MULTI-VALUED MULTI-LEVEL LOGIC SYNTHESIS

by

Minxi Gao

Memorandum No. UCB/ERL M01/2

21 December 2000

MULTI-VALUED MULTI-LEVEL LOGIC SYNTHESIS

by

Minxi Gao

Memorandum No. UCB/ERL M01/2

21 December 2000

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Multi-valued multi-level logic synthesis

Minxi Gao

November 20, 2000

The Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94704

Submitted in partial satisfaction of the requirements
for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:
Project Report Committee:

Professor Robert K. Brayton
Research Advisor

Date

Professor Tom A. Henzinger

Date

Abstract

Multi-valued multi-level logic synthesis

Minxi Gao

Master of Science in Engineering-Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Robert K Brayton, Chair

We address the problem of multi-valued network minimization. Each node in the multi-valued network is a logic function with multiple multi-valued inputs and a single multi-valued output. An important step in network optimization is extracting new nodes representing logic functions that are common factors of other nodes. This includes methods for finding common sub-expressions, semi-algebraic resubstitution, decomposing a multi-valued network, and factoring an expression. Network reconstruction also involves methods including collapse, elimination and merge. We concentrate on the generalization of these operations from the binary domain to the multi-valued domain. We start with the core algorithms for semi-algebraic division and factorization for multi-valued networks. These include a satisfiability-matrix based method and a maximum graph matching method. We then look at MV-network manipulation operations such as kernel extraction, factorization, decomposition, resubstitution, collapsing, elimination and node merge. We introduce a system called MVSIS in which we have implemented the above methods. In the end, we present the results of MVSIS and in particular compare them with SIS on binary examples.

To my family and Zhong Dong

Acknowledgements

Acknowledgements

The first person, and foremost, that I am obliged to gives thanks to, is my research advisor, Robert K. Brayton. My work had not been possible without his guidance. A lot of the ideas originated from/belonged to him. The weekly meetings with him have taught me invaluable skills to tackle research problems. The interactions with him have also taught me to strive for new insights when defining a research problem and for perfection when finalizing research results.

William Jiang, whom I worked with on the MVSIS project, deserves my special acknowledgements. He has generously helped me in the early stages of my project, as well as given valuable discussions and suggestions later in the course of the project implementation.

I would also like to thank Subarna Sinha, another member in our group. She had built the infra data structure for MVSIS which I realized most of my work on and I have constantly turned to her for help regarding the implementation of MVSIS.

This work was supported by the SRC under contract 683.004 and through the California Micro program by Fujitsu, Synopsys, and Cadence.

Contents

List of Figures

List of Tables

Life is not about the result. Life is about the process.

- Minxi

Chapter 1

Introduction

Multi-valued minimization is useful in high-level hardware/software synthesis. One approach has been to first encode the multi-valued variable, optimize in the binary domain, and then translate back to the multi-valued domain. During this process, some optimization opportunities are lost. Direct multi-valued minimization on these applications may be able to uncover these opportunities. Multi-valued minimization is also found useful in artificial intelligence and some asynchronous applications [LFS+]. In this chapter, we will review previous research on multi-valued logic synthesis. We will then address the multi-level multi-valued logic synthesis problem in particular. We will briefly preview the optimization methods to be presented in this report. In the end, we will give an outline of the topics covered in this report.

1.1 Multi-valued logic synthesis overview

Research on multi-valued logic synthesis for hardware was originally motivated by the FSM state encoding problem. The two level multi-valued logic synthesis problem was addressed earlier in the work of Rudell et.al. [RSV88] and efficient implementations have been developed in ESPRESSO-MV. Techniques developed there were applied to the encoding problem of two-level implementations [VSV89][VSV90]. Beyond this, multi-level multi-valued minimization includes node minimization and common sub-expression identification. The node minimization problem in this context was recently studied in the work of Jiang [JB00].

Identifying and substituting a common sub-expression is performed via the "division" operation that has been used extensively in the binary domain. Technology dependent multi-level synthesis had been reported in the literature [WLC94]. The technology-independent multi-level synthesis has been proposed in the work of Lavagno

et.al. [LMBSV90] with its implementation MIS-MV, targeting the state assignment problem for FSM's. Lavagno's work, however, was restricted to networks with only 1 multi-valued input variable at each node, usually the state variable. Although applicable to the state assignment problem, this soon becomes a limitation on general multi-valued systems optimization. Also the implementation was extremely computationally intensive. An algebraic division procedure that forms the core of multi-level logic synthesis was first studied in the work of Hui Min Wang el at [WLC94]. In their work, however, a rather restrictive set of notations has been employed that makes the application to general multi-valued problems hard. In conclusion, an efficient set of techniques and its implementation are still missing in the work of multi-valued multi-level logic manipulation and optimization. This is largely due to the associated combinatorial value explosion and the flexibility introduced by semi-algebraic operations, both of which are unique to multi-valued systems.

1.2 Multi-level multi-valued network optimization methods

We look at a pure multi-valued logic network where each of its nodes has multi-valued inputs and a single multi-valued output. The primary cost function we target is the number of cubes in the network. Consequently, the cost function we assume in the following chapters is the number of cubes needed to represent the network, unless otherwise specified. We have also implemented the algorithms in MVSIS with the cost function being the number of literals needed to represent the network. When the cost function is the number of literals, MVSIS gives almost the same results as SIS on binary examples. We will discuss briefly the techniques for handling the literal cost function in Section 3.10.

We first consider the core set of techniques associated with multi-level multi-valued minimization: algebraic division and factorization methods. The notion of algebraic methods in binary logic [BHV90] is generalized to multi-valued logic. Here we introduce a new concept in the next chapter that is unique to multi-valued logic: semi-algebraic methods. The notations and operations used are similar to binary operations. The complexity with multi-valued minimization, however, is much higher, due to the

explosion in the number of literals with different combinations of values in a variable, and to the additional flexibility in semi-algebraic division. For instance a variable of n values has 2^n distinct literals. Two techniques have been developed that select only the dividend with the fewest allowed values in each literal: the satisfiability-matrix based method and the maximum graph matching method.

We then look at the multi-level minimization operations in the system MVSIS. The operations that employ the semi-algebraic division and factorization techniques include fast extract, decomposition, re-substitution and factorization. The operations that do not directly apply the semi-algebraic methods but which are necessary for multi-level network manipulations are collapse, eliminate and merge. Merge is a unique operation for MV-logic. Because the operation, fast extract, extracts common sub-expressions that later will become a new node with a single binary output, a merge operation is considered useful. This combines multiple nodes into a single node with one MV output, possibly with larger set of values than the original nodes, but no more than 2^k values if k binary nodes are merged. As in the binary case, these operations are key in manipulating the initial representation of the logic function for finding an equivalent optimal multi-level structure.

There have been various representations for multi-valued logic functions and relations [BK99]. The most commonly used ones are: the sum-of-product forms (SOP) and multi-valued decision diagrams (MDD) for two-level logic and MV-networks for multi-level logic. In this report, we use the notations and definitions from [BK99] [Bra99]. Some of the notations will be repeated in this report for convenience.

1.3 Outline of this report

In Chapter 2, we first present the notations and definitions necessary for explaining the algorithms. The main body of Chapter 2 is devoted to the two methods used in the semi-algebraic division and factorization process: the satisfiability-matrix based method and the maximum graph matching method. Both the theory and the implementation heuristics will be given for the two methods.

In Chapter 3, we discuss the methods employed in implementing the multi-level optimization operations in MVSIS, namely fast extract, factorization, decomposition, re-substitution, collapsing, eliminate and node merge. At the end of this chapter, we briefly discuss the multi-level multi-valued logic optimization problem when targeting cube and literal cost functions.

In Chapter 4, we present the experimental results on various application examples. The results include quality and speed issues. We apply MVSIS on both MV-examples and binary examples. On binary examples, we compare the results with SIS.

We conclude this report in Chapter 5. Directions for future works are given also.

Chapter 2

Semi-algebraic methods

In this chapter, we first give notations and definitions necessary for further discussion on techniques presented in this report. We then concentrate on the two algorithms we have developed in semi-algebraic division and factorization on multi-valued logic functions: the satisfiabiliy-matrix based method and the maximum graph matching method. In these methods, we treat binary and multi-valued variables uniformly. We give implementation realizations of these algorithms and filtering techniques to speed up the operations.

2.1 Notations and definitions

Definition 2.1 (Multi-valued Variable) *A variable x_i is multi-valued if it takes on values from a set $P_i=\{0,1,\ldots,|P_i|-1\}$.*

Example 2.1 *x taking on 5 values from $\{0,1,2,3,4\}$ is a multi-valued variable.*

Definition 2.2 (Multi-valued Literal) *A multi-valued literal is a subset of values of a multi-valued variable. A literal evaluates to 1 if and only if it takes on one of the values in the subset.*

Example 2.2 *$x^{\{0,2\}}$ and $x^{\{1,2,3,4\}}$ are literals of x*

Definition 2.3 (Multi-valued Cube) *An MV cube is a conjunction of MV literals and evaluates to 1 if and only if each of the literals evaluates to 1. Literals containing all values of the corresponding variable do not appear in the cube form.*

Definition 2.4 (SOP) *A sum-of-product (SOP) is the OR of a set of cubes, which evaluates to 1 if any of the cubes evaluate to 1.*

Note that such an SOP is a function with a single binary output and multiple multi-valued input variables.

Definition 2.5 (Supercube) *The supercube of a set of cubes f, denoted $\sigma(f)$ is the smallest cube containing f.*

Example 2.3 *If x takes on values {0,1,2,3,4} and y takes on values {0,1,2}, then* $\sigma(x^{\{0,1\}}y^{\{0,2\}}, x^{\{1,4\}}y^{\{1\}})=x^{\{0,1,4\}}y^{\{0,1,2\}}=x^{\{0,1,4\}}$.

Definition 2.6 (Cube-free) *An expression f is cube-free if* $\sigma(f) = 1$.

Example 2.4 $f = a^{\{1,3\}}b^{\{2,3\}}+a^{\{0,3\}}b^{\{1,3\}}$ *is not cube-free since* $\sigma(f) = a^{\{0,1,3\}}b^{\{1,2,3\}} \neq 1$

Definition 2.7 (Complement Cube) *The complement cube* $\tilde{c}$ *of cube c is defined as the cube containing values not in c.*

Example 2.5 *If* $c = a^{\{1,3\}}b^{\{2,3\}}$, *then* $\tilde{c} = a^{\{0,2\}}b^{\{0,1\}}$.

Definition 2.8 (Common Cube) *An expression has a common cube if for each variable, there is no literal (except for the literal 1) appearing in the cubes of the expression that contains all other literals of that variable in the expression.*

Example 2.6 $g = a^{\{1,3\}}b^{\{1,2,3\}}+a^{\{0,1,3\}}b^{\{1,3\}}$ *has a common cube* $a^{\{0,1,3\}}b^{\{1,2,3\}}$ *because the literal* $a^{\{0,1,3\}}$ *contains* $a^{\{1,3\}}$ *and* $b^{\{1,2,3\}}$ *contains* $b^{\{1,3\}}$. *However* $f = a^{\{1,3\}}b^{\{2,3\}}+a^{\{0,3\}}b^{\{1,3\}}$ *has no common cube*

Note we want to make a distinction between supercube and common cube since we do not always want to make an expression cube free. For instance, making f cube free would give us

$$f = a^{\{1,3\}}b^{\{2,3\}}+a^{\{0,3\}}b^{\{1,3\}} = a^{\{0,1,3\}}b^{\{1,2,3\}}(a^{\{1,2,3\}}b^{\{0,2,3\}}+a^{\{0,2,3\}}b^{\{0,1,3\}})$$

which is not an improvement. An expression is simpler if we only extract the common cube, e.g., $g = a^{\{0,1,3\}}b^{\{1,2,3\}}(a^{\{1,3\}}+b^{\{1,3\}})$. This is the technique used in **MVSIS** in a large number of operations.

