MULTI-VALUED OPTIMIZATION AND POST NETWORKS

by

Yinghua Li

Memorandum No. UCB/ERL M02/18

21 December 2001

MULTI-VALUED OPTIMIZATION AND POST NETWORKS

by

Yinghua Li

Memorandum No. UCB/ERL M02/18

21 December 2001

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Multi-valued optimization and Post Networks

by Yinghua Li

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Robert K. Brayton
Research Advisor

(Date)

* * * * * * *

Professor Alberto Sangiovanni-Vincentelli

(Date)

Multi-valued Optimization and Post Networks

Abstract

Multi-valued optimization and Post networks

Yinghua Li
Master of Science in Engineering - Electrical Engineering and Computer Science
University of California at Berkeley
Professor Robert K. Brayton, Chair

We address the problem of extending multi-valued optimization methods from i-set mode expressions to Post mode expressions. The i-set mode is a format used for general multi-valued logic, where a set of multi-valued input and binary output functions are used to represent a multi-valued output function. Post mode is another format for general MV logic and can be obtained from i-set mode by a transformation. However, direct minimization of the multi-level network expressed by Post mode expressions is desirable since they are preferred in some applications and can achieve some simplifications. We examine some optimization methods on i-set mode expressions and attempted to extend them to Post mode expressions. We successfully extended some, including two important semi-algebraic optimization methods. The difficulties encountered on the extension of other methods are analyzed. The second part of this report summarizes further developments of the multi-valued multi-level synthesis tool MVSIS.

To my family

Acknowledgements

Acknowledgements

I would like to give my sincere thankfulness to my research advisor Prof. Robert K. Brayton, for his support and guidance throughout the past one and a half years of study. A smooth beginning of my research work is impossible without his patient instructions. The weekly meetings with him have taught me invaluable skills to tackle research problems. I am also grateful for his insightful and detailed comments on this report.

Minxi Gao, who helped me start my work on MVSIS, deserves my special acknowledgements. She built up the semi-algebraic optimization part of MVSIS which plays an important role in my work and I constantly turned to her for help at the early stages of my work.

I would also like to thank other MVSIS group members Yunjian Jiang, Jiehong Jiang and Subarna Sinha, for their generous help, the valuable discussions and suggestions in the course of the implementations on MVSIS, and for being best friends.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multi-valued minimization can be used in hardware/software synthesis as a high level representation. One approach for MV minimization is to encode the multi-valued variable and use binary logic optimization, and then translate back to the multi-valued domain. However, in such a process, some optimization opportunities are lost. Thus direct multi-valued minimization is more desirable. In this chapter, we review some research in multi-valued logic synthesis and its state-of-the-art applications. Then we describe some specific problems of interest. Finally, we outline the remainder of 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 [VKBSV97]. The two-level multi-valued logic synthesis problem was addressed in the work of Rudell et.al [RSV88], and efficient implementations were developed in ESPRESSO-MV. Techniques developed there were applied to the encoding problem for two-level implementations of finite state machines [VSV90]. For multi-level implementation, the synthesis problem formulated there was to minimize a multi-level logic network, with only one multi-valued variable appearing at the primary inputs. The application is state assignment for FSM's with a multi-level circuit implementation. Lavagno el.al. developed a theory for algebraic decomposition and factorization for this type of application [LMBSV90] and completed an implementation called MIS-MV.

Recently, multi-level multi-valued minimization has been studied more generally. Theories on node minimization [JB00] and common sub-expression identification

[MB00] were developed. An efficient software package MVSIS for multi-level multi-valued minimization [GJJLSR01] is available and continues to be developed. MVSIS can be used to optimize general multi-valued logic: variables can have different ranges and the logic can be multi-level. In MVSIS, the format of multi-valued functions is a natural extension of that for binary functions. Each multi-valued function is represented by a set of multi-valued input, binary-output functions. Many sophisticated algorithms in binary logic synthesis have been extended to the multi-valued case.

In summary, multi-valued synthesis has developed to a stage where general multi-valued logic can be optimized without constraints on its structure. Although most algorithms were designed for a particular format for multi-valued functions, their extension to other formats for representing multi-valued functions is worth studying.

1.2 Motivation

Multi-valued functions can have different representations based on different function sets that are logically complete. One such format is based on binary logic AND, OR and a multi-valued operation literal (we give the formal definitions in Chapter 2). This format is used in MVSIS, since it has no constraint on the range of variables and the number of levels of the logic. Another format is based on Post Algebra [P21]. This format has different base functions: MIN, MAX and literal; also the value set for a multi-valued variable is totally ordered. Usually the multi-valued logic in this format is two-level.

In order to optimize logic based on Post algebra, one approach is to convert it into the AND-OR format, use existing algorithms, and finally convert back. More direct optimization is desirable since some optimization opportunities may be lost. Some attempts have been made to minimize multi-valued logic in Post algebra [DJR01]. In this report, we will analyze the minimization algorithms used in MVSIS and try to extend them to the format based on Post algebra.

1.3 MVSIS overview

Since all implementations covered in this report are done in MVSIS, we give first a brief introduction on it. MVSIS is a program modeled after SIS, but the logic network it works on can have all variables multi-valued, each with its own range. It is an interactive tool for multi-valued technology-independent synthesis.

1.3.1 Input specification

The input to MVSIS can be an MV circuit represented by a netlist of MV-nodes (command: *read_blifmv*). Binary networks can also be read (*read_blif*). Internally, a design representation used in MVSIS is an **MV-network** of nodes; each node represents an MV-function with a single multi-valued output. Each variable can have a separate **range** of its own, which can be represented by the set $\{0, 1, \ldots, n_k-1\}$. The function whose onset gives the set of minterms for which $f_k = i$, i.e. the function at node k equals the value i, is called the **i-set** of function f_k. There is an edge connects from i to j if any of the i-sets of j depends explicitly on the variable of node i. The network has a set of **primary inputs** and a set of nodes, designated as the outputs of the network.

1.3.2 Combinational optimization

We mention all the commands for technology-independent transformations in this section. They can be organized into 4 groups.

simplify, *fullsimp* and *reset_default* are used for node simplification. Satisifiable don't cares (SDC) are used in *simplify*. *fullsimp* is a stronger node simplification where not only SDC but also compatible observability don't cares (CODC) are used. For each node, one of the i-sets is designated as the **default** and by definition is the complement of the other i-sets. Command *reset_default* can be used to choose a new default based on the cost of the i-sets used to minimize the nodes.

Command *fx*, *decomp* and *resub* are for kernel and cube extraction. *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. *decomp* does a complete multi-valued factoring of the i-sets of each node and decomposes the nodes according to these factorizations. Finally, *resub* performs algebraic substitution of one node into another.

Command *collapse*, *eliminate*, *merge*, *encode* and *undo* execute network manipulations. *collapse* operates exactly as in SIS. *eliminate* compares the cost of eliminating some internal node against a given threshold, and eliminates the node if the cost is less than the threshold. *merge* takes a list of nodes and forces a merge of them into a single multi-valued node. *encode* tries to find a good binary encoding for each multi-valued variable in order to convert the network into a binary one. *undo* replaces the current network with the previous one.

The last group of commands is for printing debug information, reading in input files and writing out optimization results. It includes commands *write_blifmv*, *read_blifmv*, *read_blif*, *print*, *print_factor*, *print_range* and *print_stats*.

Command *validate* verifies the combinational equivalence of two networks by simulating the networks on random vectors. There was no formal verification in MVSIS.

1.4 Outline of the report

In Chapter 2, we present some notation and definitions, introduce the theory of Post algebra and define the format based on it. The main part is devoted to extending algorithms used in MVSIS to this new format. Detailed algorithm modification is described for successful extensions where the operation can be done directly in this format and analysis is given on unsuccessful parts.

In Chapter 3, we present part of the implementation of the extensions to the new format. Then we discuss commands which we added to improve MVSIS, namely *qcheck*, *elim_part*, *pair_decode*, and *validate* for formal verification. The extension to handle external don't cares is also discussed. Some examples and experimental results are also given in this chapter.

We conclude and summarize this report in Chapter 4. Directions for future work are also given.

Chapter 2

Multi-valued Minimization in Post Algebra

There are different formats to represent multi-valued functions. The values of one multi-valued variable can be unordered, i.e., we don't say one value is "greater" than another; all values have the same priority. This is the format used in MVSIS. **Chain-based Post algebra**, an algebraic system also used for manipulation of multiple-valued functions, provides another format. Its main characteristic is that the values of each multi-valued variable are totally ordered. Such a characteristic leads to certain requirements on different minimization techniques. In this chapter, we first give some notation and definitions on multi-valued logic and introduce chain-based Post algebra. Then we concentrate on extending minimization techniques used in MVSIS to the minimization of multi-valued functions in Post algebra.

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 The multi-valued variable x takes on 5 values from the set $\{0, 1, 2, 3, 4\}$.

Definition 2.2 (Multi-valued Literal) A multi-valued literal x^S where S is a subset of values of x is a binary function on x. It evaluates to 1 if x takes on one of the values in the subset S.

Example 2.2 $x^{\{0,2\}}$ is a literal of x. It evaluates to 1 if x takes on value either 0 or 2.

Definition 2.3 (Multi-valued Cube) An MV cube is a conjunction of MV literals and evaluates to 1 only if each of the literals evaluates to 1. Literals containing all values of the corresponding variable always evaluate to 1 so they are suppressed in the cube form.

