightarrow B \tag{4.12}$$

```
mdd_t *
mdd_lt_s(mgr, mvar1, mvar2)
mdd_manager *mgr;
int mvar1, mvar2;
{
    mvar_type x, y;
    bvar_type bx, by;
    mdd_t *t, *tt, *ttb;
    int i;
    array_t *mvar_list;
    array_t *bvar_list;

    mvar_list = ((mdd_hook_type *)mgr->hooks.mdd)->mvar_list;
    bvar_list = ((mdd_hook_type *)mgr->hooks.mdd)->bvar_list;
    x = array_fetch(mvar_type, mvar_list, mvar1);
    y = array_fetch(mvar_type, mvar_list, mvar2);
    if (x.values != y.values)
    fail("mdd_lt_s: 2 mvars have incompatible value ranges.");

    t = zero;
    for (i=(x.encode_length-1); i>=0; i--) {
        bx = array_fetch(bvar_type, bvar_list, x.start_vertex + i*x.stride);
        by = array_fetch(bvar_type, bvar_list, y.start_vertex + i*y.stride);
        ttb = mdd_ite(by.node, one, t, 1, 1, 1);
        tt  = mdd_ite(by.node, t, zero, 1, 1, 1);
        t = mdd_ite(bx.node, tt, ttb, 1, 1, 1);
    }
    return t;
}
```

Figure 4.8: Simplified C code for *mdd_lt_s*.

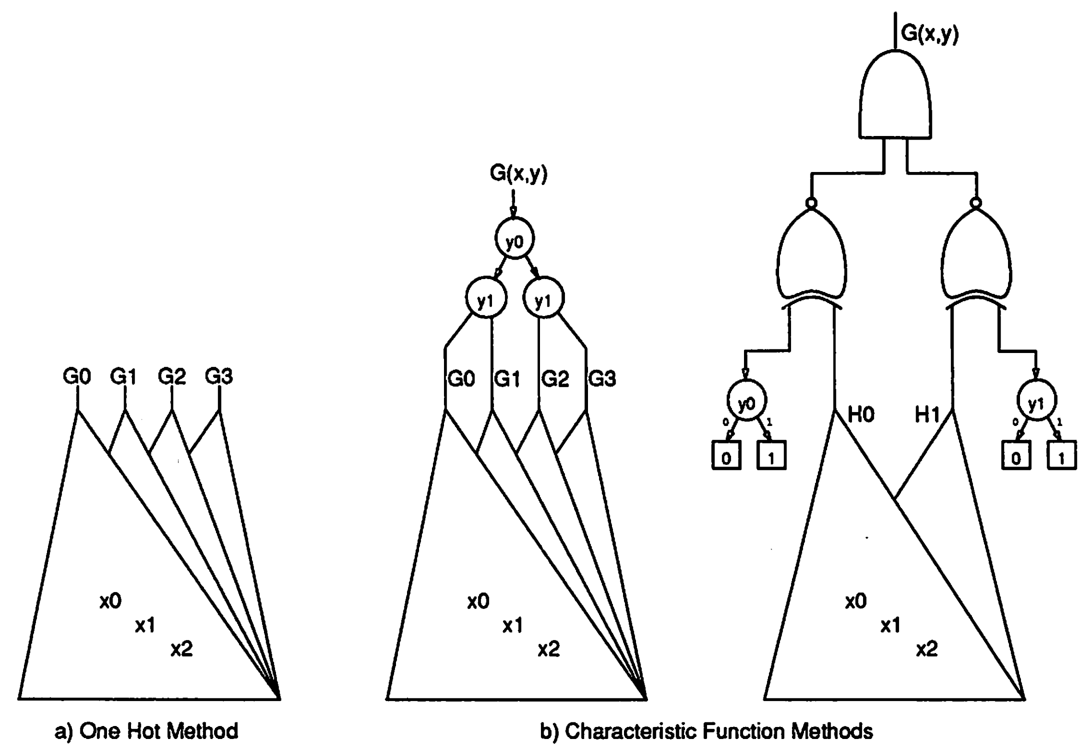


Figure 4.9: Three ways to represent multi-valued output functions.

$$G(x,y) = \begin{cases} 1 & \text{if } \mathcal{F}(x) = y \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$
$$\textit{where } x \in P_1 * P_2 * \ldots * P_n, \textit{ and } y \in Y$$

Effectively, this treats the multi-valued output as another multi-valued input. There are two ways to represent this characteristic function. First we could construct the one-hot binary functions G_1, G_2, G_3, G_4 as before. Then we can use the CASE operator to select one of them according to the value to y,

$$\begin{aligned} G(x,y) &= (F(x) = i)\ if\ (y = i) && (4.14) \\ &= CASE(y,\ (F(x) = 0),\ (F(x) = 1), \ldots,\ (F(x) = m - 1)) && (4.15) \end{aligned}$$

This is shown in the middle diagram in Figure 4.9. Of course, the ordering $y_j < x_i$ need not be used.

Alternatively, we could construct the encoded binary-valued functions H_0 and H_1:

$$H_0(x,y) = \begin{cases} 0 & \text{if } \mathcal{F}(x) = 0 \ or\ 2 \\ 1 & \text{if } \mathcal{F}(x) = 1 \ or\ 3 \end{cases} \tag{4.16}$$

$$H_1(x,y) = \begin{cases} 0 & \text{if } \mathcal{F}(x) = 0 \ or\ 1 \\ 1 & \text{if } \mathcal{F}(x) = 2 \ or\ 3 \end{cases} \tag{4.17}$$

In this case we construct $log_2(m)$ H_i BDD's instead of m G_i BDD's for the previous case. Each H_i has to be compared with the corresponding encoded variables y_i of y.

$$G(x,y) = \begin{cases} 1 & \text{if } (H_0 = y_0) \text{ AND } (H_1 = y_1) \\ 0 & \text{otherwise} \end{cases} \tag{4.18}$$

As shown on the right diagram of Figure 4.9, $G(x,y)$ is realized by XNORing H_0 and y_0 together, and XNORing H_1 and y_1 together. Then these resultant BDD's are ANDed together to realize the characteristic function $G(x,y)$.