Definition 2.9 (Semi-algebraic Division) *A division is semi-algebraic if the product of the divisor and the dividend is neither 1 nor NULL, and if the divisor and the quotient do not contain a common literal.*

Example 2.7 $f = x^{\{0,1,3\}}(y^{\{1,2\}}+x^{\{0,2\}}y^{\{0\}})$ *is a semi-algebraic division since variable x appears in both the divisor* $x^{\{0,1,3\}}$ *and the quotient* $x^{\{0,2\}}y^{\{0\}}$. *However the literals* $x^{\{0,1,3\}}$ *and* $x^{\{0,2\}}$ *that represent x are not the same.*

Note in semi-algebraic division, the divisor and the quotient are allowed to have non-disjoint sets of variables. This is the fundamental difference between MV-logic operations and binary logic. However, this permits the explosion of the number of literals in a variable, but allows flexibility in division. In a sense, semi-algebraic division is a form of division which lies between pure algebraic and pure boolean division.

Example 2.8 $f = x^{\{0,1,3\}}(y^{\{1,2\}}+x^{\{0\}}y^{\{0\}})= x^{\{0,1,3\}}(y^{\{1,2\}}+x^{\{0,1\}}y^{\{0\}})$. *We see that division by* $x\{0,1,3\}$ *does not yields a unique quotient.*

We want to use the above notations and definitions to state our problem:

Problem definition 2.1 (Factorization) *Give a logic function f in SOP form, find a good factored form.*

Example 2.9 *If* $f = a^{\{0,1,2\}}c^{\{3\}}+b^{\{1,2,3\}}c^{\{3\}}+a^{\{0\}}c^{\{1\}}+a^{\{0\}}b^{\{1,2,3\}}c^{\{0\}}$

Then $f = (c^{\{3\}}+a^{\{0\}}c^{\{0,1,2\}})(a^{\{0,1,2\}}c^{\{1,3\}}+b^{\{1,2,3\}}c^{\{0,3\}})$

Problem definition 2.2 (Division) *Given a logic function f and a divisor d, find a quotient f/d using semi-algebraic division.*

Example 2.10 *If* $f = a^{\{0,1,2\}}c^{\{3\}}+b^{\{1,2,3\}}c^{\{3\}}+a^{\{0\}}c^{\{1\}}+a^{\{0\}}b^{\{1,2,3\}}c^{0\}}$,

$d = (c^{\{3\}}+a^{\{0\}}c^{\{0,1,2\}})$

then $f = d(a^{\{0,1,2\}}c^{\{1,3\}}+b^{\{1,2,3\}}c^{\{0,3\}})$

These two problems will be solved in the subsequent sections.

2.2 Satisfiability-matrix based semi-algebraic methods

The concept of satisfiability-matrix can be first found in the work of Lavagno et.al. [LMBSV90], which is limited to only a single multi-valued variable in the logic function *f*. We have developed techniques that also employ the satisfiability-matrix concept but can treat an arbitrary number of multi-valued variables in *f*. In our method, the definition for satisfiability-matrix is different. We consider a matrix *M*, each cell of which is filled with an MV-cube. Therefore *M* contains a positional arrangement of the MV cubes in it.

2.2.1 Theorems

Definition 2.10 (Value Condition) *Let I be the set of rows and J the set of columns in M in which value v of some variable appears. The value v satisfies the value condition if it appears in all entries of M given by* $\{M_{ij} \mid i \in I, j \in J\}$.

Note if *I* or *J* has only one element, then the value condition is always satisfied.

Definition 2.11 (Satisfiability-matrix) *A matrix M is satisfiable if all values of all variables in all cubes of it satisfy the value condition.*

Example 2.11 *The matrix*

row / column	1	2	3
1	$a^{[1,2]}b^{[1]}$	$a^{[1,3]}b^{[1,2]}$	$a^{[2,3]}b^{[2,3]}$
2	$a^{[2,3]}b^{[1]}$	$a^{[1,4]}b^{[1,2,5]}$	$a^{[4]}b^{[2]}$

is not satisfiable because the values $a^{[3]}$, $a^{[2]}$ and $a^{[1]}$ do not satisfy the value condition. It should have an $a^{[3]}$ in all cells, an $a^{[2]}$ in cell (2,3) and an $a^{[1]}$ in cell (2,1).

For such a matrix, we follow the procedure below:

1. For each row i, form the supercube of all cubes in that row; denote it d_i.
2. OR these supercubes together to form $d = \sum_i d_i$.
3. For each column j, form the supercube of all cubes in that column, denote it q_j.
4. OR these supercubes together to form $q = \sum_j q_j$.

Theorem 2.1 *For any satisfiable matrix M, the d and q formed above have the following property:*

$$M_{i,j} = d_i \cap q_j$$

$$\sum_{i,j} M_{i,j} = (d)(q)$$

Proof We will prove that $M_{i,j} \subseteq d_i \cap q_j$ and $M_{i,j} \supseteq d_i \cap q_j$

$M_{i,j} \subseteq d_i \cap q_j$: By definition of supercube, d_i is a cube containing every cube in row i and q_j is a cube containing every cube in column j, therefore M_{ij} is contained in both d_i and q_j.

$M_{i,j} \supseteq d_i \cap q_j$: Suppose $M_{i,j} \subset d_i \cap d_j$, then there exists a variable with a value v such that $v \in d_i \cap q_j$ but $v \notin M_{ij}$. However, v must be in M_{ik} for some k (since $v \in d_i$), and similarly v must be in M_{mj} for some m (since $v \in q_j$). Therefore, by the value condition for a satisfiable matrix, $v \in M_{ij}$, a contradiction. Hence, $M_{i,j} \supseteq d_i \cap q_j$. ■

Now we see a way to factor an expression. Given a logic function $f = c_1 + ... + c_n$, we can arrange a subset $S \subseteq \{c_1, ..., c_n\}$ in a matrix M such that M is satisfiable. The d and q obtained by the above procedure shall give us $f = (d)(q) + R$ where R contains the cubes of f that are not in S. If the goal of the factorization is to minimize the number of cubes in the factored representation of f, we would want S to have the maximum cardinality.

The next job is to find a way to search for a largest such arrangement. A brute-force method is to try all possible arrangements and select one of the largest. One well-known technique for this is the branch and bound algorithm. However, we need to find a good bounding mechanism. The above theorem on the value condition is theoretically elegant but difficult to implement in reality. Making sure that the placement of a cube at a new position (i, j) in M would involve validating the value condition of all values placed in the matrix so far*. To assist the implementation of the branch and bound algorithm for finding a largest satisfiability matrix, we have developed the following theorem:

Definition 2.12 (Lower Bound Cube) *Define cube l^{ij} (which depends on i, j) to consist of the following set of values:*

$$l^{ij} = \{v | \exists(k<j), v \in M_{ik}\} \text{ and } (\exists(m<i), v \in M_{mj})\}$$

Note that for $i=1$ or $j=1$, this is the null set.

* We assume that cubes are put in the matrix in row-column order, i.e., (1, 1), (2, 1) ... (m, 1), (2, 1), ...

Definition 2.13 (Upper Bound Cube 1) *Define cube* u_1^{ij} *(which depends on i, j) to consist of the following set of values:*

$$u_1^{ij} = \{v | \exists(k<j), v \in M_{ik}\} \text{ or } (\forall(m<i)\,\forall(n<j), v \notin M_{mn})\}$$

Note that for $i=1$ or $j=1$, this is the set of all values.

Definition 2.14 (Upper Bound Cube 2) *Define cube* u_2^{ij} *(which depends on i, j) to consist of the following set of values:*

$$u_2^{ij} = \{v | \exists(m<i), v \in M_{mj}\} \text{ or } (\forall(m<i)\,\forall(n<j), v \notin M_{mn})\}$$

Note that for $i=1$ or $j=1$, this is the set of all values.

Theorem 2.2 *A matrix of cubes M is satisfiable if and only if for each (i, j),* M_{ij} *satisfies* $l^{ij} \subseteq M_{ij} \subseteq u_1^{ij} \cap u_2^{ij}$

Proof Let v be an arbitrary value, the value condition is equivalent to the following propositions, assuming $m < i$, $k < j$:

1. *if* $v \in M_{ik}$ *and* $v \in M_{mj}$ *then* $v \in M_{ij}$
2. *if* $v \in M_{ik}$ *and* $v \in M_{mj}$ *then* $v \in M_{mk}$
3. *if* $v \in M_{mk}$ *and* $v \in M_{ij}$ *then* $v \in M_{mj}$
4. *if* $v \in M_{mk}$ *and* $v \in M_{ij}$ *then* $v \in M_{ik}$

Let x, a, b, c stand for the following propositions:

- $x \leftrightarrow v \in M_{ij}$,
- $a \leftrightarrow \exists(k < j),\ v \in M_{ik}$,
- $b_1 \leftrightarrow \exists(m \neq i)\exists(k < j),\ v \in M_{mk}$,
- $b \leftrightarrow \exists(m < i)(k < j),\ v \in M_{mk}$,
- $c \leftrightarrow \exists(m < i),\ v \in M_{mj}$,

We can restated the value conditions as:

1. $ac \Rightarrow x$
2. $ac \Rightarrow b$
3. $bx \Rightarrow a$
4. $bx \Rightarrow c$

We look at the value conditions involving the new position (i, j):

$$ac \Rightarrow x$$
$$bx \Rightarrow a$$
$$b_1 x \Rightarrow c$$

or equivalently (solving for x)

$$ac \Rightarrow x$$
$$x \Rightarrow a + \overline{b}$$
$$x \Rightarrow c + \overline{b_1}$$

These conditions are precisely the lower bound condition l_{ij} and the two upper bound conditions, u_1^{ij} and u_2^{ij} respectively. ■

We have observed some interesting properties about the bounds computed above that will later be found very useful in trimming down the search space for the branch and bound algorithm in the implementation.

Theorem 2.3 *u_1^{ij} is monotone decreasing as a function of j and u_2^{ij} is monotone decreasing as a function of i.*

Proof for u_1^{ij}: Let

$$u_1^{ij} = T_{ij} \cup S_{ij}$$

where

$$T_{ij} = \{v | \exists (k < j,\ v \in M_{ik})\}$$
$$S_{ij} = \{v | \forall (m<i) \forall (n<j), v \notin M_{mn}\}$$

It is a trivial fact that S_{ij} is a monotone decreasing function of j since the set of cubes $S_{i(j+1)}$ considers, is a super set of the cubes S_{ij} considers. Let w be any value in $u_1^{i(q+1)}$:

- Case 1: suppose $w \in S_{i(q+1)}$, then $w \in S_{iq} \subseteq u_1^{iq}$ since S_{ij} is monotone decreasing as a function of j.
- Case 2: suppose $w \notin S_{i(q+1)}$, then $w \in T_{i(q+1)} = T_{iq} \cup M_{iq}$. We have

$$M_{iq} \subseteq (u_1^{iq} \cap u_2^{iq}) \subseteq u_1^{iq} = T_{iq} \cup S_{iq}$$
$$\Rightarrow \quad w \in T_{iq} \cup (T_{iq} \cup S_{iq}) = T_{iq} \cup S_{iq} = u_1^{iq}.$$

Cases 1 and 2 are exhaustive. Therefore for any value $w \in u_1^{i(q+1)}$, $w \in u_1^{iq}$. ■

Proof for u_2^{ij}: The proof for u_2^{ij} is similar. ■

Definition 2.15 (N_{ncf}) *N_{ncf} of a cube c in function f is defined as the number of cubes in f, including c itself, that share a common cube with c.*

Example 2.12 $f = a^{\{0\}}b^{\{1,2\}}+a^{\{0,2\}}b^{\{2,3\}}+ a^{\{2\}}b^{\{3\}}$. The cube $a^{\{0\}}b^{\{1,2\}}$ has an $N_{ncf} = 2$ because besides itself, it shares a common cube $a^{\{0,2\}}$ with another cube $a^{\{0,2\}}b^{\{2,3\}}$. Similarly, the cube $a^{\{0,2\}}b^{\{2,3\}}$ has an $N_{ncf} = 2$ and $a^{\{2\}}b^{\{3\}}$ has an $N_{ncf} = 1$.