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

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

Definition 2.5 (Multi-valued function) An n-variable multi-valued function $f(x_1,...,x_n)$ is a mapping $f: P_1 \times P_2 \times .... \times P_n \rightarrow M$ with the variable x_i taking values from the sets $P_i = \{0,1,...,p_i - 1\}$ and the function f taking its value from the set $M = \{0,1,...,m-1\}$. Each set contains at least 2 values.

Example 2.3 The following truth table defines a multi-valued function on variables x_1 and x_2.

$x_2 \backslash x_1$	0	1	2
0	0	1	0
1	2	2	2
2	0	1	0

Table 2.1 An example of multi-valued function

Of course we want a more compact method to represent multi-valued functions.

Definition 2.6 (i-set mode) A multi-valued function f with m output values can represented by m i-sets. The jth i-set is a function whose onset gives the set of minterms for which $f = j$. Such expression is said to be in the i-set mode format.

Notice that the jth i-set function of f is the same as the literal function $f^{\{j\}}$ by definition.

Example 2.4 The function in Example 2.3 can be written in the i-set mode as:

$$f^{\{0\}} = x_1^{\{0,2\}} x_2^{\{0,2\}}$$

$$f^{\{1\}} = x_1^{\{1\}} x_2^{\{0,2\}}$$

$f^{\{2\}} = x_2^{\{1\}}$

The i-set mode format is the main one used in MVSIS. All the commands discussed in Chapter 1 are based on this mode.

2.2 Post algebra and Post mode

Chain-based Post algebra was first developed by Emil Post in 1921.

Definition 2.7 (chain-based Post algebra) Chain-based Post algebra is an algebra $<M$; $+, \bullet, L$; **0**, **1**$>$, such that the elements of M form a totally ordered chain:

$$\mathbf{0} = 0 < 1 < \ldots. < m\text{-}1 = \mathbf{1}$$

"+" and "•" are the binary operations maximum (**MAX**) and minimum (**MIN**), respectively, and $L = \{x^0, x^1, \ldots, x^{m-1}\}$ is a set of basis literals specified by Definition 2.2 (Note a basis literal takes on values **0** and **1** instead of 0 and 1).

It has been shown that chain-based Post algebra is functionally complete for multi-valued functions [P21]. So it can be used for expression and manipulation of multi-valued functions. For the function specified by Definition 2.5, the chained set is the union of the input domains of the variables and the output domain of the function, i.e. $M \cup (\cup_{i=1}^{n} P_i) = \{0,1,\ldots,\max\{m, P_i\}\}$. Maximum **MAX** (x_i, x_j) and minimum **MIN** (x_i, x_j) are binary operations of type $P_i \times P_j \rightarrow M$. Similarly we have cubes and sum-of-products based on **MAX** and **MIN**, where **MIN** takes the place of **AND**, and **MAX** takes the place of **OR**.

Definition 2.8 (Post Cube) A Post cube is a product of literals of type $\alpha \bullet x_1^{S_1} \bullet x_2^{S_2} \ldots. \bullet x_n^{S_n}$ where $\alpha \in M$ is a constant and product "•" is **MIN**. For a Post cube c, let $\eta(c)$ denote its coefficient α and $\gamma(c)$ denote the product of the literals.

Actually a Post cube can be defined as a product of multi-valued variables since **MIN** is defined for more than binary valued variables.

Definition 2.9 (≥) Post cubes c_1, c_2 have relation $c_1 \geq c_2$ if the value of c_1 is greater or equal to the value of c_2 under any input vector.

Thus the coefficient associated with c_1 must not be less than the coefficient for c_2; also the cube part of c_1 must contain that of c_2, i.e., $\eta(c_1) \geq \eta(c_2)$, $\gamma(c_1) \supseteq \gamma(c_2)$. Similarly, we can define relations such as $\leq$, $=$.

Definition 2.10 (SOP) A sum-of-products expression over Post algebra is a sum of Post cubes, with sum being **MAX**.

Note: In this report we use **cube** to refer to cubes in i-set mode and **Post cube** to refer to cubes defined by Definition 2.8. Similarly, **SOP** refers to sum-of-products in i-set mode while **Post SOP** refers to that defined in Definition 2.10.

Example 2.5 The function shown in Figure 1 can be written as:

$$f = 1 \cdot x_1^{\{1\}} \cdot x_2^{\{0,2\}} + 2 \cdot x_2^{\{1\}}$$

We use the notion **jth i-set** to denote the sum-of-products expression formed by all the cubes with constant j before them. For example, the 1st compressed i-set of the above function is $x_1^{\{1\}} \cdot x_2^{\{0,2\}}$.

Since we use the **MAX** operation between cubes, a cube with a higher-valued coefficient may be combined with a cube with a lower-valued coefficient as shown in the following lemma [DJB01].

Lemma 2.1 Let $a < b$ be constants in M and g, h, k be products of multi-valued literals. If $f = a \cdot g + b \cdot h$, then $f = a \cdot k + b \cdot h$ where $g \subseteq k \subseteq (g \cup h)$.

While the proof of Lemma 2.1 is straightforward, it provides useful information: we can use the jth i-set as don't cares to simplify the kth i-set for all $j > k$. Such don't cares are called **priority don't cares**. For the function in Example 2.5, such simplification yields a simpler result: $f = 1 \cdot x_1^{\{1\}} + 2 \cdot x_2^{\{1\}}$. The resulting i-sets are called **compressed i-sets**. In fact, the following theorem holds.

Theorem 2.1 Suppose we use the don't cares described in Lemma 2.1 to minimize each i-set and suppose that such minimizations yield minimum covers $\{g^{m-1}, g^{m-2}, \ldots, g^1\}$. Then the associated chain-based Post algebra form,

$$(m-1).g^{m-1}+(m-2).\ g^{m-2}+\ \ldots\ +1.\ g^1$$

is minimum, i.e. has the minimum number of cubes.

Proof Suppose there is a Post expression with fewer cubes. Then one of the g^k can be replaced by h^k which has fewer cubes,

$$(m-1).g^{m-1}+(m-2).\ g^{m-2}+\ \ldots+k.h^k+\ldots+1.\ g^1$$

However, h^k must cover all minterms that are in the care set of the k^{th} i-set. Since h^k is assumed smaller than g^k and g^k must also cover all such minterms, then both are covers, but since g^k was assumed minimum, we have a contradiction.

Thus we have a method for obtaining minimum chain-based Post expressions.

Definition 2.11 (Post mode) The expression of a multi-valued function which results from the simplification using priority don't cares is called a Post mode format.

It can be expected that Post mode format is more compact than the i-set mode format. In some applications such as code generation, the Post mode format is more desirable. For example, if we want to generate C code from a multi-valued logic function f with i-sets f_i, $i = 0..m-1$, the natural code structure would be:

```
If $f_0$ then
        Return $f = 0$;
Else if $f_1$ then
        Return $f = 1$;
......
else
        Return $f = m-1$;
```

We can use f_0 as don't cares to simplify f_1, ..., f_{m-1}, because when we reach those branches, f_0 has been evaluated to be false. Such simplification results in a Post mode format for f.

2.3 Extension of MVSIS commands to Post mode

MVSIS is designed to work on the i-set mode format of multi-valued functions. We want to extend it to work on the Post mode format for the following reasons. First, in some applications Post mode expressions are preferred. Second, for most multi-valued functions the Post mode format is smaller than i-set mode format because of the use of priority don't cares. Optimization can be sped up since the time complexity of all optimization algorithms is typically proportional to the size of the input logic.

One possible approach for extending multi-level multi-valued synthesis to Post expressions is to convert the Post format into i-set format, optimize in the i-set format and then convert back. In this process no speed improvement is possible, while some optimization opportunities may be lost if we want the Post format as the final output. So it is desirable to develop direct multi-valued minimization algorithms on the Post format if possible.

In MVSIS, there are two main kinds of optimization methods: semi-algebraic optimization and don't care-based simplification. We will focus on extending these two classes of commands to Post mode.

2.3.1 Semi-algebraic optimization methods

Semi-algebraic optimization methods for multi-valued division and factorization are important minimization methods for multi-valued functions. Two methods that apply to the i-set mode are implemented by commands *fx*, *decomp* and *factor* in MVSIS. There are two types: the **satisifable-matrix** method and the **maximum graph matching** method.

2.3.1.1 Satisifiable Matrix Method

We give a brief description of the satisfiable-matrix method and explain how to extend it to Post mode expressions. Suppose A is a matrix, each cell of which is filled with an MV-cube.

Definition 2.12 (Supercube) The supercube of a set of cubes f, is the smallest cube containing f.

Definition 2.13 (Value Condition) Let I be the set of rows and J the set of columns in A in which value v_x of some variable x appears. The value v_x satisfies the value condition if it appears in all entries of A given by $\{A_{i,j} \mid i \in I, j \in J\}$.

Definition 2.14 (Satisfiable-matrix) A matrix A is satisfiable if all values of all variables in all cubes of it satisfy the value condition.

We can look for a factorization of the type $f = (d)(q) + r$ for a multi-valued input, binary-valued output function by fitting some of the cubes of f into a satisfiable-matrix [GB00]. Then d and q can be formed by the following procedure:

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 $d = \Sigma_i d_i$.
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 $q = \Sigma_j q_j$.

The cubes not in A form the remainder r. If the goal of the factorization is to minimize the number of cubes in the factored representation of f, we would want the satisfiable-matrix to be as large as possible. The "size" of the matrix is measured in terms of the number of distinct cubes in it.

