PART II OF THE CLASS TEXTBOOK. FUNCTIONAL DECOMPOSITION IN LOGIC DESIGN AND DATA MINING

Marek A. Perkowski
Department of Electrical Engineering,
Portland State University,
P.O. Box 751, Portland, Oregon 97207
tel. (503) 725-5411

October 11, 2006

October 11, 2006

Contents

List of Figures

List of Tables

Part I

INTRODUCTION TO DECOMPOSITION

Chapter 1

Ashenhurst Decomposition

1.1 Introduction.

One of the most general approaches to solve engineering and science optimization problems is to apply the principle of decomposition: a problem of a larger dimension is decomposed to several problems of smaller dimensions, which can be solved separately and with less effort. The problem of decomposition of a Boolean function F is to find the realization of F which is a composition of simpler Boolean functions. In the late 1950's, Ashenhurst and Curtis determined the necessary and sufficient conditions for the existence of a given type of decomposition. These decomposition types are now being called Ashenhurst Decomposition and Curtis Decomposition, respectively. In a related effort, Roth and Karp developed a method using covers of ON-set and OFF-set of the function. This method used cube calculus instead of the decomposition charts, and generalized the Curtis method for incompletely specified functions. Although popularly called the *Roth-Karp decomposition*, this is still the Curtis Decomposition with different representation of Boolean functions. Therefore, from the perspective of this book, this is still a *Curtis Decomposition Type*. We will talk, however, about the *Roth-Karp decomposition method.*

The goal of this book is to develop a general decomposition model, that will be more powerful than the existing ones, and will include many of them as particular cases. In order to do this, we will have first to present in a unified way the known decompositions. We will characterize them from various orthogonal points of view.

As the result of the analysis of the above approaches, as well as other most successful decomposition methods and programs for Ashenhurst-Curtis Decomposition, we arrived at several conclusions that we next used to create a new unified theory and program. This new program, called *Generalized Unified Decomposer* **(GUD)**, has been designed with the goal to outperform all the existing approaches to those decompositions that are not technology-related but minimize certain general cost functions related to function complexity. GUD-1 is now a part of our MULTIS decomposition environment that ultimately will include FRED, TRADE (not yet connected), GUD, DEMAIN, SOP minimizers, and several variable partitioning and format changing programs.

Such a superior program was requested for the high-demand applications that are of interest to Pattern Theory group at Avionics Labs of Wright Laboratories, U.S. Air Force. Our generalization of the decomposition is for multiple-valued logic, fuzzy and continuous logic, and information systems as well. It can be also used for designs implemented with multiplexers and generalized multiplexers in these signal representations.

In a similar ways we generalized the decompositions for multiple-valued logic, fuzzy and continuous logic, and information system as well. We created variants that can be used also for designs implemented with multiplexers and generalized multiplexers. However, at present these variants are not as general as our binary decompositions.

In this book only binary and some multiple-valued decompositions will be discussed. Other multiple-valued decompositions, fuzzy decompositions and continuous decompositions will be presented in separate papers.

In this book we totally unify the methods developed in the past by the four aforementioned groups: Portland State University, Warsaw Technical University, XBOOLE group at Freiberg University in Germany, and Wright Laboratories, and we add several new methods not known from the literature. Therefore, the GUD program will also be able to execute decompositions that have never been formulated before. It will be a test-bed to develop and compare several known and new partial ideas related to decomposition.

While creating the program, our emphasis has been on the following topics:

1. representation of data and efficient algorithms for data manipulation,
2. variable ordering methods for variable partitioning to create bound and free sets of input variables; heuristic approaches and their comparison,
3. the column compatibility problem (called also the column (multiplicity) minimization),

4. the subfunction encoding problem (column encoding problem),
5. the use of partial and total symmetries in data to decrease the decomposition search space,
6. methods of dealing with strongly unspecified functions which are typical for Machine Learning applications,
7. other decomposition types, especially XBOOLE decompositions and their generalizations.
8. controlling the strategies to select decomposition types and decomposition variable sets, such strategies allow to create some kinds of decomposed structures (such as cascades, trees without variable repetition, etc).

Since we have to introduce many different subjects in this book, some of them quite complex, and we would like our book to be understood by non-specialists in logic synthesis, we will apply a tutorial approach below. The book will be full of examples and graphical illustrations. This book is only a first draft and we welcome our readers to provide us with a strong criticism which will help us to prepare improved next versions.

The plan of this book is the following:
First we introduce, in an elementary way, various types of decomposition: some known from the literature, some developed by us. The groups of these decompositions are:

- disjoint Ashenhurst decomposition,
- non-disjoint Ashenhurst decomposition,
- disjoint Curtis decomposition,
- non-disjoint Curtis decomposition,
- disjoint PUB decomposition,
- non-disjoint PUB decomposition,
- strong AND, OR, EXOR decomposition of Bochmann and Steinbach,
- weak AND, OR, EXOR decomposition of Bochmann and Steinbach,
- multiplicity-based AND, OR, EXOR "goal-oriented" decomposition of Wei Wan and Perkowski.

We introduce here function representations as well: the Partition Calculus of Luba and our new *Cube Diagram Bundles* (CDBs for short), and demonstrate how all the abovementioned decompositions can be implemented using partitions and CDBs.

In part III we discuss the very important problem of finding good free and bound sets of variables. (This book is devoted also to solving the column compatibility problem). This part discusses new variable-partitioning methods and compares them with the well-known methods:

- r-admissability (Luba et al).
- symmetries (Kim and Dietmeyer).
- heuristic search in tables (C. Files, H. Wu).
- cube calculus based correlation between variables (Wei Wan),
- entropy (C4.5, Zwick).

- reuse of the previously found partitions (Ashenhurst, Curtis, Walliuzzaman).

The problem of bound set encoding, and the related problem of function reusing, will be described also in Part III.

The presented book is organized as follows. Section ?? defines the decomposition problem. In section ?? we introduce a new representation of Boolean and Multi-valued Functions - *Cube Diagram Bundles (CDBs).* This general-purpose representation combines Cube Calculus [?], Decision Diagrams [?] and Rough Partitions [?, ?, ?], and is especially efficient for very strongly unspecified functions. CDBs incorporate also a new concept of generalized don't cares for multiple-valued logic. This representation is next used to solve various decomposition problems that are important for Machine Learning and circuit design applications. Section ?? explains the design with multiplexers and section ?? introduces several basic types of decompositions based on patterns found in this representation. Ashenhurst decomposition [?, ?, ?] for completely and incompletely specified functions is introduced in sections ?? and ??, respectively. A new approach to Ashenhurst Decomposition is also presented in section ?? - it is shown that contrary to the more general case of Curtis Decomposition [?], the column minimization problem is polynomially complete, and we give an efficient algorithm to solve it. Section 9 introduces also the decomposition of Perkowski, Uong and Brown, called PUB. Curtis decomposition is introduced in section ??. Section 11 presents a new approach to Curtis Decomposition, which belongs to the *Basic Decompositions* of the system. Although in some respects similar to the approach from [?], we use the new representation, and several of its partial problems are significantly improved. For instance, a new very efficient algorithm for coloring of the Column Incompatibility Graph is proposed, that utilizes the similarity of the graph coloring and the set covering problems, and thus gives an exact minimal coloring for any graph that corresponds to a non-cyclic set covering problem.

