MIS-MV: OPTIMIZATION OF MULTI-LEVEL LOGIC WITH MULTIPLE-VALUED INPUTS

by

Luciano Lavagno, Sharad Malik, Robert K. Brayton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M90/68

13 August 1990

COVER PAGE

MIS-MV: OPTIMIZATION OF MULTI-LEVEL LOGIC WITH MULTIPLE-VALUED INPUTS

by

Luciano Lavagno, Sharad Malik, Robert K. Brayton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M90/68

13 August 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

MIS-MV: OPTIMIZATION OF MULTI-LEVEL LOGIC WITH MULTIPLE-VALUED INPUTS

by

Luciano Lavagno, Sharad Malik, Robert K. Brayton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M90/68

13 August 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

MIS-MV: Optimization of Multi-level Logic with Multiple-valued Inputs.

Luciano Lavagno Sharad Malik Robert K. Brayton Alberto Sangiovanni-Vincentelli

Dept. of EECS
University of California, Berkeley

Abstract

We present techniques for the optimization of multi-level logic with multiple-valued input variables. The motivation for this is to tackle the input encoding problem in logic synthesis, where binary codes need to be found for the different values that a symbolic input variable can take. Multi-level multiple-valued optimization is used to generate constraints that are then used to determine the codes. The state assignment problem in sequential logic synthesis can be approximated as an input encoding problem by ignoring the next state field, which is reasonable when the output logic dominates the next state logic.

Common factor extraction is an important step in multi-level optimization and thus far it was not clear how this could be done with multiple-valued variables. We present a novel technique for extracting common factors with multiple-valued variables. We then show how the other multi-level optimization techniques are easily extended with multiple-valued variables. These ideas have been implemented as algorithms in the program MIS-MV. We present the practical issues involved in the implementation of these ideas, as well as results of using MIS-MV for input encoding on some benchmark examples.

1 Introduction

Real life variables tend to be multiple-valued (MV) while digital circuit signals are restricted to binary values. Thus, any digital circuit that involves these variables as part of its function must use some binary encoding for the various values of the MV variable. An example of this is the MV "instruction" of a machine architecture which takes values over the different instructions. However, in the controller of the machine each instruction value is represented by a unique binary "opcode". The size of the circuits in general is a function of the encoding chosen for the different values of the MV variable. This gives rise to the encoding problem in the automatic synthesis of logic circuits, wherein an encoding is to be determined for a MV-variable that results in "simple" logic. The version of this problem in which the MV variable is an input of the logic description is called the input encoding problem. We will restrict ourselves to this problem for this paper. Orthogonal to the type of encoding problem is the form of the target logic, i.e. two-level or multi-level. For the case of two-level logic, satisfactory solutions to the input encoding problem were obtained by first using a multiple-valued two level logic minimization step (e.g. [13]) and then using the result of this to generate constraints that the encoding had to satisfy (e.g. [11, 15]). However, for multi-level logic as the target implementation the approaches currently used tend to be "predictive" in as much as they determine encodings for which a multi-level logic optimizer such as MIS [4] is likely to find common divisors (e.g. [5, 8]). In addition, they only concentrate on finding common divisors that are cubes (or single product terms) and do not consider larger common divisors. Our approach to the input encoding problem is directed towards overcoming the shortcomings of existing multi-level encoding programs while following the paradigm that was so successful in the case of two-level logic. We have developed techniques for multi-level optimization of logic with MV input variables. This permits us to view the optimizations largely independent of the encoding and then use the encoding in the final step. Also, using this approach we were able to consider common divisors that were not restricted to single cubes.

Multi-level logic optimization involves the use of several different techniques. One very important technique involves finding common sub-expressions (factors, or divisors) among a set of logic expressions. The common sub-expressions can then be implemented only once resulting in substantial savings in logic. Techniques for determining common sub-expressions (kernel intersections) efficiently with binary variables were first presented in [3]. However, it was not clear how to extract common sub-expressions for expressions with MV inputs while guaranteeing that this extraction and a subsequent encoding would be sufficient to obtain all possible kernel intersections for all possible encodings. A major contribution of this work is to present such a technique. In fact, this technique can potentially result in common factors that are Boolean and hence even stronger than the algebraic kernel intersections. (This work is a development of the work we presented earlier in [9], so some similarity may be found in these two presentations.) The discovery of MV factorization techniques completed the missing link in multi-level MV minimization since the other multi-level optimization techniques were extended easily to MV inputs. These techniques have been been implemented in the program MIS-MV, which is the multiple-valued extension to MIS, the successful multi-level logic optimizer.

The organization of this paper is as follows. In Section 2 we discuss the representation of circuits and functions with multiple-valued inputs. Section 3 describes the MV factorization technique. Next, in Section 4 we describe the extensions of the other multi-level optimization techniques to MV variables. Section 5 describes the practical issues in implementing MIS-MV as well as the final encoding step. In Section 6 we present some results using MIS-MV as part of the encoding procedure. Section 7 presents the concluding remarks regarding this work.

2 Circuit and Function Representation

Since multiple symbolic variables can be mapped into a single symbolic variable[1], we restrict ourselves to a single symbolic variable throughout the paper. The rest of the variables are binary valued.

Let the symbolic variable v take values from $V = \{v_0, v_1, \ldots, v_{n-1}\}$. v may be represented by a multiple-valued variable, X, restricted to $P = \{0, 1, \ldots, n-1\}$, where each symbolic value of v maps onto a unique integer in P [2]. Let $B = \{0, 1\}$. A binary valued function f, of a single MV variable X and $m-1$ binary-valued variables, is a mapping: f : $P \times B^{m-1} \rightarrow B$. Each element in the domain of the function is called a **minterm** of the function. Let $S \subseteq P$. Then X^S represents the Boolean function:

$$X^S = \begin{cases} 1 & \text{if } X \in S \\ 0 & \text{otherwise} \end{cases}$$

X^S is called a **literal** of variable X. If $|S| = 1$ then this literal is also a minterm of X. For example, $X^{\{0\}}$ and $X^{\{0,1\}}$ are literals and $X^{\{0\}}$ a minterm of X. If $S = \phi$, then the value of the literal is always 0. If $S = P$ then the value of the literal is always 1. For these two cases, the value of the literal may be used to denote the literal. We note the following:

(1) $X^{S_1} \subseteq X^{S_2}$ if and only if $S_1 \subseteq S_2$. (2) $X^{S_1} \cup X^{S_2} = X^{S_1 \cup S_2}$. (3) $X^{S_1} \cap X^{S_2} = X^{S_1 \cap S_2}$.

