Multi-valued Logic Synthesis

Robert K Brayton   Sunil P Khatri
University of California
Berkeley, CA 94720
{brayton, linus}@ic.eecs.berkeley.edu

Abstract

We survey some of the methods used for manipulating, representing, and optimizing multi-valued logic with the view of both building a better understanding of the more specialized binary-valued logic, as well as motivating research towards a true multi-valued multi-level optimization package.

1. Introduction

Logic design is normally thought of in terms of binary signals; however for higher level design it is natural to think of variables with symbolic values. For example, it is easier to conceive of a traffic light processor with a signal *light* taking on three values *red*,*yellow*, and *green* rather than dealing with $light_0 = 1$, $light_1 = 0$ to stand for the light being red. The process of converting these multi-valued variables to binary signals is called encoding. In many cases the encoding is done initially, mostly arbitrarily, and then binary valued logic synthesis is applied to the resulting circuit. An alternative is to first manipulate and optimize the logic directly as multi-valued logic. Then the resulting form of the network can be used (possibly) intelligently to select a good encoding. Once the encoding is done, further optimizations, not possible in the purely multi-valued form, can be applied to the resulting binary network. The intelligent encoding should take into account this additional optimization which will depend on the final binary codes selected.

However, this alternative approach is not used often because:

- There is no good multi-valued multi-level logic optimization package for a multi-valued logic network (such as SIS for binary networks).
- Although many of the algorithms in logic synthesis have been generalized to multi-valued logic, a complete suite of algorithms has not been developed
- The encoding problem is hard for large circuits since it is difficult to see how an encoding decision ultimately affects the logic that results after powerful logic optimizations are applied.

Multi-valued logic is a generalization. One advantage in dealing with generalizations is that it can lead to increased insight into the specialized problem. A generalization helps differentiate the special properties from the general ones. Often a property that is known for the special case can be a general property in disguise or a specialization of a more general property. When this is understood, frequently there is a sense of ”oh, is that what I was really doing”. Thus the attempt to generalize helps understand the special case better.

In this paper, we survey several of the concepts, algorithms, and optimizations that have found extensions from binary to multi-valued logic. We first deal with two-level logic where most of the concepts directly generalize. Then we look at several methods for representing multi-valued logic; sum-of products (SOPs), multi-valued decision diagrams (MDDs), and multi-level multi-valued networks (MV-networks). We look at algorithms for manipulating Boolean networks (decomposition, factorization using kernels, and extensions of don't cares (SPFDs)) and see their generalizations to MV-networks. We discuss extensions to a popular RTL language (Verilog) to MV-variables, and use this to build a front-end to VIS, an MV-logic optimization and verification package. Finally, the state assignment problem is revisited and we conclude the paper with a discussion of some open problems and work for the future.

2. Notation

Definition 1 *A* multi-valued variable X_i *can take on values from* $P_i = \{\alpha_0, \alpha_1, \cdots, \alpha_{|P_i|-1}\}$.

Since each symbolic value α_i can be associated with a unique integer i, we henceforth only consider multi-valued variables with integer values, for uniformity, and $P_i = \{0, 1, \cdots, |P_i| - 1\}$.

X_1	X_2	$\mathcal{F}$
0	0	1
0	1	2
1	0	1
1	1	1
2	0	0
2	1	2

Figure 1. A Multi-valued Function of 2 variables

Definition 2 *A* vertex *is a point in the space $P_1 \times P_2 \times \cdots \times P_n$*

Definition 3 *A* multi-valued function *$\mathcal{F}$ is a function which maps vertices in $P_1 \times P_2 \times \cdots \times P_n$ to $P_{\mathcal{F}}$, formally, $\mathcal{F} : P_1 \times P_2 \times \cdots \times P_n \mapsto P_{\mathcal{F}}$.*

An example of a multi-valued function is shown in Figure 1. Assume that $P_1 = \{0,1,2\}$, $P_2 = \{0,1\}$, and $P_{\mathcal{F}} = \{0,1,2\}$.

If $P_{\mathcal{F}} = \{0,1,*\}$, $\mathcal{F}$ is a multi-valued function with a binary-valued output. If a vertex (minterm) is mapped to the 1 value, it is said to be in the *on-set* of $\mathcal{F}$, mapped to 0, in the *off-set*, and to $*$, in the *don't-care set*. The binary ideas of *implicants*, *prime implicants*, *covers*, and *prime covers* can be extended to multi-valued functions for functions $\mathcal{F}$ with binary valued outputs.

Definition 4 *A* multi-valued literal *$X_i^{c_i}$ is a logic function of the form*

$$X_i^{c_i} = (X_i = \gamma_1) + \cdots + (X_i = \gamma_k), \text{ where } \gamma_j \in c_i \subseteq P_i$$

Definition 5 *A* cube *$c = c_1 \times c_2 \times \cdots \times c_n$ can be written as a product of MV-literals in the form:*

$$X_1^{c_1} X_2^{c_2} \cdots X_n^{c_n}$$

Note that if $c_i = P_i$, we can omit $X_i^{P_i}$ from the expression of the cube, since $X_i^{P_i} = 1$. If variable X_i is binary valued, the literal X_i can be written in the new notation as $X_i^{\{1\}}$. Similarly, the literal $\overline{X_i}$ can be written as $X_i^{\{0\}}$. If the variable X_i takes on both its values (also written as a "-"), this is written as $X_i^{\{0,1\}}$.

The next four definitions apply specifically to binary valued functions of multi-valued inputs.