Section ?? presents the decomposition of Multiple-Valued functions and information systems. Section ?? presents fundamental formalisms for Partition-Based Decompositions. Section ?? shows how BDDs can be used to encode these partitions for data compression. Section ?? presents a basic theorem for Curtis decomposition. Section ?? discusses decomposition of decision tables. Section ?? presents approaches to the Column Minimization Problem. Section ?? presents the set covering approach to this problem. Section ?? presents the graph coloring approach to this problem. Section ?? discusses AND, OR and EXOR strong decompositions. Section ?? discusses AND, OR and EXOR weak decompositions. Section ?? presents our entire decomposition strategy. The *Immediate Decompositions* happen rarely but are of a good quality: Strong Gate Decompositions, and the Ashenhurst Decomposition. A subsection introduces another new concept in logic synthesis: goal-oriented reduction schemes, which generalize the EXOR transformation of Curtis-nondecomposable functions from [?]. Any function can serve as a goal function, and three reduction types (EXOR, OR and AND) of reducing a given function to a goal function are presented. Another subsection presents the "last-resort" decompositions and synthesis methods, used when all other efforts fail to find a good decomposition. They include *Weak AND* and *Weak OR* Decompositions, ESOP, and other. Finally, we present our decomposition algorithm based on the concepts of Basic, Immediate, Goal-oriented, and Last-Resort decompositions. In section ?? we present an even more comprehensive unified approach to decompositions, which we even did not start to implement. Section ?? presents some other applications of CDBs. Section ?? concludes our book.

When reading this book, the reader has to bear in mind the following:

- We will refer to maps only for illustration purposes.
- Multiplexers and logic gates such as AND and OR in the decomposed functions are also mostly used for illustration. They will not imply the realization method of the function.

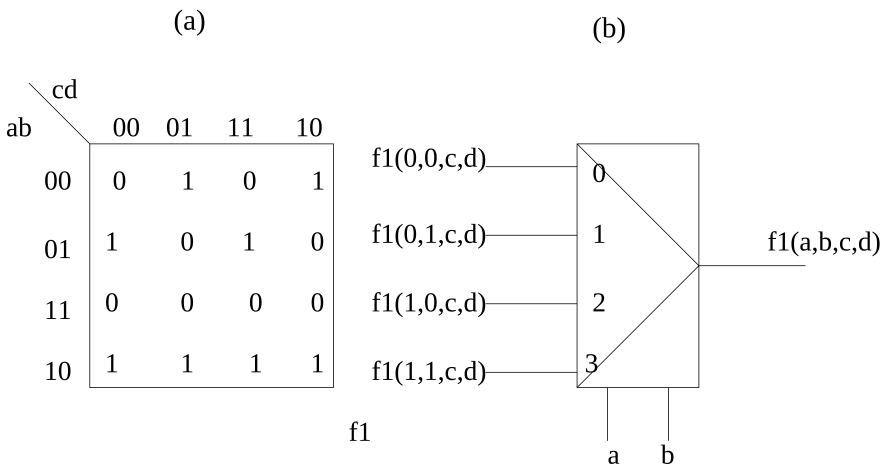


Figure 1.1: Explanation of the concept of cofactors in the Kmap and their relation to the data inputs of a multiplexer, (a) A K-map of function $f1$, (b) realization of $f1$ using a multiplexer with address variables whose combinations of values correspond to the rows of the K-map

1.2 Designs Realized With Multiplexers.

Let us first describe how Karnaugh maps (K-maps for short) can be used to implement a single-output function using multiplexers.

Example 1.2.1

Let us assume that we want to realize the four-input function $f1(a, b, c, d)$, shown in Fig. 1.1a, using a 4×1 multiplexer with two select lines (2 bit address).

Let as further assume that inputs a, b were selected as address inputs of the multiplexer. We will say that the *set of address variables* is $\{a, b\}$ or that variables a and b are the *address variables*. Then, for $a = 0$ and $b = 0$ a function described in row $\overline{a}\,\overline{b}$ of the K-map should be realized as the function $f1(a = 0, b = 0, c, d)$. Let as denote the function from row $\overline{a}\,\overline{b}$ as $f1(0, 0, c, d)$. The other notation used by us will be $f1_0(c, d)$. To realize function $f1(a, b, c, d)$ we will then provide a function $f1(0, 0, c, d)$ on the data input number 0 (which denotes $[a, b] = [0, 0]$) in the multiplexer. Similarly, the function from row 2, $a = 0$, $b = 1$, will be provided at data input 1, the function from row 4, $a = 1$, $b = 0$, will be provided at data input 2 of the multiplexer, and the function from row 3 ($[a, b] = [1, 1]$), will be provided at data input 3. This is shown in Figure 1.1b.

Now our task is to find the realizations of the functions:

$f1(0, 0, c, d) = f1_0(c, d)$,
$f1(0, 1, c, d) = f1_1(c, d)$,
$f1(1, 0, c, d) = f1_2(c, d)$,
$f1(1, 1, c, d) = f1_3(c, d)$.

These functions, called *cofactors* with respect to the set of (address) variables $\{a, b\}$, are specified by the corresponding loops in the K-map: $\overline{a}\,\overline{b}$, $\overline{a}\,b$, $a\,\overline{b}$, and $a\,b$, respectively. The functions $\overline{a}\,\overline{b}$, $\overline{a}\,b$, $a\,\overline{b}$, and $a\,b$ we will denote as *loop functions*. For better manipulation, the inexperienced user can rewrite each cofactor in the form of a separate K-map. As we see, all of these cofactors are functions of the variables c and d only, as shown in the K-maps in Fig. 1.2.

