ALGEBRAIC METHODS FOR MULTI-VALUED LOGIC

by

Robert K. Brayton

Memorandum No. UCB/ERL M99/62

7 December 1999

ALGEBRAIC METHODS FOR MULTI-VALUED LOGIC

by

Robert K. Brayton

Memorandum No. UCB/ERL M99/62

7 December 1999

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Algebraic Methods for Multi-Valued Logic

Robert K. Brayton
Electrical Engineering and Computer Sciences Dept.
University of California, Berkeley

December 7, 1999

Abstract

We give several algebraic (more correctly semi-algebraic) methods for manipulating multi-valued logic functions. The methods treat binary and multi-valued variables uniformly. They include methods for finding common sub-expressions, semi-algebraic division, decomposing a multi-valued network using kernel extraction, factoring an expression, and simplifying a factored form using "redundant values". The algorithms have been implemented in a prototype system (in APL) and tested for quality (not speed) on a small set of made-up examples. The methods seem to work satisfactorily but more experimentation needs to be done.

1 Introduction

Multi-valued (MV) logic can be useful for the initial manipulation of a hardware description before it is encoded into binary and processed by standard logic synthesis programs. It is a natural way to describe logic at a higher level. Additionally, it may be useful as a front end to a software compiler, since software lends itself naturally to the evaluation of multi-valued variables in a single cycle. Although there has been a lot of work on multi-valued logic manipulation and optimization, one set of techniques that have not been developed is the algebraic methods which are at the foundation of efficient binary logic synthesis.

Algebraic methods are used after first casting the logic expression into a minimized sum-of-products. Then the result is manipulated as an algebraic expression, ignoring the Boolean identities $x\overline{x} = 0, xx = x, x + \overline{x} = 1$. The intuition is that if two functions have a common subexpression or divisor, then often this will be seen in their minimized sum-of-products expressions. This results in faster methods for manipulating the logic, such as factoring and finding common divisors. Although some optimality is lost, this can be recovered by using Boolean methods later.

In this paper, we develop, more fully, algebraic type methods for MV-logic. The basis for these ideas originate in the paper of Lavagno et. al. [3].

2 Notation

In general, a MV-logic function can have MV input variables and an MV output. A function with a single binary output is called an MV logic function, or simply an *MV-function.* A function with k output values can be represented by k MV-functions or as a single function with k output values. In this paper, we use the first method, although ultimately we hope to treat the output as a single MV variable.

An **MV-network** is a network of nodes; each node represents a multi-valued output function. There is one MV variable associated with the output of each node. An edge connects node i to node j if the function at j depends explicitly on the variable associated with node i, typically denoted, y_i. The network has a set of primary inputs and a set of nodes which are designated as the outputs of the network. An intermediate format for representing such a network is BLIF-MV used in the system VIS [1].

A **literal** of an MV-variable x is associated with a set of values for that variable. For example, suppose x can take on 5 values $\{0, 1, 2, 3, 4\}$. Then $x^{\{0,2\}}$ and $x^{\{1,2,4\}}$ are literals of x. The interpretation of $x^{\{0,2\}}$ is that it is a binary logic function which is 1 if x has either the value of 0 or 2, and 0 otherwise. Note that $x^{\{0,1,2,3,4\}} = 1$ since all five values appear in the literal. A **product term** or

cube is a product of literals which evaluates to 1 if and only if each of the literals evaluates to 1. Additionally, a cube can be thought of as simply a set of values. We use the notation $\tilde{c}$ to denote the cube consisting of all values **not** in the cube c. A **sum-of-products** is the OR of a set of product terms, which evaluates to 1 if any of the products evaluates to 1. In general, the set of values for variable x_i is denoted $P_i = \{0, 1, ..., |P_i| - 1\}$.

The **supercube** of a set of cubes (an expression) is the cube formed by taking the union of all the values in all the cubes.

The **cofactor** of a set of cubes d with respect to a cube c, denoted d_c, is the set of cubes obtained by eliminating cubes of d that do not intersect c and then adding to each remaining cube those values not in the cube c, i.e. the values in $\tilde{c}$. For example, if x and y each have 5 values, $d = x^{\{0,1,4\}} + y^{\{1,2,4\}} + x^{\{0\}}y^{\{0,3\}}$ and $c = x^{\{1,2\}}y^{\{1,3\}}$, then $d_c = x^{\{0,1,3,4\}} + y^{\{0,1,2,4\}}$.