Theorem 2.4 *If a satisfiability-matrix M has m rows and n columns, then each cube in the matrix has its N_{ncf} at least MAX(m,n).*

Proof $f = (\sum_{i=1}^{m} d_i)(\sum_{j=1}^{n} q_j) + \{r\} = \{M_{ij}, 1 \le i \le m, 1 \le j \le n\} + \{r\}$

Here $M_{ij}=d_iq_j$. The cubes M_{ij}, $1 \le j \le n$ share a common cube d_i. The cubes M_{ij}, $1 \le i \le m$ share a common cube q_j. ■

2.2.2 Implementation heuristics and techniques

Theorem 2.2 enables us to focus only on the new position. This makes the implementation of the satisiability-matrix branch and bound algorithm much easier. Here we present one method of implementation and some important heuristics employed to speed up the process. Experimental results of the implementation are presented in Chapter 5, together with results of MVSIS user commands related to algebraic methods such as *decomp*, *resub*, etc.

There are two types of operations that we have associated with a satisfiability-matrix. We call these **inexact division** and **exact division**. The first is used generally in factorization, the second in algebraic division. Generally, in factorization, the divisor is not known. Given a logic function f, the cubes of f can be arranged in the satisfiability matrix to obtain the cubes of the divisor d and the quotient q by computing the supercubes of the row and column entries, as stated above. In this case, any set of cubes of f may be in the factored form. On the other hand, in algebraic division, the user

specifies the divisor d. Assuming the rows are to form the divisor, their supercubes have to be exactly the cubes in d. We call the satisfiability-matrix operation with this additional constraint **exact division**.

However, sometimes even for factorization, we may obtain a divisor that is likely to be contained in the factored form, and we can use it to guide the branch and bound search which generally speeds up the process. The divisor could come from a pair of cubes in the function f that share a common cube. A second source of such a divisor is similar to what is done in SIS [BHV90]. We perform inexact division twice. The first time we use $d_1 = 1 + 1$ in search for a satisfiability matrix to obtain a quotient q_1. We then use q_1 as the divisor d_2 and search for a satisfiability matrix a second time to obtain a better quotient q_2 and possibly a different divisor d_2'. In this case, we do not require d_2' to be exactly the same as d_2 but d_2 acts as a guide in the search.

To incorporate the above operations, we have developed additional constraints besides the value condition. Assume the supercubes of the rows are to form the divisor. In **inexact division**, the constraint is that the cubes in the matrix only have to be contained in the divisor cube corresponding to this row. In **exact division**, the constraint is that the supercubes of the rows have to be exactly the same as the divisor cubes.

One special case of factorization with the above constraint is to set the divisor as $1+1+...+1$. Since any cube is contained in the 1 cube, the above constraints are essentially ineffective.

We will present one method of implementing the branch and bound algorithm. We give some very effective implementation heuristics and techniques used. First we look at **inexact division**.

Given a logic function f with n_f cubes and a divisor d with n_d cubes, we want to find a largest (in terms of the number of cubes) quotient q. The resulting divisor d' does not have to be the same as d.

The basic steps are:

a. Layout a matrix M with n_d rows and $\frac{n_f}{n_d}$ columns. Each row corresponds to a cube in d.

b. Fill each cell of M with a cube in f, in row-column order. At each position, compute the lower bound l^{ij} and the two upper bounds u_1^{ij} and u_2^{ij}. The cubes of f cannot be repeated. Semi-algebraic methods do not allow the repition of cubes. The cubes already used are marked unavailable for later positions.
c. If an available cube c satisfies the bounds and the constraint that $c \subseteq d_i$, mark it used and proceed in row-column order to the next cell. Here d_i is a cube in d corresponding to row i in M.
d. If all available cubes have been tried for a position, record the best (measured as the number of columns) arrangement of cubes so far as *best_matrix*. Backtrack to try a different positional arrangement of the cubes. A matrix found later with a larger number of columns than the previously recorded *best_matrix* will replace the previous *best_matrix.*
e. Repeat d until all combinations of cube arrangement have been tried.
f. Compute the supercubes of the rows of the final *best_matrix* to form the divisor d', and the supercubes of the columns to form the quotient q.

The additional condition $c \subseteq d_i$, for any cube $c \in f$ in row i, is not necessary for the correctness of the result but it helps to trim down the satisfying cubes and to speed up the process. Because of this condition, $d' \subseteq d$.

For **exact division**, everything is the same except that in the end, the supercubes of the rows of the satisfiability matrix have to be exactly as the given d. This constraint gives rise to the possibility that the final satisfiability matrix is not the largest one.

Althugh the worst case for branch and bound algorithm, unfortunately, is exponential, our implementation incorporates a few trimming techniques that speed up the run time considerably:

1. The rows and columns of a satisfiability-matrix can be permuted arbitrarily, we have restricted the search such that the cubes in the first row and the first column are ordered. This reduces the search space by $m!n!$ terms for a $m \times n$ matrix.
2. The additional condition $c \subseteq d_i$ rules out (for row i) the cubes in f that are not contained in d_i.

3. Theorem 2.3 allows early backtracking of the search.
   a. Suppose at position (i, j), the current *best_matrix* has k $(k>j)$ columns and that fewer than $k-j+2$ available cubes satisfy u_1^{ij}. We will not get a better matrix with at least $k+1$ columns because we would need at least $k-j+2$ available cubes satisfying u_1^{ij} to fill the cells (i, n), $k+1 \geq n \geq j$. We therefore backtrack.
   b. Similarly, we can backtrack if at position (i, j) if we have a *best_matrix* with k $(k > i)$ rows, and fewer than $(k - i + 2)$ available satisfy u_2^{ij}.
4. Theorem 2.4 can be used to prune the cubes in several ways:
   a. All cubes with $N_{ncf} < m$ are pruned in the beginning, where m is the number of cubes in the divisor
   b. Order the cubes in decreasing order of N_{ncf} in the hope that a large satisfiability matrix could be found early in the process to serve as a good bound for subsequent search.
   c. Once we have discovered a satisfiability matrix with n columns, we prune the cubes with $N_{ncf} \leq n$. This could avoid the situation of later having to explore satisfiability matrices with $k \leq n$ columns. This also leads to non-chronological backtracking: we jump directly back to the earliest position that currently has a cube with $N_{ncf} \leq n$. We will replace this cube with the next available cube that has $N_{ncf} > n$. This is because any cube in the matrix with $N_{ncf} \leq n$ is not going to lead us to a satisfiability matrix with more than n columns.
   d. Part of N_{ncf} of a cube may come from cubes that are already pruned. We can therefore update N_{ncf} (subtract the contribution of N_{ncf} that comes from pruned cubes) as cubes are thrown out of the candidate pool. As the N_{ncf} of a cube c is reduced this way, we may prune it too, once it falls below n, the largest number of columns seen so far.

The heuristics are less effective for **exact division**. We have to check at each position whether the row supercubes are equal to the given divisor. The upper bounds heuristics 3 and 4 is now the column number of the largest matrix that has row

supercubes equal to the divisor. This number is generally much smaller than that of the largest matrix for **inexact division**. For this reason, even though **exact division** has more constraints, the satisfiability-matrix based method is slower than **inexact division**.

The implementation of the satisfibility-matrix with the above heuristics shows that the run time exceeds seconds when the number of cubes in f gets close to *35*. In Chapter 3, we present the utilization of this implementation in algebraic operation such as *factor*, *decomp* and *resub*. Some experimental results are presented in Chapter 5.

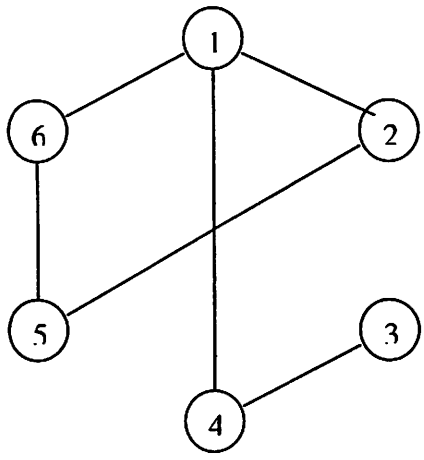
2.3 The maximum graph-matching method

We mentioned that the satisfiability-matrix based method is slower for **exact division**, even though **exact division** has more constraints than **inexact division**. This is particularly undesirable in *fast exact*, a key operation in multi-level logic minimization which finds common divisors. In our current implementation of *fast exact*, the candidate kernels are divided into all nodes to evaluate their values. This process involves a large number of exact divisions and we desire a faster process.

We have developed a second semi-algebraic method that applies only to **exact division** and that has a run time of $O(n^3)$. We call this method the maximum graph matching method because it can be reduced to this problem. For convenience, we restate the definition of the maximum graph-matching problem:

Definition 2.16 (Graph Matching Problem) *Given a graph G, the maximum graph matching problem is finding the maximum number of connected pairs of vertices, under the constraint that no vertex is allowed to repeat.*

In the following graph, a solution to the graph-matching problem is the set of node pairs *{(1,2), (3,4), (5,6)}*



The maximum graph matching method stems from the observation that, given a logic function f and a divisor d, there is a set of candidate quotient cubes S^{d_i} associated with each cube of d. Unlike the binary case, each quotient cube is not unique.

Theorem 2.5 (Multi-valued Division) *Given a cube d_i of the divisor and a cube c_j of f, the associated quotient cube is given by*

$$c_j \subseteq k_{ij} \subseteq \sigma(c_j + \tilde{d}_i)$$

Here $\tilde{d}_i$ denotes the complement cube of d_i. We call cubes $k_{i_1 j_1}$ and $k_{i_2 j_2}$ **compatible** if they can be made the same within their ranges allowed by the above inequality.

Like weak division in the binary case [BHV90], the sets are "intersected" to find the maximum subset of quotient cubes common to all $\{ S^{d_i} \}$. A cube k is in this intersection if there exists a set $\{ j_i, i = 1,...,|d| \}$ such that k satisfies the above inequalities for all k_{ij_i}.

Since the above inequalities can be represented as a cube in a larger space, common intersection is equivalent to pair-wise intersection, i.e., a common cube exists among $\{ k_{ij_i} \mid i = 1,...,|d| \}$ if and only if each pair $k_{i_1 j_{i_1}}$ $k_{i_2 j_{i_2}}$ intersects. The algorithm allows a cube c_j to participate in several of the S^{d_i}, i.e., we allow for duplication of cubes of f. The set of all intersecting cubes $\{k\}$ is the resulting quotient.

If the divisor d is restricted to have only two cubes, the above problem reduces to finding the largest number of cube pairs between S^{d_1} and S^{d_2}. If the cost function is cube number, there is no benefit in duplication of cubes of f. For this reason, in the current implementation, duplication of cubes of f is not allowed. This further simplifies the problem to the maximum graph matching problem by the following mapping: each cube is a vertex and there is an edge between c_1 from S^{d_1} and c_2 from S^{d_2} if and only if c_1 and c_2 are compatible. To formally state the problem:

Problem definition 2.3 (exact division) *Given a set of cubes f and a set of divisor cubes* $\{d_i, 1<i<n\}$, *find the largest set of quotient cubes* $\{q_j, 1<j<m\}$.

We give the method for finding a solution for $n = 2$ $(d = d_1 + d_2)$ which reduces to the maximum graph matching problem.