The literal of a binary-valued variable y is defined as either the variable or its Boolean complement. A **product term** is a Boolean product (AND) of literals. If a product term evaluates to 1 for a given minterm, it is said to contain the minterm. A **cube** is a product term in which all the literals are of binary valued variables. Note this distinction between a product term and a cube; the latter does not involve any MV variables. A **sum-of-products** (SOP) is a Boolean sum (OR) of product terms. For example: $X^{\{0,1\}}y_1y_2$ is a product term, y_1y_2 a cube and $X^{\{0,1\}}y_1y_2 + X^{\{3\}}y_2y_3$ is a SOP. A function f may be represented by a SOP expression f. In addition f may be represented as a factored form. A factored form is defined recursively as follows.

Definition 2.1 *An SOP expression is a factored form. A sum of two factored forms is a factored form. A product of two factored forms is a factored form.*

$X^{\{0,1,3\}}y_2(X^{\{0,1\}}y_1 + X^{\{3\}}y_3)$ is a factored form for the SOP expression given above.

A logic circuit with a multiple-valued input is represented as a MV-network. A **MV network** η, is a directed acyclic graph (DAG) such that for each node n_i in η there is associated a binary valued, MV input function f_i, expressed in SOP form, and a Boolean variable y_i. There is an edge from n_i to n_j in η if f_j explicitly depends on y_i. Further, some of the variables in η may be classified as **primary inputs** or **primary outputs**. These are the inputs and outputs (respectively) of the MV-network. The MV-network is an extension of the well known Boolean network [4] to permit MV input variables. Since each node in the network has a binary valued output, the non-binary(MV) inputs to any node must be primary inputs to the network.

[1]For example, if there are two symbolic variables $v1$ and $v2$ taking values from sets $V1$ and $V2$ respectively, then these may be replaced by a single symbolic variable v taking values from $V1 \times V2$. This is in fact better than considering $v1$ and $v2$ separately since the encoding for v takes into account the interactions between $v1$ and $v2$.

[2]The notation presented in this section is the same as that used in two-level multiple-valued minimization [13].

3 Multiple-Valued Factorization

In this section we present the multiple-valued factorization technique along with its interesting properties with respect to the final encoded circuit. In this direction, we first review the process of common sub-expression extraction when there are no MV variables.

3.1 Kernels and Kernel Intersections

Common sub-expressions consisting of multiple cubes can be extracted from Boolean expressions using the algebraic techniques described in [3]. We review some definitions presented there.

Definition 3.1 *A* **kernel**, *k, of an expression g is a cube-free*[3] *quotient of g and a cube c. A* **co-kernel** *associated with a kernel is the cube divisor used in obtaining that kernel.*

As an example consider the expression $g = a\epsilon + b\epsilon + z$ and the cube ϵ. The quotient of g and this cube; g/ϵ is $a + b$. No other cube is a factor of $a + b$, hence it is a kernel of g. The cube ϵ is the co-kernel corresponding to this kernel.

An important result concerning kernels is that two expressions may have common sub-expressions of more that one cube if and only if there is a kernel intersection of more than one cube for these expressions [3]. Thus, we can detect all multiple-cube common sub-expressions by finding all multiple-cube kernel intersections. In [2] algorithms for detecting kernel intersections are described by defining them in terms of the rectangular covering problem. Because of the ease in understanding the concepts involved we use the rectangular covering approach for developing our ideas in the rest of the paper. However, the ideas hold with any technique for kernel extraction.

As an example of the rectangular covering formulation, consider the expressions g_1 and g_2:

$$g_1 = \underset{1}{ak} + \underset{2}{bk} + \underset{3}{c} \quad ; \quad g_2 = \underset{4}{aj} + \underset{5}{bj} + \underset{6}{d}$$

The integer below a cube is a unique identifier for it. The kernels of g_1 and g_2 and the rectangular covering formulation for this example are shown below.

$expression$	$co-kernel$	$kernel$
g_1	1	$ak + bk + c$
g_1	k	$a + b$
g_2	1	$aj + bj + d$
g_2	j	$a + b$

	a	b	ak	bk	aj	bj	c	d
1	0	0	1	2	0	0	3	0
k	1	2	0	0	0	0	0	0
1	0	0	0	0	4	5	0	6
j	4	5	0	0	0	0	0	0

The table on the right is a **co-kernel cube matrix** for the set of expressions. A row in the matrix corresponds to a kernel, whose co-kernel is the label for that row. Each column corresponds to a cube which is the label for that column. A non-zero entry in the matrix specifies the integer identifier of the cube (in the original expressions) represented by the entry.

A rectangle $\mathcal{R}$ is defined as a set of rows $S_r = \{r_0, r_1, \ldots, r_{m-1}\}$ and a set of columns $S_c = \{c_0, c_1, \ldots, c_{n-1}\}$ such that for each $r_i \in S_r$ and each $c_j \in S_c$, the (r_i, c_j) entry of the co-kernel cube matrix is non-zero. $\mathcal{R}$ covers each such entry. $\mathcal{R}$ is denoted as: $\{R(r_0, r_1, \ldots, r_{m-1}), C(c_0, c_1, \ldots, c_{n-1})\}$.