Definition 6 *An* implicant *is a cube c such that for all vertices $v \in c$, $\mathcal{F}(v) \neq 0$.*

Definition 7 *A* prime implicant *is an implicant c such that there is no implicant d such that $d \supset c$.*

Definition 8 *A* cover *of $\mathcal{F}$ is a set of implicants whose union contains every point in the onset of $\mathcal{F}$ and no points in the offset.*

Definition 9 *A* prime cover *of $\mathcal{F}$ is a cover, each of whose elements is prime.*

The multi-valued function in Figure 1 can be written in the form of a sum of cubes for each of its values. One such cover for $\mathcal{F}$ is,

$$F^{\{0\}} = X_1^{\{2\}} X_2^{\{0\}}$$
$$F^{\{1\}} = X_1^{\{0,1\}} X_2^{\{0\}} + X_1^{\{1\}} X_2^{\{0,1\}}$$
$$F^{\{2\}} = X_1^{\{0,2\}} X_2^{\{1\}}$$

A convenient representation of literals and cubes utilizes *positional notation*:

Definition 10 Positional Notation: *A literal $X_i^{c_i}$ is assigned positions (or columns) $v_0, v_1, \cdots v_{|P_i|-1}$, such that*

$$v_j = \begin{cases} 1 & \text{if } j \in c_i \subseteq P_i \\ 0 & \text{otherwise} \end{cases}$$

For example, the multi-valued function in Figure 1 can be written in positional notation as:

X_1	X_2	$\mathcal{F}$
001	10	0
110	10	1
010	11	1
101	01	2

3. Generalizations

3.1. Boolean Algebra

A Boolean algebra is a set of objects on which there are two operations defined. The operations obey a certain set of rules. A Boolean algebra is often associated with binary functions of binary variables. The Boolean algebra in this case is the algebra of the manipulation of binary logic functions. Each such function can be thought as a set of points, its onset. It is just the characteristic function of its onset, i.e. it is 1 when applied to a point in its onset and 0 otherwise. Two functions *AND*ed together is the same as taking the intersection of their onsets. Similarly *OR*ing corresponds to taking the union. It is known that any Boolean algebra is isomorphic to the Boolean algebra of sets where union and intersection are the two operations. Note that nothing has been said about the size of the domain space. In fact one can use multi-valued variables to describe a point in some space. For example, suppose we use two variables, x with 5 values, and y with 3 values. Then there are 15 points in the domain space. A point (or minterm) in the space is given by assigning each of the variables a value from their domains, e.g. ($x = 3$, $y = 1$). A function is just an arbitrary subset of such minterms. Thus the mathematics of Boolean algebras directly applies to binary functions of multi-valued variables. Considering each output value as a separate function, one can treat the case where the range of the function is also multi-valued. Thus for example, the set of points where the signal *light* is *red* is the onset of one function, the points where *light* is *yellow* another function, etc.

3.2. One-Hot Encoding and Multi-valued Signals.

One of the first methods used to treat multi-valued variables in logic was the use of a one-hot encoding for the signals, with an associated set of don't cares. For example, consider the traffic light processor and signal *light*. A one-hot encoding would create three three signals $light_r$, $light_y$, $light_g$ with the set of don't cares given by the logic expression,

$$light_r \cdot light_y + light_r \cdot light_g + light_y \cdot light_g$$

which says that we don't care for example that both $light_r$ and $light_y$ are 1, since it will never occur. This formulation is fully equivalent to manipulating multi-valued signals and its advantage is that it maps the problem back to the binary case and hence the fully developed binary algorithms apply directly. Further, future developments in binary methods can be used when they develop. The disadvantages are that many more signals are introduced and the associated don't cares can become very large. In the area of two-level logic optimization, these latter reasons were enough to spur the development of ESPRESSO-II, a package for two-level multi-valued logic optimization. (However, in the multi-level case, this motivation has not been sufficient so far). An interesting footnote is that when ESPRESSO-II was completed and compared to the original ESPRESSO where both were applied to purely binary functions, ESPRESSO-II was faster. The explanation was that the generalization to multi-valued logic led to a superior method of representation of the functions for computer manipulations.

3.3. Multi-valued Logic Minimization in ESPRESSO-II

For a multi-valued function with a binary-valued output, most of the binary logic minimization theory can be generalized. As already discussed, the concepts of *implicants, prime implicants, covers* and *prime covers* are easily extended to such functions. As in the binary case, the process of logic minimization involves generating primes, generating a covering table, and solving this covering table. The notions of cofactors and the Shannon expansion theorem have also been generalized to the multi-valued case.

Definition 11 *The* cofactor *of a function f with respect to a MV literal X^s, denoted f_{X^s}, is obtained by eliminating all cubes of f that are disjoint to s, and expanding the remaining cubes by unioning into the X position all values not in s.*

The cofactor with respect to a MV-cube is obtained by taking the sucessive cofactors with respect to each MV-literal in the cube.

Theorem 3.1 Multi-valued Shannon Expansion Theorem: *Let f be any function and $\{c^1, c^2, \cdots, c^t\}$ any set of MV-cubes such that*

$$\sum_{i=1}^{t} c^i = 1$$

Then,

$$f = \sum_{i=1}^{t} c^i f_{c_i}$$

It follows from the above that

$$f \equiv 1 \;\; iff \;\; f_{c^i} = 1 \;\; for \;\; each \;\; i.$$

An algorithm for multi-valued tautology can be devised based on this, much like in the binary case, where typically the cubes $x, \overline{x}$ are used.