1. Find all candidate cubes for d_1 and d_2: any cube c in d_i is a candidate cube for d_i:

   d_1: $c_{11}, c_{12}, \ldots, c_{1n_1}, c_{1i} \in d_1, 1 \le n_1 \le m$

   d_2: $c_{21}, c_{22}, \ldots, c_{2n_2}, c_{2i} \in d_2, 1 \le n_2 \le m$

2. For each candidate cube c_{ij}, compute a candidate quotient cube q_{ij} that has a lower and an upper bound on the values it can take (as in Theorem 2.5):

$$d_i c_{ij} \subseteq q_{ij} \subseteq \sigma(c_{ij} + \tilde{d}_i)$$

3. Express q_{ij} (called a flexible cube) in "positional" notation where we denote c^{kl} as the value l of variable k in cube c:

$$q_{ij}^{kl} \begin{cases} 0, (d_i c_{ij})^{kl} = 1 \\ 1, (d_i c_{ij})^{kl} = 1 \\ 2, otherwise \end{cases}$$

   Here 0 means the value must not be in the cube and 1 means the value must be in the cube. So we have:

   d_1: $q_{11}, q_{12}, \ldots, q_{n_1}, 1 \le n_1 \le m$

   d_2: $q_{21}, q_{22}, \ldots, q_{n_2}, 1 \le n_2 \le m$

The candidate quotient cubes for d_1 and d_2 form a bi-partition B_1 and B_2. We want to find a maximum compatibility of it. A quotient cube q_1 from B_1 is compatible with a cube q_2 from B_2 if and only if $k = q_1 * q_2 \neq NULL$ where $k = q_1 * q_2$ is defined as:

$$k^{ij} = q_1^{ij} * q_2^{ij} = 1 \Leftrightarrow \begin{cases} q_1^{ij} = 1, q_2^{ij} = 1, or \\ q_1^{ij} = 1, q_2^{ij} = 2, or \\ q_1^{ij} = 2, q_2^{ij} = 1 \end{cases}$$

$$k^{ij} = q_1^{ij} * q_2^{ij} = 0 \Leftrightarrow \begin{cases} q_1^{ij} = 0, q_2^{ij} = 0, or \\ q_1^{ij} = 0, q_2^{ij} = 2, or \\ q_1^{ij} = 2, q_2^{ij} = 0, or \\ q_1^{ij} = 2, q_2^{ij} = 2 \end{cases}$$

$$k^{ij} = q_1^{ij} * q_2^{ij} = NULL, otherwise$$

Example 2.13

	X_1	X_2	X_3
q_1	*1111*	*1020*	*1102*
q_2	*1111*	*1000*	*1122*
k	*1111*	*1000*	*1102*

So $k = X_2^{\{0\}} X_3^{\{0,1\}}$. Note we want $k_{ij}=0$ if $q_1^{ij} = q_2^{ij} = 2$ in order to minimize the number of values in the selected quotient cube. In fact, this is where the flexibility comes into play with multi-valued division. We could have also obtained $k = X_2^{\{0\}} X_3^{\{0,1,3\}}$ as a correct solution. It is not clear though whether literal $X_3^{\{0,1\}}$ is simpler than literal $X_3^{\{0,1,3\}}$. It will largely depend on what cost function is being used in selecting from a "flexible" set of solutions.

4. Reduce the problem instance to a bipartite graph: each candidate quotient cube is a vertex; there is an edge from $q_1 \in B_1$ to $q_2 \in B_2$, if and only if q_1 is compatible with q_2. There are no edges between cubes in B_1 or between cubes in B_2.
5. Solve the maximum graph-matching problem.

Note that since a solution for the maximum graph matching does not allow repetition of vertices, there is no duplication of cubes in the final solution. The complexity of the maximum graph-matching problem is $O(n^3)$.

Example 2.14 $f = a^{\{0,1,2\}}c^{\{3\}}+b^{\{1,2,3\}}c^{\{3\}}+a^{\{0\}}c^{\{1\}}+a^{\{0\}}b^{\{1,2,3\}}c^{\{0\}}$

$d = c^{\{3\}}+a^{\{0\}}c^{\{0,1,2\}}$

1. Candidate cubes

$c^{\{3\}}$:	$a^{\{0,1,2\}}c^{\{3\}}$	$b^{\{1,2,3\}}c^{\{3\}}$
$a^{\{0\}}c^{\{0,1,2\}}$:	$a^{\{0\}}c^{\{1\}}$	$a^{\{0\}}b^{\{1,2,3\}}c^{\{0\}}$

2. Corresponding quotient cubes

$c^{\{3\}}$:	$a^{\{0,1,2\}}c^{S_1}$	$b^{\{1,2,3\}}c^{S_2}$
$a^{\{0\}}c^{\{0,1,2\}}$:	$a^{S_3}c^{S_4}$	$a^{S_5}b^{\{1,2,3\}}c^{S_6}$

Here

$$\{3\} \subseteq S_1 \subseteq \{0,1,2,3\},\ \{3\} \subseteq S_2 \subseteq \{0,1,2,3\},\ \{0\} \subseteq S_3 \subseteq \{0,1,2,3\},$$

$$\{1\} \subseteq S_4 \subseteq \{1,3\},\ \{0\} \subseteq S_5 \subseteq \{0,1,2,3\},\ \{0\} \subseteq S_6 \subseteq \{0,3\}$$

3. Find the maximum compatible pairs of quotient cubes in the two sets:

$c^{\{3\}}$:	$a^{\{0,1,2\}}c^{\{1,3\}}$	$b^{\{1,2,3\}}c^{\{0,3\}}$
$a^{\{0\}}c^{\{0,1,2\}}$:	$a^{\{0,1,2\}}c^{\{1,3\}}$	$b^{\{1,2,3\}}c^{\{0,3\}}$

So the quotient cubes are $a^{\{0,1,2\}}c^{\{1,3\}}$ and $b^{\{1,2,3\}}c^{\{0,3\}}$, with $S_1 = S_4 = \{1,3\}$, $S_2 = S_6 = \{0,3\}$, $S_3 = \{0,1,2\}$ and $S_5 = \{0,1,2,3\}$. This is what we would get by using the satisfiability-matrix method on **exact division**.

The satisfiability-matrix based method and the maximum graph matching method form the core algorithms used in a set of MV algebraic minimization operations implemented in MVSIS. We have seen powerful minimization results from MVSIS, both in binary examples and multi-valued examples. The experimental results are presented in Chapter 5. The next chapter is devoted to this set of MV multi-level minimization operations, namely, *fx*, *factor*, *decomp*, *resub*, *collapse*, *eliminate* and *merge*.

Chapter 3
Multi-valued multi-level minimization in MVSIS

As pointed out in Chapter 2, semi-algebraic methods are key algorithms in several important multi-level operations. In this chapter, we describe these operations in detail. We extend the previous work of multi-level operations in the binary domain to the MV domain. The basic concepts remain unchanged. The algorithms used, however, due to the combinational value explosion of MV literals, have to be either modified or completely replaced.

3.1 Preliminary

3.1.1 Overview

A multi-valued combinational logic network, or MV-network, is a network of nodes; each node represents a multi-valued function, or MV-function, with a single multi-valued output and multi-valued inputs. There is a directed edge from node i to node j, if the function at node j explicitly depends on the output variable at node i. The optimization problem of such a MV-network is to find an equivalent optimal MV-network. The optimization criterion is a function of the total number of nodes and the size of the MV-function contained in each node. This exact cost function will depend on the final target of implementation.

Together with the node-minimization methods developed in [JB00], the algebraic methods form the core of the current version of a multi-valued logic synthesis system called MVSIS. MVSIS is specifically targeted for multi-valued circuits and supports a design methodology that allows the designer to search a larger solution space than was

possible previously. In the current version of MVSIS, we have developed and included techniques for combinational optimization of an MV-network. Like SIS[SSL+92], MVSIS is an iteractive tool. When applied to purely binary networks, it behaves almost exactly like the technology independent part of SIS. In the sequel, the main multi-level minimization components of MVSIS are described.

3.1.2 The default value

Before attempting the operations used in MVSIS, it is useful to clarify the concept of "default value" and how it is handled in MVSIS. In the multi-valued domain, the concept of "complement" is generalized to "default value". For each node, one of the value functions is selected as the default value. As in SIS where the complement of a node is not stored, we do not store the cover for the default value in a node. We compute it on the fly when we need it. For example, if the output of a node is used in a fanout in the form of its default value, and the node is eliminated or collapsed into that node, the default value function must be computed to effect the elimination. The values of the nodes and statistics of the network are based only on the primary values and not the default value. However, there is one command *reset_default* that looks at each node and chooses a default value for it based on the costs of the value functions. For example, if the cost function is the number of cubes, each value function will be minimized with *simplify*, and the default value will be chosen to be the value whose function has the most cubes. In this way, we somewhat take into consideration the output phase assignment problem.

3.2 fx

The command *fx* looks at all the nodes in the network and tries to extract good common factors and create new nodes in the network, re-expressing other nodes in terms of these newly introduced nodes. It is one of the transforms used to break down large functions into smaller pieces. It has two options, *–q* and *–g*. The first option corresponds to iterative calls to *fxl -q* (described below) until no kernels can be identified for further

improvement in the overall network; the second option corresponds to an iteration of *fx1 –g*.

The *–q* option is similar to that used in SIS where we use the two-cube divisor method [BHV90] to extract kernels. The two-cube divisors of an expression is the set

$$\tau(f)=\{common\_cube\_free(c_i,c_j)|c_i,\ c_j\in f\}$$

Notice we emphasize the fact that we are using the *common_cube_free*, not the *super_cube_free* kernel of c_i and c_j, for the reason described in Chapter 2. Note also that there are at most $\frac{|f|(|f|-1)}{2}$ two-cube divisors for an expression f, where $|f|$ is the number of cubes in f. It was demonstrated in [BM82] that the use of such divisors in an extract process leads to little loss of optimality over the use of kernels but is more efficient. Their method processes two-cubes as well as single-cube divisors at the same time by using the rectangle-covering technique [BHV90]. This is the method implemented as *fx* in SIS. In MVSIS, if the cost function is cube number, only two-cube divisors are considered. All such two-cube divisors are extracted this way from all node functions and kept in a hash table.

The *–g* option is different from the SIS *fx –g* operation. Instead of finding all kernels as candidate divisors, *fx –g* in MVSIS only finds some additional divisors that may not be discovered by *fx –q*. In the binary domain, the following is true: all terms with cube number ≥ 2 in the factored form of f can be found in the set $\tau(f)$. In the MV domain, however, this does not hold any more, again due to the flexibility in semi-algebraic division. As an example, consider

$$\begin{aligned} f &= (a^{\{0,1\}}b^{\{2,3,4\}}+a^{\{0,2\}}b^{\{0,4\}})(a^{\{0,1\}}b^{\{0,2\}}+a^{\{1,2\}}b^{\{3,4\}}) \\ &= a^{\{0,1\}}b^{\{2\}}+a^{\{0\}}b^{\{0\}}+a^{\{1\}}b^{\{3,4\}}+a^{\{2\}}b^{\{4\}} \end{aligned}$$

so $a^{\{0,1\}}b^{\{2,3,4\}}+a^{\{0,2\}}b^{\{0,4\}}$ is a valid divisor but in not an element of $\tau(f)$. The *fx –g* method generates an additional set (in addition to $\tau(f)$) of candidate double cube divisors, one for each function in each node, by factoring each node in the network. Some of these divisors cannot be found by *fx –q*. This method may take more CPU time if the number of cubes in the nodes is large because factorization in the MV domain is expensive. There is a timer implemented in the factorization (using the satisfiability-matrix branch and bound searching) algorithm that will terminate the search if the time limit is exceeded. The time

limit can be set by the user by invoking the command *set time_limit n*, where *n* (in *seconds*) can be any real number. *fx –g* always performs better than *fx –q* because *fx –g* calls *fx –q* first, and if there is no more kernels with positive merit to be found with *fx –q*, it uses the factorization method to find additional divisors.