A rectangular covering of the matrix is defined as a set of rectangles that cover the non-zero integers in the matrix at least once (and do not cover a 0 entry). Once an integer is covered, all its other occurrences are replaced by don't cares (they may or may not be covered by other rectangles). Each rectangle that has more than one row indicates a kernel intersection. A covering for the above co-kernel cube matrix is: $\{R(2,4), C(1,2)\}$, $\{R(1), C(7)\}$, $\{R(3), C(8)\}$.

The kernel intersection $a + b$ between the two expressions is indicated by the first rectangle in the cover. The resulting implementation suggested by the covering is:

$$g_1 = kg_3 + c \quad ; \quad g_2 = jg_3 + d \quad ; \quad g_3 = a + b$$

We will use the total number of literals in the factored form of all the Boolean expressions as the metric for circuit size [4]. The above description has two fewer literals than the original description.

[3]No cube is an algebraic factor of k.

3.2 Kernels and Multiple-Valued Variables

Now consider the case where one of the input variables may be MV. The following example has a single MV variable X with six values and six binary valued variables.

$$f_1 = \underset{1}{X^{\{0,1\}}ak} + \underset{2}{X^{\{2\}}bk} + \underset{3}{c} \quad ; \quad f_2 = \underset{4}{X^{\{3,4\}}aj} + \underset{5}{X^{\{5\}}bj} + \underset{6}{d}$$

The integers below each product-term are unique identifiers for that product-term.

The definitions and matrix representations given in Section 3.1 are for binary valued expressions of binary valued variables. We now extend these to binary valued expressions with MV variables. As in Section 2 we consider only one of the variables to be MV, and the others are binary valued.

We modify the definitions for the kernel and co-kernel for a binary valued expression f of MV variables as follows:

Definition 3.2 *A* **kernel** *of an expression f is a product-term free quotient of f and a product term. The* **co-kernel** *is the product-term divisor used in obtaining the kernel.*

The **co-kernel cube matrix** is defined as follows:

Definition 3.3 *Each row represents a kernel (labeled by its co-kernel product-term) and each column a cube (labeled by this cube). Each non-zero entry in the matrix now has two parts. The first part is the integer identifier of the product-term in the expression* (**product-term part**)*. The second part is the MV literal (*MV **part**) *which when ANDed with the cube corresponding to its column and the co-kernel corresponding to its row forms this product term.*

The kernels of f_1 and f_2 and the corresponding co-kernel cube matrix is given below:

exp.	*co-kernel*	*kernel*
f_1	1	$akX^{\{0,1\}} + bkX^{\{2\}} + c$
f_1	k	$aX^{\{0,1\}} + bX^{\{2\}}$
f_2	1	$ajX^{\{3,4\}} + bjX^{\{5\}} + d$
f_2	j	$aX^{\{3,4\}} + bX^{\{5\}}$

	a	b	ak	bk	aj	bj	c	d
1	0	0	1	2	0	0	3	0
	0	0	$X^{\{0,1\}}$	$X^{\{2\}}$	0	0	1	0
k	1	2	0	0	0	0	0	0
	$X^{\{0,1\}}$	$X^{\{2\}}$	0	0	0	0	0	0
1	0	0	0	0	4	5	0	6
	0	0	0	0	$X^{\{3,4\}}$	$X^{\{5\}}$	0	1
j	4	5	0	0	0	0	0	0
	$X^{\{3,4\}}$	$X^{\{5\}}$	0	0	0	0	0	0

In the co-kernel cube matrix the product-term part is given above the MV part for each entry. The adjectives MV and binary will be used with co-kernel cube matrices and rectangles in order to distinguish between the multiple-valued and the binary case.

A rectangle is defined as in the case for all binary variables with one modification. Now the rectangle is permitted to have zero entries also. Associated with each rectangle $\mathcal{R}$ is a **constraint matrix** $M^{\mathcal{R}}$ whose entries are the MV parts of the entries of $\mathcal{R}$. For example, for the MV co-kernel cube matrix given above, M_1, is the constraint matrix for the rectangle {R(2,4), C(1,2)}.

$$M_1 = \begin{bmatrix} X^{\{0,1\}} & X^{\{2\}} \\ X^{\{3,4\}} & X^{\{5\}} \end{bmatrix}$$

A constraint matrix is said to be **satisfiable** if:

$$X^{\alpha} \subseteq \bigcup_j M(i,j) \bigwedge X^{\alpha} \subseteq \bigcup_i M(i,j) \Rightarrow X^{\alpha} \subseteq M(i,j)$$

i.e. if a particular value of X occurs somewhere in row i and also somewhere in column j, then it must occur in $M(i,j)$. M_1 given above can be shown to be satisfiable. If a constraint matrix is satisfiable then it can be used to determine a common factor between the expressions corresponding to its rows as follows. The union of the row entries for row i, ($\bigcup_j M(i,j)$) is ANDed with the co-kernel product-term corresponding to row i. Similarly the union of the column entries for column j (

$\bigcup_i M(i,j)$) is ANDed with the kernel cube corresponding to that column. Thus, this results in the following factorization of the expressions f_1 and f_2.

$$f_1 = X^{\{0,1,2\}}k \quad (X^{\{0,1,3,4\}}a + X^{\{2,5\}}b) + c \quad ; \quad f_2 = X^{\{3,4,5\}}j \quad (X^{\{0,1,3,4\}}a + X^{\{2,5\}}b) + d$$

Note that there is now a common factor between the two expressions which was not evident to start with. This common factor may be now implemented as a separate node in the MV-network and its output used to compute f_1 and f_2 as follows:

$$f_1 = X^{\{0,1,2\}}ky_3 + c \quad ; \quad f2 = X^{\{3,4,5\}}jy_3 + d \quad ; \quad f_3 = X^{\{0,1,3,4\}}a + X^{\{2,5\}}b$$

Not all matrices are satisfiable. An example of this is:

$$M_2 = \begin{bmatrix} X^{\{1\}} & X^{\{2\}} & X^{\{5\}} \\ X^{\{4\}} & X^{\{5\}} & X^{\{1\}} \end{bmatrix}$$

since $X^{\{5\}}$ occurs in row 1 as well in column 2 but is not present in $M(1,2)$.