Definition 12 *A function f is said to be* weakly unate *in X_i if there exists some value $= j$ such that changing X_i from value $= j$ to any other value does not cause f to decrease, i.e. f is not changed from 1 to 0.*

Weak unateness is one multi-valued analog of unateness. (There is another anlog, strong unateness, which for binary valued functions is the same as weak unateness.) The *unate reduction* theorem for tautology applies in the multi-valued case as well. Generation of primes and the binary routines of *essential prime generation, reduce* and *irredundant* remain essentially unchanged.

Based on the above, ESPRESSO-II handles binary valued functions of multi-valued functions. Positional notation is used to specify the multi-valued portion of the function. Symbolic variables are supported as well. MV-applications of ESPRESSO-II include state assignment [1] and PLAs where inputs are paired and decoded to form MV-inputs.

3.4. Funtional Representation

We will review several methods for representing logic functions in the MV domain.

3.4.1 Sum-of-products

One of the earliest methods used for binary functions was a two-level sum-of-product representation. Early logic synthesis work was done on this type of representation. Although it is inherently simple, there are certain functions (like the odd or even parity function) which have exponential sized representations. As we have already seen, multi-valued functions can be represented in a two-level sum-of-product scheme. Logic minimization on such functions can be performed in ESPRESSO-II. For certain functions, this scheme has the drawback of giving rise to exponential-sized SOPs.

3.4.2 MV-networks

Another powerful representation technique is the multi-level boolean network, each of whose nodes are two-level sum-of-products. This scheme has the ability to represent implementable boolean functions very compactly. A good deal of research on this type of representation has been performed, fuelled by the introduction of SIS, a sequential optimization and synthesis tool. The multi-level network of multi-valued nodes (called an MV-network) is a direct generalization of this. It is similar to a multi-level boolean network except that each node is, in general, a multi-valued function. VIS (Verification Interacting with Synthesis) is a research tool whose input is such a network. The input format format of VIS is called *blif-mv*. (VIS is discussed in more detail in Section 4). It is hoped that tools like VIS will result in increased research in synthesis for multi-valued networks. The drawback of these network representations (as well as sum-of-products) is that there are multiple ways to represent a given function under these schemes.

3.4.3 MDDs

This drawback is eliminated in a boolean function representation scheme called Reduced Ordered Binary Decision Diagrams (henceforth abbreviated as BDD). BDDs have the appealing property that they are canonical, and hence the problem of checking for functional equivalence is trivial. Yet, they also have the drawback that for some implementable circuits, the BDD is exponential in the number of input variables.

BDDs have been generalized to the multi-valued case, resulting in a Multi-valued Decision Diagram (MDDs). MDDs apply to multi-valued functions with binary-valued outputs. However, if a multi-valued function has an n-valued output, where $n > 2$, multi-valued functions (MVFs) are created first. Essentially, we construct MDDs for each value of the multi-valued output variable. So, for example, if the multi-valued function f has 3 values, then the MVF(f) has 3 MDDs, f_a, f_b and f_c.

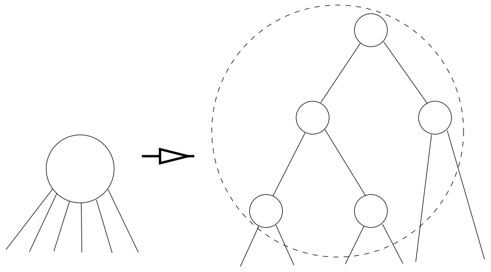


Figure 2. An MDD Node and its Corresponding BDD Nodes.

MDDs are a simple extension of BDDs. Each node in an MDD has k children instead of just two, where k is the number of values the variable associated with the node can take. The result is a DAG with the root node representing the function, and the leaf nodes representing 0 and 1. A pure MDD package was built and experimented with several years ago in Berkeley [2].

Another option is to encode each multi-valued node with k children using $log_2(k)$ binary variables. Thus for example an MDD node with 6 children would be split into 3 binary variables. In Figure 2, the MDD node on the left is transformed to the group of nodes on the right. Note that in both cases, the number of children is 6. Although with 3 binary variables, it is possible to represent 8 children, the extra two leaves are used as don't cares in the process in a somewhat arbitrary but specific way.

An MDD package was also developed at Berkeley based on this conversion to binary variables. The MDD package was constructed as a high level interface to a BDD package. In fact any BDD package can easily be used. For the user, only multi-valued variables are observable; the conversion to binary variables is internal and transparent. The advantages of this approach are:

- The continuing development of BDD packages can be leveraged in the MDD package.
- Any newly developed BDD package that proves to be superior can be easily slipped under the covers.
- The binary variables associated with a multi-valued variable do not have to be kept adjacent in the binary variable ordering, whereas with a purely multi-valued version, the effect is as if the associated binary variables are constrained to be together in the ordering. In some examples, this leads to a significant increase in MDD size. Thus in this case the initial and arbitrary binary encoding used does not seem to have any negative consequence.

3.5. Multi-valued Redundancy Removal

Recent methods [3] [4] for binary redundancy removal avoid the use of state traversal. Additionally, [4] finds multiple compatible redundancies simultaneously. These powerful advances in the field of binary redundancy removal were extended in [5] to perform redundancy removal for multi-valued networks. This method works in the following manner.

First a one-hot encoding of all the multi-valued variables of the design is performed. Multi-valued variables are written out as binary variables, using this one-hot encoding. The binary network is equivalent to the multi-valued network modulo encoding.