In *fxl –q*, the kernels are extracted and kept in a hash table. Each of the kernels is divided into all the nodes in the network. Each kernel is given a figure of merit by keeping track of the saving accumulated through this division process. The divisor with the greatest merit is chosen, implemented as a separate node with two values and substituted into all the nodes in which it appears. The substitution is performed using the fast exact semi-algebraic division. The two-cube candidate divisors make good use of the maximum graph matching method, which can only be applied to divisors with exactly two cubes in the current implementation. After the function of the new node is substituted into other nodes, its complement is also tried for division in case there is extra saving there. Even though the kernels are two cube divisors, their complement may not be. Single cube covers can be trivially dealt with. Covers with more than two cubes have to apply the satisfiability-matrix based method.

A faster way to implement *fxl –q* is to estimate the merit of the kernel by the number of its hits in the hash table. The number of hits of a kernel in the hash table is directly related to the number of its appearances in the node functions. However, this number is not exact. Sometimes a kernel *k* is divisible by a node *np* but *np* may not generate a hit for *k* because *k* does not appear in $\pi(np)$, the *common_cube_free* kernels of *np*. This is the same reason that accounts for the difference between *fxl –q* and *fxl –g*. This method, however, avoids having to divide each kernel into all nodes. This method is in the process of being implemented and we expect improvement in the run time.

In this process, binary and MV variables are treated uniformly. However, for the MV variables, we have the additional problem that both the cube-free kernels and the division itself are not unique. The satisfiability-matrix method and the maximum graph matching method only take care of flexibility in the division. To identify common sub-expressions optimally, however, we would need a method that uses the redundant values in the kernels as well. Currently we have used the minimal form of the cube-free

expressions (the one with the least number of values). This is an area that is still under investigation.

In *fxl –g*, first *fxl –q* is called. If *fxl –q* cannot find any kernels with positive merit, each function at each node is factored and the best two-cube kernels found this way are kept at each node in an auxiliary field. These kernels are then each divided into all nodes in the network to collect a value. The one with the largest value is extracted into a new node and re-substituted into all other nodes. Therefore *fxl –g* always performs no worse than *fxl –q*. In Figure 3.1, we show an example of *fx* where *fx –q* does not discover the kernels while *fx –g* does.

```
mvsis> print
{q}{1} = z{2}w{3}+z{3}w{2}+z{0}w{1}+z{1}w{0}
{q}{2} = z{3}w{3}+z{0}w{2}+z{1}w{1}+z{2}w{0}
{q}{3} = z{0}w{3}+z{1}w{2}+z{2}w{1}+z{3}w{0}
w{1} = x{2}y{3}+x{3}y{2}+x{0}y{1}+x{1}y{0}
w{2} = x{3}y{3}+x{0}y{2}+x{1}y{1}+x{2}y{0}
w{3} = x{0}y{3}+x{1}y{2}+x{2}y{1}+x{3}y{0}

mvsis> print_stats -f

adder_mod4:  2 nodes,  1 POs,  24 cubes(sop),  48 lits(sop),  48 lits(fact.)

mvsis> fx -q

mvsis> print_stats -f

adder_mod4:  2 nodes,  1 POs,  24 cubes(sop),  48 lits(sop),  48 lits(fact.)

mvsis> fx -g

mvsis> print_stats -f

adder_mod4:  4 nodes,  1 POs,  16 cubes(sop),  44 lits(sop),  38 lits(fact.)

mvsis> print

{q}{1} = z{2,3}w{2,3}new0{1}+z{0,1}w{0,1}new0{1}
{q}{2} = z{0,3}w{2,3}new0{0}+z{1,2}w{0,1}new0{0}
{q}{3} = z{0,1}w{2,3}new0{1}+z{2,3}w{0,1}new0{1}
w{1} = x{2,3}y{2,3}new1{1}+x{0,1}y{0,1}new1{1}
w{2} = x{0,3}y{2,3}new1{0}+x{1,2}y{0,1}new1{0}
w{3} = x{0,1}y{2,3}new1{1}+x{2,3}y{0,1}new1{1}
new0{1} = z{0,2}w{1,3}+z{1,3}w{0,2}
new1{1} = x{0,2}y{1,3}+x{1,3}y{0,2}
```

Figure 3.1: An example of *fx –g* where *fx –q* does not save

Figure 3.1 shows a list of operations in MVSIS. The cost function is the cube cost function. As we can see, *fx –q* does not change the network, while *fx –q* discovers the kernels new0 and new1 and saves 8 cubes.

3.3 factor

In factorization, we use **inexact division** and the divisor *d* is used only to focus and limit the search space for a satisfiable matrix. *d* can be set to *1+...+1* to allow the maximum freedom. We can also select candidate divisors by making pairs of cubes cube-free, and use them to limit the search space for a satifisable matrix. Additionally, just as in the binary case, it is not necessary to get the best result at first; in the second step, the quotient obtained in the first step can be used as the divisor in a second division, leading possibly to a better factorization. This is the basis of quick factor (QF), used in SIS [SSL+92], where the first divisor is chosen to be a level-zero kernel.

In the current implementation of MVSIS, *d* is set to *1+1* in the first step. The quotient *q* obtained in the first step is then made *common_cube_free* ($\tilde{q}$) and used as the divisor in the second step. Note again we make a distinction between *super_cube_free* and *common_cube_free* for the reason discussed earlier (See the definition for **Common Cube**). The operation in the search for a satisfiability-matrix is **inexact division** where the row expression obtained from the satisfiable rectangle need not equal $\tilde{q}$ from the first step. The divisor and quotient obtained this way are factored recursively to obtain the final result. Here is a brief description of the algorithm, which has a similar structure as in SIS:

```
GFACTOR(F) {
    D=1+1;
    If (!IS_COMMON_CUBE_FREE(F)) {
        (D,Q)=MAKE_COMMO_CUBE_FREE(F);
        // D is the common cube, Q is common_cube_free part of F
        return GFACTOR(Q)D;
    }
    (D,Q,R)=MV_FACTOR(F,D);
    // MV_FACTOR returns NULL if F is not factorizable
    If ((D,Q,R) == NULL) return NULL;
    return GFACTOR(Q)GFACTOR(D)+GFACTOR(R)
}
```

Figure 3.2: *GFACTOR* Procedure

Because the satisfiability-matrix based method only finds quotients with at least 2 cubes, single-cube factorization has to be done in a separate routine called *MAKE_FIRST_PAIR_COMMON_CUBE_FREE( )*:

```
MV_FACTOR(F,D) {
  (Q,D,R)=SAT_MATRIX_INEXACT(F,D);
  If ((Q,D,R)==NULL) {
  // SAT_MATRIX_INEXACT( ) returns NULL if it cannot find a quotient with ≥2 cubes
    (Q,D,R) = MAKE_FIRST_PAIR_COMMON_CUBE_FREE(F);
  }
  return (Q,D,R);
}
```

Figure 3.3: *MV_FACTOR* Procedure

SAT_MATRIX_INEXACT(F, D) performs a first round search for a largest satisfiability-matrix with $d = 1+1$. If a quotient q is found, it then performs a second round search using q as the divisor for a largest satisfiability-matrix. The quotient q' obtained in the second round is returned as Q, the divisor d' (d' may not be the same as q) obtained in the second round D and the rest of F is returned as R.

MAKE_FIRST_PAIR_COMMON_CUBE_FREE(F) finds the first pair of cubes in F that's not *common_cube_free*. It extracts the common-cube as D, the *common_cube_free* part as Q and the rest as R. If all pairs of F are common-cube-free, it returns *NULL*.

Because *SAT_MATRIX_INEXACT* uses the branch and bound algorithm, it may take a large amount of CPU time if the number of cubes in the function is large. A timer has been implemented which controls the maximum amount of time that can be spent in factoring a single function. This is especially useful when the factoring is only being used to estimate the value of a node or produce a readable output.

Below we show several results of the factorization algorithm applied to different functions. Only the final resulting factorizations are given. Again the initial covers are the covers obtained by multiplying out the expressions.

$$x^{[1]} = c^{[0,1]}d^{[0,1]}(b^{[0]}(c^{[0]}d^{[0]}e^{[1]}+e^{[0]})+b^{[1]}c^{[0]}d^{[0]}e^{[0]})+a^{[0]}$$

$$y^{[1]} = ((b^{[1]}e^{[2]}+b^{[2]})(d^{[2]}+c^{[2]})+(d^{[1]}+c^{[1]})b^{[2]}e^{[2]})a^{[2]}e^{[1,2]}$$

Figure 3.4 Examples of *factor*

3.4 decomp

The command *decomp* does a complete factoring of each node, but instead of creating a factored form for each, decomposes the node according to its factorization. Each node is decomposed into smaller nodes that cannot be factored further. More intermediate nodes are produced this way. gx1 or fx1 may not have produced such intermediate nodes, since they do not necessarily gain a positive merit by substitution into

other nodes. However, such nodes can always be collapsed back by the command *eliminate*. After *decomp*, *resub* (see below) should be executed to take advantage of the newly produced nodes, followed by *sweep* to eliminate duplicate factors. Then *eliminate* can be done to clean up the network. *eliminate* may reproduce some of the nodes previously decomposed. The decomposition process is to make the internal factors of a node available to other nodes, thus creating a better candidate pool of nodes for *resub*. One such example is given below:

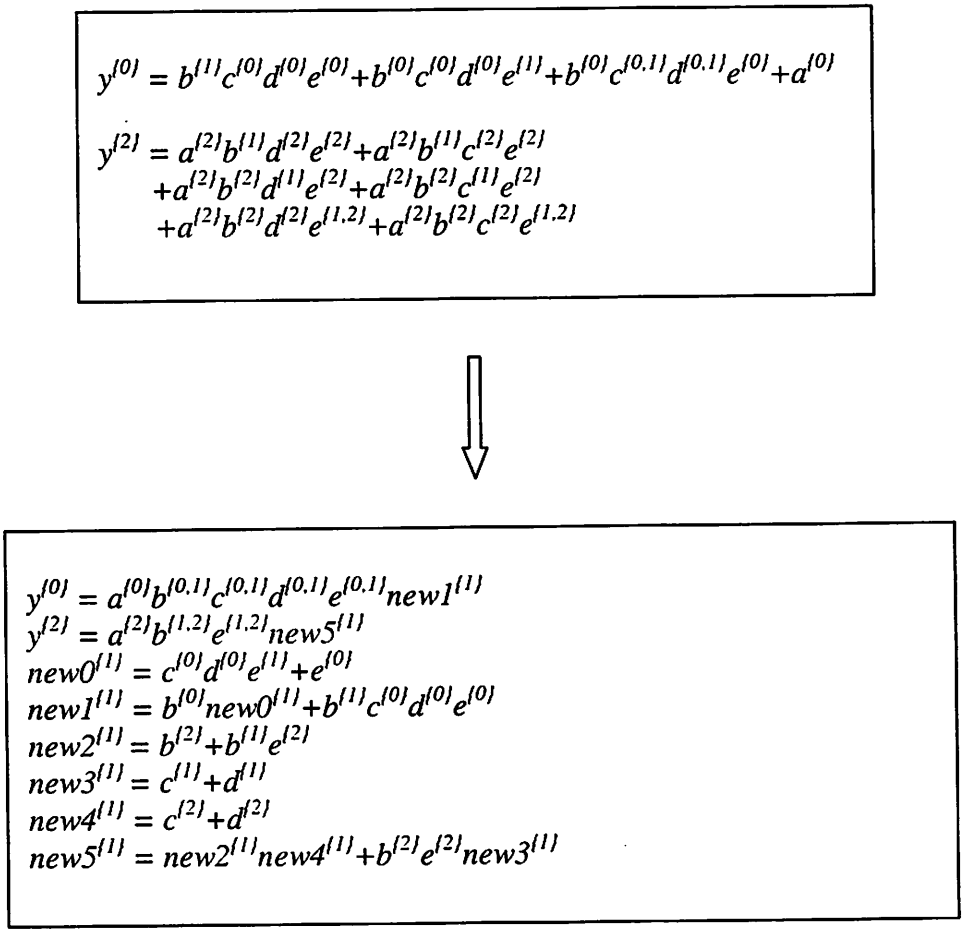


Figure 3.5: Examples of *decomp*

3.5 *resub*

Re-substitution of one node into another is performed in MVSIS by using *resub*. *resub* takes a list of nodes as argument for re-substitution into all nodes in the network. If

no argument is given, all nodes are considered. This uses the two new methods of "exact" semi-algebraic division, developed for multi-valued logic in Chapter 3. Again, if the divisor is a two-cube divisor, then the fast method based on maximum graph matching is used. Otherwise, the slower branch and bound method based on a satisfiability matrix is used. Because the default value is selected to be the most costly value function by the function *reset_default*, sometimes it is beneficial not to consider the default value in re-substitution for speed considerations. The option *–d* in resub enables the option to also consider default value.