A non-satisfiable matrix cannot be used to generate a common factor as shown above. This is because the addition to the row co-kernel product term ($X^{\{1,2,5\}}$ for row 1 in this example), and that to the column kernel cube ($X^{\{2,5\}}$ for column 2 for M_2) for the "offending" rows and column when intersected yield extra terms that are not present in $M(i,j)$. (The intersection is $X^{\{2,5\}}$ which contains $X^{\{5\}}$ which is not present in $M(1,2)$.)

Satisfiable constraint matrices are not the only source of common factors. In fact the condition can be relaxed as we see below.

Definition 3.4 *M_r is a* **reduced constraint matrix** *of M if $\forall i,j \quad M_r(i,j) \subseteq M(i,j)$*

Note that this definition includes the original constraint matrix. An example of a reduced constraint matrix for M_1 is:

$$M_3 = \begin{bmatrix} X^{\{1\}} & X^{\{2\}} \\ X^{\{3,4\}} & X^{\{5\}} \end{bmatrix}$$

A reduced constraint matrix that is satisfiable can be used to generate a common factor by covering the remaining entries separately. For example with the original expressions and M_3 we can obtain the following factorization.

$$f_1 = X^{\{1,2\}}k(X^{\{1,3,4\}}a + X^{\{2,5\}}b) + X^{\{0\}}ak + c \quad ; \quad f2 = X^{\{3,4,5\}}j(X^{\{1,3,4\}}a + X^{\{2,5\}}b) + d$$

Note that $X^{\{0\}}ak$ must be covered separately.

Let us now see what the MV-factorization process gets us in terms of the final encoded circuit. We are interested in obtaining large common factors in the encoded implementation. The work we reported in [9] was carried out with the underlying assumption that the common factors in the final encoded implementation depended on the encoding chosen. This assumption was not questioned and we presented encoding techniques that would result in large common factors in the encoded implementation. However, continuing the work in this area led to some rather surprising results. The following theorem showed that our initial assumption was not valid.

Theorem 3.1 *Let $\mathcal{E}$ be an encoded implementation and let f be a factor in this network. f can be obtained by first finding a factor in the MV-network and then using the same encoding.*

Proof. Omitted for brevity. ∎

This was a surprising result, since if factors do not depend on the initial encoding then we need not worry about the encoding since we can always go ahead and select an encoding and find the common factors later. However, the catch here is that our techniques for finding common factors in Boolean networks are algebraic and thus not strong enough to discover Boolean factors. Thus, even though a common factor may exist in the Boolean network it would remain undetected. Using conventional algebraic factorization after selecting an encoding exposes only a fraction of the common factors present. What is desirable is that we should at least be able to obtain all see all the factors that can be obtained using algebraic techniques and some encoding. In fact, the MV-factorization process, using reduced constraint matrices, does exactly that as shown by the following theorem.

Theorem 3.2 *Let $\mathcal{E}$ be an encoded implementation and let k be a kernel intersection in this network. k can be obtained by first finding a common factor in the MV-network and then using the same encoding.*

Proof. Omitted for brevity. ■

This result is significant because it gives us the satisfaction that we are not missing out on any possible algebraic factors in the encoded implementation that are dependent on the encoding. In fact, the MV factorization process described is even stronger in as much as it can potentially discover Boolean factors in the encoded implementation that could not have been found using algebraic techniques. Thus by using a procedure that is an extension of the known algebraic factorization process, we have been able to see all possible kernel intersections in all possible encoded implementations as well as additional Boolean factors.

4 Other Boolean Network Concepts

In the previous section we demonstrated how we could extract common factors with MV-variables. We now show how the other common multi-level optimization techniques used with Boolean networks, *viz.* node simplification can be used with almost no modification in the context of the MV-network. Node simplification involves using a two-level logic minimizer for each node function. (Recall that each node function is stored in SOP form.) Since we know how to do two-level minimization with MV variables [13], nothing new needs to be developed here.

In [1] it was shown how implicit don't cares in the Boolean network could be used in the node simplification process. We now examine these in the context of the MV-network. The satisfiability don't care set (SDC) for a Boolean network is defined as the set of signal values in the network that are inconsistent with the network. Mathematically this is expressed as $SDC = \bigcup_i SDC_i$ and $SDC_i = f_i\overline{y_i} + y_i\overline{f_i}$ i.e. the SDC at node i is the set of values for which the function computed with the node is not consistent with the output value of the node. Note that each node function in the MV-network is binary valued, therefore its complement is defined. Thus, the equation of SDC_i is still valid in the MV-network, only the domain now is $P \times B^{\{m-1\}}$.

For a Boolean network, the observability don't care (ODC) at a node is the set of primary input values for which the output of this node is inconsequential (or cannot be observed) at the primary outputs. For node i in a Boolean network $ODC_i = \bigcap_j (Fj_{y_i} \overline{\oplus} Fj_{\overline{y_i}})$. Here Fj is the function at primary output j and Fj_{y_i} indicates the cofactor operation of this node function with signal y_i. Again note that in the MV-network each node function is binary valued, hence cofactor with respect to a signal and its complement are defined and therefore the equation for ODC_i is still valid.

From the above we see that node simplification with implicit don't cares is done in the MV-network in exactly the same way as in the Boolean network. It can be similarly shown that other Boolean network logic simplification techniques such as simplification using permissible functions [12] have direct extensions to the MV-network.