Next, binary redundancy removal is invoked on the resulting network. We only check for signals *stuck-at-0* in the binary network. In case a binary signal s_i feeding binary gate t_j is determined to be *stuck-at-0 redundant*, this means that the multi-valued signal s in its fanout to multi-valued signal t is a don't care for value j. Hence we can choose to remove the i^{th} value of variable s occuring in any MV-cubes of t with output j. Since each table has a default value, this has the effect of making the output of such a cube restricted to $s = 1$ equal to the default value. This simplifies the table for t by reducing or removing cubes. We do not need to worry about *stuck-at-1* redundancies in the binary network, since because the signals are one-hot, a *stuck-at-1* on a value of s, has to be associated with *stuck-at-0*'s on all the other values of s.

All redundant binary signals are recorded in a file during the binary redundancy processing of the binary network. Then the original multi-valued network is modified as above, based on the binary redundancies thus computed.

Initial experiments using this technique show a 10-20% reduction in the size of the multi-valued description.

3.6. Multi-Valued Factorization

One of the more effective methods for treating multi-level Boolean networks has been the use of *kernels* for finding common factors among several binary logic functions. The common factor can then be removed as a separate function and used to simplify some of the functions. To see how this concept is extended to multi-valued functions, consider the following two functions

$$\begin{aligned} f_1 &= X^{\{0,1\}} \cdot a \cdot k + X^{\{2\}} \cdot b \cdot k + c \\ f_2 &= X^{\{3,4\}} \cdot a \cdot j + X^{\{5\}} \cdot b \cdot j + d \end{aligned}$$

We will show that the function

$$X^{\{0,1,3,4\}} \cdot a + X^{\{2,5\}} \cdot b$$

is a common factor of both f_1 and f_2, and thus the network can be rewritten as

$$\begin{aligned} f_1 &= X^{\{0,1,2\}} \cdot k \cdot y_3 + c \\ f_2 &= X^{\{3,4,5\}} \cdot j \cdot y_3 + d \\ y_3 &= X^{\{0,1,3,4\}} \cdot a + X^{\{2,5\}} \cdot b \end{aligned}$$

The first step is to find all the kernels and co-kernels by successive co-factoring by single binary literals. For this example, we obtain the following table

Exp	co-kernel	kernel
f_1	1	$a \cdot k \cdot X^{\{0,1\}} + b \cdot k \cdot X^{\{2\}} + c$
f_1	k	$a \cdot X^{\{0,1\}} + b \cdot X^{\{2\}}$
f_2	1	$a \cdot j \cdot X^{\{3,4\}} + b \cdot j \cdot X^{\{5\}} + d$
f_2	j	$a \cdot X^{\{3,4\}} + b \cdot X^{\{5\}}$

We put this in a *co-kernel cube matrix M* as follows

$$\begin{array}{ccccccccc} & a & b & a \cdot k & b \cdot k & a \cdot j & b \cdot j & c & d \\ 1 & 0 & 0 & X^{\{0,1\}} & X^{\{2\}} & 0 & 0 & 1 & 0 \\ k & X^{\{0,1\}} & X^{\{2\}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & X^{\{3,4\}} & X^{\{5\}} & 0 & 1 \\ j & X^{\{3,4\}} & X^{\{5\}} & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Note that the binary parts of the cubes of the kernels are extracted out at the top of each column. A rectangle of such a matrix is a set of rows and a set of columns. For example $\{(2,4),(1,2)\}$ is a rectangle. Associated with a rectangle is a matrix of MV entries, e.g.

$$\begin{array}{cc} X^{\{0,1\}} & X^{\{2\}} \\ X^{\{3,4\}} & X^{\{5\}} \end{array}$$

Such a rectangle can give rise to a common factor provided that the matrix is *satisfiable*, which means for every variable, e.g. X, if a value occurs somewhere in row i and the same value occurs somewhere in column j, then that value must also occur in entry M_{ij}. The above matrix is satisfiable. For a satisfiable rectangle, we can extract the common factor as follows. For each row of the rectangle, the union of row entries is *AND*ed with the co-kernel associated with that row. Similarly, for each column of the rectangle, the union of all column entries is *AND*ed with the binary cube attached to the column. The kernel is then the *OR* of the results for all the columns of the rectangle. In the above example, this yields for column 1, $a \cdot X^{\{0,1,3,4\}}$, and for column 2, $b \cdot X^{\{2,5\}}$, and the kernel $a \cdot X^{\{0,1,3,4\}} + b \cdot X^{\{2,5\}}$. For row 2 we get, $k \cdot X^{\{0,1,2\}}$ and for row 4, $j \cdot X^{\{3,4,5\}}$, yielding a factorization

$$\begin{aligned} f_1 &= k \cdot X^{\{0,1,2\}}(a \cdot X^{\{0,1,3,4\}} + b \cdot X^{\{2,5\}}) + c \\ f_2 &= j \cdot X^{\{3,4,5\}}(a \cdot X^{\{0,1,3,4\}} + b \cdot X^{\{2,5\}}) + d \end{aligned}$$

It has been proved that if κ is a kernel found by the usual Boolean kerneling process for some encoding, then it will be found by the above MV factoring process. In addition, the MV process can find some "Boolean factors" for an encoding.

Matrices that are not satisfiable can be "reduced" to satisfiable matrices by considering for each M_{ij} a subset of values in order to remove any offending value in an entry. In addition, don't cares can be expressed as $X^{\{0,1,2\}[6,7]}$ if the values of $X = 6$ or 7 are don't care for the function. Then for a given entry M_{ij} one has the option of including the values 6,7 in order the make the matrix satisfiable.

