[52] Y. Watanabe and R. Brayton, "State minimization of pseudo non-deterministic FSM's," in *Proceedings of the European Conference on Design Automation*, 1994.

[53] 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 IEEE International Conference on Computer-Aided Design*, pp. 254–261, Nov. 1996.

Chapter 4

MULTIPLE-VALUED LOGIC SYNTHESIS AND OPTIMIZATION

Elena Dubrova

Abstract Some Boolean logic problems can be solved more efficiently in multiple-valued domain. This chapter covers a part of the theory of multiple-valued logic related to applications in CAD. Basic methods for representation and optimization of multiple-valued functions are described.

4.1 INTRODUCTION

Multiple-valued logic is a generalization of classical Boolean logic. One reason for considering generalizations is that it can lead to deeper understanding of the specialized problem. To generalize a structure, some of its properties must be dropped and some must be preserved. Thus, an attempt to generalize helps distinguish non-essential properties from essential ones.

Another reason for studying multiple-valued logic is that it can help solve some Boolean problems more efficiently. For example, a well-known approach to represent a multiple-output Boolean function is to treat its output part as a single multiple-valued variable and convert it to a single-output characteristic function. Such an approach is used in ESPRESSO-MV [45] and in MVSIS [19]. Other applications of multiple-valued logic include design of PLAs with input decoders [47], optimization of finite state machines [36], testing [24] and verification [14].

Multiple-valued logic also provides a theoretical base for designing electronic circuits with more than two logic levels, such as three- and four-valued PLAs and memories. Multiple-valued circuits have a number of theoretical advantages over standard binary circuits. For example, the interconnect on and off chip can be reduced if signals in the circuit assume four or more levels rather than only two [2]. In memory design, storing two instead of one bit of information per memory cell doubles the density of the memory in the same die size [40]. Applications using arithmetic circuits often benefit from using alternatives to binary number systems. For example, residue and redundant

number systems can reduce or eliminate the ripple-through carries which are involved in normal binary addition or subtraction, resulting in high-speed arithmetic operations. These number systems have a natural implementation using multiple-valued circuits [21, 49]. However, the practicality of these potential advantages heavily depends on the availability of circuit realizations, which must be compatible or competitive with present-day standard technologies.

The purpose of this chapter is to stimulate the reader's interest beyond the binary case. We first cover a part of the theory of multiple-valued logic related to applications in CAD: multiple-valued functions, functionally complete sets and multiple-valued algebraic systems. We then look at several possibilities for representing multiple-valued functions: positional cube notation, multiple-valued networks and decision diagrams. We give a review of basic methods for two- and multi-level optimization of multiple-valued functions. Finally, we conclude with a discussion of some open problems.

4.2 MULTIPLE-VALUED FUNCTIONS

A multiple-valued function is a discrete function whose input and output variables take two or more values. Multiple-valued functions are often defined as $f : P_1 \times P_2 \times \ldots P_n \rightarrow Q$, where the variables x_i of the function take values from the sets $P_i = \{0, 1, 2, \ldots, p_i - 1\}$, $p_i > 1$, $i \in \{1, 2, \ldots, n\}$ and the function takes values from the set $Q = \{0, 1, 2, \ldots, q\}$, $q > 1$. However, in this chapter it is neither necessary nor particularly useful for us to distinguish between the sets P_i and Q. To simplify the notation and make the presentation more clear, we use the following definition.

Definition 4.2.1 *An n-variable m-valued function is a mapping $f : M^n \rightarrow M$ over a finite set of values $M = \{0, 1, 2, \ldots, m - 1\}$.*

Using this simpler definition does not affect the generality of the results presented in this chapter[1], because any function of type $g : P_1 \times P_2 \times \ldots P_n \rightarrow Q$ can be represented as an incompletely specified function of type $f : M^n \rightarrow M \cup \{-\}$, defined as follows: $f = g$ for all $(x_1, \ldots, x_n) \in P_1 \times P_2 \times \ldots P_n$ and $f =$ " $-$ " for all $(x_1, \ldots, x_n) \in M^n - (P_1 \times P_2 \times \ldots P_n)$, where $M = (\cup_{i=1}^{n} P_i) \cup Q$. For example, a function $g : \{0, 1, 2\} \times \{0, 1\} \rightarrow \{0, 1\}$ specified in Figure 4.1 can be represented by the function $f : \{0, 1, 2\} \times \{0, 1, 2\} \rightarrow \{0, 1, 2, -\}$ shown in Figure 4.2. Note, that the definition $f : M^n \rightarrow M$ does not to require the output of the function to range over *all* values of M.

Multiple-valued functions have been originally studied to provide support for designing m-valued logic circuits. Such circuits are assumed to employ the signals which vary over a final set of m discrete values, $m \geq 2$. Later, techniques for manipulating multiple-valued functions have been used to design binary-valued circuits that implement some encoding of the multiple values. Examples are synthesis of PLAs with input encoders [47] and optimization of finite state machines [36]. Most of these applications employ characteristic functions of type $f : M^n \rightarrow \{0, 1\}$, with the function output ranging over a binary set. Throughout the chapter, we refer to such functions as *multiple-valued input binary-valued output* functions.

[1] Efficient implementation of algorithms may well keep these domains distinct.

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

Figure 4.1. An example of function of type $\{0, 1, 2\} \times \{0, 1\} \rightarrow \{0, 1\}$.

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

Figure 4.2. Function from Fig.4.1 represented as $\{0, 1, 2\} \times \{0, 1, 2\} \rightarrow \{0, 1, 2, -\}$.

4.3 FUNCTIONAL COMPLETENESS

In this section we explore the concept of a set of functions being complete. Completeness is essential for any system which is to be used as a basis for practical logic design. Once a complete set of functions is identified, any logic circuit can be constructed from the gates implementing the primitive functions from this set.

4.3.1 COMPLETE SETS OF FUNCTIONS

Definition 4.3.1 *A set A of functions over M is complete if it is possible to define any function over M as a composition of functions from A.*

The set $A = \{\text{AND, OR, NOT}\}$ is an example of a complete set for the Boolean functions $\{0, 1\}^n \rightarrow \{0, 1\}$. A proof of this requires the definition of each function $\{0, 1\}^n \rightarrow \{0, 1\}$ as a composition of AND, OR and NOT. First, we define constants as $0 = x \cdot x'$ and $1 = x + x'$. Then, by applying Shannon decomposition

$$f(x_1, \ldots, x_n) = x_i' \cdot f|_{x_i=0} + x_i \cdot f|_{x_i=1}$$

we can derive sum-of-products expressions for all other n-variable Boolean functions.

Another example of a complete set for the Boolean functions is the set $A = \{\text{AND, NOT}\}$. This follows from de Morgan's laws, since $x + y = (x' \cdot y')'$, so all occurrences of OR in the sum-of-products expression of any Boolean function can be replaced by an expression using AND and NOT.

The set M of m elements together with the set A of functions over M and two distinct elements $\mathbf{0}, \mathbf{1}$ in M generate a *finite algebra of degree* m $\langle M; A; \mathbf{0}, \mathbf{1}\rangle$. For example, the set $B = \{0, 1\}$ with the functions AND, OR and NOT generate the Boolean algebra $\langle B; +, \cdot, '; 0, 1\rangle$ of degree 2. The constants $\mathbf{0}$ and $\mathbf{1}$ are two particular elements of M which have special properties. They are called the *zero* and the *unit* of the algebra, respectively.

We say that and a given algebra is *functionally complete* if it is based on a complete set of functions. For example, Boolean algebra is functionally complete for the Boolean functions $\{0, 1\}^n \rightarrow \{0, 1\}$. However, if the set B takes more than two values, then $A = \{\text{AND, OR, NOT}\}$ is not complete. For example, it not possible to express the 3-valued 1-variable function $f(x)$ shown in Figure 4.3 as a composition of AND, OR and NOT.

x	$f(x)$
0	2
1	0
2	0

Figure 4.3. A function which cannot be expressed using AND, OR and NOT.

In the next section we show how the operations AND, OR and NOT have to be extended to yield a complete set of functions over M.