Combinational logic verification (also referred to as Boolean comparison) is an important part of any combinational synthesis system. Almost all verification techniques use some form of Shannon's decomposition ([7]) ($f = xf_x + \overline{x}f_{\overline{x}}$). This decomposition is for binary valued x. However, it is easily extended to multiple valued X [13] as follows: $f = X^{\{0\}}f_{X^{\{0\}}} + \dots X^{\{n-1\}}f_{X^{\{n-1\}}}$, where $f_{X^{\{i\}}}$ is the value of f when X is equal to i. This extension enables us to use all the Boolean network comparison techniques for MV-networks (see also [14]).

5 Implementation

5.1 General approach

The full functionality of *misII* was extended to the case of multiple-valued inputs. Namely we are able to perform the following area-optimization operations on a multi-valued boolean network:

simplify using either the full satisfiability don't care set or a suitable subset.

algebraic common factor extraction using kernels and product-terms.

local collapsing of low value nodes.

decomposition of complex nodes.

The following subsections detail some of the problems that we encountered during the implementation of *mis-MV*, outlining the solutions, their strength and weakness. We will deal in particular with:

The cost function : how to verify if the operation being applied effectively decreases the implementation cost of the network.

Don't cares : some aspect of don't care computation that are specifically related with the multiple-valued variable are described.

Constraint matrices : how to reduce a constraint matrix that initially does not satisfy the intersection condition.

Encoding : how to derive an optimal encoded implementation from a multiple-valued boolean network.

5.2 Cost function estimation

Almost all algorithms used by *misII* are greedy. At each decision step (which kernel intersection to extract, which node to simplify first, ...) they make the choice that gives the best (expected) cost function reduction. So having a "good" cost function is extremely important. In the binary-valued case, the number of literals in the best available *factored form* of a node function is used whenever possible, since it has an excellent correlation with the area of the final implementation.

In the multiple-valued case we have two problems to solve:

1. Find a good factored form for the node function. This uses the kernel extraction algorithms described in Section 3 and a straightforward extension of the cube (now product-term) extraction and algebraic division algorithms from the boolean case.

2. Estimate the cost of the MV-literals appearing in the leaves of the factoring tree.

The latter problem proved to be the most difficult one, and it is still subject of active research. The cost of a MV-literal, such as $X^{\{1,3,4,5,6\}}$, depends on the encoding, that will be defined for all the MV-literals at the same time. So we cannot make a purely local estimate.

A first approximate solution assumed that each MV-literal could be implemented as a single cube[4] using minimum encoding length. This was overly optimistic, and it led to poor results.

A somewhat better approach is:

1. Perform an encoding step on the multiple-valued network. Do not replace MV-literals with their encoded implementation, just keep the codes aside.

2. Use the codes to estimate the cost of each MV-literal, creating a minimal sum of products expression for the function denoted by the encoded MV-literal.

E.g. the estimated cost of $X^{\{1,3,4\}[2]}$ with the codes:
$e(X^{\{1\}}) = \overline{c_1}\overline{c_2}\overline{c_3}$ $e(X^{\{2\}}) = \overline{c_1}\overline{c_2}c_3$ $e(X^{\{3\}}) = \overline{c_1}c_2\overline{c_3}$ $e(X^{\{4\}}) = \overline{c_1}c_2c_3$ $e(X^{\{5\}}) = c_1\overline{c_2}\overline{c_3}$ $e(X^{\{6\}}) = c_1\overline{c_2}c_3$
would be 1, since the minimum sum of products expression for $\overline{c_1}\overline{c_2}\overline{c_3} + \overline{c_1}c_2\overline{c_3} + \overline{c_1}c_2c_3$ with the explicit don't care $\overline{c_1}\overline{c_2}c_3$ and the implicit don't cares (unused codes) $c_1c_2\overline{c_3} + c_1c_2c_3$ is just c_1.

This solution has the advantage that the estimated cost is valid for at least one encoded implementation. It has the disadvantage that it is very hard to predict what is the gain of an optimization step, since the optimal encoding for the network after the optimization can be very different. According to preliminary experiments the cost seems to be relatively "smooth" over time, so that there are no big changes between the old estimate and the new one after each optimization step.

5.3 Don't cares

There are two kinds of don't cares that are somewhat specific to the multiple-valued optimization case:

1. The product-terms in each node cover need not be maximally expanded in the multiple-valued variable. These shall be denoted *encoding don't cares* in the rest of this paper.

2. A subset of the observability don't care sets can (and should) be computed before the encoding phase.

[4] I.e. that all the face-embedding constraints from all the nodes in the network could be simultaneously satisfied.

5.3.1 Encoding don't cares

The encoding don't care set EDC_c is computed for each product-term c of a node N whenever the user applies the *simplify* command to N. The algorithm is:

- perform node simplification with the appropriate don't care set
- reduce as much as possible the multiple-valued variable
- for each product-term c
  - let EDC_c be the set of all values that appear in the expanded MV-literal but do not appear in the reduced one

For example if the expanded cover obtained from the simplification is:
$f_e = aX^{\{0\}} + bX^{\{1\}} + abX^{\{0,1,2\}}$
then the reduced cover is:
$f_e = aX^{\{0\}} + bX^{\{1\}} + abX^{\{2\}}$
and we have an encoding don't care $X^{[0,1]}$ for the last product-term:
$f_e = aX^{\{0\}} + bX^{\{1\}} + abX^{\{2\}[0,1]}$
Here $X^{\{2\}[0,1]}$, assuming that X can have values $\{0,1,2,3\}$, denotes the boolean function that has value

$$X^S = \begin{cases} 1 & \text{if} X \in \{2\} \\ 0 & \text{if} X \in \{4\} \\ either0or1 & \text{otherwise} \end{cases}$$

The encoding don't cares are specific to a particular sum-of-product form of the node function. So they must be discarded whenever that form is changed, that is whenever a new *simplify* or *collapse* operation is performed on the node. On the other hand they remain valid whenever an algebraic operation is performed on the node.