Chapter 5

Variable Ordering

We can frequently suffer from exponential time (and/or space) complexities if we neglect the issue of variable ordering. As with other graph-based techniques such as BDD's, the worst case space and time complexities for constructing an MDD for any discrete function is exponential to the number of variable values of the function. Bryant proved that to represent integer multiplication with BDD's, the BDD sizes grow exponentially in the word size regardless of the ordering of the input variables. Luckily in real life, most discrete functions have reasonable representations provided that a good variable ordering is chosen.

The goal in this chapter is to find a good variable ordering so as to minimize the size of the final MDD. In our mapped-BDD implementation, we have the additional freedom of ordering the encoded binary variables. The ordering process thus consists of two steps: first we order the MDD variables, and then within this order the BDD variables. We present one heuristic for ordering the MDD variables, and two other heuristics for ordering the BDD variables.

5.1 Ordering of MDD Variables

From the graph structure inherent in the application (see Chapter 3), we can derive the following ordering heuristic that works well for the applications we tried.

- Form a digraph from the problem. Each vertex corresponds to one or more multi-valued variables for the MDD. A directed edge is drawn between vertex i to vertex j if variable i is directly related to variable j by a constraint.

- Break the cycles of the digraph arbitrarily to form a DAG.
- Levelize the DAG using topological ordering.
- Assign indices such that the variables at the highest level of the graph have lower indices (i.e., are at the top) in the MDD.
- For vertices on the same level, assign lower indices to vertices that are on longer directed paths in the digraph.
- Further ties are broken arbitrarily.

The idea behind this heuristic is that variables on the top level of the digraph have more freedom and hence will participate in more solutions. By placing them on the top levels of the MDD, more compact MDD's should result.

Example Figure 5.1 is a partial flow graph from Figure 3.2. Following the ordering heuristics outlined above, we obtain the ordering $t1 \prec t2 \prec t3 \prec t4$ or $t1 \prec t2 \prec t4 \prec t3$ which give the minimum MDD size of 8 vertices. Note that the graph has a dual-graph whose edges are related by $\leq$ instead of $>$, pointing in the opposite direction. Applying our heuristics, the ordering for the dual-graph is $t4 \prec t3 \prec t2 \prec t1$ or $t3 \prec t4 \prec t2 \prec t1$ and both of them also minimize the final MDD size. Other orderings generate MDD's of larger sizes. For example, 11 vertices are needed for the ordering $t3 \prec t4 \prec t1 \prec t2$. The adverse effect of a bad ordering on the MDD size is more serious when the multi-valued variables can take large value ranges. As shown in Figure 5.1, the MDD obtained following our heuristic is 1/4 the size of that formed with an arbitrary ordering. We note in comparing Figure 5.1 with Figure 3.2 that the $if-then$ constraints are not involved in the ordering since they are not topologically structured and cannot be easily taken into consideration when ordering the variables.

Nevertheless, the ordering heuristic for MDD variables has proved to be very useful for the hardware resource scheduling problems. We tried some large real-life digital filter examples (results are given in Section 7.1). Surprisingly, they can be solved within one minute if the ordering heuristic is followed. The variable ordering chosen by our heuristic for the 11th-order FIR filter is shown in Figure 5.2.

$$\begin{tikzcd} & t1 \arrow[d, ">"] & \\ & t2 \arrow[dl, ">"'] \arrow[dr, ">"] & \\ t3 & & t4 \end{tikzcd}$$

ordering	t1 t2 t3 t4	t4 t3 t2 t1	t3 t4 t1 t2	t1 t2 t3 t4	t3 t4 t1 t2
range	2 3 2 2	2 3 2 2	2 2 2 3	20 30 20 20	20 20 20 30
no. of vertices	8	8	11	62	254

Figure 5.1: Effects of variable ordering on MDD size.

5.2 Ordering of Mapped-BDD Variables

For the second- and third-generation MDD package, MDD's are represented as mapped-BDD's. A limitation of the current Berkeley BDD package is that once a binary variable is registered, it's position in the order list is fixed. Variables can only be appended to the end of the order list. Thus the variable ordering process is performed before the mapping process. The lifting of such a limitation will open up opportunities for new ordering heuristics. For example, it would enable us to change the order list dynamically as the function is being constructed with the variables. Also we could experiment on the effects of alternate variable ordering after a final MDD has been constructed.

In this section, we first present two ordering heuristics which represents two ends of the spectrum of variable ordering possibilities. The natural ordering discussed in Section 5.2.1 works well for problems which involve many multi-valued variables, each having small value ranges. The interleaved ordering discussed in Section 5.2.2 performs well with problems which have only a few variables but each having large value ranges. In Section 5.2.3, a mixed approach of these heuristics is introduced and its implementation is discussed.