We say that an expression is **cube-free** if for each variable there is no literal (except 1) that contains all other literals of that variable in the expression. For example, $a^{\{1,3\}}b^{\{1,2,3\}} + a^{\{0,1,3\}}b^{\{1,3\}}$ is not cube-free because the literal $a^{\{0,1,3\}}$ contains $a^{\{1,3\}}$ and $b^{\{1,2,3\}}$ contains $b^{\{1,3\}}$. The containing set of literals (in this case, $a^{\{0,1,3\}}b^{\{1,2,3\}}$) is called the common cube of the expression. We can make an expression cube-free in several ways.

1. cofactor the expression by the common cube (e.g. $a^{\{1,3\}}b^{\{1,2,3\}} + a^{\{0,1,3\}}b^{\{1,3\}} = a^{\{0,1,3\}}b^{\{1,2,3\}}(a^{\{1,2,3,4\}} + b^{\{0,1,3,4\}})$),

2. remove all literals appearing in the common cube (e.g. $a^{\{1,3\}}b^{\{1,2,3\}} + a^{\{0,1,3\}}b^{\{1,3\}} = a^{\{0,1,3\}}b^{\{1,2,3\}}(a^{\{1,3\}} + b^{\{1,3\}})$).

In the first case, values (not in the common cube) are added to the expression. In the second case extra values are added but only to make the common cube literals equal 1. In this paper we use the second method of making an expression cube free in order to keep the number of values in an expression minimal. In fact, the two cases represent upper and lower bounds for the cube-free results. The values not in the common cube are called **redundant values** and we can use them like don't cares to obtain many cube-free expressions. Thus, unlike the binary case, the cube-free expression is not unique. This makes the problem of finding common divisors among two or more expressions more difficult than in the binary case.

As an example, consider the following two expressions.

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

At first glance, f and g seem to have no common divisor. However, inside the parentheses of f, 2 and 4 are redundant values for x, and inside the parentheses of g, 1 and 3 are redundant values for x. If we choose to include 2 in the first, and remove 3 in the second, we get

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

and hence a common divisor of $y^{\{1,2\}} + x^{\{0,2\}}y^{\{0\}}$. In summary, the cube-free divisor is not unique, and common divisors must be identified by selectively choosing which redundant values to include.

3 Satisfiable Matrices

In [3], a set of literals of a variable arranged in a two dimensional array M is **satisfiable** if it satisfies the following "value condition" for each value v of the MV-variable.

Definition 1 *Let I be the set of rows in M in which v appears and J the set of columns in M in which v appears. Then v satisfies then* value condition *if it appears in each literal in M in all positions $\{M_{i,j} | i \in I, j \in J\}$. If all values of the multi-valued variable satisfy the value condition, the M is* satisfiable.

The idea in [3] for finding common divisors of a set of binary output functions with only one MV-variable and many binary variables, is to

1. find kernels and co-kernels by factoring out binary literals using the standard binary kerneling algorithm,

2. form the co-kernel/cube matrix with each co-kernel forming a row and each column associated with each unique cube appearing in any of the kernels,

3. label the columns with the binary literals in the kernel cube,

4. set the (i,j) entry of the matrix equal to the remaining literal of the MV-variable of the kernel cube,

5. find a large rectangular sub-array (I,J) that is satisfiable.

At this point, a product of two expressions can be formed out of the (I,J) sub-array and the row and column labels.

We extend this definition and procedure slightly where in general all variables can be and the rows and columns are not associated with binary co-kernel and kernel cubes. Consider any rectangular arrangement of a set of MV-cubes. It is **satisfiable** if for all values of all variables, each value satisfies the value condition.

Note that this definition applies equally to binary as well as MV-variables. In this paper, we give new methods for factoring MV expressions. Some of the methods are new even for purely binary logic functions.

Similar to the procedure in [3], one can derive a product of two expressions from a satisfiable matrix by the following:

1. For each row i, form the supercube $e_{r,i}$ of all cubes in that row.

2. OR these together to form the *row expression*, $e_r = \sum_i e_{r,i}$.

3. For each column j, form the supercube $e_{c,j}$ of all cubes in that column.

4. OR these together to form the *column expression*, $e_c = \sum_j e_{c,j}$.

Theorem 1 *Let M be a satisfiable matrix, and $z = \sum_{i,j} M_{ij}$. Then $z = (e_r)(e_c)$.*

Proof. We claim that

$$M_{ij} = e_{r,i} \cap e_{c,j}$$

Clearly $M_{ij} \subseteq e_{r,i} \cap e_{c,j}$. Now suppose that $M_{ij} \not\supseteq e_{r,i} \cap e_{c,j}$. Then there exists a variable with a value v such that $v \in e_{r,i} \cap e_{c,j}$ but $v \notin M_{ij}$. However, v must be in M_{ik} for some k, and also $v \in M_{mj}$ for some m. Therefore, by the value condition for a satisfiable matrix, $v \in M_{ij}$ (as well as M_{mk}), a contradiction. Hence, $M_{ij} \supseteq e_{r,i} \cap e_{c,j}$. □

We want to address the following problem.