5.3.2 Observability don't cares

Here is an example of the observability don't cares associated with the multiple-valued variable. Given:
$f = aX^{\{0,1,2\}}g + d$
$g = bX^{\{0,1,3,4\}} + cX^{\{2,5\}}$
and assuming that X can have values $\{0,1,\dots 7\}$, $X^{[6,7]}$ is a don't care for the first product-term of f:
$f = aX^{\{0,1,2\}[6,7]}g + d$
$g = bX^{\{0,1,3,4\}} + cX^{\{2,5\}}$
because when we resubstitute g and multiply out, the MV-literals are intersected.

Notice that $X^{[6,7]}$ in this case was arbitrarily assigned to a product-term of f. We could also have assigned it to both product-terms of g, giving:
$f = aX^{\{0,1,2\}}g + d$
$g = bX^{\{0,1,3,4\}[6,7]} + cX^{\{2,5\}[6,7]}$
but not to both f and g.

These don't cares could have been computed using the expand/reduce algorithm outlined above, creating the appropriate don't care set for either f or g. This "algebraic" formulation, though, is faster, and it did not show an appreciable difference from the "boolean" approach in practical cases.

In the current implementation partial observability don't cares are computed locally for each node in depth-first fashion, starting either from network outputs or from network inputs, as requested by the user. Let us denote by D_i the set of all values of the multiple-valued variable that do not appear in any of the MV-literals of the i-th node function.

E.g. let us assume X can have values $\{0,1,\dots,7\}$. If $F_0 = bX^{\{0,1,3,4\}} + cX^{\{2,5\}}$ then $D_0 = \{6,7\}$.
On the other hand if $F_1 = bX^{\{0,1,3,4\}} + c$ then $D_1 = \varphi$ because the don't care MV-literal is equivalent to $X^{\{0,1,\dots,7\}}$.

The algorithm to compute the multiple-valued don't care set $IODC_c$ associated with each product-term c of node N, given the encoding don't care set EDC_c, is:

- let ODC = Universe.
- for each fanout node N_o of N

```
    • let $ODC = ODC \cap D_o$
    • for each fanin node $N_i$ of $N_o$ ($N_i \neq N$) that appears together with $N$ in a product-term of $N_o$, with both literals
      in the *positive* phase
        • let $ODC = ODC \cap D_i$
• for each product-term $c$ of $N$
    • let $IODC_c = ODC \cup EDC_c$
    • for each fanin node $N_i$ of $N$
        • let $IODC_c = IODC_c \cup D_i$
```

5.4 Constraint matrices

As was pointed out in Section 3.2, rectangles correspond to valid common factors if and only if the constraint matrix associated with them satisfies the intersection condition ("satisfiable" rectangles and matrices). In general we cannot just reject unsatisfiable rectangles, because we would miss some useful common factors.

There are two ways in which we can reduce a matrix:

1. Eliminate some rows and/or columns from the rectangle.
2. Eliminate some values from some rectangle entries (multiple-valued tags).

5.4.1 Row/column elimination

If, for example, we want to find a common kernel in the following case:
$f_1 = abX^{\{1,2\}} + acX^{\{2\}} + adX^{\{2\}}$
$f_2 = ebX^{\{2,3\}} + ecX^{\{2\}} + edX^{\{3,4\}}$
we obtain the constraint matrix:
$M = \begin{bmatrix} X^{\{1,2\}} & X^{\{2\}} & X^{\{3\}} \\ X^{\{2,3\}} & X^{\{2\}} & X^{\{3,4\}} \end{bmatrix}$
that does not satisfy the intersection condition in position M_{11} because
$\{1,2\} \neq \{1,2,3\} \cap \{1,2,3\}$ (the union over row 1 intersected with the union over column 1).

We can delete either column 1 or column 3, obtaining
either $M' = \begin{bmatrix} X^{\{2\}} & X^{\{3\}} \\ X^{\{2\}} & X^{\{3,4\}} \end{bmatrix}$ or $M'' = \begin{bmatrix} X^{\{1,2\}} & X^{\{2\}} \\ X^{\{2,3\}} & X^{\{2\}} \end{bmatrix}$
that are both satisfiable. Of course we could also have deleted either row, but the resulting single-row rectangle would not probably have been useful to extract.

We can formulate the problem of optimally choosing the rows/columns to delete as a minimum column covering problem for which good solution heuristics exist. The resulting covering matrix C has one column for each row and each column of M whose deletion can remove a conflict in the constraint matrix.

```
• for each $M_{ij}$ where the intersection condition is violated
    • let $V_{ij} = (\bigcup_i M_{ij}) \cap (\bigcup_j M_{ij}) - M_{ij}$
      (the set of values that must disappear from either column $j$ or from row $i$)
    • for each row $k$ such that $M_{kj} \cap V_{ij} \neq \phi$
      (deleting *all* these rows is one option)
        • for each column $l$ such that $M_{il} \cap V_{ij} \neq \phi$
          (deleting *all* these columns is another option)
            • create a row of $C$ with 1's in the columns associated with rows $i$ and $k$ and with columns $l$ and $j$
              (deleting either $i$ or $j$ is the last option)
• solve the weighted minimum column covering problem
• delete the chosen rows and/or columns from $M$
```

The weight of each column of C is the sum over the corresponding row or column of M of the costs of the kernel cubes that appear in the corresponding rectangle entries. The rationale for this is to have an estimate of the penalty associated with not extracting those cubes as a common factor. It is likely that more precise cost evaluations would greatly help in this phase.

5.4.2 Value elimination

As an alternative to deleting rows and/or columns from M, we could have eliminated the value 3 either from M_{21} or from M_{13} obtaining

either $M''' = \begin{bmatrix} X^{\{1,2\}} & X^{\{2\}} & X^{\{3\}} \\ X^{\{2\}} & X^{\{2\}} & X^{\{3,4\}} \end{bmatrix}$ or $M'''' = \begin{bmatrix} X^{\{1,2\}} & X^{\{2\}} & X^{\{\}} \\ X^{\{2,3\}} & X^{\{2\}} & X^{\{3,4\}} \end{bmatrix}$

We can again formulate the problem as minimum column covering. This time each column c_i of the covering matrix M is associated with a particular set of values in a particular matrix entry, whose deletion can remove a conflict. I.e. it is associated with a triple *(row, column, set_of_values)*.