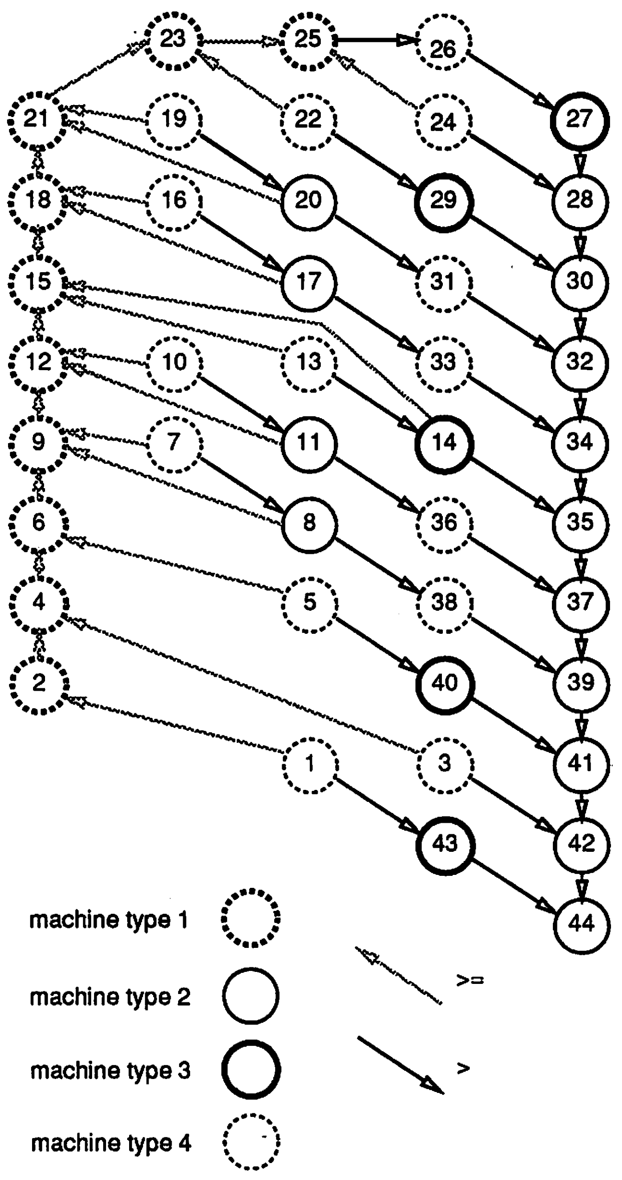


Figure 5.2: Flow graph of 11th order FIR filter and derived ordering.

5.2.1 Natural Ordering

After encoding the MDD variables with binary variables, we can order these encoded variables as well. A natural way is to cluster together all the encoded variables which correspond with the same multi-valued variable of the MDD. Within each cluster, the binary variable which corresponds to the most significant bit (MSB) is assigned the least index. Higher indices are assigned to variables which correspond to lesser significant bits. A BDD subgraph is constructed for each MDD vertex according the method described in Section 4.3. This natural ordering is used in all the MDD examples reported in Chapter 7. The following example illustrates the natural ordering.

Example In Figure 5.3, the function $F = 1 \ if \ x > y$ is represented as an MDD on the left and a mapped-BDD on the right. Variables x and y can each take 4 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 shaded subgraph before reduction has the same number of outgoing arcs as the MDD vertex and thus the two representations are equivalent. With the mapped-BDD representation, additional redundant nodes can be reduced automatically as illustrated by the y_1 vertices. Note that the number of edges exiting from the y vertices on the right is significantly less than those on the left. The actual encoding used for the y variables affects this.

5.2.2 Interleaved Ordering

Natural ordering works well for the applications we tried so far, which include routing and scheduling. Functions representing this class of problems usually have many multi-valued variables, each taking a small range of values. As listed in the tables in Chapter 7, they have values ranges of at most 15 but the number of variable can be as high as 88.

However, we are currently applying MDD's to the formal verification of interacting FSM's. In these cases, we encounter some multi-valued variables which have large value ranges. For example, the state variable of an n-bit counter can take 2^n integer values. We anticipate difficulties if we use the current mapping and ordering methods for such variables.

The problem of using natural ordering for variables with large value ranges is illustrated by Figure 5.4. Imagine x is a variable representing a short-integer, i.e. $x \in$

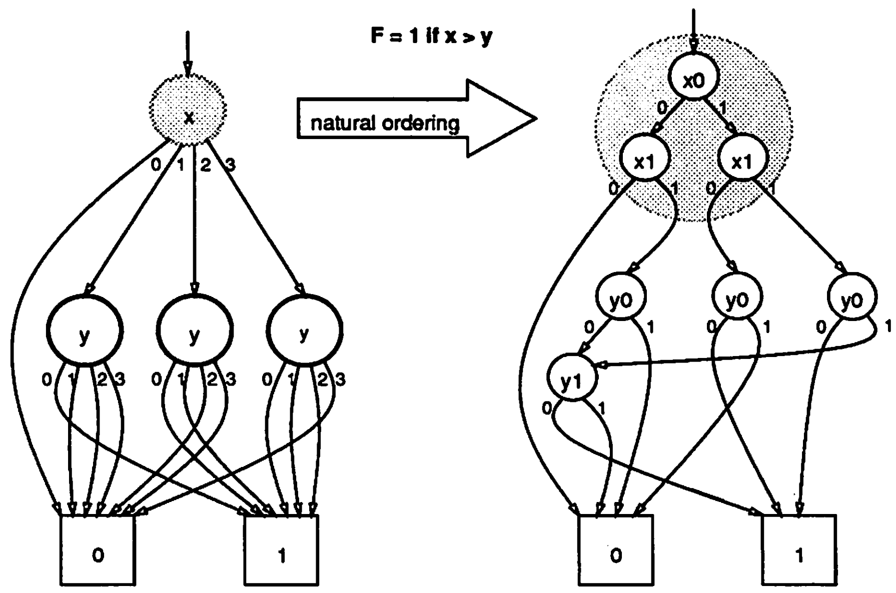


Figure 5.3: Natural ordering of encoded variables.

$\{0, 1, \ldots, 65535\}$. As mentioned before, there are 65536 outgoing edges from the mapped-BDD subgraph cluster above the dotted line. In the worst case, it would have 65536 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 can interleave the encoded variables as shown on Figure 5.4b. 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. For a single constraint, each cluster is made up of at most 3 BDD vertices, and the total number of outgoing edges is 4 which is independent of the value range of the variables. With interleaving, we can keep the shape of the mapped-BDD slim. The interleaved ordering is related but not dependent on the special mapping described in Section 4.4. Actually, we tried special mapping with natural ordering on the examples listed in Chapter 7. Interestingly, this approach is still faster than using the natural mapping approach with natural ordering. When the natural mapping is used in such an approach, for the example in Figure 4.1, the mapped-BDD subgraphs for variable y are constructed at the bottom of the mapped-BDD as shown in Figure 4.1b.