Problem: *Given a set of cubes, find a subset that can be rearranged into a (largest) satisfiable rectangle.*

We give a method for finding a solution, based on preselecting the number of rows in the matrix. Then the array is found by a branch and bound technique. Entries are selected in the matrix in column order, i.e. a cube is selected for $M_{11}, M_{21}, ..., M_{12}, ...$ At each point the entries selected are guaranteed thus far to satisfy the value condition. The bounding process is that the selected cube c for the next entry should satisfy a lower bound cube l and two upper bound cubes, u_1 and u_2, i.e.

$$l \subseteq c \subseteq u_1 \cap u_2$$

Let (i,j) be the matrix position for the next entry to be selected.

Lower Bound Cube: The cube l is made up of the following set of values:

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

Note that for $i=1$ or $j=1$, this is 0.

Upper Bound Cube 1: The cube u_1 is made up of the following set of values:

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

Alternately, u_1 consists of all values **except** the following,

$$\tilde{u}_1 = \{v|(\forall(k<j), v \notin M_{i,k}) \text{ and } (\exists(n \neq i, m<j), v \in M_{n,m})\}$$

Note that for $j=1$, this is 1.

Upper Bound Cube 2: The cube u_2 is made up of the following set of values:

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

Alternately, u_2 consists of all values **except** the following,

$$\tilde{u}_2 = \{v|(\forall(n<i), v \notin M_{n,j}) \text{ and } (\exists(m<j), v \in M_{n,m})\}$$

Note that for $j=1$, this is 1.

Theorem 2 *A matrix M is satisfiable if and only if for each (i,j), $M_{i,j}$ satisfies $l \subseteq M_{i,j} \subseteq u_1 \cap u_2$.*

Proof. Let v be an arbitrary value, and x, a, b, b_1, c stand for the following propositions.

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

The value condition can be characterized by the following expressions,

1. $ac \Rightarrow x$
2. $xb \Rightarrow a$
3. $xb_1 \Rightarrow c$

The value condition can be seen as exactly the conditions that relate diagonal entries M_{ik} and M_{mj} to M_{ij}, or M_{nk} and M_{nj} to M_{ij}. Solving for x for each of the above expressions, we get

1. $ac \Rightarrow x$
2. $x \Rightarrow \overline{b} + a$
3. $x \Rightarrow \overline{b_1} + c$

These conditions translate into the lower bound condition l and the two upper bound conditions, u_1 and u_2 respectively. □

Other Efficiencies: Since the rows and columns can be permuted arbitrarily, we can assume that the least numbered cube, in the matrix to be found, is in the (1,1) position and that the first row and first column are ordered. Having preselected the number of rows, we search for the satisfiable matrix with the most columns. Whenever a new rectangular satisfiable matrix is found , it is recorded if it is the largest seen so far. The search can be bounded if we can reason that the remaining part of the search cannot produce a larger matrix. One way to achieve this is to use the upper and lower bounds. For example, suppose that the (2,3) entry is to be chosen next, and we have seen already a satisfiable matrix with four columns. Of the remaining cubes that have not been tried for the (2,3) entry, suppose only two satisfy the upper bound u_1. Then we back-track the search to the next choice for the (1,3) entry. The reason is that u_1 is independent of column j, so only the two remaining cubes can be used for row 2, leaving no choice for a fifth column. A similar bounding can be obtained by using u_2 and the observation that u_2 is independent of row i.