4.3.2 COMPLETENESS FOR MULTIPLE-VALUED INPUT BINARY-VALUED OUTPUT FUNCTIONS

Let us first consider the case of multiple-valued input binary-valued output functions. These functions are a "nice" subset of the functions $M^n \rightarrow M$. Many notions and algorithms for the Boolean functions trivially extend to the case of $M^n \rightarrow \{0, 1\}$. We show below that Boolean AND and OR can still be used. Only NOT needs to be extended to a new unary operation, which is able to distinguish between the m values of M. Such a unary operation can be defined as follows.

Definition 4.3.2 *A literal of a multiple-valued variable x is a unary operation defined by*

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

where $S \subseteq M$.

If $m = 2$ is substituted into the definition of literal, then we get $x^{\{1\}} = x$ and $x^{\{0\}} = x'$. So, literal is indeed a generalization of x'. To simplify the exposition, we omit brackets when S is a single-element set, i.e. we write x^1 instead of $x^{\{1\}}$.

As we mentioned before, $\mathbf{0}$ and $\mathbf{1}$ denote two designated constants of M. We introduce them to have a uniform definition of literal for both cases: $M^n \rightarrow \{0, 1\}$ and $M^n \rightarrow M$. Throughout the chapter, we will be using $\mathbf{0} = 0$ and $\mathbf{1} = 1$ when considering multiple-valued input binary-valued output functions, and $\mathbf{0} = 0$ and $\mathbf{1} = m - 1$ for the general case of $M^n \rightarrow M$.

Note that the literal is a characteristic function of type $M \rightarrow \{0, 1\}$. It implies that, for multiple-valued input binary-valued output functions, the operations on the literals are Boolean operations of type $\{0, 1\}^n \rightarrow \{0, 1\}$. For example, a sum of two literals $x^i + x^j$ or a product of two literals $x^i \cdot x^j$ are binary operations of type $\{0, 1\}^2 \rightarrow \{0, 1\}$. Therefore, we can define the sum "+" as OR and the product "$\cdot$" as AND and represent multiple-valued input binary-valued output functions as sum-of-products of literals. To prove that these three operations are sufficient to define all functions of type $M^n \rightarrow \{0, 1\}$, we next show how an arbitrary function $M^n \rightarrow \{0, 1\}$ can be expressed in terms of AND, OR and literals.

First, we observe that the constants 0 and 1 can be defined as $0 = \prod_{j \in M} x_i^j$ and $1 = \sum_{j \in M} x_i^j$, for any $i \in \{1, 2, \ldots, n\}$. Next, we extend the Shannon decomposition theorem to the multiple-valued input binary-valued output functions as follows.

Theorem 4.3.1 *Every multiple-valued input binary-valued output functions $M^n \rightarrow \{0, 1\}$ can be decomposed with respect to a variable x_i, $i \in M$, as*

$$f(x_1, \ldots, x_n) = \sum_{j=0}^{m-1} x_i^j \cdot f|_{x_i = j}$$

where $f|_{x_i = j} = f(x_1, \ldots, x_{i-1}, j, x_{i+1}, \ldots, x_n)$ are co-factors of f with respect to x_i, $\sum$ stands for OR and "$\cdot$" stands for AND.

By subsequently applying Theorem 4.3.1 we can decompose an n-variable function with respect to all its variables and derive the following canonical sum-of-products expression. Let "+" = OR, "$\cdot$" = AND and $L = \{x^0, x^1, \ldots, x^{m-1}\}$ be the set of literals specified by Definition 4.3.2 with $\mathbf{0} = 0$ and $\mathbf{1} = 1$.

Theorem 4.3.2 *Every multiple-valued input binary-valued output functions $M^n \rightarrow \{0, 1\}$ has a canonical expression in terms of $\{+, \cdot, L\}$ of type*

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{m^n - 1} c_i \, x_1^{i_1} \, x_2^{i_2} \ldots x_n^{i_n}$$

where $c_i \in \{0,1\}$ are binary constants, and $(i_1 i_2 \ldots i_n)$ is the m-ary expansion of i with i_1 being the least significant digit.

For example, the function $f : \{0,1,2\}^2 \to \{0,1\}$ shown in Figure 4.4 has the following canonical form:

$$f(x_1, x_2) = x_1^0 x_2^0 + x_1^1 x_2^0 + x_1^0 x_2^1 + x_1^2 x_2^1 + x_1^1 x_2^2 + x_1^2 x_2^2.$$

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

Figure 4.4. An example function.

The following result follows directly from Theorem 4.3.2.

Theorem 4.3.3 *The algebra $\langle M; +, \cdot, L; \mathbf{0}, \mathbf{1} \rangle$, where "+" is OR, "$\cdot$" is AND, $L = \{x^0, x^1, \ldots, x^{m-1}\}$ is the set of literals, $\mathbf{0} = 0$ and $\mathbf{1} = 1$, is functionally complete for multiple-valued input binary-valued output functions $M^n \to \{0,1\}$.*

Theorems 4.3.1, 4.3.2 and 4.3.3 are special cases of Theorems 4.3.4, 4.3.5 and 4.3.6, respectively, presented in the next section. Formal proofs can be found in [38].

4.3.3 COMPLETENESS FOR GENERAL CASE

The problem of completeness of a set of functions have been intensively studied by mathematicians. The classic result for multiple-valued functions, giving a characterization of the necessary and sufficient conditions for a set of functions to be complete is due to Rosenberg [49].

In the general case of multiple-valued functions $M^n \to M$, not only NOT but also AND and OR operations have to be extended. It was proved in [38] that, to be able to generalize the Shannon decomposition theorem to the multiple-valued case, the sum- and product-type operations have to be specified in the following way.

Definition 4.3.3 *A binary operation "$\cdot$" over M is a product-type operation if and only if, for all $x \in M$, $\mathbf{0} \cdot x = x \cdot \mathbf{0} = \mathbf{0}$ and $\mathbf{1} \cdot x = x \cdot \mathbf{1} = x$.*

Definition 4.3.4 *A binary operation "+" over M is a sum-type operation if and only if, for all $x \in M$, $\mathbf{0} + x = x + \mathbf{0} = x$.*

For example, the general form of the sum- and product-type operations for $m = 3$ is given in Figure 4.5, the dashes in the tables indicating that any of the three values may occur in that position. Note, that $\mathbf{0} = 0$ and $\mathbf{1} = 2$.

MAX

$x + y$	0	1	2
0	0	1	2
1	1	-	-
2	2	-	-

MIN

$x \cdot y$	0	1	2
0	0	0	0
1	0	-	1
2	0	1	2

Figure 4.5. Specification tables for sum- and product-type operations for $m = 3$.

If sum- and product-type operations are defined as above, then the Shannon decomposition theorem extends to the multiple-valued case as follows.

Theorem 4.3.4 *Every multiple-valued function $M^n \to M$ can be decomposed with respect to a variable x_i, $i \in M$, as*

$$f(x_1, \ldots, x_n) = \sum_{j=0}^{m-1} x_i^j \cdot f|_{x_i = j}$$

where $f|_{x_i=j} = f(x_1, \ldots, x_{i-1}, j, x_{i+1}, \ldots, x_n)$ are co-factors of f with respect to x_i, $\sum$ is for a sum-type operation and "$\cdot$" is for a product-type operation.

By subsequently applying Theorem 4.3.4 to decompose an n-variable function with respect to all its variables, we can derive the following canonical sum-of-products expression. Let "+" be a sum-type operation specified by Definition 4.3.4, "$\cdot$" product-type operation specified by Definition 4.3.3 and $L = \{x^0, x^1, \ldots, x^{m-1}\}$ be the set of literals given by Definition 4.3.2 with $\mathbf{0} = 0$ and $\mathbf{1} = m - 1$.

Theorem 4.3.5 *Every multiple-valued function $M^n \to M$ has a unique expression in terms of $\{\cdot, +, L\}$ of type*

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{m^n - 1} c_i \cdot x_1^{i_1} \cdot x_2^{i_2} \cdot \ldots \cdot x_n^{i_n}$$

where $c_i \in M$ are constants, and $(i_1 i_2 \ldots i_n)$ is the m-ary expansion of i with i_1 being the least significant digit.