See [6] for an extended discussion of these ideas.

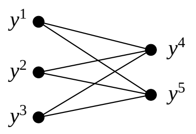
3.7. SPFDs

A new method for specifying implementational flexibility in boolean networks was introduced in [7]. This work was generalized to MV-networks in [8]. SPFDs are like don't cares but are more powerful. Unlike don't cares, which compute the flexibility of a single node in a network, SPFDs express the flexibility of a node in a network along with the nodes in its fanin.

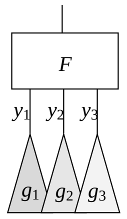
In general, SPFDs are a set of inter-related Incompletely Specified Functions (ISFs). An ISF can be represented as a complete bipartite graph on the minterms in the offsets and onsets. An edge between minterms indicates that a function that distinguishes the onset and offset minterms on that edge is required. This kind of graph has exactly two minimum colorings corresponding to implementing the onset or the offset.

In the SPFD method, we first build the complete bipartite graph of an ISF F. This gives pairs of minterms that need to be distinguished. Figure 3(a) shows an example bipartite graph with minterms $y^1, y^2, \cdots y^5$ in the input y space. Assume that the inputs to F are $y = (g_1(x), g_2(x), g_3(x))$, as shown in Figure 3(b). Then, if minterms $\{ y^1, y^2, y^3 \}$ are encoded differently from $\{ y^4, y^5 \}$ by $g(x)$, we have enough information in y to build a valid implementation of F. The task of distinguishing different pairs of minterms can be distributed to different input wires, as shown in Figure 4. Note that even though F started out as an ISF (a complete bipartite graph), the graphs for g_i are not bipartite, hence not ISFs. They are SPFDs. Any coloring of the SPFDs of the 3 wires in Figure 4 is a valid implementation of the functions $g_1(x), g_2(x)$ and $g_3(x)$. For example, input 1 has 4 possible two-colorings, corresponding to 4 possible implementations of g_1. In general, SPFDs provide the flexibility to change both the functions g which implement the SPFDs derived for these inputs, and also to re-implement F to reflect the new encoding of the inputs.

The above discussion on binary valued SPFDs is easily generalized [8], as follows.



(a) Bipartite graph for F.



(b) Structure of F.

Figure 3. SPFDs - an example

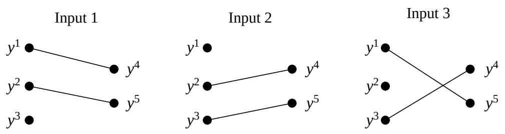


Figure 4. An implementation of F.

Definition 13 *A* SPFD *$\mathcal{F}(y)$ on domain Y is an undirected graph (V, E) where each $v \in V$ is encoded as a minterm $v = (y_1, y_2, \cdots y_k) \in Y$.*

Definition 14 *A function f(y)* implements *an SPFD $\mathcal{F}(y)$ = (V, E) if f(y), $y \in V$ is a valid coloring of $\mathcal{F}$, i.e.*

$$f(y^1) \neq f(y^2), (y^1, y^2) \in E.$$

Analogous to the binary case, each valid coloring of the SPFD gives an implementation of $\mathcal{F}$. The *chromatic number* of the graph is the minimum number of values that the resulting function is required to have in its range. Thus if this is greater than 2, multi-valued functions are required. Each valid coloring of the graph gives rise to a MV-function.

3.8. Decomposition of Multi-valued Functions

In [9], the authors extend the extensive work on the decomposition of binary functions to MV-functions. Consider set functions of the form $f : E^n \to D^m$, with n inputs $x_1, x_2, \cdots x_n$ and m outputs $y_1, y_2, \cdots y_m$ which are partially specified. Here E is a finite, nonempty set and $D = 2^E - \{\emptyset\}$; in general, f assigns to any output y_i a nonempty set of elements of E. The problem of *decomposition* of $f(x_1, x_2, \cdots, x_n)$ in the form $h(u_1, u_2, \cdots, u_r, g(v_1, v_2, \cdots, v_s))$ is addressed. Here $X = \{x_1, x_2, \cdots, x_n\}$ is the set of input variables, and $U = \{u_1, u_2, \cdots, u_r\}$ and $V = \{v_1, = v_2, \cdots, v_s\}$ are two subsets of X whose union is X. Figure 6 shows such a decomposition.

The function f is represented as a *set matrix M*, where each row consists of a $n + m$-tuple $t = t_1, \cdots, t_n, t_{n+1}, \cdots t_{n+m}$. The *input projection* of t is $t_{in} = t_1, \cdots t_n$, and the *output projection* of t is $t_{out} = t_{n+1}, \cdots t_{n+m}$. The matrix M is required to be *consistent*, which means that

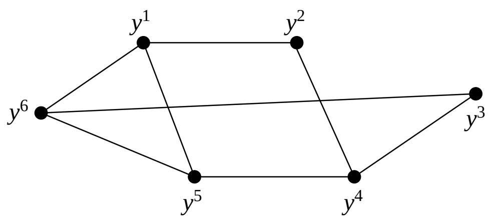


Figure 5. A multi-valued SPFD.