4 Semi-Algebraic Division

One of the applications of the above search process is given an expression (sum-of-products), d, which will serve as the divisor, and another expression f, find the dividend, e, i.e. a largest expression e such that $f = de + r$. In this equation, each side is a set of cubes and equality means that the two sets are equal. de produces a set of cubes of size $|d| \times |e|$, i.e. the cross product of the two sets. The product of two cubes is the intersection of the two sets of values for each variable. A cube may be one of the null cubes, i.e. a cube where at least one of the literals has no values in its set. If null cubes are retained, no cubes are lost in the multiplication process, and the product is like the algebraic product defined for binary valued variables. However, there is a difference. In the binary case, the algebraic product was defined only if the two expressions in question have no variables in common. Thus the product could not produce null cubes and the property $xx = x$ is not needed.

We relax the definition of algebraic product and apply it to the multi-valued case. *We do not require that the two expressions have disjoint sets of variables.* As an example, consider the product

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

When this is multiplied out, we get

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

In general, null cubes could be produced. We use non-algebraic properties in performing this product by using that the set of values obtained for a variable is the intersection of the two sets from each cube, e.g.

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

which is analogous to using $xx = x$ for the binary case.

In our division algorithms, we start with a given divisor d and search for a satisfiable matrix M formed from the cubes of f. The row expression e_r associated with M will be related to d.

In **exact division**, we require that the given divisor, $d = e_r$. The column expression, e_c, is the dividend, and the cubes of f not included in $M = (e_r)(e_c)$ form the remainder r. Each cube of d is associated with a row. Since each cube placed in a row must be contained in the associated cube of d, we can restrict the search for cubes in that row to only such cubes. In this way, the cubes of d serve limit the search and hence make it more efficient.

In **inexact division** we just require that $d \supseteq e_r$. For example, if $d = a + b$ and we "divide" this into

$$f = a\bar{b}x + a\bar{b}y + \bar{a}bx + \bar{a}by$$

we get $e_r = a\bar{b} + \bar{a}b$ and $e_c = x + y$. The initial divisor, d, just seeds the search; d is not necessarily a divisor of f. Another case is

$$f = a\bar{b}xz + a\bar{b}yz + \bar{a}bxz + \bar{a}byz$$

where the initial divisor is $a\bar{b} + \bar{a}b$. We get $e_r = z(a\bar{b} + \bar{a}b)$. In general, we can use any expression to start the process, even, for example, $d = 1 + 1 + 1$, in which case d has no information and we are just looking for a largest satisfiable matrix with 3 rows.

There are several applications where we want the row expression to be exactly the divisor. For example, suppose a common divisor is to be extracted from a set of functions. Then the candidate divisor d should be used to rewrite each function in terms of d, $f_i = de_i + r_i$. It would not do to have a result where each row expression is unique, i.e. $f_i = d_ie_i + r_i$, where each $d_i \subseteq d$.[1]

However, in an application like factoring, we use the divisor in only one function. As an example, consider the function,

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

Suppose we determine from looking at the cubes 1 and 3 that we want to divide by

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

If we require that the row expression equal this, then we get $c^{\{3\}}$ as the dividend. However, if we only use $a^{\{0,1,2\}} + b^{\{1,2,3\}}$ as a seed, we can get a larger satisfiable matrix and achieve the factorization

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

Again the row expression is contained in the original divisor, but is not equal to it.

Hence, in factorization, the divisor is used to focus and limit the search process and the row expression obtained from the satisfiable rectangle need not equal d. In this case we obtain $f = \hat{d}e + r$ which can be an acceptable start of a factoring process. Additionally, it is not necessary to get the best result at first; as a second step e can be used as the divisor leading possibly to a better factorization. This is the basis of *quick factor* (QF), used in SIS [4], where the first divisor is chosen to be any level-zero kernel. Such a factoring process has been implemented where the first seed divisor is chosen to be any two-cube divisor in analogy to the method of [5]. In our case, we look at pairs of cubes in decreasing order of size and choose the first pair which has a nontrivial common cube. These literals are extracted from the pair (by making it cube-free, according to method 2 in Section 2) and the result is used as the candidate divisor.

As examples of the resulting QF algorithm applied to several functions, we get the following results, where below we just show the final factorizations; the initial set of cubes given to the algorithm is the set obtained by multiplying out the expressions.