A slight extension to handle the constants in Post cubes, can make the satisfiable-matrix method work on Post expressions. Recall, for these cubes, each has a constant in it in addition to a set of literals.

Definition 2.15 (Constant condition) Let A be a matrix of Post cubes. Suppose the maximum of the constants in row i is denoted $const_i$ and the maximum of the constants in column j is $const_j$. The constant condition is satisfied if the constant at $A_{i,j}$ can be written as $const_i \cdot const_j$.

In Post mode, a satisfiable-matrix must satisfy both the value conditions and the constant condition.

Definition 2.16 (Post supercube) The supercube of a set of Post cubes $C = \{c_i\}$, denoted $\sigma(C)$ is the cube that satisfies:

1. $\sigma(C) \geq c_i$ for any $c_i \in C$;
2. $\sigma \geq \sigma(C)$ for any Post cube σ satisfying condition 1.

An interesting fact about the constant of $\sigma(C)$ is that it is equal to the maximum constant in C; if $\eta(\sigma(C)) > max\_const$, we can replace it with max_const to get a new Post cube σ, which satisfies Condition 1 while $\sigma < \sigma(C)$. For any Post cube σ that satisfies Condition 1, $\gamma(\sigma) \supseteq \gamma(c_j)$. So $\gamma(\sigma) \supseteq$ *supercube*$(\{\gamma(c_j)\})$ where *supercube* is defined in Definition 2.12. Thus $\gamma(\sigma(C)) =$ *supercube*$(\{\gamma(c_j)\})$. Actually these arguments show a method to compute the Post supercube.

For a multi-valued input, multi-valued output function *f*, we can fit its Post cubes into a satisfiable-matrix and then use the above procedure to find a factorization for it (of course we need to get the Post supercube instead of supercube for each row and column).

Theorem 2.2 For any satisfiable matrix *A*, the *d* and *q* formed by the above procedure have the following property:

$$A_{i,j} = d_i \cdot q_j$$

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

Proof: $A_{i,j} \le d_i \cdot q_j$ can be obtained directly from the definition of Post supercube. Since d_i is the Post supercube for row *i*, $A_{i,j} \le d_i$ for any input vector. Similarly $A_{i,j} \le q_j$. So $A_{i,j} \le MIN(d_i, q_j) = d_i \cdot q_j$.

Assume $A_{i,j} < d_i \cdot q_j$ for some input vector *m*. Note that the constants in $A_{i,j}$, d_i and q_j has the following relation: $\eta(A_{i,j}) = \eta(d_i) \cdot \eta(q_j)$ due to the constant condition. Thus $A_{i,j} < d_i \cdot q_j$ happens only if there exists a variable with a value *v* such that $v \in d_i \cdot q_j$ but $v \notin A_{i,j}$. However, *v* must be in $A_{i,k}$ for some *k*, and in $A_{m,j}$ for some *m*. Therefore by the value condition for a satisfiable matrix, $v \in A_{i,j}$, which contradicts to our assumption. So $A_{i,j} = d_i \cdot q_j$.

Example 2.6 The following multi-valued input, multi-valued output function *f* in Post mode can be fit in a satisfiable matrix:

Row \Column	1 $3 \cdot x^{\{0,2,3\}} \cdot u^1$	2 $2 \cdot x^{\{0,1,2\}} \cdot v^1$	3 $2 \cdot w^1$
1 $3 \cdot x^{\{0,1,3\}}$	$3 \cdot x^{\{0,3\}} \cdot u^1$	$2 \cdot x^{\{0,1\}} \cdot v^1$	$2 \cdot x^{\{0,1,3\}} \cdot w^1$
2 $1 \cdot y^1$	$1 \cdot x^{\{0,2,3\}} \cdot y^1 \cdot u^1$	$1 \cdot x^{\{0,1,2\}} \cdot y^1 \cdot v^1$	$1 \cdot y^1 \cdot w^1$
1 $3 \cdot z^1$	$3 \cdot x^{\{0,2,3\}} \cdot z^1 \cdot u^1$	$2 \cdot x^{\{0,1,2\}} \cdot z^1 \cdot v^1$	$2 \cdot z^1 \cdot w^1$

Table 2.2 An example of satifiable matrix

Then a factorization of f is $(3{\cdot}x^{\{0,1,3\}} + 1{\cdot}y^1 + 3{\cdot}z^1){\cdot}(3{\cdot}x^{\{0,2,3\}}{\cdot}u^1 + 2{\cdot}x^{\{0,1,2\}}{\cdot}v^1 + 2{\cdot}w^1)$. Note that sometimes factorization is not unique. For example, if also value 4 were allowed, then the first coefficient of the first expression could be 4 as well as 3.

Thus satisfiable-matrix method can be used on Post expressions for factorization, inexact division, just as it is used in i-set mode.

A branch and bound technique is used in implementing the satisfiable-matrix method to speed up the search for the largest matrix. In the Post mode, both the value condition and the constant condition can be used to identify candidate Post cubes for a matrix cell. We give a method to check the constants. This method can be combined with those for checking the value condition to form the bounding part of the algorithm for Post mode factorization.

Suppose for each row, we store the maximum constant in the row so far constructed and call it r_i. Similarly for a column, and call it c_j. In selecting a cube for A_{ij}, besides checking for the value condition, we check the constant as follows. Let the coefficient of the proposed cube be c.

1. If $c < \min(r_i, c_j)$ or $c < \min(r_i, c_j)$, then reject the cube.
2. At this point, either $r_i < c$ or $c_j < c$.
    a. If $r_i < c$ then test all entries in the row i and see if $\eta(A_{i,k}) = \min(c, c_k)$ for $k < j$. If this holds, replace r_i by c and accept the cube. Otherwise reject the cube.
    b. (here $c_j < c$) Test all entries in column j and see if $\eta(A_{k,j}) = \min(c, c_k)$ for $k < i$. If this holds replace c_j by c and accept the cube. Otherwise reject the cube.

Such checking has a linear time complexity to the size of satisfiable-matrix.

2.3.1.2 Maximum graph-matching method

The **maximum graph-matching** method is used for exact division, in which the divisor is given. It has the time complexity of $O(n^3)$ and is much faster than the satisifiable-matrix method. We show here that this algorithm can be used for exact division in the Post mode with little modification.

The problem of exact division can be defined as: given a set of Post cubes f and a set of divisor Post cubes $\{d_i, 1<i<n\}$, find the largest set of quotient Post cubes $\{q_j, 1<j<l\}$. The maximum graph-matching method is used only for the case $n = 2$. It can be outlined as:

1. Find all candidate Post cubes of f for d_1 and d_2. Any candidate cube c_{ij} is one satisfying $c_{ij} \le d_i$. Thus we get two groups of candidates $\{c_{1j}\}$ and $\{c_{2j}\}$.
2. For each candidate c_{ij} compute a candidate quotient q_{ij}, specified by a lower bound and upper bound: $c_{ij} \le q_{ij} \le \sigma(c_{ij}, \eta(c_{i,j})\cdot\overline{\gamma(d_i)})$, where $\bar{c}$ denotes the cube that contains all values not in c. For example, $\overline{a^{\{1\}}c^{\{2\}}} = a^{\{0,2\}}b^{\{\}}c^{\{0,1\}}$ if each variable has a range size of 3.
3. Identify compatible pairs. Two Post cubes, c_{1j}, c_{2k} are said to be compatible if $\sigma(c_{1j}, c_{2k}) \le \sigma(c_{1j}, \eta(c_{1,j})\cdot\overline{\gamma(d_1)})\cdot \sigma(c_{2k}, \eta(c_{2,k})\cdot\overline{\gamma(d_2)})$.
4. Reduce the problem to a bipartite graph: each candidate cube is a vertex. There is an edge between two cubes if and only if they are compatible.
5. Solve the maximum graph-matching problem. The cubes that are paired form the columns of a satisfiable matrix.

The only modification is the concept of Post supercube and the part of handling the constants in the Post cubes in the calculation of lower and upper bounds of quotients. The key part of this algorithm is about the compatibility of two Post cubes.

Lemma 2.2 If two Post cubes c_1 and c_2 are compatible, then there exists a quotient q such that $c_1 = q\cdot d_1$ and $c_2 = q\cdot d_2$.

Proof For two compatible Post cubes, we have $\sigma(c_1, c_2) \le \sigma(c_1, \eta(c_1)\cdot\overline{\gamma(d_1)})\cdot \sigma(c_2, \eta(c_2)\cdot\overline{\gamma(d_2)})$. Let $q = \sigma(c_1, c_2)$. Then $q\cdot d_1 = \sigma(c_1, c_2)\cdot d_1 \ge c_1\cdot d_1 = c_1$. Since $\overline{\gamma(d_1)}$ only contain values that are not included in $\gamma(d_1)$, $q\cdot d_1 \le \sigma(c_1, \eta(c_1)\cdot\overline{\gamma(d_1)})\cdot d_1 = \eta(c_1)\cdot$ $supercube(\gamma(c_1),\overline{\gamma(d_1)})\cdot d_1 = \eta(c_1)\cdot(supercube(\gamma(c_1),\overline{\gamma(d_1)})\cdot\gamma(d_1)) = \eta(c_1)\cdot\gamma(c_1) = c_1$. So $c_1 = q\cdot d_1$. Similarly, we can show $c_2 = q\cdot d_2$.

Lemma 2.2 also provides a method to compute the quotient q for two compatible Post cubes. In the following we show an example of this algorithm.