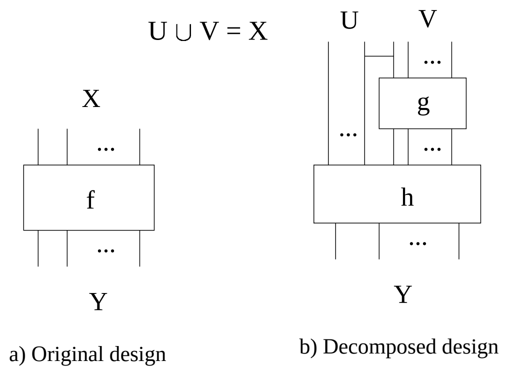


Figure 6. Multi-valued decomposition

if the input projections of a set of rows cover an input vertex, then the corresponding output projections should have a common value. The function f is evaluated at vector e by taking the intersection of the y values of all the rows that cover e. The decomposition proceeds by first finding sets U and V, then finding a *blanket* β_g from M. From β_g, g and h can be constructed.

Definition 15 *Given a set S, a* blanket $\beta = \{B_1, B_2, \cdots B_k\}$ *is a set of sets of nonempty, distinct but not necessarily disjoint subsets of S called* blocks, *whose union is S.*

For example, if $S = \{1,2,3\}$, then a blanket of S is $\beta^1 = \{\{1,2\},\{2,3\},\{1\}\}$.

Definition 16 *The* blanket product *of two blankets* β *and* β^* *is a blanket given by*

$\beta * \beta^* = undup(ne\{B_i \cap B_j | B_i \in \beta, = B_j \in \beta^*\})$,

where $ne\{B_i\} = \{B_i\} - \{\emptyset\}$. $undup(\beta)$ *removes the duplicate entries in* β.

Consider $\beta^2 = \{\{1\},\{1,3\},\{2\}\}$. Then $\beta^1 * \beta^2 = \{\{1\},\{2\},\{3\}\}$.

Definition 17 $\beta \le \beta'$ *if for each* $B_i \in \beta$, *there is a* $B_j \in \beta'$ *such that* $B_i \subseteq B_j$.

In the above examples, $\beta^1 * \beta^2 \le \beta^1$.

In the remainder of this section, we refer to blankets of rows of set matrices.

1	$\{1\}$	$\{0,1,2\}$	$\{0\}$
2	$\{1\}$	$\{1\}$	$\{0,2\}$
3	$\{0,1,2\}$	$\{0,1\}$	$\{0\}$
4	$\{1\}$	$\{0,1\}$	$\{0\}$
5	$\{0\}$	$\{0,1\}$	$\{1,2\}$

Table 1. Set matrix to illustrate row blanket

Definition 18 *The* row blanket β_f *of a set matrix M for f having h rows and k columns is given by*

$\beta_f = ne\{T_{\supseteq e}\}$ *where*

$T_{\supseteq e} = \{t \in T | t \supseteq e\}$

where T is the set of rows of M, and $e \in E^k$.

Consider the set matrix M given in Table 1. Note that $T_{\supseteq 000} = \{3\}$ and $T_{\supseteq 100} = \{1,3,4\}$. Listing all the unique blocks corresponding to the minterms of the table, we get the row blanket for this matrix $\beta_f = \{\{3\},\{5\},\{1,3,4\},\{1,2,3,4\},\{2\},\{1\}\}$.

In the following, $X = U \cup V$ and the variables V correspond to the support of the function g in the decomposition. s_{in}^Z refers to the projection of tuple s_{in} on the Z space.

Definition 19 *For all tuples t and u, if there exist multi-valued minterms d and e such that* $t_{in}^V \supseteq d$, *and* $u_{in}^V \supseteq e$, *then t and u appear in the same block of* β_f. *In this case,* β_f *is said to* correspond to g with respect to V.

Theorem 3.2 *Let set function f(X) be specified by a consistent set matrix T of tuples, and let U, V of X be such that U ∪ V = X. For every blanket* β_g *satisfying*

$$\beta_f^V \le \beta_g \text{ and } \beta_f^U * \beta_g \le \beta_f \qquad (1)$$

there exists a decomposition (g,h) *of* f *such that* β_g *corresponds to g with respect to V.*

Consider the set matrix in Table 2. The set $U = \{x_1\}$ and $V = \{x_2, x_3\}$. Hence

$$\beta_f^U = \{\{1\},\{2,4\},\{3,4\}\},$$
$$\beta_f^V = \{\{1,3,4\},\{1,4\},\{2,3\},\{2\},\{2,3,4\},\{2,4\}\},$$
$$\beta_f = \{\{1\},\{2,4\},\{4\},\{3,4\},\{2\},\{3\}\}.$$

Note that $\beta_g = \{\{1,2,4\},\{2,3,4\},\{1,3,4\}\}$ satisfies equation 1. Now encode these three blocks using a multi-valued variable with values 0, 1, 2 respectively.

The construction of g from β_g proceeds as follows. For each multi-valued minterm in V, we enumerate the rows of T covering this minterm. Now all the blocks B_i of β_g are

Row	x_1	x_2	x_3	f_1	f_2
1	$\{0\}$	$\{0\}$	$\{0,2\}$	$\{0,1\}$	$\{0\}$
2	$\{1\}$	$\{1,2\}$	$\{0,2\}$	$\{0,2\}$	$\{1\}$
3	$\{2\}$	$\{0,1,2\}$	$\{0\}$	$\{1,2\}$	$\{2\}$
4	$\{1,2\}$	$\{0,2\}$	$\{0,2\}$	$\{0,1\}$	$\{1,2\}$

Table 2. Example set matrix to illustrate decomposition

x_2	x_3	$\beta_V \leq$	β_g	codes	g
0	0	$\{1,3,4\}$	$\{1,3,4\}$	2	2
0	2	$\{1,4\}$	$\{1,2,4\},\{1,3,4\}$	0, 2	2
1	0	$\{2,3\}$	$\{2,3,4\}$	1	1
1	2	$\{2\}$	$\{1,2,4\},\{2,3,4\}$	0, 1	1
2	0	$\{2,3,4\}$	$\{2,3,4\}$	1	1
2	2	$\{2,4\}$	$\{1,2,4\},\{2,3,4\}$	0, 1	0

Table 3. Construction of g from β_g

listed such that these blocks contain the rows of T covering the minterm. From the feasible B_i for this minterm, we choose one B_i as the implementation of g for that multi-valued minterm. Finally these B_i are encoded.