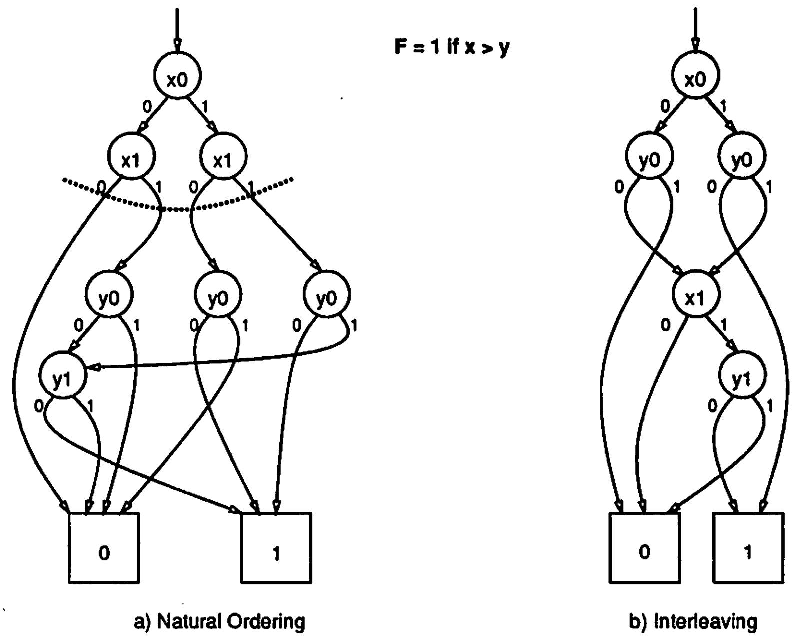


Figure 5.4: Comparison between natural and interleaved ordering.

When the interleaved ordering is used and the CASE on the encoded variable x_1 is next performed, the BDD package will automatically interleave the x_1 BDD vertices in between the y_0 and y_1 vertices. As a result, the final mapped-BDD with interleaved ordering is the same as that obtained by the special mapping but the speed of construction is faster using the special mapping than the general mapping.

5.2.3 Mixed Natural/Interleaved Ordering

Usually as CAD problems are formulated using MDD's, some closely-related variables should be interleaved as discussed in Section 5.2.2. However, there are usually other variables or groups of variables which are not related to each other and should be put in a natural ordering as in Section 5.2.1. This situation calls for the use of a mixture of both the natural and interleaved ordering within a single mapped-BDD. The current MDD package allows variables to be ordered by a mixture of such heuristics. This is accomplished by a parameter called *stride* for each MDD variable stored in the *mdd_manager.*

Example Suppose a problem has six multi-valued variables with the following constraints:

$$\begin{aligned} u &> v; \\ w &= 1; \\ x &< y; \\ y &< z; \end{aligned}$$

The encoded binary-variables for u should be interleaved with those for v as they are related by the first constraint. Similarly variables x, y and z are related by $x < y < z$ and their encoded variables should be interleaved with one another. Encoded variables of w can just follow a natural ordering. In addition, there is no relationship between the groups of variables $\{u, v\}$, $\{w\}$, and $\{x, y, z\}$ so their encoded variables can be ordered one after another. A desired order list of all encoded variables, *bvar_list*, which respects these groupings is:

$$u_0 \prec v_0 \prec u_1 \prec v_1 \prec u_2 \prec v_2 \prec w_0 \prec w_1 \prec x_0 \prec y_0 \prec z_0 \prec x_1 \prec y_1 \prec z_1$$

With the MDD package, the user needs to specify only the information in Table 5.1 and the above *bvar_list* is derived automatically. The number of binary-variables used to

MDD Variable	value range	stride
u	8	2
v	8	2
w	4	1
x	4	3
y	4	3
z	4	3

Table 5.1: MDD variables information stored in *mdd_manager*.

encode the multi-valued variable u is $\lceil log_2(range(u))\rceil$ [1]. The parameter *stride* specifies the number of multi-valued variables that are in the same interleaving group as the variable. e.g. x, y and y are specified with strides of 3 as the associated 2 encoded variables of each three multi-valued variables are interleaved with one another. Similarly, u and v have strides of 2 while w, not interleaved with other variables, has a stride of 1. If *stride* of an MDD variable is not specified, the stride is by default 1 which corresponds to the case of natural ordering.

As the *mdd_manager* contains the order list as well as the stride information, various algorithms mentioned in Chapter 4 can locate the encoded variables they need in the order list. For example, if an algorithm after processing with variable u_0 needs the next variable u_1 for Shannon decomposition, u_1 can be found 2 (= $u.stride$) locations down the list.

[1] $\lceil k\rceil$ is the smallest integer larger than k.

Chapter 6

MDD Package User Interface

To solve the problems formulated in Chapter 3, the first- and second-generation of the MDD packages take their input from constraint input files derived from the problems. Another version of the MDD package is needed when we start to apply MDD to new applications, such as the formal verification of interacting FSM's. The input format as a constraint input file is not convenient for these applications. A clean user-interface is needed where the user can construct and manipulate MDD's by calling a common set of MDD function calls. This function call interface will be described in Section 6.2. Also since the original MDD package was part of the MIS [BRSVW87] environment, a portable stand-alone MDD package is required for general use. Thus such a third-generation MDD package has been developed and is described in this chapter.

6.1 Constraint Input File Interface

The header of the constraint input file consists of three lines. The first has one number, the total number of multi-valued variables used. The second line contains a list of all the variable names; the third line is a list of their respective value ranges. The implied multi-valued variable ordering will be followed. The rest of the file is a line by line listing of all the constraints. Each constraint can take the following formats:

- $variable_1 > variable_2$
- $variable_1 >= variable_2$
- $variable_1 < variable_2$

- $variable_1 <= variable_2$
- $variable_1 == variable_2$
- $variable_1\ != variable_2$
- if $constraint_1$ then $constraint_2$

6.2 Function Call Interface

The following is the set of function calls available to the user, which serves as the user-interface to the third-generation MDD package. For backward compatibility, a conversion program has been written that takes the constraint input file and converts it into an equivalent set of function calls for constructing the corresponding discrete function.