Example 2.7 Exact division using the maximum graph matching method.

$$f = 3\cdot x^{\{0,3\}}\cdot u^{1} + 2\cdot x^{\{0,1\}}\cdot v^{1} + 1\cdot x^{\{0,2,3\}}\cdot y^{1}\cdot u^{1} + 1\cdot x^{\{0,1,2\}}\cdot y^{1}\cdot v^{1}$$

$$d = 3\cdot x^{\{0,1,3\}} + 1\cdot y^{1}$$

- Find candidate cubes. Since $3\cdot x^{\{0,3\}}\cdot u^{1} \leq d_{1} = 3\cdot x^{\{0,1,3\}}$, and $2\cdot x^{\{0,1\}}\cdot v^{1} \leq d_{1}$, they are the candidate cubes for d_1. Similarly, $1\cdot x^{\{0,2,3\}}\cdot y^{1}\cdot u^{1}$ and $1\cdot x^{\{0,1,2\}}\cdot y^{1}\cdot v^{1}$ are candidate cubes for d_2.
- Compute the lower and upper bounds of the quotient cubes. We use q_{jlow} and q_{jup} to refer to the lower and upper bound of the jth cube.

	1 $3\cdot x^{\{0,2,3\}}\cdot u^{1}$	2 $2\cdot x^{\{0,1\}}\cdot v^{1}$	3 $1\cdot x^{\{0,2,3\}}\cdot y^{1}\cdot u^{1}$	4 $1\cdot x^{\{0,1,2\}}\cdot y^{1}\cdot v^{1}$
lower bound	$3\cdot x^{\{0,3\}}\cdot u^{1}$	$2\cdot x^{\{0,1\}}\cdot v^{1}$	$1\cdot x^{\{0,2,3\}}\cdot y^{1}\cdot u^{1}$	$1\cdot x^{\{0,1,2\}}\cdot y^{1}\cdot v^{1}$
Upper bound	$3\cdot x^{\{0,2,3\}}\cdot u^{1}$	$2\cdot x^{\{0,1,2\}}\cdot v^{1}$	$1\cdot x^{\{0,2,3\}}\cdot u^{1}$	$1\cdot x^{\{0,1,2\}}\cdot v^{1}$

Table 2.3 Lower and upper bounds of the quotient cubes

- Then look for compatible pairs. We can see that $\sigma(q_{1low}, q_{3low}) = \sigma(q_{1up}, q_{3lup}) = 3\cdot x^{\{0,2,3\}}\cdot u^{1}$, and $\sigma(q_{2low}, q_{4low}) = \sigma(q_{2up}, q_{4lup}) = 2\cdot x^{\{0,1,2\}}\cdot v^{1}$. Hence these are the compatible pairs. Actually they are the maximal compatible pairs. For a larger example, we can build a graph and use the maximal graph-matching algorithm to find them. Thus the exact division result is $q = 3\cdot x^{\{0,2,3\}}\cdot u^{1} + 2\cdot x^{\{0,1,2\}}\cdot v^{1}$, and $r =$ null, i.e., $f = (3\cdot x^{\{0,1,3\}}\cdot u^{1} + 1\cdot y^{1})\cdot(3\cdot x^{\{0,2,3\}}\cdot u^{1} + 2\cdot x^{\{0,1,2\}}\cdot v^{1})$.

Thus we have extended the main algorithms for semi-algebraic optimization used in MVSIS from the i-set mode to the Post mode. Some new features were introduced in this extension. First, the divisor obtained from the satisfiable-matrix method is no longer a binary output function, but a multi-valued output function. However, extractions of such divisors can result in functions of the following form: $f = d\cdot(q_1 + q_2 \ldots + q_n) + r$, where d is a multi-valued variable. This is no longer a Post expression as defined in Section 2.2 since multi-valued variables are used in cubes besides literals; a multi-valued variable is not a literal. If we restrict the result to contain only literals, d has to be replaced by $\sum_{l=0}^{n-1} l \cdot d^{\{l\}}$ in some cubes, so such extractions may not be a gain as far as the network cost is concerned.

Another feature is that the extraction of a common sub-expression can be done considering the entire multi-valued output function instead of separate i-sets. In the i-set mode, a multi-valued output function is treated as a set of binary output functions, each for one i-set. This restricts the search for good common sub-expression, e.g., the sub-expression cannot contain cubes from different i-sets. However, in the Post mode, all the cubes of a multi-valued output function are considered simultaneously. However, at this point it is unclear what difference such features would make in practical applications.

2.3.2 Don't care-based logic network minimization

Algebraic methods can be used to derive an appropriate structure for the MV-network. Once the structure has been decided, the multi-valued function at each node can be optimized using the maximal permissible behavior allowed for this node. This flexibility is given by satisfiability don't cares (SDC), observability don't cares (ODC) and observability partial cares (OPC) [JB00]. Such optimization is called don't care-based minimization and forms another important part of MV function optimization. In MVSIS, command *simplify* and *fullsimp* are implementations of such minimization methods.

An SDC is easy to compute. For a MV function node y_i in i-set mode, let $(f^0, f^1, \ldots, f^n)$ denote its i-sets. Suppose its input variables are $\{x_1, x_2, \ldots, x_r\}$ and $x_j \in \{0, \ldots t_j\} \equiv P_j$. f^l is a binary function $P_1 \times P_2 \times \ldots . \times P_r \rightarrow B$ which defines the set of minterms in $P_1 \times P_2 \times \ldots . \times P_r$ that produce output value l for y. We can express the SDC for y as:

$$SDC_{y_i} = \sum_{l=0}^{n} y_i^{U-\{l\}} f^l$$

where U represents the universal set {0, 1, …, n}, since the SDC just expresses variable combinations that can't happen. Here, the SDC expression is in SOP form rather than a MAX of MIN form.

In Post mode, such a function is given as

$$y_i = n \cdot g^n + (n-1) \cdot g^{n-1} + ... + 1 \cdot g^1$$

where $\cdot$ and + are **MIN** and **MAX**, and g^l can treated as a binary function since it takes on either value n or value 0. Priority don't cares may have been used to simplify function y, so $f^l \subseteq g^l \subseteq f^l \cup \sum_{i=n}^{l+1} f^i$.

Lemma 2.3 The SDC we can get directly from the Post expression of y is:

$$SDC'_{y_i} = \sum_{l=0}^{n} y_i^{\{0..l-1\}} g^l$$

where $\{0..l-1\}$ represents the set of values from 0 to l-1.

Proof We want to use SDC to express variable combinations that can't occur. Since $f^l \subseteq g^l \subseteq f^l \cup \sum_{i=n}^{l+1} f^i$, the maximum onset for g^l is $\sum_{j=n}^{l} f^j$. So we have g^l=1 $\Rightarrow$ $y_i \notin \{0,...,l-1\}$. This is expressed by SDC'_{y_i}. $y_i^{\{n..l+1\}} g^l$ can occur and thus cannot be included in SDC. So SDC'_{y_i} is the largest SDC we can obtain in the Post mode.

In the above we tried to obtain an expression of SDC without first converting to i-set mode and then computing SDC. We can see in the following example that because of the use of priority don't cares, we lose a part of SDC to keep the calculation correct.

Example 2.8 The SDC of Post mode function $y = 2 \cdot a^{\{1\}} b^{\{0\}} c^{\{1\}} + 1 \cdot a^{\{1\}}$ is SDC$_y$ = $y^{\{0,1\}} \cdot a^{\{1\}} b^{\{0\}} c^{\{1\}} + y^0 \cdot a^{\{1\}}$. $a^{\{1\}}$ includes the whole function f^2 to obtain the simplest form, and $y^2 \cdot a^{\{1\}}$ contains $y^2 \cdot a^{\{1\}} b^{\{0\}} c^{\{1\}}$ so it cannot be included in SDC. This causes the loss of SDC part $y^2 \cdot (a^{\{1\}} b^{\{1\}} + a^{\{1\}} b^{\{1\}} c^{\{1\}})$.

The computation of CODC is more complicated. The maximum ODC (**MODC**) for the input edge x_j is the set of minterms in the primary input space, such that the output MV-function y is insensitive to all values of x_j, i.e., for any such minterm, no matter how x_j is changed to some other value, the value of y does not change. This set of minterms can be used as don't cares for the minimization of the source function of x_j, since the minterms have no effect on the output value of y. We first compute the set of don't care minterms MODC in the local input space of y_i, under which the value of x_j are indistinguishable. This gives the maximal ODC for edge $x_j \to y_i$. MODC is defined as:

$$MODC(y_i, x_j) = \{m \mid f(m[x_j = 0]) = \ldots = f(m[x_j = t_j]), m \in P_1 \times \ldots. \times P_r\}$$

Here the notation $m[x_j = t]$ is the minterm obtained from *m* by changing x_j to the value *t*. It can be proved that MODC can be computed using multi-valued cofactoring:

$$MODC(y_i, x_j) = \sum_{l=0}^{n} \prod_{k=0}^{t_f} f^{l}_{x_j^k}$$

The validity of MODC's for a particular input edge requires other input edges to produce certain values. So we need the MODC's to be "compatible" with each other. One approach is to implicitly order the input edges and to compute the CODC for each input edge by making the associated MODC compatible with all the preceding edges in the ordering. Given an ordering $x_1 \prec \ldots \prec x_j \prec \ldots \prec x_r$, the CODC for edge x_j is defined as:

$$CODC(y_i, x_j) = \{m \in MODC_j \mid \forall l < j, (m \notin CODC_l) \vee$$

$$(\text{for any value } t \text{ of } x_l,\ m[x_l = t] \in MODC_j)$$

It can be shown that CODC can be computed using the following approach:

$$CODC(y_i, x_j) = P_1(P_2(\ldots. P_{j-1}(MODC(y_i, x_j)))) + CODC_{y_i}$$

$$P_k(F) = \overline{CODC_{x_k}} \cdot F + \forall x_k \cdot F$$

$$CODC_{x_j} = \prod_{i \in fanout(x_j)} CODC(y_i, x_j)$$

We give the algorithm to compute CODC's that has been implemented in MVSIS [JTHE] in Figure 2.1.