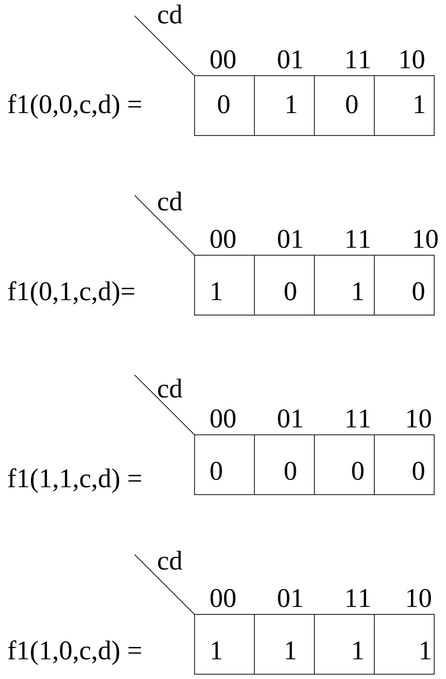


Figure 1.2: Cofactors from Fig. 3 shown as separate functions of variables c and d

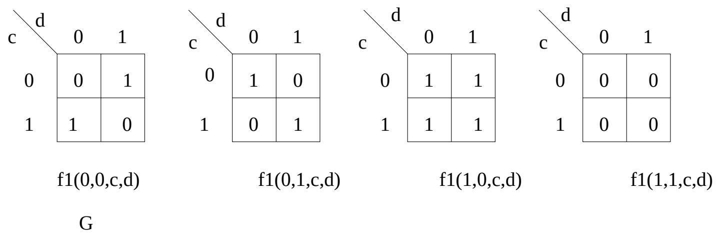


Figure 1.3: Cofactors from Figure 4 rewritten to a more familiar form of K-maps

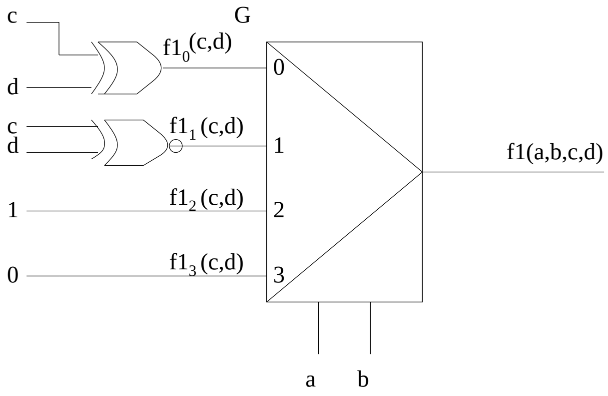


Figure 1.4: Realization with address variables a, b

Before getting some experience in manipulating such maps, we can transfer these functions to K-maps of a more familiar form, as shown in Fig. 1.3. After solving few examples, however, the reader should get enough proficiency to be able to omit this step and operate directly on the initial submaps of the K-map.

The data input functions $f1_0$, $f1_1$, $f1_2$, $f1_3$ can be now realized using any of the well-known Boolean minimization methods. The corresponding circuit realization is shown in Fig. 1.4.

Note that, in contrast to the well-known approaches to decomposition, we do not need to use Marquand or decomposition charts to find multiplexer realizations for other sets of address variables. The process of finding the circuit when some other input variables, c and d, are assumed as the address variables, is shown in Fig. 1.5.

This realization costs less than the previous one since an inverter is used instead of an equivalence gate (EXNOR). Similarly, assuming address variables a and c (Fig. 1.6a), the functions $f1_0(b,d)$, $f1_1(b,d)$, $f1_2(b,d)$, $f1_3(b,d)$ will be as shown in Fig. 1.6b and the realization as shown in Fig. 1.6c. Note the inverse order of variables in the headings of some of maps from Fig. 1.6b. If the designer has difficulties in using the "inverse" K-maps, he can rewrite some of them as shown in Fig. 1.6d. After solving few examples, however, the reader achieves enough proficiency to be able to omit this step, and operate directly on the "inverse submaps" of the K-map. The realization shown in Fig. 1.6c has the same cost as one from Fig. 1.4.

We are now able to formulate the design method for designs implemented with multiplexers. For each possible subset of variables that are used as the address variables, the corresponding loops in the K-map must be drawn. Fig. 1.7 shows loops for all the possible sets of address variables for 4-variable functions. It is easy to start with the loop corresponding to the product made up of noninverted address variables (such products are shown in bold - for instance group 3 for ab in Fig. 1.7a). Drawing of the remaining loops for these variables can be quickly achieved by taking the mirror image of the initial loop with respect to all the possible combinations of values of corresponding address variables. For instance the group 1 in Fig. 1.7a is a mirror image with respect to variable a, the group 2 with respect to variable b, and group 0 with respect to both variable a and variable b.

We denote the set of address variables by $ADDR$.

The design method for circuits implemented with multiplexers can be then summarized as follows:

1. Draw the K-map for the function $F(X)$.
2. Find all the possible subsets $ADDR$ of input variables to be used as the address variables.
3. Draw the initial loop and then all the remaining loops for each set $ADDR$ of address variables, as shown in Fig. 1.7.

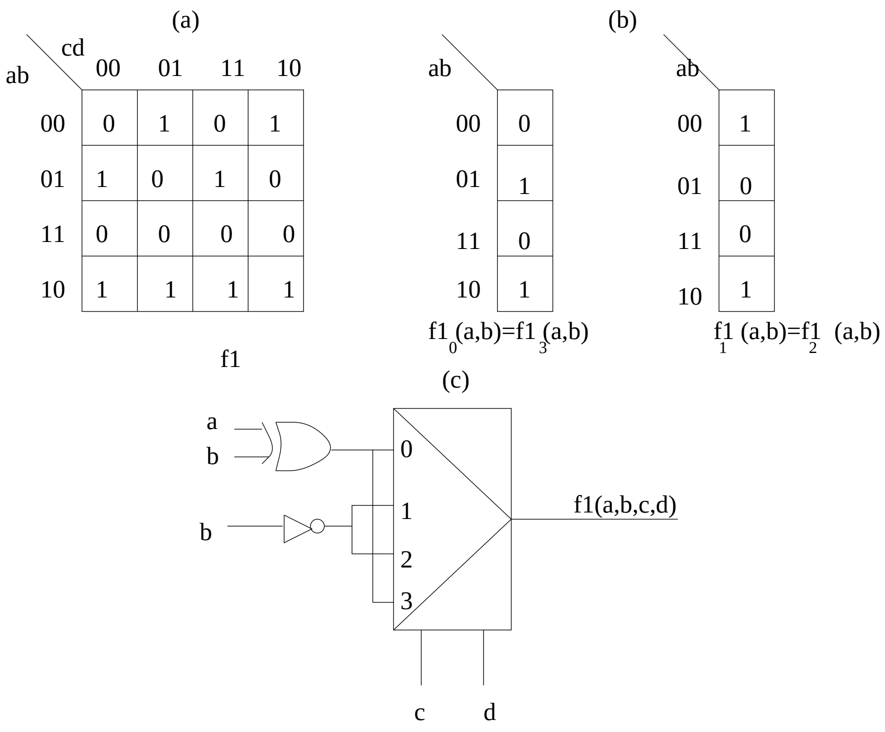


Figure 1.5: Realization with address variables c and d

4. For each subset $ADDR$ of address variables:
   - draw loop functions $f_i(X - ADDR)$ for all the cofactors of this address set.
   - realize loop functions $f_i(X - ADDR)$ using any Boolan minimization method (including realization with multiplexer, recursively), but use the same function for as many multiplexer inputs as possible.
   - evaluate the cost of the solution by adding the realization costs of various cofactors.
5. Select the best solution for all the address sets out of those found in step 4.