$$\begin{aligned}
S3 &= (a^{\{0,1\}} + c^{\{1,4\}} + a^{\{2\}}b^{\{1,2,3\}}c^{\{3,4\}}) \\
&\quad ((f^{\{1,2\}} + f^{\{0,2\}}g^{\{1\}})e^{\{1\}}g^{\{1,2\}} + \\
&\quad f^{\{2\}} + d^{\{0,2,4,5\}}g^{\{1,2,3\}} + \\
&\quad d^{\{0,1,2,4,5\}}f^{\{0,1\}}g^{\{3\}} + d^{\{2,3,4,5\}}e^{\{0\}}g^{\{2\}}) \\
GP &= (a^{\{0\}}b^{\{0\}} + c^{\{0\}}d^{\{0\}} + e^{\{0\}}f^{\{0\}}) \\
&\quad (a^{\{1\}}b^{\{1\}} + c^{\{1\}}d^{\{1\}} + e^{\{1\}}f^{\{1\}}) \\
GQ &= (b^{\{1\}} + c^{\{1\}} + a^{\{1\}}f^{\{1\}})(d^{\{1\}} + e^{\{1\}} + a^{\{1\}}g^{\{1\}})
\end{aligned}$$

Note that even though the first divisor used in the QF algorithm consists of only two cubes, by using the cube-

[1] Since $d_i \subseteq d$, there exists a function g_i such that $d_i = g_id$, hence $f_i = dg_ie_i + r_i$, so d is a Boolean divisor of f_i. However, the combination g_ie_i may be more complex than the original function f_i.

free column expression as the second divisor, we obtain a much stronger factorization. Note also that the second and third expressions have only binary inputs and the factorization is not algebraic since the second uses the Boolean identity $a\bar{a} = 0$ and the third uses $aa = a$. GP is an example where the initial sum-of-products for it includes appropriate null cubes.

$$\begin{aligned} GP &= a^{\{\}}b^{\{\}} + a^{\{0\}}b^{\{0\}}c^{\{1\}}d^{\{1\}} + a^{\{0\}}b^{\{0\}}e^{\{1\}}f^{\{1\}} + \\ &\quad a^{\{1\}}b^{\{1\}}c^{\{0\}}d^{\{0\}} + c^{\{\}}d^{\{\}} + c^{\{0\}}d^{\{0\}}e^{\{1\}}f^{\{1\}} + \\ &\quad a^{\{1\}}b^{\{1\}}e^{\{0\}}f^{\{0\}} + c^{\{1\}}d^{\{1\}}e^{\{0\}}f^{\{0\}} + e^{\{\}}f^{\{\}} \end{aligned}$$

Without the null cubes, we would obtain

$$\begin{aligned} GP &= (c^{\{1\}}d^{\{1\}} + e^{\{1\}}f^{\{1\}})a^{\{0\}}b^{\{0\}} + \\ &\quad (e^{\{1\}}f^{\{1\}} + a^{\{1\}}b^{\{1\}})c^{\{0\}}d^{\{0\}} + \\ &\quad e^{\{0\}}f^{\{0\}}(a^{\{1\}}b^{\{1\}} + c^{\{1\}}d^{\{1\}}) \end{aligned}$$

We will discuss next how appropriate null cubes can be inserted into an expression to yield a better factorization.

5 Supplementing with Null Cubes

A good factorization can be lost because multiplying it out results in null cubes which are usually thrown away. Then the factorization process which must reverse this has difficulty finding the original factorization. However, one can assume that all null cubes are implicitly part of any set of cubes. One procedure for using these is to insert an unspecified null cube into the matrix, when there are no regular cubes that will fit. As an example, suppose in the division process we arrive at a rectangle with 3 rows and in the third column, we have succeeded in finding a satisfiable arrangement of cubes, except for the (3,3) element. Then if none of the cubes are a candidate for this position, we can insert a null cube whose values are determined later. Consider the following example.

$$\begin{aligned} GP &= a^{\{0\}}b^{\{0\}}c^{\{1\}}d^{\{1\}} + a^{\{0\}}b^{\{0\}}e^{\{1\}}f^{\{1\}} + \\ &\quad a^{\{1\}}b^{\{1\}}c^{\{0\}}d^{\{0\}} + c^{\{0\}}d^{\{0\}}e^{\{1\}}f^{\{1\}} + \\ &\quad a^{\{1\}}b^{\{1\}}e^{\{0\}}f^{\{0\}} + c^{\{1\}}d^{\{1\}}e^{\{0\}}f^{\{0\}} \end{aligned}$$

Assume that the cubes are numbered 1-6 in the order shown above, with a null cube labelled 7. Suppose we divide by $a^{\{0\}}b^{\{0\}} + c^{\{0\}}d^{\{0\}} + e^{\{0\}}f^{\{0\}}$. Then in forming a satisfiable matrix we would like to get the following matrix,