- for each M_{ij} where the intersection condition is violated
    - let $V_{ij} = (\bigcup_i M_{ij}) \cap (\bigcup_j M_{ij}) - M_{ij}$
      (the set of values that must disappear from either column j or from row i)
    - for each k such that $M_{kj} \cap V_{ij} \neq \phi$
      (deleting $M_{kj} \cap V_{ij}$ from *all* such M_{kj}'s is one option)
        - for each l such that $M_{il} \cap V_{ij} \neq \phi$
          (deleting $M_{il} \cap V_{ij}$ from *all* such M_{il}'s is one option)
            - let c_1 be the column whose tag is $(k, j, M_{kj} \cap V_{ij})$
              if no such column exists, add a new one
            - let c_2 be the column whose tag is $(i, l, M_{il} \cap V_{ij})$
              if no such column exists, add a new one
            - create a row of C with 1's in c_1 and c_2
- solve the weighted minimum column covering problem
- delete the chosen rows and/or columns from M

The weight of each column of C is the cost of the kernel cube that appears in the corresponding rectangle entry (using only the deleted part of the MV-literal). The rationale for this is to have an estimate of the penalty associated with not extracting those points in the common factor. Again this estimation is very crude, and it is likely that a more precise algorithm can achieve better results.

This algorithm neglects the fact that the union of two sets of values to be eliminated from an entry can "cover" another set to be eliminated from that entry. The constraint matrices that appear in practice, though, seem to be very easy to make satisfiable, so this does not appear to be a serious problem.

The points of the MV-literal that has been deleted from the rectangle entry tag cannot be extracted, and they must be covered separately. For example, let us assume that we chose to delete value 2 from M_{13}. Then we have:
$f_1 = abX^{\{1,2\}} + acX^{\{2\}} + adX^{\{\}} + adX^{\{3\}}$
$f_2 = ebX^{\{2,3\}} + ecX^{\{2\}} + edX^{\{3,4\}}$
where now the common kernel can be extracted without harm:
$f_1 = ag + adX^{\{3\}}$
$f_2 = eg$
$g = bX^{\{1,2,3\}} + cX^{\{2\}} + dX^{\{3,4\}}$

5.5 Encoding algorithm

Given a multiple-valued multi-level network, output of *mis-MV*, we have to solve the following problem:

> *find an encoding for the multiple-valued input variable such that the cost of the encoded boolean network is minimum*

We do not know how to solve this problem if the cost function is the sum of the literals in the factored forms of the node functions. Also we neglect the fact that further optimizations are still possible after the encoding[5]

We currently solve the problem using the sum of the literals in the sum-of-product forms of the node functions as the network cost. In the experiments we made to assess the validity of *mis-MV* we used a simulated annealing-based input encoding algorithm, since no other algorithm available to us was able to handle don't cares in the MV-literals.

The algorithm is:

```
• assign initial random codes
• for each temperature
    • repeat until equilibrium
        • exchange two codes (one of them can be a unused code)
        • let $C = 0$
        • for each node
            • add to $C$ the sum of the cost of all MV-literals in the sum-of-products expression for $N$
        • let $= C - C_{old}$
        • accept the move always if $< 0$,
          with probability $e^{-\frac{}{T}}$ if $>= 0$
```

The computation of the cost of the MV-literal is very expensive[6]. Since the simulated annealing encoding algorithm is currently the bottleneck of *mis-MV*, we are currently studying some deterministic alternative.

6 Experimental results

We made two kinds of experiments to verify the validity of *mis-MV* as *input encoder*:

- compare the relative importance of the various multi-valued optimization steps.
- compare *mis-MV* with some existing *state assignment* programs, such as *NOVA* ([15]), *MUSTANG* ([5]), *JEDI* ([8]) and *MUSE* ([6]). Notice that the comparison is not completely appropriate, since these programs embody also heuristics for the output encoding problem, that *mis-MV* does not handle.

In both sets of experiments we used the MCNC '89 set of benchmark state machines, and we encoded their present state input *only*. The output was left one-hot.

The experiments were conducted as follows:

- minimum-length encoding was always used.
- a single simplified boolean script (using *simplify* only once) was used both for multi-valued and binary valued optimization.
- the script was run twice in all cases.
- *mis-MV*:
    1. *espresso* was run on the unencoded machine.
    2. all or part of the first script was run in *mis-MV*'s multi-valued mode.
    3. the inputs were encoded, using the simulated annealing algorithm.
    4. the remaining part of the first script and the second script were run in binary-valued mode.
- *NOVA*, *MUSTANG*, *JEDI* and *MUSE*:

[5]For example we can still extract common kernels that involve only code variables, since it is impossible to have a good estimate of the value of these kernels in the multiple-valued kerneling phase.

[6]We use a single shot of the *expand* phase of *espresso*

1. each program was run in *input oriented* mode ("-e ih" for *NOVA*, "-p -c" for *MUSTANG*, "-e i" for *JEDI* and "-e p" for *muse*) to generate the codes.
2. *espresso* was run on the unencoded machine.
3. the symbolic input was encoded.
4. *espresso* was run again, using the invalid states as don't cares.
5. the script was executed twice.

We performed five experiments on each machine, two using *NOVA*, *MUSTANG*, *JEDI* and *MUSE* and three using *mis-MV*. The experiments on *mis-MV* differed in the point when encoding was perfomed:

1. at the beginning, to verify the encoding algorithm itself (that at this point has exactly the same two-level information as the other programs).
2. after *simplify*, to verify multiple-valued boolean resubstitution.
3. after algebraic optimization (*gkx*, *gcx*, ...), to verify the full power of *mis-MV*.

Table 1 contains the results, expressed as factored form literals.