Experienced designers can practically apply this method in K-maps using not more than ten variables. In the age of computers, of course, we do not advocate to use K-maps, we would like to point out, however, that good understanding of maps, decision diagrams and partitions is crucial in quick acquisition of ideas from the research literature and also in creating new, improved and more general decompositions. Many design short-cuts are possible to find good solutions without investigating all address sets.

In a forthcoming report we will illustrate also how these representations can be used to develop design methods with generalized multiplexers and respective decompositions in multiple-valued, fuzzy and continuous logics.

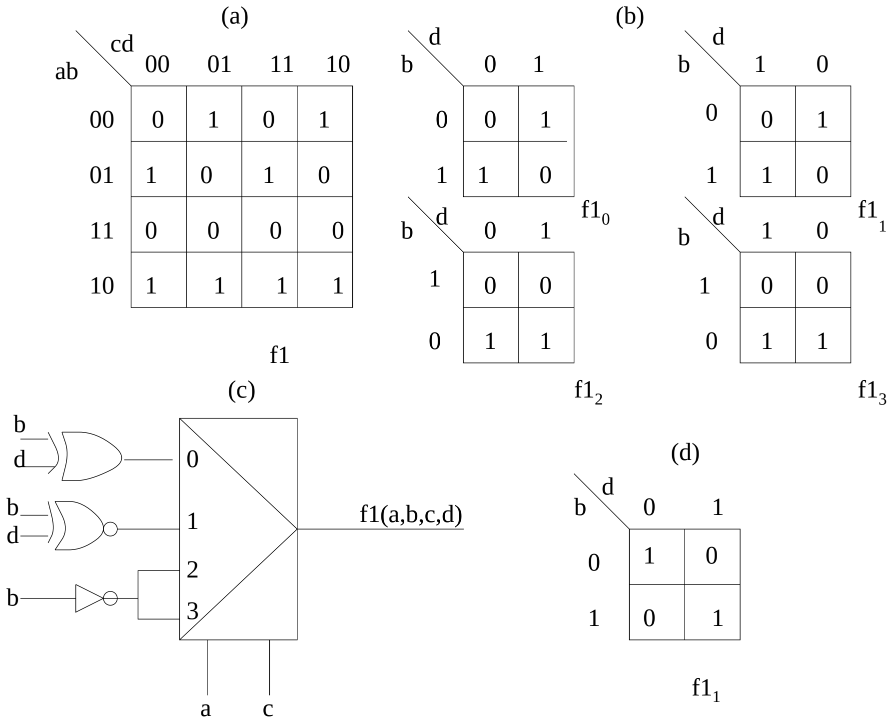


Figure 1.6: Realization with address variables a, c

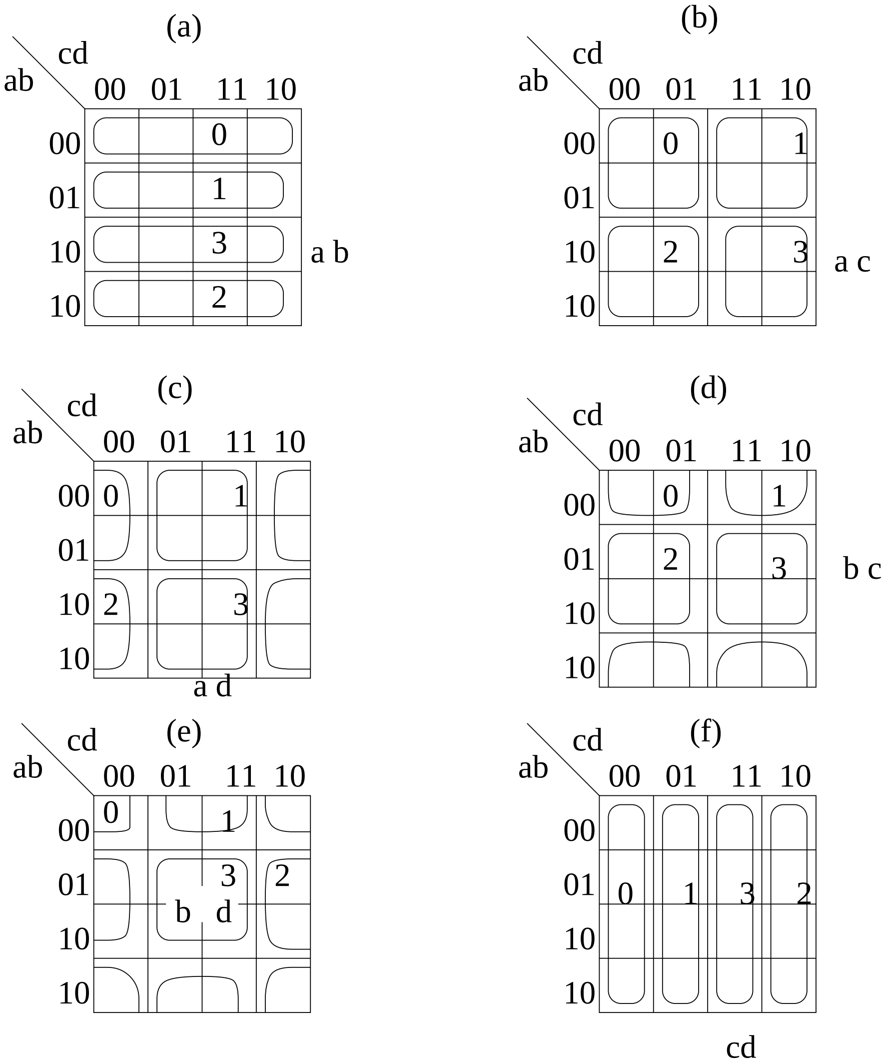


Figure 1.7: Patterns of cofactors for various combinations of variables; (a) variables ab, (b) ac, (c) ad, (d) bc, (e) bd, (f) cd

1.3 Basic Patterns for Decompositions.

There exist only three types of decomposition in the literature that are essentially distinct and that make use of the concept of partitioning of input variables to *bound and free sets*:

- Curtis Decomposition [?],
- Steinbach et al (XBOOLE) Decomposition [?],
- Perkowski/Uong/Brown (PUB) Decomposition [?, ?, ?].