Let's see if we can extend this complex algorithm to Post mode. The first step is to compute MODC and CODC in the local input space.

Theorem 2.2 If MV-function y_i has the Post expression, i.e., $y_i = n \cdot g^n + (n-1) \cdot g^{n-1} + \ldots. + 1 \cdot g^1$, the MODC can be computed as:

$$MODC(y_i, x_j) = \sum_{l=0}^{n} \left(\left(\prod_{k=0}^{t_f} g^{l}_{x_j^k} \right) \cap \overline{\sum_{p=l+1}^{n} \sum_{k=0}^{t_f} g^{p}_{x_j^k}} \right)$$

Proof: We show that $\left(\prod_{k=0}^{t_f} g^l_{x^k_j}\right) \cap \overline{\sum_{p=l+1}^{n}\sum_{k=0}^{t_f} g^p_{x^k_j}} = \prod_{k=0}^{t_f} f^l_{x^k_j}$. Since $f^l \subseteq g^l \subseteq f^l \cup \sum_{i=n}^{l+1} f^i$, for any g^l, and $f^l \cap \sum_{i=n}^{l+1} g^i = null$, any minterm m in $\prod_{k=0}^{t_f} f^l_{x^k_j}$ must be in $\prod_{k=0}^{t_f} g^l_{x^k_j}$ while not in $\sum_{p=l+1}^{n}\sum_{k=0}^{t_f} g^p_{x^k_j}$. Otherwise, if $m \notin \prod_{k=0}^{t_f} g^l_{x^k_j}$, we have that $x^k_j \cdot m \in \prod_{k=0}^{t_f} f^l_{x^k_j}$ while $x^k_j \cdot m \notin \prod_{k=0}^{t_f} f^l_{x^k_j}$. This contradicts that $f^l \subseteq g^l$. Similarly, we can show that $m \notin \sum_{p=l+1}^{n}\sum_{k=0}^{t_f} g^p_{x^k_j}$. Finally any minterm m' in $\left(\prod_{k=0}^{t_f} g^l_{x^k_j}\right) \cap \overline{\sum_{p=l+1}^{n}\sum_{k=0}^{t_f} g^p_{x^k_j}}$ must be in $\prod_{k=0}^{t_f} f^l_{x^k_j}$ because otherwise we have $x^k_j \cdot m \in \prod_{k=0}^{t_f} g^l_{x^k_j}$ while $x^k_j \cdot m' \notin f^l \cup \sum_{i=n}^{l+1} f^i$.

The computation shown in Theorem 2.2 in some sense implicitly converts from the Post mode to the i-set mode. So it seems more complicated than in the i-set mode. Note that the size of $\sum_{k=0}^{t_f} g^p_{x^k_j}$ cannot exceed that of g^p. Thus the above method of calculating MODC is at least as fast as the straightforward way, which explicitly converts y_i to i-set mode and then calculates the MODC. During the convertion to i-set mode, we need to use complement too: $f^l = g^l \cap \overline{\sum_{i=n}^{l+1} g^i}$.

Once we have computed the MODC, the CODC in the local space can be computed using the algorithm in i-set mode without modification. Then we need to map CODC into the primary input space by variable substitution, complement the result to obtain the care set, and then map it to the local input space by image computation. Mapping CODC into the primary input space is realized by variable substitution, i.e., we replace any literal of intermediate nodes $y^{\{l\}}$ by its function $f^{\{l\}}$ until we reach the primary inputs. For image computation, recursive range computation is used in MVSIS. In the local input space of y_i, each input variable is cofactored by *A(x)*, the complement of the don't care set. A generalized cofactor operation is used here. This array of cofactored functions gives the transition functions that map the entire primary input space *PI* into the local care set of node y_i.

$$F_{A(x)} = \left[(f_1)_{A(x)}, (f_2)_{A(x)}, \ldots, (f_r)_{A(x)}\right]$$

$$A(x) = \overline{CODC_{PI}^{y_i}}(x)$$

Once we have the range function, we apply output cofactoring to carry out the recursive image computation:

$$\begin{aligned} &\overline{CODC_{local}^{y_i}} \\ &= IMAGE(\overline{CODC_{PI}^{y_i}}) = RANGE(F_{A(x)}) = RANGE(f_1, f_2, \ldots f_r) \\ &= \sum_{k=0}^{|P_1|} y_1^k \cdot RANGE([f_2, \ldots, f_r]_{f_1^k}) \end{aligned}$$

The range computation is applied recursively to the list of functions, successively cofactored. We give a more detailed analysis on these two processes to see if extension to Post mode is possible.

The first step is to collapse CODC into the primary input space, where we need to use the function for literals, such as $f^{\{l\}}$. Unfortunately, it seems impossible for one to get exact $f^{\{l\}}$ from $g^{\{l\}}$ without converting back into i-set mode. Since $f^{\{l\}} \subseteq g^{\{l\}}$, if we use $g^{\{l\}}$ to replace $f^{\{l\}}$, we will get a larger CODC set in the primary input space. So when we use $A(x) = \overline{CODC_{PI}^{y_i}}(x)$ to compute the care set, we lose some part of the real care set which is unacceptable. One idea to fix this would be to try to recover the lost care part during the image computation. However, from the following graph, we can see that it is impossible.

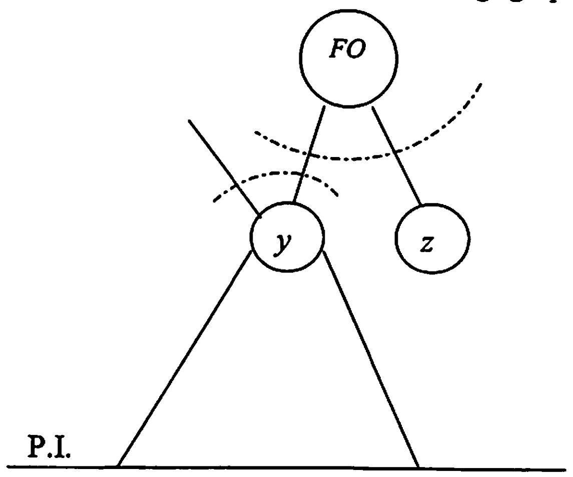

Figure 2.2 CODC collapsing

Suppose we have computed the CODC for *y* in terms of the inputs to the fanout cone of *y*. Node *z* which is a fanin to this CODC can be outside of the transitive fanins of *y*. So when we do collapse using the Post mode functions, some care set may be lost due to node *z*. In image computation, we only use the nodes in the transitive fanins of *y*, and thus this could not recover the lost part due to *z*.

We conclude that the algorithm based on collapsing and image computation cannot be extended to the Post mode. The main reason is that we cannot get the exact function of a literal during collapsing. Therefore it seems the best way to use CODCs for simplifying multi-valued nodes in Post mode is to convert the network to i-set mode, use the previous algorithm, and convert back.

```
Algorithm [CODC-based MV-network minimization]:
Input: MV-network ntk, external don't care $XDC_j$ at each primary output j
Local $CODC_i$: CODC set for node i
Local $DC_i$: complete don't care set for node i
Traverse each node j in ntk in reverse DFS order
    If j is primary output
        $CODC_j = XDC_j$
        Continue;
    For each fanout node k
        $D = MODC(f_k, y_j)$
        For each fanin node i of k that is already visited
            $D = [complement(CODC_i) + \forall y_i] \cdot D$
        $D = D + CODC_k$;
        $CODC_j = CODC_j \cap D$;
    Collapse $CODC_j$ into primary input space
    Remove the supporting variables not in the transitive fanin cone of $y_j$
    $DC_j = \neg image(\neg CODC_j)$
    $MINIMIZE(ONSET_j, DC_j)$
End
```

Figure 2.1 CODC-based MV-network minimization algorithm

2.3.3 Other commands in MVSIS

Besides semi-algebraic optimization and don't care-based minimization, there are other commands in MVSIS, such as *merge*, *collapse*, *eliminate*, *pair_decode*, and *eliminate_part*. We would like to extend them to Post mode if possible.

merge is used to combine small nodes into one larger node. Given *n* nodes, np_1, $np_2,\dots,np_m$, each takes values from the set P_1, P_2, …, P_n respectively. The merged node *new_np* has the i-set functions as:

$$f_{new\_np}^{\{0\}} = f_{np1}^{\{0\}} \cap f_{np2}^{\{0\}} \cap \dots \cap f_{npn}^{\{0\}}$$

$$\dots\dots$$

$$f_{new\_np}^{\{|P_1|\times\dots\times|P_n|-1\}} = f_{np1}^{\{|P_1|-1\}} \cap f_{np2}^{\{|P_2|-1\}} \cap \dots \cap f_{npn}^{\{|P_n|-1\}}$$

Theorem 2.3 If the given nodes are all in Post mode, then the merged node obtained by the following method is a node in Post mode which has the same functionality as node *new_np* shown above.