1	2	7
6	7	5
7	4	3

or

$a^{\{0\}}b^{\{0\}}c^{\{1\}}d^{\{1\}}$	$a^{\{0\}}b^{\{0\}}e^{\{1\}}f^{\{1\}}$	ϕ_3
$e^{\{0\}}f^{\{0\}}c^{\{1\}}d^{\{1\}}$	ϕ_2	$e^{\{0\}}f^{\{0\}}a^{\{1\}}b^{\{1\}}$
ϕ_1	$c^{\{0\}}d^{\{0\}}e^{\{1\}}f^{\{1\}}$	$c^{\{0\}}d^{\{0\}}a^{\{1\}}b^{\{1\}}$

The strategy would be to try a null cube for the current position when all other options are exhausted. Note that a null cube is labelled 7 to force it to be the last cube tried for a given position. The matrix we obtain in the search process has its first row and column in order. We can also limit our choices to only one null cube per row and column to make the process more efficient.

The only question is when is a null cube appropriate for a given position? It is important to give the inserted null cube a set of values because its values must satisfy the upper and lower bounds for that position. Also its values are used later to derive upper and lower bounds for subsequent positions. We use the following strategy if $i \neq 1$ and $j \neq 1$. For these positions, we have already computed lower and upper bounds for the current position. If the lower bound is not a null cube, then there is no null cube appropriate for that position and we have to backtrack. Otherwise, we create a null cube with no values in the null variables of the lower bound cube and equal to the upper bound in the other variables. For $j = 1$, we create a null cube for the position and when the column is finished we recompute the null cube so it is compatible with that column. For $i = 1, j \neq 1$ the null cube can only appear if this is the last column. In this case, we try to create one more column but in reverse row order. Thus the null cube appears only when the column is done and we have enough information to create a correct one. For $i = 1, j = 1$ it is not possible to have a null cube in this position.

This strategy seems to work pretty well, except it is possible to create a matrix containing the created null cubes but which is not satisfiable. Hence when inserting null cubes we must check the satisfiability with a special subroutine before proceeding in the search.

Thus in the above example, we get $\phi_1 = c^{\{\}}d^{\{\}}$, $\phi_2 = e^{\{\}}f^{\{\}}$, and $\phi_3 = a^{\{\}}b^{\{\}}$. We still get the same row and column expressions as expected,

$$\begin{aligned} e_r &= a^{\{0\}}b^{\{0\}} + e^{\{0\}}f^{\{0\}} + c^{\{0\}}d^{\{0\}} \\ e_c &= c^{\{1\}}d^{\{1\}} + e^{\{1\}}f^{\{1\}} + a^{\{1\}}b^{\{1\}} \end{aligned}$$

Multiplying this out leads to same null cubes that were created.

The use of null cubes is not always beneficial as the following example shows. Without using null cubes our factoring algorithm gives,

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

and using null cubes, we get

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

In the first case there are 24 literals in the factored expression and in the second, 27 literals.

6 Using Redundant Values in Factored Forms

A factored form is a parse tree where the leaves are cubes, and internal nodes are either the AND operator or the OR operator. Redundant values are like observability don't cares, and although less powerful, they are simpler to compute and use. We describe a technique for deriving redundant values for subtrees of a factored form. Once redundant values are derived they can be used like don't cares where in the subexpression they can be included in each cube or excluded. We derive sufficient conditions for values to be redundant.

1. Consider AND(e_1, e_2). Let s_1 = supercube(e_1) and s_2 = supercube(e_2). Then $\tilde{s_1}$ are redundant values for e_2 and $\tilde{s_2}$ are redundant values for e_1.

2. Consider OR(e_1, e_2). Let t_1 = supercube$(\overline{e_1})$ and t_2 = supercube$(\overline{e_2})$. Then $\tilde{t_1}$ are redundant values for e_2 and $\tilde{t_2}$ are redundant values for e_1.

With these two propositions, we can construct an algorithm which operates on a factored form, computes redundant values down to the leaves, and simplifies the leaves. The algorithm operates in two passes. On the first pass, we compute recursively for each node, the supercube of the associated expression and the supercube of the complement. On the second pass, the redundant values are propagated down to the leaves in depth-first order. At a leaf, the redundant values are used to simplify the leaf. This may change the supercube and the supercube of the complement for that leaf. Hence these supercubes are updated before we proceed to the next branch. As we move up the tree, the supercubes and the supercubes of the complements are updated. This gives us an order-dependent but compatible way of using the redundant values.