Beside the MDD graph structure, there is a run-time *mdd_manager* (similar to the *bdd_manager*) which maintains all the bookkeeping information for the MDD's. It includes a pointer (*mdd.hook*) to two arrays, *mvar_list* and *bvar_list*. *mvar_list* contains a list of the multi-valued variables (*mvar*) while *bvar_list* contains a list of all the encoded binary variables (*bvar*). They also contain enough information such that one can easily find the correspondence between *mvar*'s and *bvar*'s.

6.2.1 Multi-valued Operator Function Calls

mdd_manager *
mdd_init(mvar_sizes)
array_t *mvar_sizes;

Given an array of mvar values (mvar_sizes), it initializes and returns a new mdd_manager which includes mvar_list and bvar_list information. The name of each mvar has a prefix of "mv" followed by its mvar id number.

mdd_manager *
mdd_init_name(mvar_sizes, mvar_names)
array_t *mvar_sizes;
array_t *mvar_names;

Given an array of mvar values (mvar_sizes) and another array of mvar names (mvar_names), it initializes and returns a new mdd_manager which includes mvar_list and bvar_list information.

void
mdd_quit(mgr)
mdd_manager *mgr;

Free up all resources associated with the mdd_manager (and bdd_manager).

mdd_t *
mdd_case(mgr, mvar, child_list)
int var;
array_t *child_list;

Given an mvar id and an array of child-functions id's, the MDD CASE operation is performed on all the child-functions and the resultant MDD function is returned.

mdd_t *
mdd_literal(mgr, mvar, values)
mdd_manager *mgr;
int mvar;
array_t *values;

Given an mvar id and an array of values which the mvar can have, the MDD representing a multivalued literal is returned. The literal is an OR of terms of the form mvar=value. If mvar can take 5 values {0,1,2,3,4}, to specify the literal (mvar=2 or mvar=4) we would put in the array "values" the values {2,4}.

mdd_t *
mdd_eq_g(mgr, mvar1, mvar2)
mdd_manager *mgr;
int mvar1;
int mvar2;

Given two mvar id's, the MDD representation of the relation (mvar1 = mvar2) is returned.

mdd_t *

mdd_neq_g(mgr, mvar1, mvar2)

mdd_manager *mgr;

int mvar1;

int mvar2;

Given two mvar id's, the MDD representation of the relation (mvar1 $\neq$ mvar2) is returned.

mdd_t *

mdd_leq_g(mgr, mvar1, mvar2)

mdd_manager *mgr;

int mvar1;

int mvar2;

Given two mvar id's, the MDD representation of the relation (mvar1 $\leq$ mvar2) is returned.

mdd_t *

mdd_lt_g(mgr, mvar1, mvar2)

mdd_manager *mgr;

int mvar1;

int mvar2;

Given two mvar id's, the MDD representation of the relation (mvar1 $<$ mvar2) is returned.

mdd_t *

mdd_geq_g(mgr, mvar1, mvar2)

mdd_manager *mgr;

int mvar1;

int mvar2;

Given two mvar id's, the MDD representation of the relation (mvar1 $\geq$ mvar2) is returned.

mdd_t *

mdd_gt_g(mgr, mvar1, mvar2)
mdd_manager *mgr;
int mvar1;
int mvar2;

Given two mvar id's, the MDD representation of the relation (mvar1 > mvar2) is returned.

mdd_t *
mdd_eq_s(mgr, mvar1, mvar2)
mdd_manager *mgr;
int mvar1;
int mvar2;

Given two mvar id's, the MDD representation of the relation (mvar1 = mvar2) is returned. Special mapping is used to speed up the MDD construction.

mdd_t *
mdd_neq_s(mgr, mvar1, mvar2)
mdd_manager *mgr;
int mvar1;
int mvar2;

Given two mvar id's, the MDD representation of the relation (mvar1 ≠ mvar2) is returned. Special mapping is used to speed up the MDD construction.

mdd_t *
mdd_leq_s(mgr, mvar1, mvar2)
mdd_manager *mgr;
int mvar1;
int mvar2;

Given two mvar id's, the MDD representation of the relation (mvar1 ≤ mvar2) is returned. Special mapping is used to speed up the MDD construction.

mdd_t *
mdd_lt_s(mgr, mvar1, mvar2)

mdd_manager *mgr;

int mvar1;

int mvar2;

Given two mvar id's, the MDD representation of the relation (mvar1 $<$ mvar2) is returned. Special mapping is used to speed up the MDD construction.

mdd_t *

mdd_geq_s(mgr, mvar1, mvar2)

mdd_manager *mgr;

int mvar1;

int mvar2;

Given two mvar id's, the MDD representation of the relation (mvar1 $\geq$ mvar2) is returned. Special mapping is used to speed up the MDD construction.

mdd_t *

mdd_gt_s(mgr, mvar1, mvar2)

mdd_manager *mgr;

int mvar1;

int mvar2;

Given two mvar id's, the MDD representation of the relation (mvar1 $>$ mvar2) is returned. Special mapping is used to speed up the MDD construction.

mdd_t *

mdd_cofactor(mgr, fn, cube)

mdd_manager *mgr;

mdd_t *fn;

mdd_t *cube;

Perform the cofactoring of a function 'fn' with respect to a multivalued cube 'cube' using mdd_and_smooth. First all mvars present in 'cube' are extracted, then 'fn' and 'cube' are ANDed together and smoothed with the mvars in 'cube'.

mdd_t *

mdd_smooth(mgr, fn, mvars)

mdd_manager *mgr;

mdd_t *fn;

array_t *mvars;

Given an MDD function and an array of smoothing variables, mdd_smooth smooths 'fn' with respect to all the mvars.

mdd_t *

mdd_and_smooth(mgr, f, g, mvars)

mdd_manager *mgr;

mdd_t *f;

mdd_t *g;

array_t *mvars;

Given two MDD functions and an array of smoothing variables, mdd_and_smooth first AND 'f' and 'g' together and then smooth the intermediate function with respect of the mvars. The resultant MDD function is returned.

mdd_t *

mdd_substitute(mgr, fn, old_mvars, new_mvars)

mdd_manager *mgr;

mdd_t *fn;

array_t *old_mvars;

array_t *new_mvars;

In function 'fn', multivalued variable old_vars[i] is substituted with the corresponding new_vars[i]. It checks that arrays old_mvars and new_mvars are of the same length, and within them, each corresponding pair of mvars have the same number of values.