As shown in Section 4.3.2, in the case of multiple-valued input binary-valued output functions we can express the constants 0 and 1 in terms of AND, OR and literals. It is not possible, however, to express all m constants of M

in terms of the operations $\{\cdot, +, L\}$ considered in this section. Therefore, we need to include all the constant functions as primitive functions to make the set functionally complete.

Theorem 4.3.6 *The algebra $\langle M; +, \cdot, L, C; \mathbf{0}, \mathbf{1}\rangle$, where "+" and "$\cdot$" are a sum- and product-type operations, respectively, $L = \{x^0, x^1, \ldots, x^{m-1}\}$ is the set of literals, C is a set of m constant functions, $\mathbf{0} = 0$ and $\mathbf{1} = m - 1$, is functionally complete for multiple-valued functions $M^n \rightarrow M$.*

Formal proofs of Theorems 4.3.4, 4.3.5 and 4.3.6 can be found in [38].

The conditions given by Definitions 4.3.3 and 4.3.4 are not *necessary* conditions for completeness. There are functionally complete algebras whose sum- and product-type of operations do not obey these definitions. An example is the algebra employing an underlying structure Galois field $GF(m)$, with operations addition modulo m and multiplication modulo m, where m is a power of a prime [48].

4.4 CHAIN-BASED POST ALGEBRA

In this section we present *chain-based Post algebra*, which is based on a single totally ordered chain $0 < 1 < \ldots < m - 1$ and use operations maximum, minimum and literal. This is a special case of more general Post algebras, based on a lattice [38]. Maximum and minimum operations satisfy Definitions 4.3.3 and 4.3.4 for sum- and product-type of functions. Therefore, Theorem 4.3.6 holds, stating functional completeness of chain-based Post algebra for multiple-valued functions $M^n \rightarrow M$.

Definition 4.4.1 *Chain-based Post algebra is an algebra $\langle M; +, \cdot, L; \mathbf{0}, \mathbf{1}\rangle$, such that the elements of M form a totally ordered chain $\mathbf{0} = 0 < 1 < \ldots < m - 1 = \mathbf{1}$, " + " and " $\cdot$ " are the binary operations maximum (MAX) and minimum (MIN), respectively, and $L = \{x^0, x^1, \ldots, x^{m-1}\}$ is the set of literals specified by Definition 4.3.2.*

Any multiple-valued function has a canonical expression in chain-based Post algebra in terms of MIN, MAX and literals of the form given by Theorem 4.3.5. For example, the 2-variable 3-valued function shown in Figure 4.6 has the following canonical form:

$$f(x_1, x_2) = 1 \cdot x_1^1 \cdot x_2^0 + 1 \cdot x_1^1 \cdot x_2^2 + 2 \cdot x_1^0 \cdot x_2^1 + 2 \cdot x_1^1 \cdot x_2^1 + 2 \cdot x_1^2 \cdot x_2^1.$$

The canonical form can be simplified by applying the following properties of the literals (proved in [38]) .

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

Figure 4.6. An example function.

Property 4.4.1 *The following properties of literals hold:*

(1) $x^i \cdot x^j = \mathbf{0}$, *for any* $i, j \in M, i \neq j$.

(2) $\sum_{i=0}^{m-1} x^i = \mathbf{1}$.

For example, the function in Figure 4.6 can be represented as

$$f(x_1, x_2) = 1 \cdot x_1^1 \cdot x_2^0 + 1 \cdot x_1^1 \cdot x_2^2 + 2 \cdot x_2^1$$

with "+" = MAX and "$\cdot$" = MIN. Further, due to the property of MAX $a + b = b$ for any $a \geq b$, $a, b \in M$, the following rule can also be used simplification [38].

Property 4.4.2 *Let a and b be constants in M such that $a \leq b$. Then*

$$a \cdot x_1^i + b \cdot x_2^j = a \cdot (x_1^i + x_2^j) + b \cdot x_2^j.$$

Using the above property, the function in Figure 4.6 can be reduced to

$$f(x_1, x_2) = 1 \cdot x_1^1 + 2 \cdot x_2^1.$$

We can also omit the constant 2 from product-terms, since by Definition 4.3.3, $2 = m - 1$ is the unit with respect to MIN.

Finally, we consider an alternative to Theorem 4.3.4 which provides another way to decompose a multiple-valued function. This will be used later in Sections 4.5 and 4.6. The operation literal over the function, used below, is defined in the same way as the literal over a variable, namely

$$f^i(x_1, \ldots, x_n) = \begin{cases} \mathbf{1} & \text{if } f(x_1, \ldots, x_n) = i \\ \mathbf{0} & \text{otherwise} \end{cases}$$

where $i \in M$ is a constant.

Theorem 4.4.1 *Every multiple-valued function $M^n \rightarrow M$ can be expressed as*

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{m-1} i \cdot f^i(x_1, \ldots, x_n)$$

where $f^i : M^n \rightarrow \{0,1\}$ *are literals of* f.

The above decomposition splits a function $f : M^n \rightarrow M$ with respect to each of its values $i \in M$ into $m-1$ multiple-valued input binary-valued output functions $f^i(x_1, \ldots, x_n)$. Since $f^i \cdot f^j = 0$ for any $i \neq j, i, j \in M$, the functions $f^i(x_1, \ldots, x_n)$, $i \in M$, partition the domain M^n of f into m disjoint sets.

For example, the function in Figure 4.6 can be decomposed as $f(x_1, x_2) = 1 \cdot f^1 + f^2$ with the functions f^1 and f^2 given by $f^1 = x_1^1 \cdot x_2^{\{0,2\}}$ and $f^2 = x_2^1$.

In the next section we consider how multiple-valued functions can be represented in a way convenient for computer implementation of algorithms.

4.5 REPRESENTATIONS OF MULTIPLE-VALUED FUNCTIONS

The simplest way to represent an m-valued n-variable function is by giving a *truth table* containing m^n rows, each specifying the value of the function for the corresponding values of the variables. A truth table can also be re-arranged in a rectangular n-dimensional m-valued Karnaugh map. However, both tables and maps are too large to be feasible for larger functions. In this section we consider more practical ways of representing multiple-valued functions: positional cube notation, multiple-valued networks and multiple-valued decision diagrams.

4.5.1 POSITIONAL CUBE NOTATION

Positional cube notation used for the Boolean functions can be easily extended to the multiple-valued input binary-valued output functions [45]. While a binary-valued literal, x or x', is represented by a 2-bit field, an m-valued literal can be represented by an m-bit field. Value $i \in M$ corresponds to the bit i of the field. A literal x^S has 1 in all positions $i \in S$. For example, positional cube notation for the function in Figure 4.4 is:

$$110 - 100$$
$$101 - 010$$
$$011 - 001$$

For the Boolean case, a don't care condition corresponds to the encoding of the field 11 and means either 0 or 1. For m-valued input binary-valued output functions, a don't care condition means any of m values $i \in M$, and therefore corresponds to the m-bit field of all 1s.

General multiple-valued functions can be represented in positional cube notation by first splitting a function $f : M^n \rightarrow M$ with respect to each of its values $i \in M$ into m multiple-valued input binary-valued output literals $f^i : M^n \rightarrow \{0,1\}$, and then representing each of these functions in positional cube notation. Since $f^i \cdot f^j = 0$ for any $i \neq j, i, j \in M$, the literals $f^i, i \in M$, partition the domain M^n of f into m disjoint sets. For $m = 2$, f^0 corresponds to the off-set and f^1 corresponds to the on-set of f.

4.5.2 MULTIPLE-VALUED NETWORKS

A multiple-valued network is a multi-level graph-based representation similar to a Boolean network except that each node is, in general, a multiple-valued function [29]. At high and system levels of abstraction, where the variables often range over a set of symbolic values, use of multiple-valued networks can make the design task more intuitive [3]. For example, it is easier to deal with a traffic light controller with a signal *light* taking the values *red, yellow* and *green* rather than consider the encoding $light_1$, $light_2$ to stand for the light being green. The designer can first manipulate and optimize a multiple-valued network, and then, when no further optimization is possible in the multiple-valued domain, perform a suitable encoding and derive the resulting Boolean network. This allows a better exploration of the design space, since the decision about the encoding is postponed and multiple-valued optimization is not influenced by this decision. At the final stage, several different binary encodings of a multiple-valued network might be tried to select a good one. The problem of minimization of multi-valued networks has been addressed in [25].