Given nodes in Post mode:

$$y_{npi} = (|P_n|-1)\cdot {g_{npi}}^{|P_n|-1} + (|P_n|-2)\cdot {g_{npi}}^{|P_n|-2} + \dots. + 1\cdot {g_{npi}}^{1}$$

The merged node can be computed as:

$$y_{npi} = (|P_1|\times..\times|P_n|-1)\cdot g_{new\_np}^{|P_1|\times\dots\times|P_n|-1} + (|P_1|\times..\times|P_n|-2)\cdot g_{new\_np}^{|P_1|\times\dots\times|P_n|-2} + \dots. + 1\cdot g_{new\_np}^{1}$$

where $g_{new\_np}^{i_1\times|P_2|\times\dots\times|P_n|+i_2\times|P_3|\times\dots\times|P_n|+\dots+i_n} = {g_{np1}}^{i_1} \cap \dots. \cap {g_{npn}}^{i_n}$.

Proof: We only need to show that $f^l \subseteq g^l \subseteq f^l \cup \sum_{i=|P_1|\times\dots\times|P_n|-1}^{l+1} f^i$ for any l. Suppose $l = i_1\times|P_2|\times..\times|P_n| + i_2\times|P_3|\times..\times|P_n| + \dots + i_n$, then $g_{new\_np}^{l} = {g_{np1}}^{i_1} \cap \dots. \cap {g_{npn}}^{i_n}$ and $f_{new\_np}^{l} = {f_{np1}}^{i_1} \cap \dots. \cap {f_{npn}}^{i_n}$. And we know that $f_{npj}^{i_j} \subseteq g_{npj}^{i_j} \subseteq f_{npj}^{i_j} \cup \sum_{i=|P_j|-1}^{i_j+1} f_{npj}^{i}$, so $f_{new\_np}^{l} \subseteq g_{new\_np}^{l}$. We can also get

$$g_{new\_np}^{l} \subseteq \bigcap_{j=1}^{n} f_{npj}^{i_j} \cup \sum\nolimits_{i=|P_j|-1}^{i_j+1} f_{npj}^{i} \subseteq \sum_{k_j=i_j}^{|P_j|} \bigcap_{j=1}^{n} f_{npj}^{k_j} = \sum_{i=|P_1|\times\ldots|P_n|-1}^{l} f_{new\_np}^{i} .$$

Of course this may not be the simplest form; and the new node possibly could be further simplified using priority don't cares.

collapse is the inverse operation of substitution. If two nodes are given as arguments, the fanin node is collapsed into the fanout node so that the fanout node is not dependent on the fanin node any more. Since the fanin node is usually referred to in the literal form in the fanout node, it seems that we have to convert the fanin node back into the i-set mode to get the function of its literals. No other method is known as far as we know. Actually the failure in extending the CODC computation was mainly because we did not find a way to do collapse in the Post mode.

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 that node is eliminated. Unfortunately, this command is based on *collapse*. We eliminate a node by collapsing it into all its fanout nodes and then delete it from the network. So again there seems no easy way to extend it to the Post mode. A similar thing happens to *eliminate_part*, a newly added command to reduce the number of values of a node. It eliminates the values of a node whose cost does not exceed a given threshold. When eliminating those values, we probably need to collapse the function of those values into the fanout nodes. Thus converting back to i-set mode seems necessary again.

pair_decode is a newly added command to create multi-valued nodes. It searches the whole network for the "best" pair of nodes, and creates a new node whose i-sets equal to all the decodes of the two nodes, and then re-substitutes the new node into the fanouts of the pair to get some saving. This command is introduced in more detail in Section 3.6. We will see that the method for calculating the value of a pair can be extended to Post mode without modification. Thus this command can be used for Post mode.

In summary, we examined all the optimization commands in MVSIS and made attempts to extend them to the Post mode format. We successfully extended semi-algebraic optimization methods while difficulties were found in the extension of don't care-based minimization methods, specifically, the CODC computation. Other commands were examined and analysis of their extension was given. The main difficulty encountered is the collapse process, which is unique to multi-leveled logic and is used to compute the function of a node in terms of primary inputs.

Chapter 3

Implementation in MVSIS

In this chapter we describe some further development of MVSIS. MVSIS already contains commands for node simplification and sub-expression extraction, which are quite powerful for multi-valued multi-level logic minimization. However, MVSIS cannot perform formal verification or handle external don't cares. To do formal verification, the user can write out BLIF-MV files, read it into VIS, and execute a verification command, but this is cumbersome. At the end of this chapter, we show some experimental results of an MVSIS implementation of some operations extended to Post mode. However, most of the extension work of Chapter 2 has not been implemented yet.

3.1 Cost function

We introduced the structure of multi-valued logic in MVSIS in Chapter 1. Before presenting our implementations, we discuss the cost functions used to evaluate the minimization results. The optimization criteria are functions of the total number of nodes and the size of the MV-function contained in each node. The exact cost function depends on the final type of target implementation. Minimizing the number of cubes or literals is generally beneficial for both hardware and software. For some applications, we want to minimize the number of cubes while paying relatively less attention to the number of literals. One example application is optimization in a compiler for control dominated software applications where code size or evaluation time is key, while time for compilation can be relaxed. For other applications, such as hardware applications, we would focus on reducing the number of literals.

MVSIS provides two cost functions: one is the number of cubes, the other is the number of literals in factored form. Users can specify which cost function to use.

3.2 *Qcheck*

MVSIS guarantees the result of synthesis to be consistent with the input only if the input logic network is deterministic. We call a network is deterministic if for any input vector, each primary output has exactly one value. Otherwise, because of the use of default value and other optimization methods, the resulting logic could be inconsistent with the original.

So it is desirable to include a command to quickly check if the input network is deterministic. *qcheck* is such a command which is always called automatically by *read_blifmv* since it is very fast even for large networks. In this command, we examine each node to see if its i-sets are orthogonal. If overlap parts exist, these parts are removed and put into the default i-set. Then the new network is compared with the original network using simulation. If the two networks are reported inconsistent, then the original network definitely was non-deterministic; hence a warning message is printed. The default value of 1000 is used for number of simulation vectors.

The fact that a network contains external don't cares does not immediately imply that it is non-deterministic. The argument for this is that external don't cares correspond to some part of the input space which the user does not care about (possibly because it can't occur), and we are concerned only with the care part being deterministic. So we do the same check on networks with external don't cares but restrict the simulation vectors to be in the care part of the input spaces.

Notice that *qcheck* is not a definitive test in the sense that a network can pass *qcheck* but still be non-deterministic since simulation instead of formal verification is used to compare the two networks. The purpose of this command is only to check determinism of input networks quickly as a first check after we read them in.

3.3 *eliminate_part*

eliminate_part is like *eliminate*, except it works only on multi-valued output nodes (not binary nodes) and can eliminate some of the i-sets of the nodes. *eliminate_part* can be used to reduce the number of values of multi-valued nodes, even to the point that the node becomes binary. This command examines each internal multi-valued node in the network and eliminates the i-sets whose value sum does not exceed a specified threshold. If the value of 2 or more i-sets in a node is less than or equal to the threshold, those i-sets are eliminated by two steps: a) collapsing their functions into the fanout nodes if necessary; b) merging their functions into one new i-set to replace them. The command iterates, since eliminating some i-sets may change the values of other i-sets. The iteration continues of all no further change in the network occurs.

The value of an i-set in node m is defined as the worst-case cost increase for eliminating this i-set. The worst case happens when we have to collapse the i-set into the fanout nodes of node m for each of its appearances while each literal representing m is still needed. For example, literal $m^{\{0,1\}}$ is used in its fanout node g. The worst case of eliminating $m^{\{0\}}$ happens when we have to collapse the function of $m^{\{0\}}$ into g and literal $m^{\{1\}}$ is still needed. So the value is actually an upper bound on the increase in cost. Real increase can be much lower because of three reasons: 1) the selection of a new default value, 2) the use of the new i-set (in the above example, if we finally decide to merge $m^{\{0\}}$ and $m^{\{1\}}$, then no collapse is needed) and 3) the simplification of the new i-set.

Note that we define the elimination of a subset of i-sets as merging them into one single i-set. So actually we should evaluate all possible subsets of i-sets to determine which should be eliminated. Since the complexity of such evaluation is exponential in the number of values of a node, we use estimations. We first calculate the value of each single i-set and order them in increasing order. Then we find the largest j so that the sum of the values of the first j i-sets is less than or equal to the threshold. Here we take into consideration the possible saving by using the i-set as new default. Such estimation is conservative, but still gives a good direction on how to eliminate the values of multi-valued nodes to control the cost increase. The overview of *eliminate_part* is as follows.

```
// node m contains n i-sets
value[n] = single_value_calculate( m ); // this array contains the values of i-sets.
(value[n], order[n]) = increase_order(value[n]);
max = 0;    // max keeps the number of i-sets that can be eliminated
For i from 1 to n-2
        Cost = $\Sigma_{0..i}$ value[i] – default_saving
        If cost <= threshold
                max = i;
If max >= 1
        merge i-sets from order[0] to order[ max];
        update fanout nodes of m and default value of m;
```

Figure 3.1 *eliminate_part* procedure on a single node

To see the accumulated value of the i-sets, the command *print_part_value* can be used. It shows the ordering of the i-sets and the accumulated values from least to greatest. For example, for a particular node *m* it might print out the following,

m: (110) 12 30 50 68 112 136 160 184 (6 3 0 7 8 5 2 1 4)

The name *m* is first followed by the normal value of the node (110), then a vector of the accumulated i-set values 12, 30, …., where 12 is the value of the 6-set, 30 is 12 plus the value of the 3-set, etc. If the command *eliminate_part* 30 is given, MVSIS will merge the 6-set and the 3-set, resulting in a new node with 8 i-sets. Then a new default value will be selected, since the merged i-set may be the largest.