6.2.2 Binary-valued Operator Function Calls

The following function calls are actually equivalent to the corresponding function calls from the BDD package. They are simply macro definitions. Their functions have been documented in the Berkeley BDD package which is appended in Appendix B.

Note also that the structure of an MDD node is actually the same as that of a BDD node:

```
#define mdd_t bdd_t

#define mdd_and bdd_and
#define mdd_constant bdd_constant
#define mdd_dup bdd_dup
#define mdd_equal bdd_equal
#define mdd_free bdd_free
#define mdd_is_tautology bdd_is_tautology
#define mdd_ite bdd_ite
#define mdd_leq bdd_leq
#define mdd_not bdd_not
#define mdd_one bdd_one
#define mdd_or bdd_or
#define mdd_size bdd_size
#define mdd_xor bdd_xor
#define mdd_xnor bdd_xnor
#define mdd_zero bdd_zero
```

Chapter 7

Results

The following benchmark examples were run on a VAX 8800. The results will be compared against Srinivasan's first-generation MDD package [SKMB90] which doesn't use BDD's. These comparisons should not be taken as the final verdict between the effectiveness of the straightforward MDD approach and the mapped BDD approach because efficient BDD techniques have not been exploited in the first-generation MDD package.

7.1 Scheduling Problems

To find out how large a problem the MDD approach can handle, several real-life digital filter scheduling problems were used as shown in Table 7.1. These include an 11th order FIR filter (fir11), a 5th order elliptic filter (ellip5), a 7th order quadratic filter (quad7) and two small examples at the top of Table 7.1. The columns of Table 7.1 and 7.2 represent: (1) the name of the benchmark example, (2) the number of multi-valued variables, (3) the

Example	No. of Vars.	Value Range	No. of Constr.	Size of BDD Mapped	Run Time MDD	RunTime Mapped BDD	Speed Up
sched1	22	6,1-2	42	258	2.8	1.2	2.3
sched2	24	4,2	44	327	3.9	0.5	7.8
fir11	88	14,1	256	1	109.9	20.0	5.5
ellip5	44	10,1-2	129	10503	472.3	79.4	5.9
quad7	54	15,1-2	121	22178	403.5	77.8	5.2

Table 7.1: Results on scheduling benchmarks.

Example	No. of Vars.	Value Range	No. of Constr.	Size of BDD Mapped	Run Time MDD	RunTime Mapped BDD	Speed Up
n7.5	11	11	24	213	7.7	2.0	3.9
n7.7	13	12	41	2205	72.0	12.2	5.9
n9.5	13	13	22	224	13.2	2.3	5.7
n9.6	14	13	56	13471	290.1	51.6	5.6
n9.7	14	14	30	498	46.9	5.8	8.1
asc	20	9	70	5371	112.7	8.6	13.1

Table 7.2: Results on routing benchmarks.

value ranges for variables (for Table 7.1, the first range is for time step and the second is for number of machines for each type), (4) the number of constraints, (5) the number of BDD vertices in the final mapped-BDD, (6) the run-time in seconds for the direct MDD implementation by Srinivasan, (7) the run-time in seconds for the MDD package using mapped-BDD's, (8) the speedup factor using the mapped MDD package as compared to the direct MDD implementation.

fir11 is the most complicated example. The flow graph and variable ordering is shown in Figure 5.2. It has 88 MDD variables and 256 constraints. Allowing only one function unit per type, it cannot not be scheduled in 14 time steps. The final MDD, a single zero BDD terminal vertex, can be considered a proof of the non-existence of a solution. If one of the redundant constraints in Equation 3.1 and 3.2 are removed, all these benchmarks can run within one minute. Except for the smallest example, all examples show a speed improvement of about six times or more over the direct MDD implementation.

7.2 Routing Problems

Channel routing problems (n*.*) of different sizes were also tried as well as a switchbox routing example (asc). The results are shown in Table 7.2. Again except for the smallest example, all show a speedup factor of about 6 or more. Relatively, larger examples tends to run even more efficiently with the MDD package using mapped-BDD's. To give a perspective on how well the MDD approach works, Table 7.3 gives a comparison with Devadas results [Dev89], using a subset of the routing examples running on the same machine, VAX 8800.

Example	No. of Vars.	Value Range	RunTime Boolean Sat.	RunTime Mapped BDD	Speed Up
n7.5	11	11	126	2.0	63
n7.7	13	12	174	12.2	14
n9.7	14	14	246	5.8	42

Table 7.3: Comparison with Devadas's results.

Chapter 8

Conclusions

A rigorous theoretical basis for MDD's has been provided. Also developed are efficient algorithms that map MDD into BDD's, and variable ordering heuristics that work well on most problems. For problems which need large value ranges, the concept of variable interleaving has been introduced, and an alternate set of mapping and ordering techniques have been developed. After three versions of development, a highly efficient and general purpose MDD package is available.

With MDD's, we can approach CAD problems in a natural setting, using multi-valued symbolic variables which associate directly with the problem statement. MDD's implicitly enumerate all the solutions. Finding an optimum solution is guaranteed if it exists. Otherwise the non-existence of a solution is proved. Because of these desirable properties, MDD's can be used as a complement to traditional heuristic methods in some domains of application. A variety of CAD applications have been formulated using the MDD approach. Results of realistic benchmark examples are encouraging.

This work represents just a beginning for exploring the capabilities of MDD's. There are many other applications of MDD's, one of which is the formal verification of interacting FSM's. The two ordering heuristics presented in this report work well for two ends of the spectrum of CAD problems. We believe that future experimentation with ordering heuristics will be key for using MDD's to solve many other CAD related problems.