If *resub* is invoked with no argument, all pairs of nodes have to be considered in theory, implying as many as $(mn)^2$ divisions, where m is the average number of functions in each node and n is the number of nodes in the network. As in SIS, filters are used to circumvent most of these divisions. The function f_j is not an algebraic divisor of f_i if:

1) f_j contains a variable not in f_i,

2) f_j has more terms than f_i,

3) for any variable, the total number of times its literals occur in f_j exceeds the total number of times the corresponding *sub_literals* occur in f_i.

4) f_i is in the transitive fan-in of f_j

Definition 3.1 (sub_literal) *a literal l_1 is a sub_literal of another literal l_2 if $l_1 \subseteq l_2$.*

For instance, the literal $a^{\{0,2\}}$ is a *sub_literal* of the literals $a^{\{0,2\}}$ and $a^{\{0,1,2\}}$, assuming a takes on *3* values. The set of *sub_literals* of $a^{\{0,2\}}$ is $\{a^{\{0\}}, a^{\{2\}}, a^{\{0,2\}}\}$.

Note compared with the same set of filters in SIS, we have replaced "literal" in filter 1) in SIS with "variable" in MVSIS. Also we have replaced filter 3) in SIS "for any literal, the number of times it occurs in f_j exceeds that in f_j", with the above filter 3) in MVSIS. This is because we have relaxed algebraic methods to semi-algebraic methods for MV. We illustrate the new modifications with an example:

$$f_i = a^{\{0,1\}}b^{\{2\}} + a^{\{0\}}b^{\{0\}} + a^{\{1\}}b^{\{3,4\}} + a^{\{2\}}b^{\{4\}}$$
$$= f_j(a^{\{0,1\}}b^{\{0,2\}} + a^{\{1,2\}}b^{\{3,4\}})$$
$$f_j = a^{\{0,1\}}b^{\{2,3,4\}} + a^{\{0,2\}}b^{\{0,4\}}$$

Note f_j contains a literal $a^{[0,2]}$ not in f_i, but filter 1) fails the check because we are now looking at variables. The old filter 3) would hold also because of this. With the new filter 3), however, the set of literals of variable a in f_j is $L = \{a^{[0,1]}, a^{[0,2]}\}$. The total number they occur in f_j is 2. The corresponding set of *sub_literals* of L is $L_s = \{a^{[0,1]}, a^{[0]}, a^{[1]}, a^{[0,2]}, a^{[2]}\}$. The total number of times they occur in f_i is 4. Therefore filter 3) also fails the check for variable a. Filter 3) is based on the observation that any literals in f_j cannot be expanded in f_i when multiplied out by a potential quotient.

With these filters and the fast two-cube maximum graph matching based division technique, the *resub* operation is fairly fast in MVSIS. Some of the experimental results are presented in Chapter 5.

Although the operations *collapse* and *eliminate* are not directly related to algebraic methods, they are the inverse of *resub* and play an important role in MV-network manipulations. Often, reconstructing the network by eliminating most of the current structure leads to the discovery of better common sub-expressions. Again, an overall good solution in minimizing the network is the collective work of many kinds of operations, often through a script.

3.6 *collapse*

"Collapsing" is the inverse operation of "substitution". If no arguments are given, this command collapses the entire multi-level network so that each output is in terms of the primary inputs only. All intermediate nodes are eliminated. If only one argument is given as the node name, then this node alone is collapsed and expressed in primary inputs only. All other nodes are unchanged. If two arguments are given, which are names of nodes, a node and one of its fanins, then the fanin node is collapsed into the fanout node so that the fanout node is not dependent on the fanin node any more.

In general, the fanout node gets enlarged after collapsing and we want to minimize the fanout node after the operation. The weakest node simplification is SNOCOMP where only a single iteration is carried out in espresso-MV and the reduced offset is used to compute the complement. For a network with a large number of nodes, however, collapsing nodes close to the primary outputs may involve a large number of

collapsing of node pairs. As the node function gets larger in this process, simplifying the node each time is collectively costly and could result in a large amount of CPU time. Filters should be used and techniques weaker than *node_simplify* should be exploited. If fanin g is to be collapsed into fanout f, as in the binary case, we use the following algorithm for collapsing g into each value function f^i of f:

```
collapse(f^i, g) {
    for each value j of node g {
        ( f^i_{g^j}, r) = mv_alg_cofactor(f^i, g^j);
        // f^i_{g^j} = multi-valued algebraic cofactor of the i^th function f^i of f with respect to the
        // literal g^(j). r = terms of f^i that do not contain the variable g
    }
    result = r;
    For each j {
        If g^j is constant 1, then result = cover_join(result, f^i_{g^j} );
        else if g^j is constant 0, then do nothing;
        else {
            temp = cover_intersect(g^j, f^i_{g^j} );
            result = cover_join(result, temp);
        }
    }
    return result;
}
```

Figure 3.6: *collapse* node pair procedure

Definition 3.2 (MV_alg_cofactor) The multi-valued algebraic cofactor $f^i_{g^j}$ of an MV-function f is the algebraic cofactor of the binary function f^i (the i^{th} value function of f) of f with respect to the literal $g^{(j)}$. The algebraic cofactor of a binary function f^i with respect to the literal $g^{(j)}$ is the set of cubes in f^i in which the literal $g^{(j)}$ appears.

Example 3.1

$$f^{[0]} = a^{[1]}g^{[0,2]} + a^{[0]}g^{[2]} + a^{[0,1]} + a^{[2]}g^{[3]}$$

$$f^i_{g^0} = a^{[1]}g^{[0,2]}$$

$$f^i_{g^1} = NULL$$

$$f^i_{g^2} = a^{[1]}g^{[0,2]} + a^{[0]}g^{[2]}$$

$$f^i_{g^3} = a^{[2]}g^{[3]}$$

$$r = a^{[0,1]}$$

Note $f^i_{g_j} + r$ preserves the onset of f^i whenever $g = j$, i.e., $g^j \cdot (f^i_{g^j} + r) = g^j \cdot f^i$. Note also that *MV_alg_cofactor* is a modified version of a special case of the multi-valued cofactor [BK99], where the set of cubes that partitions the input space of f is the set of single-valued literals of the input variable g. The above collapse algorithm is can be viewed as based on a modified Multi-valued Shannon Expansion Theorem [BK99], $f^i = \sum_{j=1}^{|P|} g^j f^i_{g^j} + r$. This algorithm is a generalization of the algorithm used in SIS node collapse where each node has only one value function.

The filters are for special cases where $g^{[j]} = 1$ *or* or $g^{[j]} = 0$. The sub-routines *cover_join* and *cover_intersect* apply techniques that reduce the cover size but do not have to call *node_simplify*. These techniques include eliminating cubes in the cover that are contained in other cubes, and merging cubes that are distance *1* apart. In some of the examples that we experimented with, the above method shows superior results in terms of run time, compared to the alternative method that calls *node_simplify* after all values have been collapsed. These examples are often the ones with a large number of nodes, where collapsing a single node close to the primary outputs involves many node pair collapses.

3.7 eliminate

The command *eliminate* eliminates all the nodes in the network whose value does not exceed a specified threshold. The value of a node represents the increase in cost in the network if the node is eliminated. If the value is less than (or equal to) the specified threshold, the node will be eliminated by collapsing the node into each of its fanouts. Of course, a primary input or a primary output will not be eliminated. The command iterates, since eliminating one node may affect the value of other nodes. The iteration continues until all remaining nodes have a value greater than the threshold.

If the cost function is the number of cubes in the SOP form, a fake *collapse* (the actual network is not changed) is done to obtain the covers of the fanouts after eliminating the node. The covers are minimized to accurately reflect the savings. The difference in cost is then computed before and after the fake *collapse*, and assigned as the value of the node. This method is exact but may be expensive if the fake collapsed node is large. Alternatively, faster methods can be tried to only give an estimate of the cost.

If the cost function is the number of literals in the factored form, evaluating the node value is costly since it has to invoke the satisfiability-matrix based branch and bound algorithm to perform the factorization. A heuristics has been developed to estimate the node value without factorization. Suppose the node being evaluated is *np*, then:

$$value(np) = \sum_{i \in FO(np),\, j \in range(i),\, k \in range(np)} l_{ijk} f_k \; - \sum_{i \in FO(np),\, j \in range(i),\, k \in range(np)} l_{ijk} \; - \sum_k f_k$$

Here i is the i^{th} fanout of *np*, j is the j^{th} value of the fanout and k is the k^{th} value of *np*. l_{ijk} is *1* if the value k of *np* appears in the literals of the j^{th} value function of its i^{th} fanout and *0* otherwise. f_k is the number of distinct literals in the k^{th} value function of *np* in SOP form. For instance, $f_1 = 5$ in the function

$$f^{[1]} = a^{[0,1,2]}c^{[3]} + b^{[1,2,3]}c^{[3]} + a^{[0]}c^{[1]} + a^{[0]}b^{[1,2,3]}c^{[3]}$$

$c^{[3]}$, $a^{[0]}$ and $b^{[1,2,3]}$ appear multiple times but are counted only once. Note $a^{[0,1,2]}$ and $a^{[0]}$ are distinct literals. The term $\sum_{i \in FO(np),\, j \in range(i),\, k \in range(np)} l_{ijk} f_k$ is an estimate of the number of literals in the fanout terms involving *np* after collapsing. We use the term $\sum_k f_k$ as a lower bound for the number of literals in the factored form of the k^{th} value function of *np*. This is because the number of literals in the factored form is at least the number of

distinct literals in the SOP form. Similarly, $\sum_{i \in FO(np), j \in range(i), k \in range(np)} l_{ijk} f_k$ and $\sum_{i \in FO(np), j \in range(i), k \in range(np)} l_{ijk}$ are both lower bounds in that we only count distinct appearance of literals and values of *np* in the fanouts. Overall, *value(np)* gives an estimate of the increase in the number of literals in factored form after collapsing.

This method of evaluating a node avoids having to factor the node and its fanouts. In the eliminate operation, because affected nodes have to be updated after each collapse, the number of factorization involved may be large and this provides an way to give an estimate of the effectiveness for eliminating a node.

3.8 merge

We mentioned in the operation *fx* that the new node extracted from the set of kernels is a binary node with its positive phase representing the kernel. In a way, this is not done in a purely multi-valued fashion.

One way to produce multi-valued kernels is to extract all kernels and find a collection of kernels that are mutually orthogonal and that gives the best saving in the network. This collection of kernels will form a new multi-valued node, each of whose value functions is one of the kernels in the collection. This method has not been implemented and thus we do not know yet the effectiveness of the method.