The decompositions of Ashenhurst [?, ?, ?]; Luba et al [?]; Lai, Pedram et al [?], and many other are just special cases, or different realizations, of the Curtis decomposition [?]. While most authors differentiate between disjoint and non-disjoint decompositions, the concept of the Repeated Variable Maps (RVMs) allows to explain them in a uniform way, and the CDBs allow to realize all these decompositions uniformly in software (including those from [?, ?, ?, ?]).

In RVM, the rows of the map correspond to the Row Variables, and the columns correspond to the Column Variables. As we see, the Row Variables can be represented as $A \cup C$, and the Column Variables can be represented as $B \cup C$. Using Curtis terminology, set $B \cup C$ is a *bound set*, and set $A \cup C$ is a *free set*. If $C = \emptyset$ the decomposition is disjoint and the RVM becomes a standard Karnaugh Map. If $C \neq \emptyset$ the decomposition is non-disjoint and the RVM is incompletely specified, even if the original function is completely specified. Every variable in C is called *a repeated variable*. Let us observe, that every repeated variable creates a map of one dimension higher, in which all newly introduced cells are don't cares. For instance, if the original map is completely specified and has 4 variables, a, b, c, d, the bound set is $\{a, c, d\}$ and the variable a is a repeated variable, the new 4 * 8 map will have three variables for columns and two variables for rows (variable a stands in both rows and columns). Half of the RVM are don't cares. If variables a and c were repeated, and $\{a, c, d\}$ were a bound set, the new 8 * 8 map will have three variables for columns, and three variables for rows. It will have 75% of don't cares. As we see, even starting with a completely specified function, by repeating variables, very quickly one has to deal with very strongly unspecified functions. In addition, in ML applications, even the initial data can have more than 99.99% of don't cares. It is than absolutely crucial to be able to represent and manipulate such functions efficiently.

The main observation of our unified and generalized approach is the observation that all decompositions [?, ?, ?, ?, ?, ?] use certain fundamental patterns in cofactors. These patterns can be easily observed in rows and columns of the RVM. Let us remind that both the rows and the columns of the RVM correspond to cofactors with respect to cubes on literals created from row and column variables, respectively.

Let us concentrate in this section on binary-valued-output multi-valued input functions.

We will distinguish the following patterns in cofactors:

1. Pattern of don't cares. We will call it the DC Pattern.
2. Pattern of ones (and possibly don't cares). We will call it the ON Pattern.
3. Pattern of zeros (and possibly don't cares). We will call it the OFF Pattern.
4. Pattern of function F with zeros, ones, (and possibly don't cares). We will call it the F Pattern.
5. Pattern of function $\overline{F}$ with zeros, ones, (and possibly don't cares) We will call it the $\overline{F}$ Pattern.
6. Pattern being either the DC Pattern or the ON Pattern. We will call it the DC/ON Pattern.
7. Pattern being either the DC Pattern or the OFF Pattern. We will call it the DC/OFF Pattern.

Similarly other combined patterns of DC, ON, OFF, F, and $\overline{F}$ can be defined. Let us observe that if a function has in every column either a DC/ON pattern or a DC/OFF pattern, then it is independent on the variables from the free set.

Analogically, if a function has in every row either a DC/ON pattern or a DC/OFF pattern, then it is independent on the variables from the bound set. Let us observe, that some column can be characterized as having either an ON pattern or an F pattern. There exist more characteristic patterns that we do not discuss here, and all possible decomposition methods are based on finding these patterns in functions. Other patterns and their uses will be discussed in our forthcoming reports.

1.3.1 Definitions of Patterns

Definition 1.3.1 Row OR decomposition *exists with respect to the set of row variables RV if there exists at least one row that has the ON Pattern.*

Definition 1.3.2 Row AND decomposition *exists with respect to the set of row variables RV if there exists at least one row that has the OFF Pattern.*

Let us observe that in the above two cases, patterns OR and AND can be found immediately, without comparing them to other patterns of rows.

Definition 1.3.3 Row EXOR decomposition *exists with respect to the set of row variables RV if all patterns for rows can be Patterns F and $\overline{F}$.*

Let us observe, that in this case every DC, ON and OFF row must be here characterized as either an F or $\overline{F}$ pattern, if possible. This case is then more difficult than the first two and requires a global comparison/minimization procedure.

Definition 1.3.4 Column OR decomposition *exists with respect to the set of column variables RV if there exists at least one column that has the ON Pattern.*

Analogically one can define the **Column AND decomposition** and the **Column EXOR decomposition**. Row and Column decompositions are also called Weak Decompositions [?]. There exist then **Weak AND, Weak OR**, and **Weak EXOR** decompositions.

Definition 1.3.5 Strong OR decomposition *exists with respect to a set of row variables RV and a set of column variables CV if there exists Row OR Decomposition, and next, after replacing the ON rows with don't cares, every column has either a ON/DC or a OFF/DC pattern.*

Definition 1.3.6 Strong OR decomposition *exists with respect to a set of column variables CV and a set of row variables RV if there exists Column OR Decomposition, and next, after replacing the ON columns with don't cares, every row has either a ON/DC or a OFF/DC pattern.*

In other words a strong OR decomposition exists if and only if there is a set of ON/DC rows and columns that covers all ones in the table and covers no zeroes. In yet another words a strong OR decomposition exists if and only after removing ones from the decomposition with respect to rows (columns) the remaining function has all row (column) variables as vacuous.

Definition 1.3.7 Strong AND decomposition *exists with respect to a set of row variables RV and a set of column variables CV if there exist Row AND Decomposition, and next, after replacing the OFF rows with don't cares, every column has either a ON/DC or a OFF/DC pattern.*

Definition 1.3.8 Strong AND decomposition *exists with respect to a set of column variables CV and a set of row variables RV if there exist Column AND Decomposition, and next, after replacing the OFF columns with don't cares, every row has either a ON/DC or a OFF/DC pattern.*

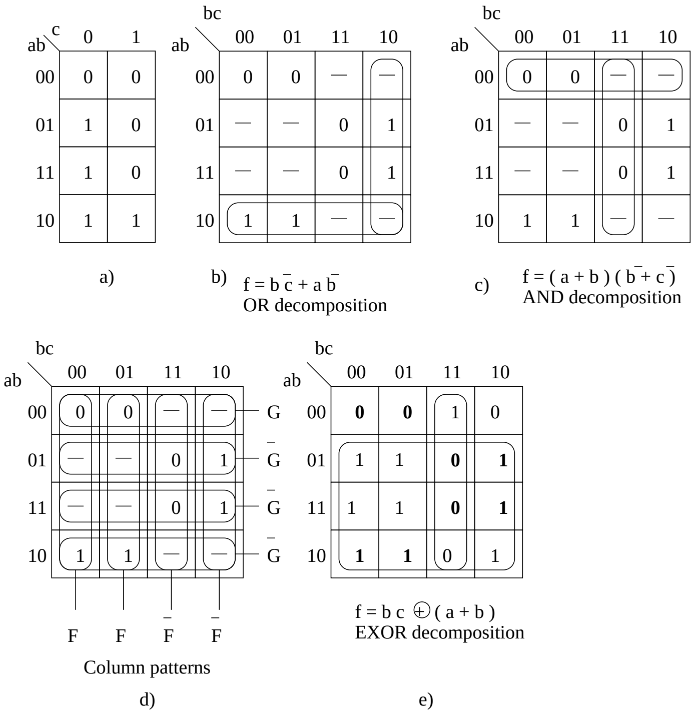


Figure 1.8: Basic patterns of OR, AND and EXOR decompositions

In other words a strong AND decomposition exists if and only if there is a set of OFF/DC rows and columns that covers all zeroes in the table and covers no ones.