An example of this in operation is the following,

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

which, using redundant values, reduces to

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

where the number of values for a, b, c, e, f, g = $3, 4, 5, 2, 3, 4$ respectively. Another example is

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

where a, b, c each have 4 values. Using the above operation, this reduces to 1.

Redundant values are easy to compute. The only comment on their computation is how to compute the supercube of the complement without complementing the expression. This is done by observing that the supercube of the complement of an expression is upper bounded by the following procedure:

1. Remove all cubes of the expression except single literal cubes.

2. Complement the resulting expression, which is the cube consisting of the product of the complement literals.

The result is only an upper bound, since an expression may not be minimized. For example, there may be a single literal implicant, but it is not apparent from the initial expression.

7 A Kerneling Process

We seek a kerneling process similar to that used for binary functions [2, 5]. We follow a process analogous to the two-cube divisor method of [5].

The **two-cube divisors** of an expression is the set

$$\tau(f) = \{\text{cubefree}(c_i, c_j) | c_k \in f\}$$

Note that there are at most $\frac{|f|(|f|-1)}{2}$ two-cube divisors of an expression. It was demonstrated in [5] that the use of such divisors in a decomposition process leads to little loss of optimality over the use of kernels but is much faster. A method was given for processing divisors as well as common cubes at the same time, also identifying when divisors were complements of each other, to obtain a more accurate figure of merit.

A divisor is given a figure of its merit by keeping track of the number of times it or its complement appears in all the expressions being considered. The divisor with the greatest merit is chosen, implemented as a separate function and substituted into all the expressions in which it appears. The substitution is performed using the algebraic division process described previously. Once a network has been decomposed by this process, functions can be selectively eliminated if their figure of merit in implementing the network is below a given theshhold.

As an example of this process consider the following derived network, where the numbers in the parentheses of the input variables give the number of values for the variable.

.in $a(2), b(2), c(2), d(2), e(2), f(2), g(2), h(2)$

.out GP, GO, GR, GQ, GS, gg

$$\begin{aligned}
GP &= y_3^{\{1\}} \\
GO &= y_3^{\{1\}} + a^{\{1\}}b^{\{1\}}c^{\{1\}}d^{\{1\}}e^{\{1\}}f^{\{1\}} + \\
&\quad a^{\{0\}}b^{\{0\}}c^{\{0\}}d^{\{0\}}e^{\{0\}}f^{\{0\}} \\
GR &= (c^{\{0\}} + d^{\{0\}})(e^{\{0\}} + f^{\{0\}})(a^{\{0\}} + b^{\{0\}}) + \\
&\quad (a^{\{1\}} + b^{\{1\}})(c^{\{1\}} + d^{\{1\}})(e^{\{1\}} + f^{\{1\}}) \\
GQ &= y_1^{\{1\}}y_2^{\{1\}} \\
GS &= a^{\{0\}}((e^{\{0\}} + c^{\{0\}})d^{\{1\}} + d^{\{0\}}e^{\{1\}} + \\
&\quad c^{\{0\}}f^{\{0\}}h^{\{1\}} + c^{\{0\}}e^{\{1\}}(h^{\{1\}} + f^{\{0\}}g^{\{1\}})) \\
gg &= y_1^{\{1\}}y_2^{\{1\}} \\
y_1 &= b^{\{1\}} + c^{\{1\}} + a^{\{1\}}f^{\{1\}} \\
y_2 &= a^{\{1\}}g^{\{1\}} + d^{\{1\}} + e^{\{1\}} \\
y_3 &= (a^{\{0\}}b^{\{0\}} + c^{\{0\}}d^{\{0\}})e^{\{1\}}f^{\{1\}} + \\
&\quad a^{\{1\}}b^{\{1\}}e^{\{0\}}f^{\{0\}} + a^{\{0\}}b^{\{0\}}c^{\{1\}}d^{\{1\}} + \\
&\quad c^{\{1\}}d^{\{1\}}e^{\{0\}}f^{\{0\}} + a^{\{1\}}b^{\{1\}}c^{\{0\}}d^{\{0\}}
\end{aligned}$$

The y_i subexpressions were not in the original description, and are the result of the kerneling process followed by selective elimination of expressions. Note that even though only two-cube divisors were initially extracted, the elimination process resulted in larger subexpressions.

The above network was a binary one. The next example is a MV-network.