3.4 External don't cares

A previous version of MVSIS only supported inputs that are completely specified and only deterministic multi-valued logic. We extend MVSIS to permit user specified external don't cares and to be able to extract external don't cares from incompletely specified networks. Command *fullsimp* is modified correspondingly to use external don't cares as a part of the compatible observability don't cares.

Command *read_blifmv* is extended to recognize the syntax *.exdc* in the input .blif or .blifmv file, which can be used to specify external don't cares. The user can also specify don't cares by putting '-' on the primary output and the corresponding cubes are automatically recognized as external don't cares. An example of such an input file is shown in the following example.

Example 3.1 a27.blif

```
model s27.bench
.inputs G0 G1 G2 G3
.outputs G17
.wire_load_slope 0.00
.latch   G10 G5  0
.latch   G11 G6  0
.latch   G13 G7  0
.names G11 G17
0 1
.names G14 G11 G10
00 1
01 -
........
.exdc
.inputs G6 G5
.outputs G10 G11 G13 G17
.names G6 G5 dc[1093]
11 1
.names dc[1093] G10
1 1
.names dc[1093] G11
1 1
.names dc[1093] G13
1 1
.names dc[1093] G17
1 1
.end
```

Table 3.1 An input file with external don't cares

An external don't care network will be created as a part of the main network if there are such specifications in the input file. The EXDC network can be multi-level. Each primary output node is a binary node and its onset specifies the don't care set for the output node which has the same name in the main network. For sequential logic, the latch inputs are treated as primary outputs and can have corresponding nodes in the external don't care network.

We can extract external don't cares from some incompletely specified networks. If one primary output is not specified on some part of the primary input space, that part can be extracted to become the external don't care for this primary output. However if the initial network is multi-level, to determine such unspecified input minterms is based on global sensitization and becomes very complicated. Another possible way is to collapse the whole network into a two-level network and then extract the unspecified part, but such an operation can be quite expensive too. So currently we only do such extraction on two-level networks. In such networks, each primary output depends directly on primary inputs, the unspecified part of the output node can be easily obtained by complementing the OR of all i-sets. This routine is called automatically by *read_blifmv*. The extracted external don't cares are merged with any external don't care network specified.

Command *write_blifmv* is also extended to write out the external don't care network in the same format as shown in the above example.

3.5 Formal verification

MV-networks can be verified in MVSIS either by simulation or by formal verification based on MDD comparison. The command for verification is:

validate [-m method] [-n num] [-b] file1 [file2]

file1 is compared with the current network when *file2* is not specified. The primary input and output variables of the two networks are associated by their names. The *-m* option specifies the verification method. If method is *sim* (default), two networks are simulated using random vectors. If method is *mdd*, the MDD's for the global functions at the primary outputs are constructed for both networks and compared. The *-b* option indicates that the file(s) are in BLIF format. The default is BLIF-MV. The *-n num* specifies the number of random vectors to be used in the simulation. The default is 1000.

Formal verification is based on MDD operations [KB90]. In formal verification, we first assign variable indexes to all the primary inputs of the two networks. The primary inputs with the same names are assigned the same index and thus will be treated as the same variable by the MDD package. Then we traverse the whole network from primary inputs to primary outputs and build up the local MDD for each node function. Then we eliminate internal variables using MDD operations to obtain the global functions of the primary outputs.

Formal verification works not only on combinational networks but also sequential networks. It also permits input networks to contain external don't care networks. For sequential networks, full sequential equivalence is not checked in general. We simply treat the latches as primary inputs and the latch inputs as primary outputs. The number of latches is compared first, and then the global functions for latch inputs. Further development is needed here. If external don't care networks exist, the global function for the external don't care network is subtracted from the global network functions to obtain global care functions, and these are compared. If the comparison result is that two networks are different, one minterm in the primary input space on which the two networks yield different output values will be printed. The main procedure is as follows.

```
Formal verification( network1, network2 ) {
        Compare primary inputs and primary outputs;
        If different then return FALSE;
        If networks are sequential then
              Compare latches;
              If different then return FALSE;
        Assign indexes to the primary inputs and latches of network1;
        Based on the assignment to network1, assign indexes to PIs and latches of
network2;
        // it is very important that variables that match by name are assigned the
same index so that they are treated as the same variable by BDD package.
        If exdc networks exist then
              Assign indexes to PIs based on the assignment of network1;
        For each PO o1 in network1 and PO o2 that matches it by name in network2
              For the ith  i-set of o1 and o2
                    Mdd_compute( o1, i,  mdd1 );  // mdd1 is the MDD for this i-set
                    Mdd_compute(o2, I, mdd2 );
                    Mdd_exclude( mdd1, exdc_mdd1);
                    Mdd_exclude( mdd2, exdc_mdd2);
                    if( ! mdd_equal( mdd1, mdd2) )
                          print a vector;
                          return FALSE;
        return TRUE;
}
```

Figure 3.2 Formal Verification Procedure

Formal verification is usually called after the execution of optimization commands to verify the correctness of all the operations.

3.6 *Pair_decode*

pair_decode does bit pairing to create new multi-valued nodes. It looks for a "best" pair of signals to pair together. Then it creates a new node (if its value is greater than a specified threshold) with i-sets equal to all the decodes of the pair. This new node then is algebraically substituted into nodes which depend on at least one of these decodes. Finally, any set of values of the new node, which always appear together in the fanouts, is merged into a single value of the new node. After this step, *simplify* should be executed to effect full substitution.

The format of this command is:

pair_decode [-i num] [-t num] [threshold]

The threshold option is used to specify the upper bound of the cost increase due to creating the new node. Its default value is 0. The *-i* option can be used to specify the number of iterations. The default case is to continue iterating until the network does not change. The *-t* option specifies a time limit in terms of seconds for the whole process. When a timeout occurs, the intermediate result obtained so far is returned.

Pairing two nodes into one can bring savings. A simple example is shown in the following, in which we suppose a and b are binary signals.

Example 3.2 $g = a^{\{0\}}b^{\{0\}}c^{\{1\}}d^{\{1\}} + a^{\{1\}}b^{\{1\}}c^{\{1\}}d^{\{1\}}$

$\Rightarrow$ $newab^{\{0\}} = a^{\{0\}}b^{\{0\}}$

$newab^{\{1\}} = a^{\{0\}}b^{\{1\}}$

$newab^{\{2\}} = a^{\{1\}}b^{\{0\}}$

$newab^{\{3\}} = a^{\{1\}}b^{\{1\}}$

$g = newab^{\{0,\,1\}}c^{\{1\}}d^{\{1\}}$

$$\Rightarrow \quad newab^{\{0\}} = a^{\{0\}}b^{\{0\}} + a^{\{1\}}b^{\{1\}}$$

$$g = newab^{\{0\}}c^{\{1\}}d^{\{1\}}$$

Pairing a and b enables the merge of two cubes in g. Since the values 0 and 1 of newab are always used together, they are merged into a single value. Thus *newab* can be simplified. The total number of literals reduces from 8 to 7. We have found that *pair_decode* is quite useful in finding XOR sub-expressions in some examples.

```
// estimate the savings of pairing up a and b
Pair_value(a, b) {
    // a has m values and b has n values. They can be MV nodes
    Value = 0;
    For each node out with a and b as inputs
        Value =+#literals ( or #cubes);
        Remove a and b from all cubes and eliminate all duplicated cubes;
        Value = - #literals ( or #cubes );
    If cost = #literals then
        Value = -2*(m*n-1);        // #literals in newab
    Else // cost = #cubes
        Value = -(m*n-1);          // #cubes in newab
    Return value;
}
pair_decode( network ) {
    while( time_remaining and changed and iteration_remaining) {
        choose the best pair a and b  using procedure pair_value;
        create newab from a, b;
        use node_simplify to resub newab into fanout nodes of a and b;
        merge values of newab if possible;
    }
}
```

Figure 3.3 *pair_decode* Procedure

The key part of this command is how to estimate the savings of a pair. We evaluate each pair of nodes by counting the savings introduced by merging cubes if they are paired up. This is a lower bound on the real saving. After the best pair is chosen and a new node has been generated, we use an existing procedure *node_simplify* in MVSIS to find fanouts for this new node, but only the fanout nodes of the original pair are considered as candidate fanouts for the new node. Such restriction cuts execution time and yet gives good performance. If the use of the new node in a candidate fanout can lead to any savings, *node_simplify* will automatically re-substitute the new node into it. In Figure 3.2, we present the main procedures of *pair_decode*.

```
UC Berkeley, MVSIS 0.95 (compiled 16-Oct-01 at 4:42 PM)
changing to short-name mode
mvsis> rl matmul-c
mvsis> s
mvsis> saf
matmul:  4 nodes,  4 POs,  96 cubes(sop),  320 lits(sop),  160 lits(fact.)
mvsis> pd 1
m{0} = a{0}e{2} + e{0}
m{1} = a{0}e{1}
m{3} = a{1}e{2} + a{2}e{1}
n{0} = a{0}f{2} + f{0}
n{1} = a{0}f{1}
n{3} = a{1}f{2} + a{2}f{1}
o{0} = e{0}c{2} + c{0}
o{1} = e{0}c{1}
o{3} = e{1}c{2} + e{2}c{1}
p{0} = f{0}c{2} + c{0}
p{1} = f{0}c{1}
p{3} = f{1}c{2} + f{2}c{1}
q{0} = b{0}g{2} + g{0}
q{1} = b{0}g{1}
q{3} = b{1}g{2} + b{2}g{1}
r{0} = b{0}h{2} + h{0}
r{1} = b{0}h{1}
r{3} = b{1}h{2} + b{2}h{1}
s{0} = g{0}d{2} + d{0}
s{1} = g{0}d{1}
s{3} = g{1}d{2} + g{2}d{1}
t{0} = h{0}d{2} + d{0}
t{1} = h{0}d{1}
t{3} = h{1}d{2} + h{2}d{1}
matmul:_ 12 nodes,  4 POs,  64 cubes(sop),  184 lits(sop),  160 lits(fact.)
```

Figure 3.4 an example for *pair_decode*

At first glance *pair_decode* looks similar to another command *merge* but they are different. First, *merge* will replace the pair of nodes with a new node, while *pair_decode* will keep this pair of nodes if eliminating them provides no cost savings. Also the methods to calculate savings are different. *merge* counts the saving of a pair of nodes by comparing the size of the resulting node with that of the two nodes, while *pair_decode* focuses on the impact of creating a new node, on the fanout nodes of the original pair and can result in impressive results in some cases. In Figure 3.3 we show an example where *pair_decode* can reduce the cost significantly.