Definition 1.3.9 Strong EXOR decomposition *exists with respect to a set of row variables RV and a set of column variables CV if there exist Row EXOR Decomposition, and Column EXOR Decomposition.*

In other words, a strong EXOR decomposition exists if and only if there is a set of rows and columns such that each 0 is covered by an even number of rows/columns and each 1 is covered by an odd number of rows/columns.

Definition 1.3.10 Strong OR/AND decomposition *exists with respect to a set of row variables RV and a set of column variables CV if there exist ON Patterns of rows, and next, after replacing the ON rows with don't cares, there exists a Strong AND decomposition.*

Definition 1.3.11 Strong OR/AND decomposition *exists with respect to a set of column variables CV and a set of row variables RV if there exist ON Patterns of columns, and next, after replacing the ON columns with don't cares, there exists a Strong AND decomposition.*

There are several other complex patterns of this type. AND, OR and EXOR decompositions will be called the **Basic Gate Decompositions**. OR/AND, AND/OR and other of this type will be called the **Complex Gate Decompositions**.

Example 1.3.1

An example of the RVM is shown in Figure 1.8. Figure 1.8a presents a standard Kmap of 3-input function f. Assuming b to be a repeated variable, the Bound Set $\{b,c\}$ (the columns) and the Free Set as $\{a,b\}$, one creates a RVM from Figure 1.8b. Let us observe that ON Patterns $b\ \overline{c}$ and $a\ \overline{b}$ exist in this RVM, which lead to Strong OR Decomposition: $f = b\ \overline{c} + a\ \overline{b}$.

Similarly, for the same RVM in Figure 1.8c, the OFF Patterns in row $(\bar{a}\ \bar{b})$ and column $(b\ c)$ are found, which leads to the Strong AND Decomposition: $(a + b) \cdot (\overline{b} + \overline{c})$. Finally, for the same RVM in Figure 1.8d, the Column Patterns F, and $\overline{F}$ and the Row Patterns G, and $\overline{G}$ are found as shown with loops on the map in Figure 1.8d. These patterns lead to Strong EXOR Decomposition: $(b\ c) \oplus (a + b)$ from Figure 1.8e. Figure 1.8e clearly shows the incomplete patterns from Figure 1.8d after their completion with 0's and 1's. Bold symbols correspond to original cares from the RVM. Thin symbols correspond to the specification of the cares created by the decomposition. As we see, an overlap of an even number of group (in this case, two or none) creates a **0**, and an odd number of groups creates a **1**.

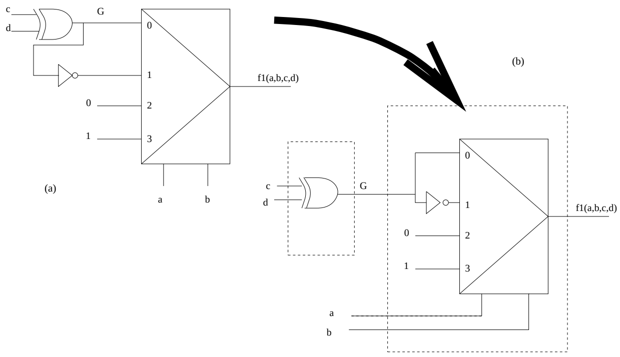


Figure 1.9: Explanation of the concept of Ashenhurst decomposition for function $f1(a, b, c, d)$: (a) using a multiplexer and an inverter gate, (b) a corresponding block diagram of the Ashenhurst decomposition

1.4 Ashenhurst Decompositions

1.4.1 Disjoint Ashenhurst Decomposition of Completely Specified Functions.

Example 1.4.1

Let us first observe that function $f1_0(c, d)$ in Fig. 1.4 is the negation of function $f1_1(c, d)$. Therefore, the realization from Fig. 1.4 can be re-drawn as shown in Fig. 1.9.

Next, the function from Fig. 1.9 can be represented in a block diagram form as in Fig. 7.2. This figure represents a general scheme of the classical (simple) *disjunctive decomposition* of a single output Boolean function with the *set of bound variables* {c, d} and the *set of free variables* {a, b}. Such decompositions are discussed, among others, in [?].

At this point the question arises:

"How to find from the function's K-map whether this function is decomposable for the set of address variables selected".

It is obvious from Fig. 1.9 that if we want to have a single connection from the predecessor to the successor block then all of the data inputs to the multiplexer have to be a constant 0, a constant 1, an output of the predecessor, or a negation of the predecessor (the negation is created inside of the successor block). This is basically the proof of the Ashenhurst theorem [?].

The well-known Ashenhurst theorem can now be expressed as follows:

The function $f(X)$ is decomposable with free set A and bound set B if each of the subfunctions $f_i(A)$, called *cofactors of the free set*, is one of at most four following types:

- constant 0,
- constant 1,
- some function, call it $G(B)$,
- negation $\overline{G(B)}$ of this said function $G(B)$.

Let us observe, that the above condition is equivalent to the following:

The function $f(X)$ is decomposable with free set A and bound set B if each of the subfunctions $f_i(B)$, i.e the cofactors of the bound set, is at most one of the following types:

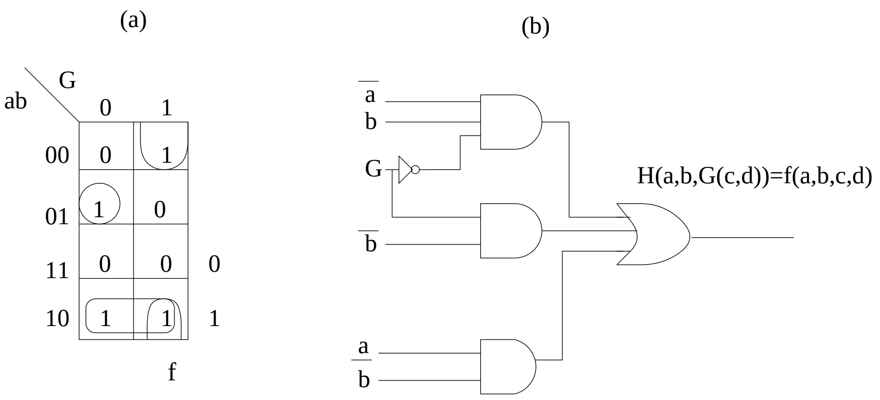


Figure 1.10: The SOP realization of the image function H to Example 6.2

- some function, call it $K(B)$,
- some other function, call it $L(B)$,

If one draws a K-map with the set of colums corresponding to the bound set variables, the functions $K(B)$ and $L(B)$ will be called the *column functions*, and the number of different column functions will be called the *column multiplicity index*. Analogously, for a given map, one can define the *row multiplicity index*. Function is Ashenhurst-decomposable with the bound set B if the column multiplicity index is equal to two.