An example of a tool whose input takes a multiple-valued network is Verification Interacting with Synthesis (VIS) [53]. It has a Verilog front end, which generates a *blif-mv* description of a multiple-valued network. Part of the VIS system is a Verilog translator which supports a multiple-valued extension to Verilog. 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 type. Later,

```
signal light color
```

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

A drawback of network representation is that it is not canonical, i.e. a given function may have many different networks representing it. Consequently, testing for functional equivalence or satisfiability can be quite difficult. This problem is eliminated in the representation described in the next chapter.

4.5.3 MDDS

Multiple-valued Decision Diagrams (MDDs) are a straightforward generalization of Binary Decision Diagrams (BDDs) [26]. Functions are represented by directed, acyclic graphs with nonterminal vertices v labeled by a variable index $index(i), i \in \{1, 2, \ldots, n\}$. Each vertex v has m out-coming edges directed towards children vertices, denoted by $child_j(v), j \in M$. Each terminal vertex v has as attribute a value $value(v) \in M$. To obtain a reduced ordered decision diagrams, the conditions for ordering of variables and reduction of graphs, similar to the Boolean case, apply. An example of a 3-valued decision diagram, implementing a function from Figure 4.6 is shown in Figure 4.7.

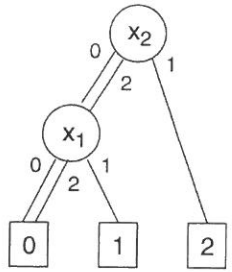

Figure 4.7. MDD for the example function.

If v is a non-terminal vertex with $index(v) = i, i \in \{1, 2, \ldots, n\}$, then the function represented by node v, f_v, is given by

$$f_v(x_1, \ldots, x_n) = \sum_{j \in M} x_i^j \cdot f_{child_j(v)}(x_1, \ldots, x_n).$$

Similarly to the Boolean case, the value for the function for a given assignment of the variables is determined by tracing the path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The function value is then given by the terminal vertex at the end of the path. For example, the function $f(x_1, x_2)$ represented by the MDD in Figure 4.7 evaluates to 1 for the input assignment $x_2 = 2, x_1 = 1$.

MDDs inherit all the strong features of BDDs: canonicity, easy manipulation algorithms and compactness for many functions. They also inherit the variable ordering problem. For example, the MDD in Figure 4.7 would have twice more non-terminal nodes if the variables were ordered as $\langle x_1, x_2 \rangle$.

An extended survey of MDDs and their extensions can be found in [46].

4.6 TWO-LEVEL LOGIC OPTIMIZATION

The objective of two-level multiple-valued logic optimization is to reduce the size of the sum-of-products expression (SOP) representing a multiple-valued function. Transformations of two-level forms into multiple-level forms will be described in the next chapter.

4.6.1 MINIMIZATION OF SOPS OF MULTIPLE-VALUED INPUT BINARY-VALUED OUTPUT FUNCTIONS

For multiple-valued input binary-valued output functions, most of the Boolean logic minimization theory generalizes easily [45]. A *cube* is a product of literals in the form $x_1^{S_1} x_2^{S_2} \ldots x_n^{S_n}$. The *on-set*, the *don't care-set* and the *off-set* of f are the sets of minterms that are mapped by f to 1, – and 0, respectively. The Boolean definitions of implicant, prime implicant, essential implicant, cover, prime and irredundant cover remain unchanged. The notions of cofactors is extended as follows.

Definition 4.6.1 *The cofactor of a cube $S = x_1^{S_1} x_2^{S_2} \ldots x_n^{S_n}$ with respect to a cube $T = x_1^{T_1} x_2^{T_2} \ldots x_n^{T_n}$ is empty if S and T are disjoint. Otherwise, it is the product $x_1^{S_1 \cup \overline{T}_1} x_2^{S_2 \cup \overline{T}_2} \ldots x_n^{S_n \cup \overline{T}_n}$.*

Definition 4.6.2 *The cofactor of a function $f : M^n \rightarrow \{0, 1\}$ with respect to a cube S, denoted by $f|_S$, is the sum of the cofactors of each cube of F with respect to S.*

As an example, consider the function $f : \{0, 1, 2\}^2 \rightarrow \{0, 1\}$ from Figure 4.4. For this case, $x_1^{\{0,1\}} x_2^{\{0,1\}}$ is a cube which is *not* an implicant of the function. $x_1^{\{0,1\}} x_2^0$ is a prime implicant. None of the implicants is essential. $x_1^0 x_2^0 + x_1^2 x_2^0 + x_1^0 x_2^1 + x_1^2 x_2^1 + x_1^1 x_2^2 + x_1^2 x_2^2$ is a cover which is not prime and $x_1^{\{0,1\}} x_2^0 + x_1^{\{0,2\}} x_2^1 + x_1^{\{1,2\}} x_2^2$ is a prime cover. The cofactor of a cube $x_1^{\{0,1\}} x_2^0$ with respect to the cube $x_1^0 x_2^{\{0,1\}}$ is $x_1^{\{0,1,2\}} x_2^{\{0,2\}} = x_2^{\{0,2\}}$. The cofactor of the cover $x_1^{\{0,1\}} x_2^0 + x_1^{\{0,2\}} x_2^1 + x_1^{\{1,2\}} x_2^2$ with respect to the cube $x_1^0 x_2^{\{0,1\}}$ is $x_2^{\{0,2\}} + x_2^{\{1,2\}} = 1$.

As in the Boolean case, the process of logic minimization involves generating prime implicants, generating a covering table and solving this covering table. A powerful package for two-level minimization is ESPRESSO-MV developed at Berkeley. Below, we briefly review the concept of ESPRESSO-MV. See [45] for an extended discussion of these ideas.

The basic paradigm for manipulating multiple-valued functions in ESPRESSO-MV is to use the multiple-valued extension of the Shannon decomposition

theorem to recursively split the functions into smaller ones until each of the smaller functions becomes unate. A simple test is then performed on the unate function to quickly determine the result. Finally, the results of each branch of the recursion are merged to produce the answer to the original problem. Generation of prime implicants and the Boolean routines REDUCE, IRREDUNDANT and ESSENTIAL remain basically unchanged. The interesting cases are extensions of Shannon decomposition theorem and unateness property.

Shannon decomposition theorem is extended in the following way.

Theorem 4.6.1 *Let f be a multiple-valued input binary-valued output function and $\{c_1, c_2, \ldots, c_k\}$ be a set of cubes satisfying $\sum_{i=1}^{k} c_i = 1$ and $c_i \cdot c_j = \emptyset$ for all $i \neq j$, $i, j \in \{1, 2, \ldots, k\}$. Then,*

$$f = \sum_{i=1}^{k} c_i \cdot f|_{c_i}$$

Using Theorem 4.6.1, it is easy to show that the operations of tautology and complementation can be computed using the properties:

$$f \equiv 1 \text{ iff } f|_{c_i} \equiv 1 \text{ for each } i \in \{1, 2, \ldots, k\}$$

$$f' = \sum_{i=1}^{k} c_i \cdot f'|_{c_i}$$

It is not immediately obvious how to define unateness for multiple-valued functions. Two extensions have been presented in [45]: *weak unateness* and *strong unateness*. For the Boolean case these two concepts coincide.

Definition 4.6.3 *A function is weakly unate in variable x_i if there exists a $j \in M$ such that changing the value of x_i from j to any other value in M causes the function value, if it changes, to change from 0 to 1. A function is weakly unate if it is weakly unate in all its variables.*

A simple test for whether a function is weakly unate in a variable x_i is to form the supercube of all cubes of f which do not include a literal of type $x_i^{S_i}$ with $S_i = M$. In positional cube notation, if this supercube has a 0 in any positions of x_i, then f is weakly unate in x_i. For example, the function $f : \{0, 1, 2, 3, 4\}^3 \rightarrow \{0, 1\}$ given by the following positional cube notation:

11111 – 00001 – 11110
01100 – 00011 – 01010
01010 – 00100 – 11111
00110 – 01001 – 11010
00001 – 11111 – 10110

is weakly unate because it is weakly unate in position 1 of variable 1, position 1 of variable 2, and position 5 of variable 3. The *unate reduction* theorem for tautology can be applied in the multiple-valued case for weakly unate variables.