An example of the construction of g given β_g is shown Table 3.

Similarly, in the construction of h from β_g, we first list, for each multi-valued minterm of U, rows of T that cover it (see first two columns below in Table 4. For each minterm of U, we list the possible multi-valued minterms of g, along with their implemented code from the step above. Intersecting the two sets gives us an element B_k. For each such element, we list all elements $B_j \in \beta_f$ such that $B_k \leq B_j$. Choose one element as the implementation. The outputs are chosen by intersecting the outputs of the rows corresponding to the chosen implementation element.

An example of the construction of $h = (h_1, h_2)$ given β_g is shown in Table 4. Note that h is kept as a set function for maintaining flexibility for further decompositions.

Finding β_g is not simple, but an algorithm for this starts

x_1	β_U	g	β_g	$\beta_U * \beta_g$	$\leq \beta_f$	h_1	h_2
0	$\{1\}$	2	$\{1,3,4\}$	$\{1\}$	$\{1\}$	0, 1	0
1	$\{2,4\}$	2	$\{1,3,4\}$	$\{4\}$	$\{2,4\},\{4\},\{3,4\}$	0, 1	1, 2
1	$\{2,4\}$	0	$\{1,2,4\}$	$\{2,4\}$	$\{2,4\}$	0	1
1	$\{2,4\}$	1	$\{2,3,4\}$	$\{2,4\}$	$\{2,4\}$	0	1
2	$\{3,4\}$	2	$\{1,3,4\}$	$\{3,4\}$	$\{3,4\}$	1	2
2	$\{3,4\}$	0	$\{1,2,4\}$	$\{4\}$	$\{2,4\},\{4\},\{3,4\}$	0, 1	1, 2
2	$\{3,4\}$	1	$\{2,3,4\}$	$\{3,4\}$	$\{3,4\}$	1	2

Table 4. Construction of h from β_g

with β_V, and merges blocks to get β' such that $\beta_V \leq \beta'$. Now check if $\beta_U * \beta' \leq \beta_f$.

4. The VIS System

VIS (Verification Interacting with Synthesis) is a software tool distributed by the University of California, Berkeley, and the University of Colorado, Boulder. VIS is a tool integrating verification, simulation and synthesis of finite-state hardware systems. It has a Verilog front end, which generates a *blif-mv* description of the network. *blif-mv* is a format for representing MV-networks. VIS supports formal verification (fair CTL model checking, language emptiness checking and equivalence checking), hierarchical synthesis from a multi-valued description, and cycle based simulation of the multi-valued input. In this way, VIS provides a strong platform for research in formal verification and in the future, hierarchical multi-valued synthesis.

4.1 Multi-valued extensions to Verilog

Part of the VIS system is a Verilog translator (*vl2mv*) which which supports a multi-valued extension to Verilog (as well as nondeterminism). The user can declare that a variable is of a particular type with its range of values given by referring to a *typedef* statement. For example,

```
   typedef color { red,yellow,green }
```

declares *color* as a type. Later,

```
   signal light color
```

declares the variable *light* to have type *color*. The Verilog translator, translates the input into an MV-network represented in a file using *blif-mv*.

4.2 Blif-mv

blif-mv is an intermediate format that is output by the Verilog translator. It represents an MV-network using tables to represent multi-valued functions. Each table is a cover of MV-cubes of the corresponding multi-valued function. These tables are *fully specified* (all multi-valued vertices are assigned some output value) and *deterministic* (each multi-valued vertex is assigned a unique output value). *blif-mv* is a simple extension of *blif*, the intermediate format used in SIS. *blif-mv* includes for convenience some higher level constructs not in *blif*. One such that is particularly useful for multi-valued variables is the ”equal” construct. Consider a multiplexor with a single binary control and two multi-valued inputs a and b. In the pure table format, we would have to say

x	*a*	*b*	*output*
0	0	-	0
0	1	-	1
0	2	-	2
0	3	-	3
⋮	⋮	⋮	⋮
1	-	0	0
1	-	1	1
⋮	⋮	⋮	⋮

With the *equal* construct, the table is compacted into two lines, no matter how many values are in the range of *a* and *b*

x	*a*	*b*	*output*
0	-	-	=*a*
1	-	-	=*b*

4.3 VIS internals

The MV-network in the *blif-mv* format is translated inside VIS into a set of MVFs before any formal verification or simulation is performed. A simple multi-valued simulation in provided in VIS. It is performed by using the MDDs of the MVFs of the functions to be simulated. Assume that a function has an MVF with n MDDs, each corresponding to the n values of the function. For the i^{th} MDD, simulation proceeds by cofactoring this with respect to the vector corresponding to the system inputs. If the result is a “1”, then the simulation output is i, and the remaining MDDs are not evaluated, (since the multi-valued functions in VIS are deterministic). If the result is a “0”, the $i+1^{th}$ MDD is checked. If $n-1$ MDDs return a “0”, then the simulation output is n. (This is because the multi-valued function is fully specified.)