Let us now observe that the decomposition process can be performed graphically with use of K-maps even faster than the process of design with multiplexers. First, all the loops are drawn for the selected free variables, as previously. Next, we discard all the loops that include only zeros or only ones. Next, we denote one of the remaining loop functions as G. When all the remaining loops describe functions that are either G or $\overline{G}$, we have found a decomposition. If we find at least one function that is not a G and not a $\overline{G}$ either - then the Ashenhurst decomposition *does not* exist for the selected free variables. We can often quickly establish that the Ashenhurst decomposition for a given set of free variables **does not exist** without checking too many loops for this set. Therefore with some experience the designer can quickly scan the loops to find the **nonexistence** of an Ashenhurst decomposition. He does this subsequently for all other sets of free variables. There is no need to redraw the sub-K-maps. It can be more time consuming to find the **existence** of the decomposition. For K-maps with many variables it may require re-drawing of some submaps.

Example 1.4.2

Let us go back to function $f1(a,b,c,d)$ from Example 1.4.1. We assume address variables a, b. Loop $a\,\overline{b}$ of ones and loop $a\,b$ of zeros are discarded (it means, not considered further), so that only loops $\overline{a}\,\overline{b}$ and $\overline{a}\,b$ remain. Let us assume that $G = f1_0$. Now it is easy to check that $f1_1 = \overline{G}$ and the theorem is therefore satisfied. The SOP realization of the image function H is shown in Fig. 1.10a,b.

The K-map of successor is created from the K-map of Fig. 7.3, G, and $\overline{G}$, as in Fig. 1.10a. This function is minimized using two-level Boolean minimization to the circuit from Fig. 1.10b.

Example 1.4.3

For the same function let us now select variables c and d as the address variables (Fig. 1.5a). No loops can be discarded. Assuming $G = f_0(a,b)$ in column 0 (loop $\overline{c}\,\overline{d}$) we immediately see that function from loop $\overline{c}\,d$ is neither G nor $\overline{G}$, so no decomposition exists with c and d as free variables.

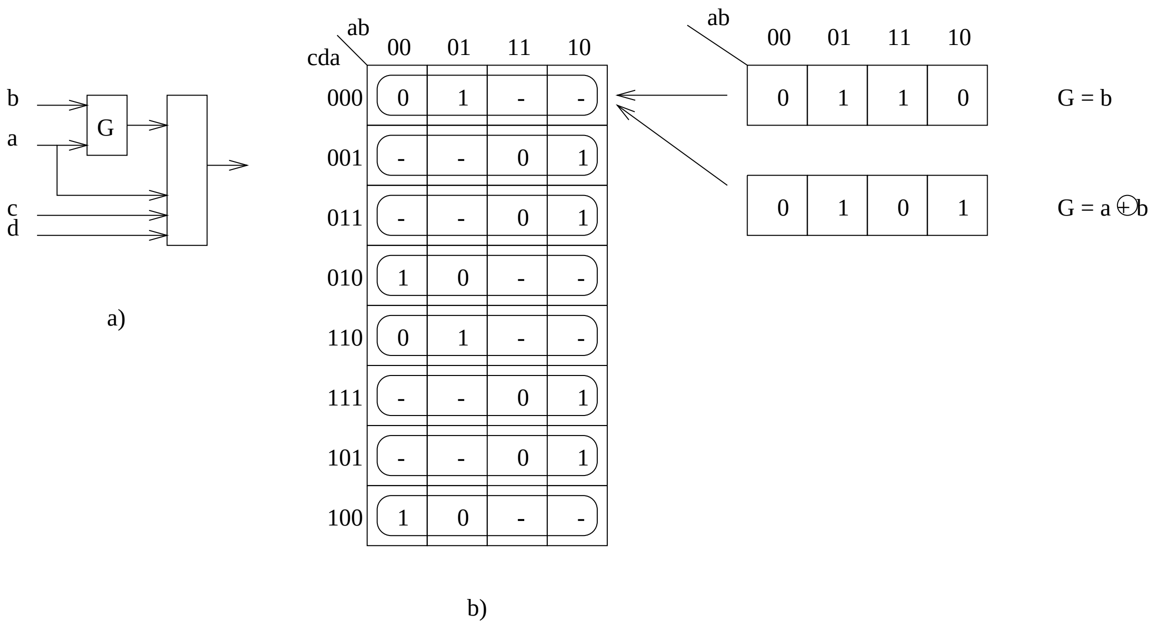


Figure 1.11: An example of a non-disjoint decomposition using a K-map to Example 6.3

Let us observe that if the decomposition exists the designer has a choice to implement each function either with a multiplexer for successor and efficient realization of data input functions, or successor can be realized using any other Boolean minimization method, for instance as a PLA, minimized with a two-level Boolean minimization program (as in Example 6.1). If the decomposition does not exist, the results of the above cofactor comparison are still useful for efficient design with multiplexers. Such designs can be found when simple cofactor functions exist, and only few types of cofactors exist (say, half of data inputs to a multiplexer - see section ??). Such functions are more likely to be found when the function is strongly unspecified.

1.4.2 Non-disjoint Ashenhurst Decomposition of Completely Specified Functions.

Example 1.4.4

Let us again consider the function from Fig. 7.3a with variables c and d forming free set $A = \{c, d\}$ and variables a and b forming bound set $B = \{a, b\}$. For such defined free and bound sets no disjoint Ashenhurst decomposition exists. However, function may be decomposed using the **non-disjoint** Ashenhurst decomposition technique. Let us consider decomposition scheme showed in Fig. 1.11a where variable a belongs to both free and bound sets $a = A \cap B$ and $A = \{c, d, a\}$, $B = \{a, b\}$.