.in $a(3), b(4), c(5), d(6), e(2), f(3), g(4)$

.out $S3, S6$

$$\begin{aligned}
S3 &= y_1^{\{1\}}(g^{\{12\}}y_4^{\{1\}} + y_2^{\{1\}} + d^{\{01245\}}g^{\{3\}}) \\
S6 &= (y_2^{\{1\}} + d^{\{01245\}}f^{\{01\}}g^{\{3\}})a^{\{01\}} + g^{\{12\}} \\
&\quad ((d^{\{0245\}} + f^{\{2\}})y_0^{\{1\}} + y_1^{\{1\}}y_4^{\{1\}}) \\
y_0 &= b^{\{123\}}c^{\{3\}} + c^{\{14\}} \\
y_1 &= a^{\{01\}} + y_0^{\{1\}} \\
y_2 &= d^{\{0245\}}g^{\{123\}} + f^{\{2\}} \\
y_4 &= (f^{\{1\}} + g^{\{1\}})e^{\{1\}} + d^{\{2345\}}e^{\{0\}}g^{\{2\}}
\end{aligned}$$

Again the y_i are sub-expressions created by the kerneling and elimination process. The process works essentially the same. We extract common subexpressions by finding cube-free pairs of cubes and choosing the ones that appear most often. These are extracted by performing algebraic division as described above. Binary and MV variables are treated uniformly. However, for the MV variables we have the additional problem that the cube-free divisor is not

unique and to identify common subexpressions optimally, we would need a method which uses the redundant values optimally. At this point we do not know how to do this, so we have used the minimal cube-free expressions (the one with the least values).

8 Conclusions

We have given several algebraic (more correctly semi-algebraic) methods for multi-valued logic functions. The methods treat binary and multi-valued variables uniformly. They include methods for

- finding common sub-expressions,
- semi-algebraic division,
- decomposing a multi-valued network,
- factoring an expression, and
- simplifying a factored form using the concept of redundant values.

The algorithms have been implemented in a prototype system (in APL) and tested for quality (not speed) on a small set of made-up examples. In particular, our system has only rudimentary methods for simplifying an expression and lacks a powerful multi-valued optimizer like ESPRESSO-MV. Nevertheless, the methods seem to work satisfactorily but more experimentation needs to be done.

The methods need to be improved along the following directions.

1. We do not know how to treat common divisors that are equivalent modulo redundant values. Although one can imagine such approaches, it remains to develop one and make it efficient.

2. All common subexpressions extracted by the methods of this paper are binary. Indeed all variables, except possibly variables in the initial description are binary. All examples that we have done so far had only binary output functions. It would be more appealing to have a method for finding common divisors which are themselves multi-valued output functions, so that multi-valued output functions can be treated directly without breaking them up into binary functions, one for each value. The hope would be to treat a multi-valued output function in a manner analogous to how multiple output functions are treated in PLA minimization.

3. Adding null cubes does not uniformly improve the factorization. We need a better filter for when to add null cubes.

4. The measure of progress currently is the number of literals in the factored form. However, it is not clear whether the literals $x^{\{0\}}$ and $x^{\{0,1,3,5\}}$ should be treated equally. Additionally, when a literal is factored out to make an expression cube-free, the number of literals is not always decreased, e.g.

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

   Even the number of values and the number and type of operations in this example do not change. Which expression is simpler to evaluate?

Finally, the use of initial multi-valued optimization followed by binary encoding and subsequent optimization needs to be evaluated for hardware implementation. We intend to experiment in two domains, hardware implementation, and software compilation where possibly binary encoding is not required. As a first step we will develop an efficient MV package using the algebraic methods developed in this paper along with other MV methods in the literature.

Acknowledgements

This work was supported by the SRC under contract 683.004.

References

[1] R. K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. P. Kurshan, S. Malik, A. L. Sangiovanni-Vincentelli, E. M. Sentovich, T. Shiple, K. J. Singh, and H.-Y. Wang. BLIF-MV: An Interchange Format for Design Verification and Synthesis. Technical Report UCB/ERL M91/97, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, November 1991.

[2] R. K. Brayton and C. McMullen. The Decomposition and Factorization of Boolean Expressions. In *Proc. of the Intl. Symposium on Circuits and Systems*, pages 49–54, May 1982.

[3] L Lavagno, S Malik, R 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.

[4] 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, Univ. of California, Berkeley, CA 94720, May 1992.

[5] J. Vasudevamurthy and J. Rajski. A Method for Concurrent Decomposition and Factorization of Boolean Expressions. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 510–513, November 1990.