Definition 4.6.4 *A function is strongly unate in variable x_i if the values of x_i can be totally ordered via "$\leq$" such that changing the value of x_i from j to k, $j \leq k$, $j, k \in M$, causes the function value, if it changes, to change from 0 to 1. A function is strongly unate if it is strongly unate in all its variables.*

A function strongly unate in x_i provides a total order for all of the parts of x_i, meanwhile a weakly unate function merely provides an order for a single part which is less than all remaining parts. It is proved in [45] that all primes of a strongly unate function are essential and that the complement of a strongly unate function is strongly unate. These results do not hold for weakly unate functions.

Based on the above, ESPRESSO-MV handles multiple-valued input binary-valued output functions. An interesting observation from [3] is that when ESPRESSO-MV was completed and compared to the original ESPRESSO in terms of the results for purely Boolean functions, ESPRESSO-MV was faster. An explanation is that the generalization to the multiple-valued logic led to a superior method of representation of the functions for computer manipulation.

4.6.2 MINIMIZATION OF SOP EXPRESSIONS OF GENERAL MULTIPLE-VALUED FUNCTIONS

Two-level sum-of-products for multiple-valued functions and associated minimization techniques have been studied since the late sixties [8, 34, 37, 51]. The algorithms for minimization of Boolean sum-of-products expressions, beginning with the generation of all prime implicants followed by the selection of a minimum cover, can be extended to the multiple-valued expressions over Post algebra, but it is quite inefficient. One of the difficulties is that, because of the MAX operation between cubes, a cube with a higher-valued coefficient $b \in M$ may be combined with a cubes with a lower-valued coefficient $a \in M$ as shown in Property 4.4.2. On one side, it gives a great flexibility; on the other side, it results in an explosion of choices for a cover.

A more efficient approach is to split the function $f : M^n \rightarrow M$ with respect to each of its non-zero values $i \in M - \{0\}$ into $m - 1$ multiple-valued input binary-valued output literals $f^i : M^n \rightarrow \{\mathbf{0}, \mathbf{1}\}$, and to apply an efficient minimizer for multiple-valued input binary-valued output functions [9, 10]. The functions f^i are minimized successively for $i = m - 1, m - 2, \ldots, 1$. By Property 4.4.2, the on-set of any function f^b can be used as don't-care sets for the function f^a for $b > a$, $a, b \in M$. Finally, we apply Theorem 4.4.1 to merge

the minimized functions into the expression over chain based Post algebra for the original function.

As an example, consider the function $f : \{0,1,2\}^2 \rightarrow \{0,1,2\}$ shown in Figure 4.6. It is split with respect to the values $i = 1$ and 2 as:

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

First, we simplify f^2 to x_2^1. Then, we minimize f^1 using x_2^1 as don't care set. This results in $f^1 = x_1^1$. Finally, we merge these two functions according to Theorem 4.4.1 to produce the following cover for f:

$$f = 1 \cdot f^1 + f^2 = 1 \cdot x_1^1 + x_2^1.$$

Note that a better result might be obtained by renaming the values in the output domain to obtain a different ordering. For example, consider the function shown in Figure 4.8 (left). It minimal sum-of-products form is

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

If we rename the values in the output domain as shown in Figure 4.8 (right), the minimal sum-of-products form reduces to

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

Therefore, in general, there is an output ordering problem to be solved to get a minimal sum-of-products representation for a multiple-valued function. This problem is an analogue to output phase optimization problem in the Boolean case.

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

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

Figure 4.8. Specification tables for sum- and product-type operations for $m = 4$.

4.7 MULTI-LEVEL LOGIC OPTIMIZATION

Multi-level networks are often preferred to two-level logic implementations. This section describes two techniques for transformations of two-level forms into multiple-level forms: factorization and decomposition. The overall level

of understanding of multiple-valued multi-level logic is at a much less mature stage than Boolean multi-level logic. Some techniques are generalized to the multiple-valued case, but many essential ones are missing. The development of effective heuristic algorithms for practical synthesis and optimization of multiple-valued multi-level logic is one of the most important open problems.

4.7.1 FACTORIZATION

An effective method for optimizing Boolean networks is the use of kernels for finding common factors among several Boolean logic functions. The common factor can then be removed as a separate function, simplifying the overall network. For the case of general multiple-valued functions $M^n \rightarrow M$, the research on factorization is still in an early stage of development. For multiple-valued input binary-valued output functions, techniques for determining common factors efficiently have been presented in [29] and implemented in multi-level optimization package MIS-MV. Below, we briefly review its main ideas.

MIS-MV handles the functions of type $f : \{0,1\}^{n-1} \times \{0,1,\ldots,m-1\} \rightarrow \{0,1\}$, i.e. only a single variable is assumed to be multiple-valued. As in the Boolean case, the process of factorization involves generating kernels and co-kernels, creating a co-kernel cube matrix and solving the rectangular covering problem. The definitions of kernel and co-kernel remain unchanged, as well as the main steps of the factorization algorithms. To see how the factorization technique extends to the multiple-valued case, consider the following two functions (the example is taken from [3]):

$$f_1 = x^{\{0,1\}} \cdot a \cdot k + x^2 \cdot b \cdot k + c$$

$$f_2 = x^{\{3,4\}} \cdot a \cdot j + x^5 \cdot b \cdot j + d$$

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

$$f_1 = x^{\{0,1,2\}} \cdot k \cdot f_3 + c$$

$$f_2 = x^{\{3,4,5\}} \cdot j \cdot f_3 + d$$

$$f_3 = x^{\{0,1,3,4\}} \cdot a + x^{\{2,5\}} \cdot b$$

As in the Boolean case, the first step is to find all the kernels and co-kernels by successive co-factoring by single Boolean literals. For this example, the following set of kernels is obtained:

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

The kernels are put in a *co-kernel cube matrix* **M**, where rows are labeled by co-kernels. The binary parts of the cubes of the kernels are extracted out at the top of each column, while the multiple-valued part is entered in the matrix:

	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

A *rectangle* in such a matrix is a sub-matrix, containing 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 multiple-valued entries, e.g.

$$\begin{matrix} x^{\{0,1\}} & x^2 \\ x^{\{3,4\}} & x^5 \end{matrix}$$

Such a rectangle can give rise to a common factor provided the matrix is *satisfiable*, which means for every variable, 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 $\mathbf{M}_{ij}$. The above matrix is satisfiable. For a satisfiable rectangle, the common factors are extracted 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 part of the cube on the top of that 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, $b \cdot x^{\{2,5\}}$, giving the kernel $a \cdot x^{\{0,1,3,4\}} + b \cdot x^{\{2,5\}}$. For row 2 we get $k \cdot x^{\{0,1\}}$ and for row 4, $j \cdot x^{\{3,4\}}$, yielding a factorization

$$f_1 = k \cdot x^{\{0,1,2\}}(a \cdot x^{\{0,1,3,4\}} + bx^{\{2,5\}}) + c$$

$$f_2 = j \cdot x^{\{3,4,5\}}(a \cdot x^{\{0,1,3,4\}} + bx^{\{2,5\}}) + d$$

Matrices that are not satisfiable can be transformed to satisfiable matrices by examining for each $\mathbf{M}_{ij}$ 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 cares for the function. Then for a given entry $\mathbf{M}_{ij}$ one has the option of including the values 6, 7 in order to make the matrix satisfiable.

It has been proved in [29] that any kernel found by the standard algebraic factoring process for some encoding will also be found by the above multiple-valued factoring process. Furthermore, the multiple-valued technique can potentially result in common factors that are Boolean factors and hence even stronger than the algebraic kernel intersection method. Other techniques which use multiple-valued logic to identify Boolean factors include [30] and [55].