Apart from a couple of cases (namely *keyb* and *tbk*) most of the gain seems to be associated with the multiple-valued *simplify*, that takes advantage of the additional don't care sets associated with unused codes. This does not mean that in general multiple-valued kerneling is not useful, just that in this set of examples most useful kernels were either independent of the encoding or they were found by mere chance[7]. Another reason is that multiple-valued kerneling extracts *boolean* common factors (as opposed to algebraic in the binary-valued case). So the computation of the value of a common kernel intersection[8] must be more sophisticated than the current algebraic-based techniques.

7 Conclusions and future work

This work proved that multiple-valued multi-level logic synthesis is both feasible and useful.

It relies heavily upon previous results in multiple-valued two-level logic synthesis [13] and input encoding [11, 15] and in binary-valued multiple-level logic synthesis [4], extending both theoretical results and algorithmic ideas.

The challenging part of the implementation is still to be able to foresee the impact of optimization decisions on the final results. Good heuristics have been developed to tackle this problem.

Possible directions for future work are:

1. analyze more examples, to explore the importance of multiple-valued kerneling.
2. improve the heuristics, especially those related with matrix reduction and kernel value computation.
3. develop a fast and efficient encoding algorithm, handling also encoding don't cares.
4. explore the output encoding problem.

Output encoding is probably the most interesting aspect, since a good solution to state encoding requires to handle both symbolic inputs and outputs. It is likely that a procedure similar to symbolic minimization ([10]) can be used for this purpose. It will require to merge the ideas presented in this paper together with a deeper understanding of the observability don't care sets that the encoding induces in the network.

[7]In some pathological cases, such as the *dk** family, kernels come almost only from the code inputs.

[8]The reduction in total area if the kernel is implemented as a node, used in guiding kernel extraction routines.

file	nova	mustang	jedi	muse	best mis-MV	beginning	simplify	algebraic optimization
bbara	106	96	96	99	84	84	84	85
bbsse	151	148	125	126	130	130	132	131
bbtas	32	37	34	36	31	35	31	31
beecount	70	65	57	60	56	62	56	58
cse	214	208	189	192	191	191	199	195
dk14	98	108	97	102	79	97	79	81
dk15	65	65	65	65	65	65	68	69
dk16	351	314	254	244	225	225	247	261
dk17	58	69	63	58	58	58	62	63
dk27	38	34	30	29	27	27	27	27
dk512	93	78	73	73	68	70	68	69
donfile	186	195	132	131	123	127	123	123
ex1	246	252	256	239	232	240	232	236
ex2	167	197	179	169	143	143	144	154
ex3	98	98	87	96	82	82	86	82
ex4	84	73	71	72	72	90	74	72
ex5	83	80	79	79	67	67	69	69
ex6	98	90	91	92	84	85	85	84
ex7	94	100	93	84	78	89	79	78
keyb	195	203	186	180	146	186	172	146
kirkman	168	181	175	195	160	169	166	160
lion	16	14	16	16	16	16	16	16
lion9	43	61	55	55	38	40	38	38
mark1	98	99	94	92	90	90	94	92
mc	32	30	32	30	30	35	30	30
modulo12	71	77	58	72	71	71	71	71
opus	82	77	83	70	70	87	70	74
planet	551	538	454	511	466	512	466	473
s1	345	377	347	291	249	335	253	249
s1a	253	264	262	195	214	217	214	225
s8	48	47	50	52	48	52	48	48
sand	542	519	552	498	509	523	509	528
shiftreg	35	34	24	25	24	24	24	24
sse	151	148	125	126	130	130	132	131
styr	501	460	413	418	438	442	438	465
tav	27	27	27	27	27	27	27	27
tbk	567	603	463	570	393	426	456	393
train11	92	88	65	79	59	60	59	59
train4	14	18	14	14	14	14	15	15
total	6163	6172	5566	5562	5087	5423	5243	5232

Table 1: Input encoding comparison

References

[1] K. Bartlett, R. Brayton, G. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic minimization using implicit don't cares. *IEEE Transactions on Computer-Aided Design*, 7(6):723–740, June 1988.

[2] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic optimization and the rectangular covering problem. In *Proceedings of the International Conference on Computer-Aided Design*, 1987.

[3] R. K. Brayton and C. McMullen. The decomposition and factorization of Boolean expressions. In *Proceedings of the International Symposium on Circuits and Systems*, 1982.

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

[5] S. Devadas, H.-K. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State assignment of finite state machines targeting multi-level logic implementations. *IEEE Transactions on Computer-Aided Design*, (12):1290–1300, December 1988.

[6] G. Hachtel, X. Du, and P. Moceyunas. Algorithms for state assignment based on multi-levelrepresentation. In *Proceedings of the Hawaii International Conference on System Sciences*, 1990.

[7] G. D. Hachtel and R. M. Jacoby. Verification algorithms for VLSI synthesis. *IEEE Transactions on Computer-Aided Design*, CAD-7(5):616–640, May 1988.

[8] Bill Lin and A. R. Newton. Synthesis of multiple level logic from symbolic high-level description languages. In *Proceedings of the International Conference on Very Large Scale Integration*, 1989.

[9] Sharad Malik, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Encoding symbolic inputs for multi-level logic implementation. In *Proceedings of the International Conference on Very Large Scale Integration*, 1989.

[10] G. De Micheli. Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros. *IEEE Transactions on Computer-Aided Design*, (4):597–616, October 1986.

[11] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design*, (3):269–285, July 1985.

[12] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The transduction method - design of logic networks based on permissible functions. In *IEEE Transactions on Computers*, 1989.

[13] R. L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, (5):727–750, September 1987.

[14] A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. submitted to the International Conference on Computer-Aided Design, 1990.

[15] T. Villa. Constrained encoding in hypercubes: Algorithms and applications to logical synthesis. Technical Report UCB/ERL M87/37, Electronics Research Lab., U. C. Berkeley, May 1987.