Appendix A

Documentation for array package

```
An array_t is a dynamically allocated array.  As elements are inserted
the array is automatically grown to accomodate the new elements.

The first element of the array is always element 0, and the last
element is element n-1 (if the array contains n elements).

This array package is intended for generic objects (i.e., an array of
int, array of char, array of double, array of struct foo *, or even
array of struct foo).

Be careful when creating an array with holes (i.e., when there is no
object stored at a particular position).  An attempt to read such a
position will return a 'zero' object.  It is poor practice to assume
that this value will be properly interpreted as, for example,  (double)
0.0 or (char *) 0.

In the definitions below, 'typeof' indicates that the argument to the
'function' is a C data type; these 'functions' are actually implemented
as macros.



array_t *
array_alloc(type, number)
typeof type;
int number;
        Allocate and initialize an array of objects of type 'type'.
        Polymorphic arrays are okay as long as the type of largest
        object is used for initialization.  The array can initially
        hold 'number' objects.  Typical use sets 'number' to 0, and
        allows the array to grow dynamically.


void
array_free(array)
```

```
array_t *array;
        Deallocate an array.  Freeing the individual elements of the
        array is the responsibility of the user.


int
array_n(array)
array_t *array;
        Returns the number of elements stored in the array.  If this is
        'n', then the last element of the array is at position 'n' - 1.


void
array_insert(type, array, position, object)
typeof type;
array_t *array;
int position;
type object;
        Insert a new element into an array at the given position.  The
        array is dynamically extended if necessary to accomodate the
        new item.  It is a serious error if sizeof(type) is not the
        same as the type used when the array was allocated.  It is also
        a serious error for 'position' to be less than zero.


void
array_insert_last(type, array, object)
typeof type;
array_t *array;
type object;
        Insert a new element at the end of the array.  Equivalent to:
                array_insert(type, array, array_n(array), object).


type
array_fetch(type, array, position)
typeof type;
array_t *array;
int position;
        Fetch an element from an array.  A runtime error occurs on an
        attempt to reference outside the bounds of the array.  There is
        no type-checking that the value at the given position is
        actually of the type used when dereferencing the array.


type
array_fetch_last(type, array)
typeof type;
array_t *array;
        Fetch the last element from an array.  A runtime error occurs
```

```
if there are no elements in the array.  There is no type-checking
that the value at the given position is actually of the type used
when dereferencing the array.  Equivalent to:
        array_fetch(type, array, array_n(array))
```