4.7.2 DECOMPOSITION OF MULTIPLE-VALUED FUNCTIONS

For simple disjoint decomposition of type

$$f(X,Y) = g(h(X),Y) \tag{4.1}$$

where $f : M^n \rightarrow M$, $h : M^{|X|} \rightarrow M$ and $g : M^{|Y|+1} \rightarrow M$ are multiple-valued functions and X and Y are sets of variables satisfying $X \cap Y = \emptyset$, the necessary and sufficient condition for a set of variables to be a bound set directly generalizes to the multiple-valued case [28].

Theorem 4.7.1 *A subset X of variables of an m-variable function f is a bound set if and only if the decomposition chart $X|Y$ of f has column multiplicity at most m.*

For example, the set $\{x_1x_2\}$ is a bound set for the 3-valued function shown in Figure 4.9, since its decomposition chart $x_1x_2|x_3$ has only 3 distinct columns.

x_1x_2	00	01	02	10	11	12	20	21	22
x_3 0	1	0	0	0	0	0	1	0	1
1	0	1	2	2	1	1	0	1	0
2	1	1	1	1	1	1	1	1	1

Figure 4.9. Decomposition chart $x_1x_2|x_3$.

Theorem 4.7.1 has a direct application to the Boolean logic synthesis, since it covers as a special case decompositions of the type $f(X,Y) = g(h(X),Y)$ with $f : \{0,1\}^n \rightarrow \{0,1\}$, $h : \{0,1\}^{|X|} \rightarrow M$ and $g : M \times \{0,1\}^{|Y|} \rightarrow \{0,1\}$. In such a decomposition the function h can be coded by $k = \lceil \log_2 m \rceil$ Boolean functions $h_1, h_2, \ldots, h_k$, giving a decomposition of the form

$$f(X,Y) = g(h_1(X), h_2(X), \ldots, h_k(X), Y) \tag{4.2}$$

with all functions being Boolean [28]. The decomposition of type (4.2) includes as a subclass simple disjoint decompositions ($k = 1$) as well as nondisjoint decompositions. As long as f is a function of more than three variables,

such a decomposition can always be found with $h_1(X), h_2(X), \ldots, h_k(X)$ and g each having fewer arguments than f, for there always exists a decomposition of the form

$$f(X, x_n) = g(h_1(X), h_2(X), x_n)$$

with $X = (x_1, \ldots, x_{n-1})$. An example is Shannon decomposition, where $h_1(X) = f|_{x_n=0}$, $h_2(X) = f|_{x_n=1}$ and $g =$ " $+$ ". Thus, a decomposition of type (4.2) is applicable to *any* Boolean function.

While some properties of bound sets directly extends to the multiple-valued case, there are also some whose generalization is far from trivial. For instance, the following property of overlapping bound sets, fundamental for the Boolean decomposition theory, extends only to the multiple-valued input binary-valued output functions [7].

Property 4.7.1 *If $\{X, Y\}$ and $\{Y, Z\}$ are bound sets, then $\{X\}, \{Y\}, \{Z\}$ and $\{X, Y, Z\}$ are also bound sets.*

This does not hold for all functions $M^n \rightarrow M$. An example is the function $\{0,1,2\}^4 \rightarrow \{0,1,2\}$ given by the following expression

$$f(x_1, x_2, x_3, x_4) = (x_1^1 + x_2^1) \cdot x_3^0 \cdot x_4^0 + 1 \cdot (x_1^2 \cdot x_2^{\{0,2\}} + x_1^{\{1,2\}} \cdot x_2^0) \cdot x_3^1 \cdot x_4^1.$$

The sets $\{x_1, x_2, x_3\}$ and $\{x_1, x_2, x_4\}$ are bound sets, but the set $\{x_1, x_2\}$ is not. The operations "+" and "$\cdot$" in the above expression are MAX and MIN, respectively.

To characterize a class of functions for which Property 4.7.1 holds, we first introduce the following definition [11].

Definition 4.7.1 *A function f is fully sensitive to a variable x_i if there exists an assignment for the other variables of f such that any change in the value of x_i causes a change in the value of f. A function is fully sensitive if it is fully sensitive to all its variables.*

Full sensitivity of f to x_i means that for some fixed assignment $(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n) \in M^{n-1}$, the subfunction $f(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n)$ range over all possible values of M. For $f(x_1, x_2, x_3)$ in Figure 4.9, the subfunctions $f(x_1, 2, 1), f(0, x_2, 1)$ and $f(0, 2, x_3)$ are such subfunctions, so $f(x_1, x_2, x_3)$ is fully sensitive.

Von Stengel has shown that Property 4.7.1 holds for any fully sensitive function [54]. Starting from this fundamental case, he has extended the Ashenhurst disjoint decomposition theory [1] to the class Σ of fully sensitive functions $M^n \rightarrow M$. He has shown that, for any fully sensitive function, there exist a *composition tree* reflecting any bound set of f. Each node of the composition

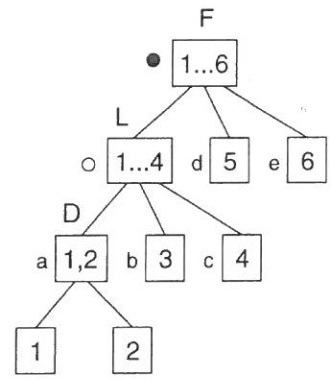

Figure 4.10. Example of a composition tree.

tree is labeled with a function that has as many variables as the node has children. Leaves are labeled with unary functions, which may be the identity. The hierarchical expression of these functions represents f. For a tree such as that shown in 4.10 (which is described more fully below), this expression is

$$f(x_1, \ldots, x_6) = g(h(a(x_1, x_2), x_3, x_4), x_5, x_6). \tag{4.3}$$

The nodes of the tree are labeled by three types of labels: "disjoint", "full" or ""linear". For the tree from the example, we used abbreviations "D", "F" and "L" for "disjoint", "full" or ""linear", respectively. The following property of composition trees holds [13].

Theorem 4.7.2 *Let $T(f)$ be the composition tree of f. A is a bound set for f if and only if*

(a) *A is a node of the tree, or*

(b) *A is the union $\bigcup_{i \in I} B_i$ of the children $B_1, \ldots, B_k$ of a "full" node, $\emptyset \neq I \subseteq \{1, \ldots, k\}$, or*

(c) *A is the union $\bigcup_{i=j}^{l} B_i$ of an interval of the children $B_1, \ldots, B_k$ (specified in linear order) of a "linear" node, $1 \leq j \leq l \leq k$.*

Based on the above, a function can decomposed in accordance with its composition tree as follows [13].

Theorem 4.7.3 *Let $T(f)$ be the composition tree of $f : M^n \rightarrow M$ and $B_1, \ldots, B_k$ be the children of the root $N = \{x_1, x_2, \ldots, x_n\}$. Then*

$$f(B_1, \ldots, B_k) = g(h_1(B_1), \ldots, h_k(B_k)) \tag{4.4}$$

for functions $h_i : M^{|B_i|} \to M$ $(1 \leq i \leq k)$ *and* $g : M^k \to M$ *in* Σ *where*

(a) *g is non-decomposable if N is labeled "disjoint",*

(b) $g(a_1, \ldots, a_k) = a_1 \bullet \cdots \bullet a_k$ *(for* $a_i \in M$, $1 \leq i \leq k$*) with an associative and commutative operation "*$\bullet$*" in* Σ *if N is labeled "full",*

(c) $g(a_1, \ldots, a_k) = a_1 \circ \cdots \circ a_k$ *with an associative and non-commutative operation "*$\circ$*" in* Σ *if N is labeled "linear".*

(d) *In (a), g is unique up to isotopy. In (b) and (c),* $\bullet$ *is unique up to isomorphy.*

This is the main representation theorem of [54]. The theorem is applied by induction, which ends if f is non-decomposable. Using the trees $T(h_1), \ldots, T(h_k)$ with the children $B_1, \ldots, B_k$ of N as roots, equation (4.4) gives a hierarchical expression containing any simple disjoint decomposition as a subexpression. As an example consider the composition tree in Figure 4.10. Letters a, b, c, d, e denote the functions associated with the nodes, and $\bullet$ and $\circ$ denote the operations. In accordance with this tree, f can be decomposed as

$$f(x_1, \ldots, x_6) = (a(x_1, x_2) \circ b(x_3) \circ c(x_4)) \bullet d(x_5) \bullet e(x_6)$$

with $\bullet$ being an associative and commutative operation and $\circ$ being associative and non-commutative.