A second alternative is to merge nodes after *fx*. This is similar to the first method except that we update the network after extracting each binary kernel, and then in the end cluster the kernels that are mutually orthogonal.

Attentive readers may have noticed that the merge concept is not restricted to mutually orthogonal nodes, neither is it restricted to binary nodes. The generalization is as follows:

Given n nodes, np_1, np_2, ..., np_m, each taking values from the set P_1, P_2, ..., P_n respectively. The merged node *new_np* of these nodes has the value functions as:

$$new\_np^{\{0\}} = np_1^{\{0\}} \cap np_2^{\{0\}} \cap \ldots \cap np_n^{\{0\}}$$

$$\ldots$$

$$new\_np^{\{i_1 \times |P_2| \times \ldots \times |P_n| + i_2 \times |P_3| \times \ldots \times |P_n| + \ldots + i_n\}} = np_1^{i_1} \cap np_2^{i_2} \cap \ldots \cap np_n^{i_n}$$

$$\ldots$$

$$new\_np^{\{|P_2| \times \ldots \times |P_n| - 1\}} = np_1^{\{|P_1| - 1\}} \cap \ldots \cap np_n^{\{|P_n| - 1\}}$$

That is, the set of numbers $(i_1, .., i_n)$ is mapped to a value $i_1 \times |P_2| \times .. \times |P_n| + i_2 \times |P_3| \times .. \times |P_n| + ... + i_n$. Figure 3.7 shows an example.

(0, 0, 0)	↔	0
(0, 0, 1)	↔	1
(0, 1, 0)	↔	2
(0, 1, 1)	↔	3
(0, 2, 0)	↔	4
(0, 2, 1)	↔	5
(1, 0, 0)	↔	6
.		
.		
.		
(3, 2, 1)	↔	23

Figure 3.7: Mapping of values from 3 nodes to a single node for $|P_1|=4$, $|P_2|=3$, $|P_3|=2$

Thus the node new_np has $|P_1| \times .. \times |P_n|$ values. It is trivial to realize that if nodes $np_1,...,np_n$ are deterministic and completely specified, then the merged node *new_np* produced above is deterministic and completely specified. Some of the value functions of *new_np* may be empty and we can eliminate them. As a special case, if nodes np_1, ..., np_n

are the binary kernel nodes produced during fx, and if they are mutually orthogonal, then only the value functions of *new_np* generated from

$$np_1^{\{0\}} \cap np_i^{\{1\}} \cap ... \cap np_n^{\{0\}} = np_i^{\{1\}},\ 1 \leq i \leq n$$

are non empty since,

$$np_i^{\{1\}} \cap np_j^{\{0\}} = np_i^{\{1\}}(1 - np_j^{\{1\}}) = np_i^{\{1\}},\ i \neq j$$

After eliminating the empty value functions from *new_np*, we obtain a *new_np* with exactly n values, each carrying the function of a binary kernel. Notice we do not write out the default value function as it is implied as the complement of other value functions.

The *merge* command is a command unique to MVSIS. If arguments are given as names of nodes, it takes this list of nodes and forces a merge of them into a single multi-valued node according to the method described above. Of course a node will not be merged if it would produce a cycle in the network. If no argument is given, *merge* looks for likely lists of candidate nodes and merges them if a gain in the value for the network can be obtained. This may result in several additional nodes through multiple merges. We call the first case the *forced merge* and the second the *optimized merge*. For the *optimized merge*, two algorithms were considered:

Method 1:

a) Construct a graph where each vertex corresponds to a node in the network and each edge between two nodes carries a weight equal to the saving obtained by merging these two nodes.

b) Find the maximum number of pairs, no nodes being repeated, that has the largest total sum of edge weights.

This problem turns out to be NP-complete. However a clever method exists that translates the problem to the unate complement problem:

i) Create n binary variables $x_1 ... x_e$, each corresponding to an edge in the network

ii) For each vertex v with degree k, create k cubes $x_i x_j$ where i and j are two different edges of v.

iii) OR all cubes obtained in part ii) for all vertices to form a cover F.

iv) Obtain the SOP form of the complement of F; call it $\overline{F}$.

v) The set of edges in the final solution is, all positive phase variables x_i in a minterm of $\overline{F}$ that has the largest total weight.

Note this method differs from the maximum graph-matching algorithm because the edges are weighted.

Example 3.2

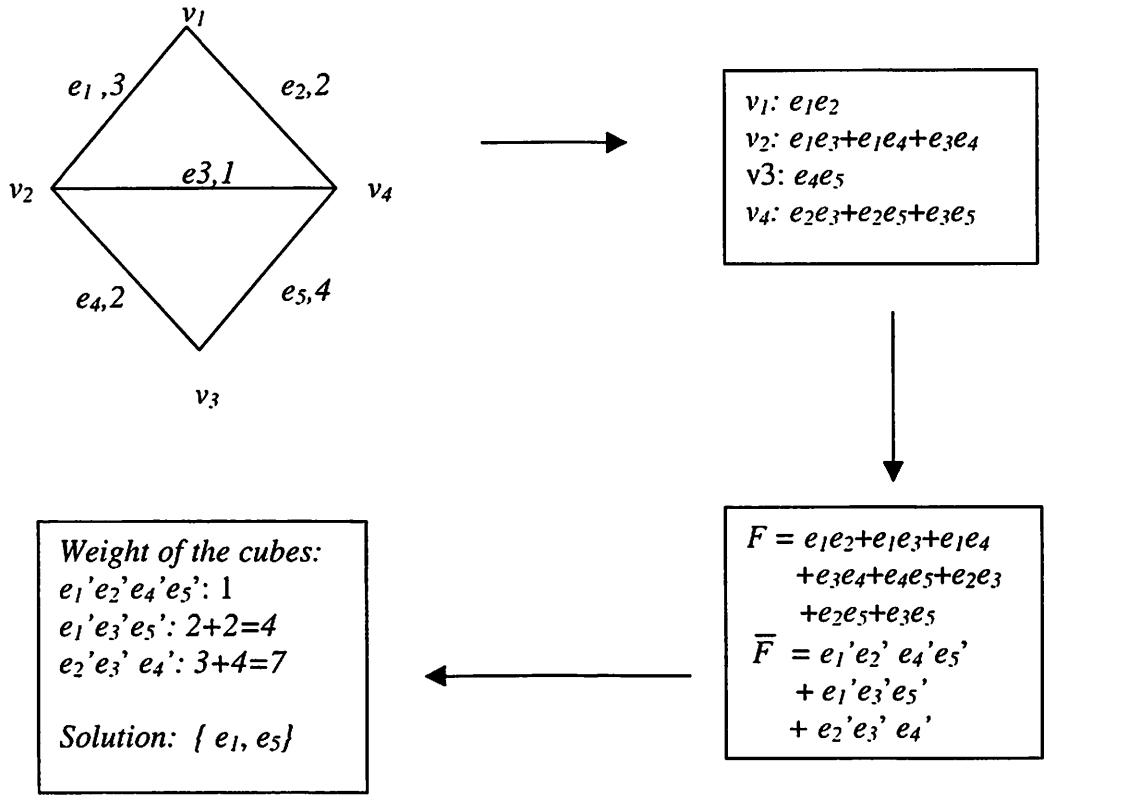


Figure 3.8: Solving for the set of non repeated edges with maximum weight

Method 2:

The second method is simpler. It merges one pair of nodes at a time with the largest saving, updates the network, and iterates until no further improvement can be made. Although both methods are implemented, currently MVSIS adopts the second

method since the merge of one pair of node may affect the weights of other node pairs. Updating the network after each merge more accurately reflects this effect.

The reverse operation of *merge* is *encode* which breaks a multi-valued node into binary nodes. This is being implemented in MVSIS and it should be interesting to see the result of network reconstruction through *merge* and *encode*.

merge and *encode* operations also makes it possible for translation between binary and MV networks. The next step in MVSIS is to apply such translation and compare the effect of two different approaches: 1. pure MV minimization; 2. encode into binary domain, perform binary minimization and decode back to MV domain.

3.9 The cube and the literal cost functions

Now that we have a feeling for the operations in MV-network minimization, we discuss in this section the impact of the cost function in the implementation of the operations.

The goal of minimizing an MV-network depends on the final types of target implementation. Minimizing the number of cubes or literals is generally beneficial for both hardware and software.

One of the applications that we are looking at is a compiler for control dominated software applications [JGSK00] where code size or evaluation time is key, while time for compilation can be relaxed. The input to the compiler is the intermediate representation (IR) of the program after the front-end lexical analysis and parsing. We decompose the IR into a data evaluation part and a control part. The control part is then translated into an MV-network representation, which is then optimized using both algebraic and don't care based algorithms. For this application, we have developed a cost function which is primarily the number of cubes, and secondarily the number of nodes in the network. The algebraic methods developed above such as the satisfiability-matrix and the maximum graph matching algorithms have been targeting at the cube cost function.

Hardware application may favor a cost function emphasizing the number of literals. The major change in algebraic methods for the cost function being number of literals is the addition of the single cube kernels in *fx*. We do not have to consider single

cube kernels with the cube cost function since they do not give savings in cube count. We have a way to extract single cube kernels as the common cubes of cube pairs in different node function. We have implemented in MVSIS both the cube cost and the literal cost function.

3.10 Other operations in MVSIS

The current MVSIS version implements the operations mentioned above. Another set of important operations is the node minimization routines. Like SIS, the node minimization commands include *simplify* and *full_simplify*. *simplify* is essentially *espresso-MV*. *full_simplify* was generalized from the binary domain in the work of [JB00]. Both are fully functional and together with the algebraic method packages, MVSIS now is a complete software package for combinational optimization of an MV network.

To make MVSIS an interative tool, we have implemented a set of user *I/O* interface commands. The commands that are similar to SIS commands include *read_blif, print, print_factor, print_stats, chng_name, reset_name, alias, unalias, delete, undo, echo, history, runtime, set, unset, sweep,* and *usage*. *read_blif* reads in ordinary *blif* files like SIS. Commands unique to MVSIS are *read_blifmv* that reads in a network in *blifmv* format, *print_range* that prints out the size of the range for each variable, and *reset_default*, which were discussed previously. In addition, we have a set of verification commands. The command *validate* compares the current MV-network with a *blif* or *blifmv* file (generally the original input file) and outputs a message indicating whether they are functionally equivalent by random test vector simulation.

3.11 Causions on using MVSIS

There are a few things about the current version of MVSIS worth mentioning:

1. We do not have a facility yet to allow external don't cares. The input patterns not specified in an input file are assumed to produce the default value.

2. The computation time for *fx*, *factor* and *full_simplify* may be long if the network/nodes contain a large number of cubes or literals. Currently there is a timer implemented in each of these commands to terminate if the specified time limit is exceeded, and a partial result is returned.

3. The computation time for *collapse* may be long if the network contains a large number of nodes and a node close to the primary outputs is collapsed.

4. For very large networks with over a few thousand cubes, operations such as *fx* and *full_simplify* may run out of memory. It is recommended to minimize the network by calling *simplify* once it is read in, and to break down large nodes into smaller ones by calling *decomp*.

Chapter 4

Experimental results

Because of the large solution space for minimizing an MV-network, finding the optimal solution is at least NP-hard. Hence, we aim for a set of heuristic minimization methods, each focusing on a particular aspect of the implementation. This set of methods includes the node-minimization methods, mentioned in Chapter 1, and algebraic operations discussed in Chapter 3. The whole minimization process is often executed through optimizing scripts, which are an assembly of the minimization methods. Scripts can heuristically lead to a good solution.

In this chapter, we first present experimental results concerning the algebraic method operations. We then give some results on MVSIS executed using scripts. In addition, to illustrate the effectiveness and speed of MVSIS, we run MVSIS on purely binary networks and compare the results with SIS. Experiments on binary examples are performed with the cost function being literal cost, since this is what is used in SIS.