3.7 Post mode

To extend MVSIS commands to Post mode operations, it is necessary to include commands for transformation between the i-set mode and the Post mode. This is because there is no effective way to extend some commands to Post mode, as shown in Chapter 2. The command *compress* that transforms a network in i-set mode to Post mode had already been included for use in code generation [DJR01]. We added a new command, *uncompress*, to transform in the other direction. Given an expression $y_i = n \cdot g^n + (n-1) \cdot g^{n-1} + ... + 1 \cdot g^1$ in Post mode, *uncompress* computes the corresponding expression in i-set mode by:

$$f^i = g^i \cap \overline{\sum_{l=i+1}^{n} g^i}$$

ESPRESSO is called to control an explosion in the number of cubes, which may be caused by complement and intersection operations.

As presented in Chapter 2, some MVSIS commands can be extended to Post mode, but the implementation for this has not been finished. However, some commands can be used directly on Post mode. They are *decomp* and *fx*. In these commands, each i-set of a node is treated as a separate function. For example, in the satisfiability matrix method, the search for a good divisor is limited to single i-sets. Then we can directly use them in Post mode. Note that these commands are not as powerful as the extensions discussed in Section 2.3.1 because the search space is limited to one compressed table entry, rather than the entire Post expression.

We mentioned that including Post mode operations in MVSIS has two positive effects: one is that the user can optimize Post mode logic directly; the other is to speed up some optimization operations since Post mode expressions can have smaller sizes than that for i-set mode. For some algorithms that are relatively time-consuming, such as the satisfiable-matrix method, Post mode can reduce execution times greatly. We did some experiments using *decomp* and *fx* to demonstrate the effectiveness of using the Post mode in improving time performance. Some interesting experiments are shown.

The MV examples used in the experiments come from 1) machine learning applications, 2) a set of multi-valued benchmarks distributed with VIS and 3) a set of examples from asynchronous applications. We did not include sequential examples since *decomp* and *fx* are mainly for combinational logic optimization. We compare two approaches, both with respect to performance and the run time:

Approach 1: *decomp*, *fx*, *eliminate*;

Approach 2: *compress*, *decomp*, *fx*, *uncompress*, *eliminate*.

The input files available are written in i-set mode, so we need to call *compress* to transform to Post mode first. The run time of Approach 2 includes the run time for *compress* and *uncompress*. *eliminate* is the last operation for both approaches since it is usually called after decomposition and kernel extraction to get rid of bad extractions. The initial characteristics of the examples are given in Table 4.1 and the experimental results are shown in Table 4.2.

	#cubes_sop	#literals_sop	#nodes
Adder_mod4.mv	24	48	2
Balance.mv	337	1348	1
Iris.mv	100	400	1
Mm3.mv	111	555	1
Mm4.mv	598	2990	1
Mm5.mv	112	506	1
Ele-ctr-det.mv	1633	3542	1446
Bakery-proc.mv	813	1788	258
Coh-dir.mv	996	1917	653
Matmul.mv	128	480	4

Table 3.2 initial characteristics of the examples

From the experimental results, we see that for most examples, the times are comparable or better with Approach 2. The run times reduce mainly because the size of network is reduced by being simplified with priority don't cares. For some examples, better results are obtained by Approach 2, mainly because of more extensive use of the complements of kernels. At the end of Approach 2 we call *uncompress* to change back to i-set mode. This involves the intersection of function with lower priority and the complement of those with higher priorities. So if a kernel has been extracted for a function with higher priority, its literal is used in that function, possibly causing the complement of that literal to be introduced into the function with lower priority when *uncompress* is executed. This can cause large savings since in i-set mode, we usually cannot use the complement of a kernel if it contains more than 2 cubes, since division by a more than 2 cube kernel would be very expensive.

	Approach1			Approach 2		
	#cubes	#literals	Time(s)	#cubes	#literals	Time(s)
Adder_mod4.mv	24	60	≤ 0.1	26	56	≤ 0.1
Balance.mv	-	-	-	230	1100	3.8
Iris.mv	57	187	2.1	31	114	0.3
Mm3.mv	26	64	1.2	16	37	≤ 0.1
Mm4.mv	140	393	65.8	41	112	0.2
Mm5.mv	126	338	7,7	78	242	1.0
Ele-ctr-det.mv	193	439	3.0	180	407	2.5
Coh-dir.mv	254	566	4.4	240	546	4.4
Bakery-proc.mv	147	338	1.4	183	650	1.4
Matmul.mv	112	248	1.2	112	304	0.6

'-' means that no result in reasonable time period

Table 3.3 Experimental results by Approaches 1 and 2

There are some examples where Approach 2 gets worse results. The main reason is that we extract kernels in Post mode but finally evaluate the results in i-set mode. A good kernel to the function that has been simplified with priority don't cares can be bad in the original function.

Since the speed improvement obtained using Post mode is desirable, it leads to the idea of using Post mode operations first if the input network is too large and then use i-set mode operations. This approach can improve both speed and the quality of the results.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

We investigated the problem of extending MV optimization methods from i-set mode expressions to Post mode expressions, which is another format used for general MV logic. Also, some further development of the multi-valued synthesis tool MVSIS was done. The work can be summarized in two main parts:

- We successfully extended two semi-algebraic optimization methods from i-set mode to Post mode, namely the satisfiability matrix method for factorization and inexact division, and the maximum graph matching method for exact division. Also attempts were made to extend don't care-based minimization methods to Post mode. The commands used in MVSIS were examined one by one to show either how they can be extended or what is the difficulty in such extension. This forms the theoretical bases for implementing such extensions.
- We also worked on the development of MVSIS. New commands were implemented, including *qcheck*, *eliminate_part*, *pair_decode*, and *validate* for formal verification. The algorithms used in these commands were presented and their effectiveness was shown on some examples. Also, experiments on using Post mode operations to speed up MV minimization were performed and the results analyzed.

4.2 Future work

Much work still needs to be done:

- Implement the extension of semi-algebraic optimization methods in MVSIS. We have mentioned that it seems those methods on Post mode can generate multi-valued sub-expressions which cannot be realized in the i-set mode. Comparing the results in the two different modes would be interesting.
- Extend don't care-based methods to Post mode. We have shown how to compute SDC in Post mode, while the computation of CODC is unsolved. Since the simplification using CODC plays an important role in MV minimization, more effort in this should be done.
- Develop a new version of MVSIS for asynchronous optimization using type of logic called delay-insensitive logic. Some discussions on using MV-synthesis to design such logic have indicated how this can be done.
- Include technology dependent optimization of MV logic in MVSIS. Circuits using current based transistors have been discussed in literature. Such circuits can realize multi-valued logic, and this poses a requirement on technology dependent optimization. Some theoretical work has been done on technology mapping, which provides a good basis for further research.

Bibliography

[P21] Post E. L. 1921 Introduction to a general theory of elementary propositions *Amer.J.Math.* **43** 163-85.

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

[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.

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

[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.

[VKBSV97] T. Villa, T. Kam, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. *Synthesis of Finite State Machine: Logic Optimization*. Kluwer Academic Publishers, 1997.

[JB00] Yunjian Jiang and R. K. Brayton. Don't cares and Multi-valued Logic Minimization. In *Proceeding of the International Workshop on Logic Synthesis*, May 2000.

[MB00] M. Gao, R. K. Brayton. Semi-Algebraic Methods for Multi-Valued Logic, *IEEE IWLS 2000, International Workshop on Logic Synthesis 2000, Dana Point, CA, Workshop Notes* 73-80, May 31 - June 2, 2000.

[JTHE] Y. Jiang. Multi-valued Logic Network Minimization and it's Application. Master thesis, 2000.

[GJJLSR01] Minxi Gao, Jie-Hong Jiang, Yunjian Jiang, Yinghua Li, Subarna Sinha and Robert Brayton. MVSIS. *International Workshop on Logic Synthesis, Tahoe City*, June 2001.

[DJR01] Elena Dubrova, Yunjian Jiang and Robert Brayton, Minimization of Multiple-Valued Functions in Post Algebra, *International Workshop on Logic Synthesis, Tahoe City*, June 2001.