A result from [12] states that a percentage of m-valued n-variable functions which are *not* fully sensitive tends to zero as n increases. This implies that, for large n, composition tree exists for most of the functions $M^n \to M$. An algorithm for constructing composition trees for multiple-valued functions has been presented in [13].

For non-disjoint decompositions, Curtis' results [6] have been partly extended to the multiple-valued case in [33]. Other techniques for decomposition of multiple-valued functions include [18] and [31].

4.8 SUMMARY

We surveyed a part of the theory of multiple-valued logic related to CAD: multiple-valued functions, functionally complete sets and multiple-valued algebraic systems. We discussed positional cube, multi-level network and MDD representations of multiple-valued functions. We described techniques for two- and multi-level optimization of multiple-valued functions.

Conceptually, the best way to benefit from multiple-valued logic is to describe a system using some multiple-valued representation and to manipulate such a representation directly. When no further optimization is possible in purely multiple-valued form, a suitable encoding can be applied to transform the multiple-valued representation into a Boolean one. However, this approach

is not widely used mainly due to the lack of mature packages for representation and manipulation of multiple-valued functions, as well as for multiple-valued synthesis, optimization and verification. To gain popularity, these packages must by competitive with recognized packages, like CUDD [52] and VIS [53]. Such multiple-valued packages are, in turn, not developed yet because: (1) A complete suite of associated algorithms is still not available. Although some techniques are generalized to the multiple-valued case, many essential ones are missing, e.g. algorithms to perform multi-level multiple-valued optimization efficiently. (2) The encoding problem is hard for large circuits since it is difficult to foresee how a given encoding affects the optimized multiple-valued multi-level representation. (3) Not the least problem is a lack of researchers working in the area. Current CAD packages have reach their level of complexity, sophistication and application largely because of the continuous development and brain-power investments. It is hoped that this chapter will motivate more research towards a mature multiple-valued CAD package.

4.9 HISTORICAL PERSPECTIVES

The development of multiple-valued logic began with work of Łukasiewicz [32] and Post [39]. The first algebra corresponding to Post's logic was first formulated by Rosenbloom [42] and then later developed by Epstein [16]. A characterization of the necessary and sufficient conditions for the completeness of a set of multiple-valued functions is due to Rosenberg [43]. A survey on the topic of completeness can be found in [44].

Multiple-valued functions were originally studied to provide support for designing m-valued logic circuits, which employ m discrete signals, $m \geq 2$. The attempts to built multiple-valued integrated circuits (ICs) of multiple-valued circuits compatible with IC technologies can be traced back to 1970, starting from early work on 3-valued designs [50, 24]. Applying multiple-valued logic has been shown to allow enhancing circuit performance in terms of chip area, operation speed and power consumption [20]. Multiple-valued circuits have matured to the point where four-valued flash and DRAM memories are now part of commercially available ICs [40].

Applications of multiple-valued logic to other sciences have been considered, including neural nets [22], threshold logic [35], molecular computing [27], optical computing [23].

Books related to the multiple-valued logic synthesis and optimization include Rine [41], Muzio and Wesselkamper [38], Butler [4], Epstein [17], Sasao [48].

Journals and conferences proceedings related to the subject include *Multiple-Valued Logic: An International Journal* (1996-), special issues of *Computer* (Sept. 1974, April 1988), *IEEE Transactions of Computers* (Sept. 1981, Feb.

1986), *International Journal of Electronics* (August 1987, Nov. 1989), Proceedings of annual *International Symposium on Multiple-Valued Logic* (1971-).

Overview papers include Hurst (1981) [24], Smith (1984) [50], Butler (1995) [5], Brayton (1999) [3] and Dubrova (2001) [15].

References

[1] R. L. Ashenhurst, "The decomposition of switching functions," in *Proc. Int. Symp. Theory of Switching*, Vol. 29, Part. I, pp. 74-116, 1959.

[2] I. Ben Dhaou, E. Dubrova and H. Tenhunen 2001 "Power efficient inter-module communication for digit-serial architectures in deep-submicron technology," in *Proc. 31st Int. Symp. Multiple-Valued Logic*, pp. 61-67, May 2001.

[3] R. K. Brayton and S. P. Khatri "Multi-valued logic synthesis," in *Proc. 12th Int. Conf. on VLSI Design*, pp. 196-206, Jan. 1999.

[4] J. T. Butler (ed), *Multiple-valued logic in VLSI*, California, IEEE Computer Society, 1991.

[5] J. T. Butler, "Multiple-valued logic," *IEEE Potentials*, Vol. 14, No. 2, pp. 11 - 14, April-May 1995.

[6] H. A. Curtis, *A New Approach to the Design of Switching Circuits* Van Nostrand, Princeton, 1962.

[7] M. Davio, J.-P. Deschamps and A. Thayse, *Discrete and Switching Functions*, Switzerland, McGraw-Hill International Book Company, 1978.

[8] L. Djordjevic, "Application of the Karnaugh map to the minimization of functions in ternary logic," *Automatika Theoretical Supplement*, Vol. T.3, No. 1-2, pp. 35-38, Zagreb 1967.

[9] G. W. Dueck and D. M. Miller, "Direct search minimization of multiple-valued functions," in *Proc. 18th Int. Symp. Multiple-Valued Logic*, pp. 218-225, May 1988.

[10] E. V. Dubrova, Y. Jiang and R. K. Brayton, "Minimization of multiple-valued functions in Post algebra," in *Proc. of Int. Workshop on Logic Synthesis*, pp. 132-138, June 2001.

[11] E. V. Dubrova, D. B. Gurov and J. C. Muzio, "Full sensitivity and test generation for multiple-valued logic circuits," in *Proc. of 24th Int. Symp. MVL*, pp. 284-289, May 1994.

[12] E. V. Dubrova, D. B. Gurov D. B. and J. C. Muzio, "The evaluation of full sensitivity for test generation in MVL circuits," in *Proc. 25th Int. Symp. on MVL*, pp. 104-109, May 1995.

[13] E. V. Dubrova, J. C. Muzio and B. von Stengel, "Finding composition trees for multiple-valued functions," in *Proc. 27th Int. Symp. on MVL*, pp. 19-26, May 1997.

[14] E. Dubrova and H. Sack, "Probabilistic verification of multiple-valued functions," in *Proc. 30th Int. Symp. on Multiple-Valued Logic*, pp. 461-466, May 2000.

[15] E. Dubrova, "Multiple-valued logic in VLSI," *Multiple-Valued Logic, An International Journal*, 2001, to appear.

[16] G. Epstein, "The lattice theory of Post algebras," *Trans. Am. Math. Soc.*, Vol. 95, No. 2, pp. 300-317, Feb. 1960.

[17] G. Epstein, *Multiple-valued logic design: an introduction*, Bristol and Philadelphia, IOP Publishing, 1993.

[18] G.C. Files and M. Perkowski, "New multivalued functional decomposition algorithm based on MDDs," *IEEE Trans. on CAD/ICAS*, Vol. CAD-14, No. 9, pp. 1081-1086, Sept. 2000.

[19] M. Gao, J.-H. Jiang, Y. Jiang, Y. Li, S. Sinha and R. Brayton, "MVSIS," in *Proc. Int. Workshop on Logic Synthesis*, pp. 138-144, June 2001.

[20] T. Hanyu, "Challenge of a multiple-valued technology in recent deep-submicron VLSI," in *Proc. 31st Int. Symp. on Multiple-Valued Logic*, pp. 241-247, May 2001.

[21] T. Hanyu and M. Kameyama, "A 200 MHz pipelined multiplier using 1.5 V-supply multiple-valued MOS current-mode circuits with dual-rail source-coupled logic," *IEEE Journal of Solid-State Circuits*, Vol. 30, No. 11, pp. 1239-1245, Nov. 1995.