The function for each MV-node is represented in SOP form. We test the algorithms on a set of binary examples and a set of MV examples. The set of binary examples is a subset of the benchmarks used for SIS. The set of MV examples come from machine learning applications, a set of multi-valued benchmarks distributed with VIS [VIS96] and a set of examples from asynchronous applications [THE]. First, we give the characteristics of the examples and explain some of the notations:

Example	*#nodes*	*#PO*	*#cubes_sop*	*#lits_sop*	*#lits_ft*
b9.blif	*117*	*21*	*195*	*256*	*256*
c8.blif	*48*	*18*	*151*	*363*	*232*
cht.blif	*36*	*36*	*120*	*374*	*236*
sct.blif	*40*	*15*	*116*	*236*	*167*
5xp1.blif	*175*	*10*	*240*	*401*	*401*
inc.blif	*153*	*9*	*266*	*357*	*357*
misex1.blif	*66*	*7*	*90*	*126*	*126*
lal.blif	*71*	*19*	*138*	*258*	*224*
misex2.blif	*141*	*18*	*152*	*264*	*264*

Table 4.1: Initial characteristics of the binary examples

Example	*#nodes*	*#PO*	*Range of inputs*	*Range of outputs*	*#cubes_sop*	*#lits_sop*	*#lits_ft*
car.mv	*1*	*1*	*4*	*4*	*46*	*259*	*106*
pal3.mv	*1*	*1*	*3*	*2*	*27*	*162*	*18*
balance.mv	*1*	*1*	*5*	*5*	*153*	*570*	*349*
iris.mv	*1*	*1*	*8*	*3*	*29*	*116*	*90*
mm3.mv	*1*	*1*	*3*	*3*	*10*	*37*	*23*
mm4.mv	*1*	*1*	*4*	*4*	*30*	*141*	*78*
monks2tr.mv	*1*	*1*	*3*	*2*	*35*	*210*	*122*
monks3tr.mv	*1*	*1*	*3*	*2*	*30*	*171*	*120*
monksltr.mv	*1*	*1*	*3*	*2*	*35*	*200*	*128*

Table 4.2: Characteristics of the MV examples after initial simplification

The mv examples balance.mv, car.mv, iris.mv, monks2tr.mv, monks3tr.mv and monksltr.mv are from machine learning applications and are generally incompletely specified. The input patterns not specified are intended as don't cares. However, currently we do not provide the facility of specifying external don't cares. For the inputs patterns

not specified in the input file, we assume a default value and assign these input patterns to the default cover.

The binary examples are the initial input files before any minimization is performed. The mv examples are obtained by reading in the initial input file followed by a single command *simplify*. This is because a large portion of the mv input files are trivial redundancies and can be greatly reduced by *simplify*.

The statistics is obtained through the command *print_stats –f*. *PO* is the number of primary outputs. *#nodes* is the number of nodes in the network, not including the primary inputs. The column *Range of inputs* gives an average number of the values the primary input nodes takes. The column *Range of outputs* gives an average number of the values the primary output nodes take. *#cube* is the total number of cubes in the network in SOP form, *#lit_sop* is the total number of literals in the network in SOP form, and *#lit_ft* total number of literals in the network in factored form.

4.1 Of algebraic methods

In the following tables, we show the results of MVSIS using just the algebraic methods on these benchmarks and compare them with SIS results:

Example	*#lits_ft (Original)*	*#lits_ft (MVSIS-fx)*	*#lits_ft (SIS-fx)*
b9.blif	*236*	*188*	*194*
c8.blif	*232*	*223*	*220*
cht.blif	*336*	*242*	*243*
lal.blif	*224*	*193*	*188*
sct.blif	*167*	*121*	*119*
5xp1.blif	*401*	*351*	*351*
inc.blif	*357*	*313*	*313*
misex1.blif	*126*	*101*	*101*
misex2.blif	*264*	*192*	*191*
Average reduction		*18%*	*18%*

Table 4.3: *fx* on binary examples

In Table 4.3, the cost function for MVSIS is the literal cost function in factored form since SIS does not consider cube cost function. For convenience, column *original* repeats the number of literals in factored form in the original network; column *MVSIS-fx* shows the number after applying command *fx* in MVSIS on the original network and column *SIS-fx* gives this number after applying SIS. The default value function is not considered in this evaluation. As we can see, the results of MVSIS are close to that of SIS.

Example	*org_cube*	*fx_cube*	*gx_cube*	*org_lit*	*fx_lit*	*gx_lit*
car.mv	46	34	34	259	155	151
pal3.mv	27	11	11	162	30	30
balance.mv	153	114	114	570	490	475
iris.mv	29	29	24	116	116	105
mm3.mv	10	10	10	37	32	32
mm4.mv	30	30	30	141	121	121
monks2tr.mv	35	32	32	210	169	155
monks3tr.mv	30	30	29	171	152	144
monksltr.mv	35	35	35	200	187	173
Average reduction		18%	19%		22%	26%

Table 4.4 *fx –q* and *fx –g* on MV examples with cube cost function

In Table 4.4, the cost function is the cube cost function. *org_cube* is the number of cubes in SOP form in the original network. *org_lit* is the number of literals in SOP form in the original network. *fx_cube* is the number of cubes in SOP form and *fx_lit* is the number of literals in SOP form, after the command *fx –q*. *gx_cube* is the number of cubes in SOP form and *gx_lit* is the number of literals in SOP form after the command *fx –g*.

Note *fx –g* always does at least as good as *fx –q*. There are cases where *fx_cube* = *gx_cube* but *fx_lit* > *gx_lit*. This is because *fx* extracts kernels that does not increase the cost of the network, including the ones that do not change the cost. When the cost function is the number of cubes, doing so generally reduces the number of literals in the network. The fact that *fx –g* gives better result shows that *fx –g* finds more kernels than *fx –q*.

4.2 Of MVSIS scripts

We have developed a full set of minimization operations in MVSIS and it makes sense to run scripts with both SIS and MVSIS on binary example to compare the results.

```
sweep
eliminate -1
simplify
eliminate -1
sweep
eliminate 5
simplify
resub
fx
resub
sweep
eliminate -1
sweep
fs
```

Table 4.5: The file script.rugged

Example	Original	MVSIS	SIS
b9.blif	236	126	121
c8.blif	232	136	139
cht.blif	336	165	165
lal.blif	224	106	105
sct.blif	167	75	78
5xp1.blif	401	136	124
inc.blif	357	206	210
misex1.blif	126	66	63
misex2.blif	264	111	104
Average reduction		52%	53%

Table 4.6: Scripts run on binary examples

Again the cost function is the number of literals in factored form.

All experiments are performed on an *Alpha 100MHz* machine with *128MB* memory. Run times for operations such as *collapse* and *eliminate* are similar to SIS. Run times for *fx* and *factor* are slower for MVSIS due to the generalization of the algorithms for MV networks. In either case, the run time for MVSIS ranges from less than one second to minutes, depending on the size of the examples.

Chapter 5

Conclusions and Future Directions

In this chapter we summarize out contributions and point out some future directions where this research can proceed.

5.1 Conclusion

We investigated multi-valued multi-level logic minimization problems. As a design effort, we implemented the algorithms developed in a test-bed system called MVSIS, which in many ways resembles the technology independent part of SIS and behaves almost exactly like SIS on binary examples. We summarize our contributions:

- The main theoretical breakthrough was the development of a method for algebraic manipulation of multi-valued expressions. We developed two types of algebraic divisions, exact and inexact. These have been implemented using the concept of a satisfiable matrix of MV-cubes. A fast branch and bound method and efficient heuristics were developed for this. In addition, for exact division, we devised a direct maximum graph matching method that runs much faster. It's complexity is $O(n^3)$.
- Both algebraic methods, as well as a full set of associated minimization operations have been implemented in MVSIS. This set of operations includes algebraic and don't care-based methods*. They form the technology independent part of multi-valued logic synthesis. Algebraic methods include *fx, factor, decomp, resub, collapse* and *eliminate*; don't care-based methods include node minimization using observability don't cares, implemented through commands

* Currently we have not implemented a way to specify external don't cares.

simplify and *fullsimp*; also a network reconstruction method that is not present in SIS is *merge*. We have implemented the operations targeting both a cube cost function and a literal cost function. The results of MVSIS on binary examples are almost equivalent to SIS, yet have the ability to handle MV-networks in general.

5.2 Future work

Some future research directions in this area are:

- Neither *fx -q* nor *fx –g* in MVSIS considers all kernels in the network. For *fx –q*, it finds the kernel of all pairs of cubes in the nodes. As mentioned, the kernel of a pair of cubes is not unique, due to the flexibility in semi-algebraic operation. Currently, we select the kernel with the least number of values. By doing this, we have thrown away possibly better kernels for the network. The satisfiability-matrix based method and the maximum graph matching method takes care of the flexibility in the exact division, but not in the selection of kernels. We still need to find a good, efficient method to combine flexibilities in both aspects.
- So far, we have concentrated only on the technology independent part of MV network minimization. Applications in technology dependent minimization such as technology mapping and software synthesis remain to be developed. Algorithmic development will depend heavily on target applications.
- To compare on a fair basis with the alternative approach which translates an MV application to a binary network for minimization, and then decode back to MV domain, we need an efficient encoding scheme. *encode* is the reverse process of *merge*. *encode* and *merge* together will not only allow for better evaluation of MV methods, but also for a broader range of applications which will benefit both hardware implementation and software implementation.

Bibliography

[BHV90] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel Logic Sythesis. In *Proceedings of the IEEE, vol.78, (no.2), p.264-300,* Feb. 1990.

[BK99] R. K. Brayton and S. P. Khatri. Multi-valued logic synthesis. In *Proceedings of the International conference on VLSI Design*, 1999

[BM82] R. K. Brayton and C. McMullen. The Decomposition and Factorization of Boolean Expressions. In Proceedings of the International Symposium on Circuits and Systems, May 1982.

[Bra99] R. K. Brayton. Algebraic methods for multi-valued logic. Technical Report UCB/ERL M99/62, Electronics Research Laboratory, University of California, Berkeley, Dec. 1999

[JB00] Yunjian Jiang and Robert K. Brayton. Don't Cares and Multi-Valued Logic Minimization. In *Proceedings of the International Workshop on Logic Synthesis*, May 2000.

[JGSK00] Yunjian Jiang, Minxi Gao, Subarna Sinha, and Robert K. Brayton. An optimizing software compiler based on multi-valued logic optimizations. The *SRC's Sith Premier Technical Conference*, Sept. 2000.

[LFS+] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev. Asynchronous design using commercial hardware synthesis tools. In *Proceedings of the Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems* (ASYNC 2000)

[LMBSV90] L. Lavagno, S. Malik, R. K. Brayton, and A. Sangiovanni-Vincentelli. MIS-MV: Optimization of multi-level logic with multiple-valued inputs. *In Proceedings of the International Conference on Computer-Aided Design*, 1990.

[RSV88] R. L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. IEEE Transaction on CAD, 1988.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, Electronics Research Laboratory, University of California, Berkeley, CA 94720, May 1992.

[THE] Theseus Logic Inc. http://www.theseus. com

[VIS96] VISgroup. VIS: A system for verification and synthesis. *In IEEE International Conference on Computer-Aided Verification*, 1996

[VSV89] T. Villa and A. L. Sangiovanni-Vincentelli. NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementations. In *Proceedings of the Design Augumation Conference*, 1989

[VSV90] T. Villa and A. L. Sangiovanni-Vincentelli. NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, September 1990.

[WLC94] H. M. Wang, C. L. Lee, and J. E. Chen. Algebraic Division for Multilevel Logic Synthesis of Multi-Valued Logic Circuits. *In the Twenty-Fourth International Symposium on Multiple-Valued Logic*, 1994.