Since an MV-network is fully represented in VIS, and the VIS system allows the use of a popular RTL to specify such networks, we have an excellent opportunity in VIS to create a multi-valued optimization package. Further, VIS allows and keeps hierarchy, so synthesis using hierarchy is enabled. However, at this point, direct synthesis inside VIS has not been developed since our first efforts were to take advantage of the SIS system. The idea is that by converting all signals into their *one-hot* (or even logarithmic encoded binary versions), we can experiment and make use of the extensive developments in SIS for binary optimizations. However, this has proved more difficult that we had first estimated and perhaps it is time to bite the bullet and do the full development inside VIS.

5. State Assignment

As mentioned earlier, an alternative way of optimizing a multi-valued logic function is to first do the manipulations in the multi-valued domain, independent of any encoding, and then to use the resulting structure to intelligently find a good encoding. Perhaps the most successful example of this is the KISS approach for state assignment of finite state machines [1]. Here the state variable is multi-valued.

Consider all next state functions, one for each state value, as many separate binary valued functions of one MV-variable and perhaps several binary valued variables. The approach taken in KISS is to minimize this set of functions with ESPRESSO-II resulting in a minimized cover of MV-cubes. Consider one such cube. In its state variable position is a MV-literal, which is a set of values. Each such cube in the cover gives such a set. The main idea in KISS is that if it is possible to encode the state variable in such a way that each set of values associated with any cube in the resulting minimized SOP cover can be also described as a cube in the space of binary encoding variables, then each MV-cube can be replaced by the binary counterpart, and the size of the cover is not increased.

This embedding of sets into faces of the cube of the encoding variables is called the *face embedding problem*; given a set of sets of points, encode each set with binary variables so that each is precisely contained in a cube in the binary space. This is always possible if enough binary variables are used, so a side constraint is to use a small or minimum number of binary variables. The above procedure is known as the input encoding problem. Note that we treated the next state function as separate binary functions. Thus the fact that the next state variables will also be encoded was ignored. Once the next state output functions are replaced by the derived codes used for the state variables, then more optimization is possible because more cubes can be combined due to sharing of the outputs.

This procedure has been extended in a program called NOVA [10] to consider both the input and output encoding of the state variables. The procedure works well for small state machines, say less than 30 states, but is ineffective for large machines, say more than 50 states. There are examples where an encoding given by the designer, possibly derived from some knowledge about the structure of the problem, leads to a much smaller implementation of the logic than an implementation derived using NOVA. One speculation is that a better encoding could be obtained by decomposing the machine into a product of smaller machines and applying NOVA to the small machines. The encoding obtained by concatenating the codes of the small machines is an encoding of the large machine. Thus a first step in the multi-valued domain would be the decomposition of the machine. Unfortunately, we do not know of any really effective way to do this and this is an area for potentially fruitful (but probably difficult) research.

6. Conclusions and Open Problems

We have surveyed two-level and multi-level logic optimizations for MV-variables. We discussed three methods for representing MV-functions (SOPs, MV-networks, and MDDs). One-hot encoding represents a way to keep the multi-valued structure, but to use binary operations for the manipulations; however this is not always successful. Conceptually, the best way to deal with MV-logic is direct manipulation and optimization followed by intelligent encoding, followed by binary optimizations. An effective package for this remains a challenge for the future. Although, as we have seen in this paper, many of the concepts necessary for suvch a package have been developed, efficient algorithms for their effective use in such a package are still missing. The VIS system represents a framework for future developments in this direction, but a significant amount of effort and research remains to be done.

Acknowledgements

We're grateful for financial support for this research provided by the SRC under contract 98-DC-324, and for the support of the California Micro program and participating industrial sponsors, Fujitsu, Motorola, Cadence, Synopsys, Intel, and Metamor, Inc.

References

[1] G. D. Micheli, R. Brayton, and A. Sangiovanni-Vincentelli, "KISS: a program for optimal state assignment of finite state machines," in *Proc. of the Intl. Conf. on Computer-Aided Design*, Nov. 1984.

[2] S. Malik, A. Srinivasan, T. Kam, and R. Brayton, "Algorithms for discrete function manipulation," in *Proceedings of the International Conference on Computer-Aided Design*, 1990.

[3] M. Iyer, D. Long, and M. Abramovici, "Identifying sequential redundancies without search," in *Proceedings of the 33rd Design Automation Conference*, 1996.

[4] A. Mehrotra, S. Qadeer, V. Singhal, R. Brayton, A. Aziz, and A. Sangiovanni-Vincentelli, "Sequential optimization without state space search," in *Proceedings of the International Conference on Computer-Aided Design*, 1997.

[5] S. Khatri, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential multi-valued network simplification using redundancy removal," in *to appear in Proceedings of the International Conference on VLSI Design*, 1999.

[6] 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.

[7] S. Yamashita, H. Sawada, and A. Nagoya, "A new method to express functional permissibilities for LUT based FPGAs and its applications," in *Proceedings of the International Conference on Computer-Aided Design*, 1996.

[8] R. Brayton, "Understanding SPFDs: A new method for specifying flexibility," in *Workshop Notes, International Workshop on Logic Synthesis*, 1997.

[9] J. Brzozowski and J. Lou, "Blanket albebra for multiple-valued function decomposition," in *Proceedings of the International Workshop on Formal Languages and Computer Systems*, 1997.

[10] T. Villa and A. L. Sangiovanni-Vincentelli, "NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementations," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 905–924, Sept. 1990.