[22] L. S. Hsu, H. H. Teh, S. C. Chan and K. F. Loe "Multi-valued neural logic networks," in *Proc. 20th Int. Symp. Multiple-Valued Logic*, pp. 426-432, May 1990.

[23] S. L. Hurst, "A survey: developments in optoelectronics and its applicability to the multiple-valued logic," in *Proc. 16th Int. Symp. Multiple-Valued Logic*, pp. 2179-2188, May 1980.

[24] S. L. Hurst, "Multiple-Valued Logic - its status and its future," *IEEE Trans. on Computers*, Vol. C-33, No. 12, pp. 1160-1179, Dec. 1984.

[25] Y. Jiang and R. K. Brayton, "Don't cares and multi-valued logic network minimization," in *Proc. Int. Conf. on Computer-Aided Design*, pp. 520-526, Nov. 2000.

[26] T. Kam, T. Villa, R. Brayton and A. Sangiovanni-Vincentelli, "Multi-valued decision diagrams: theory and applications," *Multiple-Valued Logic, An International Journal*, Vol. 4, No. 1-2, pp. 9-24, 1998.

[27] M. Hiratsuka, T. Aoki and T. Higuchi, "A model of reaction-diffusion cellular automata for massively parallel molecular computing," in *Proc. 31st Int. Symp. on Multiple-Valued Logic*, pp. 247-253, May 2001.

[28] R. M. Karp, "Functional decomposition and switching circuit design," *J. Soc. Indust. Appl. Math.*, Vol. 11, pp. 291-335, Nov. 1963.

[29] L. Lavagno, S. Malik, R. Brayton and A. Sangiovanni-Vincentelli, "MIS-MV: optimization of multi-level logic with multiple-valued inputs," in *Proc. Int. Conf. Computer-Aided Design*, pp. 560-563, Nov. 1990.

[30] S. Liao, S. Devadas and A. Ghosh, "Boolean factorization using multiple-valued minimization," in *Proc. Int. Conf. Computer-Aided Design*, pp. 606-611, Nov. 1993.

[31] T. Luba, "Decomposition of multiple-valued functions," in *Proc. 25th Int. Symp. on MVL*, pp. 256-261, May 1995.

[32] J. Łukasiewicz, "O logice trójwartościowej," *Ruch Filozoficzny*, Vol. 5, pp. 169-171, 1920.

[33] J. J. Lou and J. A. Brzozowski, "A generalization of Shestakov's functional decomposition method," in *Proc. Int. Symp. on Multiple-Valued Logic*, pp. 66-71, May 1999.

[34] C. Moraga, "A minimization method for 3-valued logic functions," in *Theory of Machines and Computations*, ed: Paz A. and Kohavi Z., New York, Academic, pp. 363-375, 1971.

[35] C. Moraga, "Multiple-valued threshold logic." in *Optical computing, digital and symbolic*, ed: Arrathoon, New York, Dekker, pp. 161-184, 1989.

[36] G. D. De Micheli, R. Brayton and A. Sangiovanni-Vincentelli, "Optimal state assignment for finite state machines," *IEEE Trans. on CAD/ICAS*, Vol. CAD-4, No. 3, pp. 269-284, July 1985.

[37] J. C. Muzio and D. M. Miller, "On the minimization of many-valued functions," in *Proc. 9th Int. Symp. Multiple-Valued Logic*, pp. 294-299, May 1979.

[38] J. C. Muzio and T. C. Wesselkamper, *Multiple-valued switching theory*, Bristol, Hilger, 1986.

[39] E. L. Post "Introduction to a general theory of elementary propositions," *Amer. J. Math.*, Vol. 43, pp. 163-185, 1921.

[40] B. Ricco, G. Torelli, M. Lanzoni, A. Manstretta, H. E. Maes, D. Monatanari and A. Modelli, "Non-volatile multilevel memories for digital applications," in *Proc. IEEE*, Vol. 86, No. 12, pp. 2399-2421, Dec. 1998.

[41] D. C. Rine (ed), *Computer science and multiple-valued logic*, 2nd edn, Amsterdam, North-Holland, 1984.

[42] P. C. Rosenbloom, "Post algebras I, postulates and general theory," *Amer. J. Math.*, Vol. 64, pp. 167-188, 1942.

[43] I. G. Rosenberg, "La structure des fonctions de plusieurs variables sur un ensemble fini," *C. R. Acad. Sci. Paris, Ser. AB*, Vol. 260, pp. 3817-3819, 1965.

[44] I. G. Rosenberg, "On closed classes, basic sets and groups," in *Proc. 7th Int. Symp. Multiple-Valued Logic*, pp. 1-6, May 1977.

[45] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Trans. on CAD/ICAS*, Vol. CAD-5, No. 9, pp. 727-750, Sept. 1987.

[46] T. Sasao, "Ternary decision diagrams: survey," in *Proc. 27th Int. Symp. Multiple-Valued Logic*, pp. 241-250, May 1997.

[47] T. Sasao, "Multiple-valued logic and optimization of programmable logic arrays," *IEEE Computer*, Vol. 21, pp. 71-80, 1988.

[48] T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.

[49] K. Shimabukuro and C. Zukeran, "Reconfigurable current-mode multiple-valued residue arithmetic circuits," in *Proc. 28th Int. Symp. Multiple-Valued Logic*, pp. 282-287, May 1998.

[50] K. C. Smith, "The prospects for multivalued logic: A technology and applications view," *IEEE Trans. om Computers*, Vol. C-30, No. 9, pp. 619-634, Sept. 1981.

[51] W. R. Smith, "Minimization of multiple-valued functions," in *Computer science and multiple-valued logic*, ed: Rine, North-Holland, Amsterdam, 1984.

[52] F. Somenzi, *CUDD: CU Decision Diagram Package, Release 2.3.0*, University of Colorado at Boulder, 1998.

[53] The VIS Group, "VIS: A system for Verification and Synthesis," in *Proc. 8th Int. Conf. on Computer Aided Verification*, Springer Lecture Notes in Computer Science, ed: Alur and Henzinger, Vol. 1102, New Brunswick, NJ, pp. 428-432, 1996.

[54] B. von Stengel, *Eine Dekompositionstheorie für mehrstellige Funktionen* Mathematical Systems in Economics, Vol. 123, Anton Hain, Frankfurt, 1991.

[55] H. M. Wang, C. L. Lee and J. E. Chen, "Factorization of Multiple-Valued Functions," *25th Int. Symp. on Multiple-Valued Logic*, pp. 164-169, May 1995.

Chapter 5

TECHNOLOGY MAPPING

Leon Stok

Vivek Tiwari

Abstract Technology mapping transforms a technology independent logic network into gates implemented in a technology library. This chapter focuses on the three phases of technology mapping : decomposition, pattern matching and covering. Traditionally, a lot of work has been focused on tree mapping algorithms, but since most practical circuits are DAGs, DAG mapping algorithms are gaining importance. Different objective functions, namely delay, area, power and reliability motivate the use of different algorithms. Future challenges are outlined.

5.1 INTRODUCTION

Most state-of-the-art logic synthesis systems [6, 22, 25] consist of three separate phases. In the technology independent optimization phase, boolean equations are minimized, terms are shared, and boolean and algebraic optimizations are executed. Most often the goal is to reduce the size of the equations and minimize the number of literals. Technology mapping translates technology independent logic equations into a network of technology cells in a particular technology. The technology mapping phase is commonly followed by technology based transformations. Depending on the quality of the mapping, the technology based transformations locally modify the network to meet specific timing, power or area goals.

Technology mapping is the only phase in logic synthesis that transforms each and every cell in the network. It therefore has a major impact on the global structure of the technology mapped logic, and its delay and area characteristics.

Conventional technology mapping can be described as a three step procedure: decomposition, pattern matching and covering. First, the technology-independent circuit is decomposed in terms of some primitive cells to have a simple logic structure to aid the technology mapping process. This phase is typically referred to as decomposition. Second, a pattern matcher performs