Appendix B

Documentation for BDD Package

```
The BDD package has been modified to be independent of misII so that it
can be used in any application.  Many of the functions which previously
existed in the BDD package have been moved to the misII package NTBDD.

Compatibility with misII Release 2.1 has not been maintained.

Release 2.2 provides the capability of maintaining several different
orderings. This is done through the concept of a bdd_manager, where a
manager is associated with a particular ordering. Thus, a manager may be
created by specifying an ordering of variables. Other variables may be
added later at the end of this ordering.

Summary:
        bdd_free();
        bdd_dup();

        bdd_equal();
        bdd_compose();
        bdd_leq();
        bdd_is_tautology();

        bdd_not();
        bdd_and();
        bdd_or();
        bdd_xor();
        bdd_xnor();
        bdd_cofactor();
        bdd_smooth();
        bdd_substitute();
        bdd_and_smooth();
        bdd_ite();
        bdd_one();
        bdd_zero();

        bdd_top_var();
        bdd_then();
        bdd_else();

        bdd_start();
        bdd_end();
        bdd_get_variable();
        bdd_create_variable();

        bdd_size();
        bdd_print();
        bdd_get_stats();
        bdd_print_stats();
        bdd_set_gc_mode();

void
bdd_free(bdd)
bdd_t *bdd;
        Frees up 'bdd'.

bdd_t *
bdd_dup(bdd)
bdd_t *bdd;
        Returns a copy of the BDD.


boolean
bdd_equal(bdd1, bdd2)
bdd_t *bdd1, *bdd2;
        Checks if the two BDDs are identical, returns '1' if they are,
        '0' otherwise. Both BDDs should belong to the same bdd manager.

bdd_t *
bdd_compose(bdd1,bdd2,bdd3)
bdd_t *bdd1,*bdd2,*bdd3;
        'bdd2' is the bdd of variable 'v'. Replaces 'v' with 'bdd3' in
        'bdd1'.  'bdd2' should be a single variable bdd.

boolean
bdd_leq(bdd1,bdd2,phase1,phase2)
bdd_t *bdd1, *bdd2;
boolean phase1, phase2;
        Checks for implications. 'phase1' and 'phase2' indicate the phases to be
        used for 'bdd1' and 'bdd2'. For example:
                bdd_leq(bdd1,bdd2,1,0)
        returns returns the value of bdd1 => bdd2'.(While this can be done using
        bdd_or and then checking if this result is a constant value, using
        bdd_leq is generally faster.)

boolean
bdd_is_tautology(f,phase)
```

```
bdd_t *f;
boolean phase;
        Checks if the given function is tautologously true. 'phase'
        indicates the phase to be used for 'f', i.e. phase==1 checks
        if f==1 and phase==0 checks if f'==1.


bdd_t *
bdd_not(bdd)
bdd_t *bdd;
        Returns the BDD for the complement of 'bdd'.

bdd_t *
bdd_and(bdd1, bdd2, phase1, phase2)
bdd_t *bdd1, *bdd2;
boolean phase1, phase2;
        Returns the BDD for the AND of 'bdd1' and 'bdd2' in the phases
        specified by 'phase1' and 'phase2'. For example if phase1 == 0
        and phase2 == 1, the function will return the bdd for
        (bdd1' and bdd2).  Both BDDs should belong to the same bdd manager.

bdd_t *
bdd_or(bdd1, bdd2, phase1, phase2)
bdd_t *bdd1, *bdd2;
boolean  phase1, phase2;
        Returns the BDD for the OR of 'bdd1' and 'bdd2' in the phases
        specified by 'phase1' and 'phase2'. For example if phase1 == 0
        and phase2 == 1, the function will return the bdd for
        (bdd1' or bdd2).  'bdd1' and 'bdd2' should belong to the same
        bdd manager.

bdd_t *
bdd_xor(bdd1, bdd)
bdd_t *bdd1, *bdd2;
        Returns the BDD for the XOR of the two BDDs.
        'bdd1' and 'bdd2' should belong to the same bdd manager.

bdd_t *
bdd_xnor(bdd1, bdd)
bdd_t *bdd1, *bdd2;
        Returns the BDD for the XNOR of the two BDDs.
        'bdd1' and 'bdd2' should belong to the same bdd manager.

bdd_t *
bdd_cofactor(bdd1, bdd2)
bdd_t *bdd1, *bdd2;
        Returns the cofactor of the 'bdd1' with respect to 'bdd2'. 'bdd2'
        should be a single cube. 'bdd1' and 'bdd2' should  belong to
        the same bdd manager.
```

```
bdd_t *
bdd_smooth(f, var_array)
bdd_t *f;
array_t *var_array;
        The smoothing function.  'var_array' is an array of bdd_t.

bdd_t *
bdd_substitute(f, old_array, new_array)
bdd_t *f;
array_t *old_array, *new_array;
        Substitute all old_array vars with new_array vars. 'old_array' and
        'new_array' are arrays of bdd_t.  Given two arrays of variables a and b
        consisting of member values (a1 .. an) and (b1 .. bn), replace all
        occurrences of ai by bi. This could be done iteratively with
        bdd_compose but would require n passes instead of one.  Thus this algorithm
        is only a performance optimization.

bdd_t *
bdd_and_smooth(f, g, smoothing_vars)
bdd_t *f, *g;
array_t *smoothing_vars;
        Smooth and AND at the same time.  'smoothing_vars' is an array of bdd_t,
        which are the input variables to be smoothed out.

bdd_t *
bdd_ite(f, g, h, f_phase, g_phase, h_phase)
bdd_t *f, *g, *h;
boolean f_phase, g_phase, h_phase;
        Returns the ite (IF-THEN-ELSE) of three bdd's: ITE(f,g,h).

bdd_t *
bdd_one(mgr)
bdd_manager *mgr;
        Returns a new copy of the 1 BDD.

bdd_t *
bdd_zero(mgr)
bdd_manager *mgr;
        Returns a new copy of the 0 BDD.


bdd_t *bdd_top_var(bdd)
bdd_t *bdd;
        Returns the bdd corresponding to the top variable of 'bdd'.

bdd_t *bdd_then(bdd)
bdd_t *bdd;
        Returns the bdd of the function when the top_var of 'bdd' evaluates
        to 1.
```

```
bdd_t *bdd_else(bdd)
bdd_t *bdd;
        Returns the bdd of the function when the top_var of 'bdd' evaluates
        to 0.


bdd_manager *
bdd_start(nvariables)
int nvariables;
        Initialize and return a bdd_manager. 'nvariables' is
        the maximum number of variables allowed in this bdd manager,
        and cannot change over the lifetime of the bdd manager.

void
bdd_end(manager)
bdd_manager *manager;
        Terminate a bdd manager.

bdd_t *
bdd_get_variable(manager, variable_i)
bdd_manager *manager;
bdd_variableId variable_i;
        Get or create a new variable and add it to the bdd.  The variable
        must be in the range 0 to nvariables, as specified in the call to bdd_start,
        else a failure occurs.  A bdd of the form (v, 1, 0) is returned.  This is
        the only acceptable way of getting or creating variables in the manager.

bdd_t *
bdd_create_variable(bdd_manager)
bdd_manager *bdd_manager;
        Returns a BDD for a new variable after registering it with the
        manager. This BDD is  of the form (v, 1, 0).  The variable is added
        to the end of the current ordering.


int
bdd_size(bdd)
bdd_t *bdd;
        Returns the size of the BDD, 0 for the constant function 0, 1 for
        the constant function 1, and the number of vertices in the BDD
        otherwise.

void
bdd_print(f)
bdd_t *f;
        Print a bdd.

void
bdd_get_stats(manager, stats)
bdd_manager *manager;
```

```
bdd_stats *stats;
        Get the statistics from the bdd package, and return 'stats'.

void
bdd_print_stats(stats, file)
bdd_stats stats;
FILE *file;
        Print the given statistics to 'file'.

void
bdd_set_gc_mode(bdd, no_gc)
bdd_manager *bdd;
boolean no_gc;
        If no_gc==0, turn on garbage collection, else if no_gc==1, turn
        off garbage collection.
```

Bibliography

[Ake78] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–516, June 1978.

[BHMSV84] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[BRB90] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. *Proc. 27th Design Automation Conference*, pages 40–45, June 1990.

[BRSVW87] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[Bry86] R.E. Bryant. Graph based algorithms for Boolean function manupilation. *IEEE Transactions on Computers*, C-35:667–691, August 1986.

[CBM89] O. Coudert, C. Berthet, and J.C. Madre. Verification of sequential machines based on symbolic execution. *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, June 1989.

[Dev89] S. Devadas. Optimal layout via Boolean satisfiability. *Proc. Int. Conf. on CAD, 1989 (ICCAD-89)*, pages 294–297, 1989.

[FFK88] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. *Proc. Int. Conf. CAD (ICCAD-88)*, pages 2–5, November 1988.

[FS87] S.J. Friedman and K.J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *Proc. 24th Design Automation Conference*, June 1987.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[HK88] Z. Har'El and R.P. Kurshan. Software for analysis of coordination. *Proc. Int. Conf. Syst. Sci. Eng.*, pages 382–385, 1988.

[Kar87] K. Karplus. Boole: an abstract data type for manupilating Boolean expressions. *Unpublished Manuscript*, 1987.

[MWBSV88] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. *Proc. Int. Conf. CAD (ICCAD-88)*, pages 6–9, November 1988.

[SKMB90] A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. *Proc. Int. Conf. CAD (ICCAD-90)*, pages 92–95, November 1990.

[TSL+90] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. *Proc. Int. Conf. CAD (ICCAD-90)*, pages 130–133, November 1990.

[YK82] T. Yoshimura and E.S. Kuh. Efficient algorithms for channel routing. *IEEE Transactions on CAD*, pages 25–35, January 1982.