K-map for this scheme is shown in Fig. 1.11b. Don't cares in that map correspond to situations which never happen. For instance, in loop $\overline{c}\,\overline{d}\,\overline{a}$ situation where bound set $a = 1$ never happens because the loop definition assumes free set $a = 0$ and, a is the same variable for both free and bound sets. Let us denote the loop function $\overline{c}\,\overline{d}\,\overline{a}$ as G. To make the function decomposable, we have to fill up DCs in such a way that each loop function be either equal to G or $\overline{G}$. That condition is met for $G = b$ ($A = \{c, d, a\}$ and $B = \{b\}$) and $G = a \oplus b$ ($A = \{c, d, a\}$, $B = \{a, b\}$).

Fig. 1.12 shows a different way of using K-map to make the same decomposition. Decomposition scheme from Fig. 1.12a is equivalent to the scheme from Fig. 1.11a. K-map is shown in Fig. 1.12b. The loops drawn in K-map correspond to free set variables $A = \{c, d, a\}$.

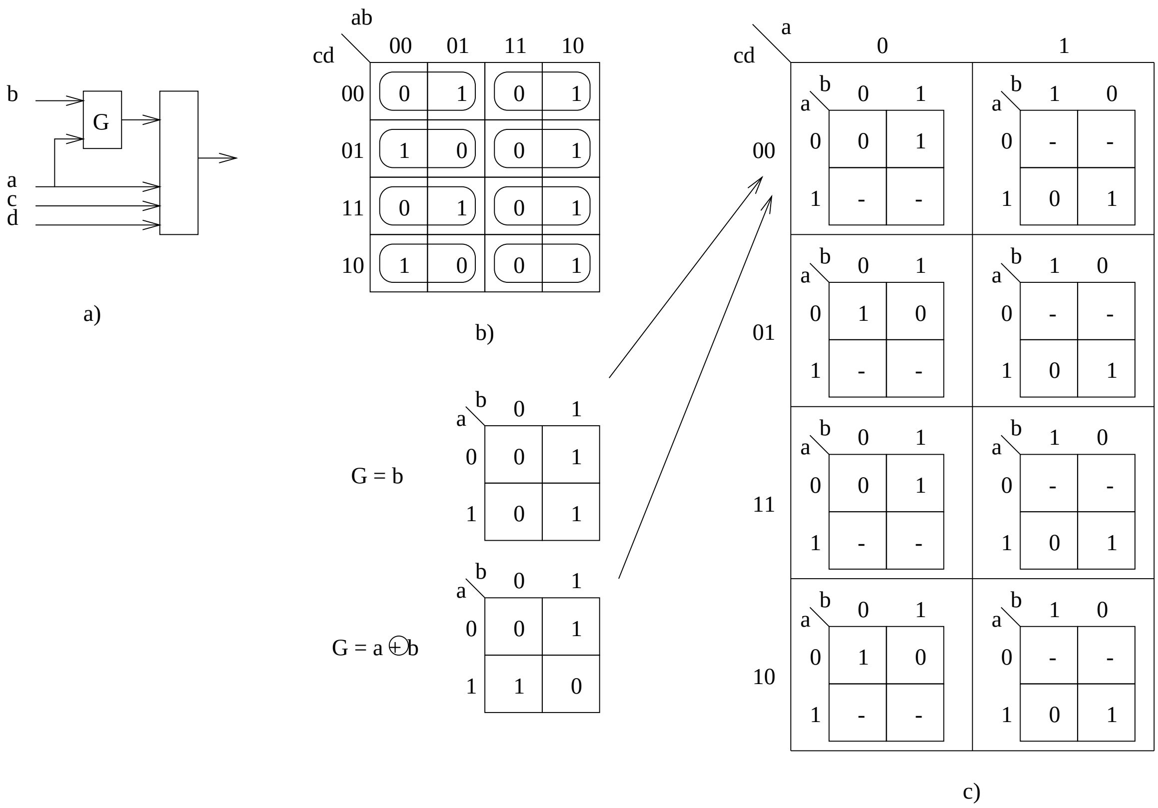


Figure 1.12: A different way of using a K-map to make the non-disjoint decomposition from Fig. 13

To form K-maps for loop functions however, we procede in a way which is slightly different from the one used in the former examples. K-map for the loop function $\overline{c}\,\overline{d}\,\overline{a}$ for instance, contains DCs for $a = 1$ (for $a = 0$ from the loop definition) and whatever is in the loop, for $a = 0$. The K-maps for all the loops are shown in Fig. 1.12c. If the DCs in these K-maps may be filled up in such a way that we end up with a function G (and possibly $\overline{G}$) then the Ashenhurst decomposition exists. In our case we may fill up DCs to get $G = b$ ($A = \{c, d, a\}$, $B = \{b\}$) or $G = a \oplus b$ ($A = \{c, d, a\}$, $B = \{a, b\}$).

1.5 Ashenhurst Decomposition of Incompletely Specified Functions

1.5.1 Disjoint Ashenhurst Decomposition of Incompletely Specified Functions

Let us observe that our graphical method can be, with very little modification, applied to incompletely specified functions as well. First we remove loops that can be completed to all zeros, or that can be completed to all ones. Next we check whether the remaining loops can be *completed* to functions G and $\overline{G}$.

Example 1.5.1

abc \ def	000	001	011	010	110	111	101	100	
000	1	-	1	0	1	0	-	0	loop $\overline{a}\,\overline{b}$
001	0	1	0	-	-	-	0	1	
011	0	-	-	0	0	-	-	-	loop $\overline{a}$ b
010	-	0	-	0	0	-	0	0	
110	1	-	-	1	1	-	-	-	loop a b
111	-	1	1	-	-	-	1	-	
101	0	1	0	-	0	1	-	1	loop a $\overline{b}$
100	-	0	1	0	1	-	-	0	

f2

Figure 1.13: Function $f2(a, b, c, d, e, f)$ to Example 7.1

Given is function $f2(a, b, c, d, e, f)$ from Fig. 1.13.

Assuming address variables a and b, the loops are as shown in Fig. 1.13. We can immediately discard loop $\overline{a}\ b$ that can be completed to all zeros, as well as the loop $a\ b$ that can be completed to all ones. To help the reader, the loops functions $\overline{a}\ \overline{b}$ and $a\ \overline{b}$ are rewritten to those of Fig. 1.14. It can be seen that both of them can be completed to the same function

$G = \overline{c}\ \overline{d}\ \overline{e}\ \overline{f} + c\ \overline{d}\ \overline{e}\ f + \overline{c}\ \overline{d}\ e\ f + c\ \overline{d}\ e\ \overline{f} + \overline{c}\ d\ e\ \overline{f} + c\ d\ e\ f + \overline{c}\ d\ \overline{e}\ f + c\ d\ \overline{e}\ \overline{f} = c \oplus d \oplus e \oplus \overline{f}$

It is important to note that with some experience one does not have to draw all the loops in the K-map, but, remembering their shapes, can verify directly the existence of functions G and $\overline{G}$.

Also, when checking the decomposition possibility (decomposability) for free (bound) sets, it is often not neccessary to generate loops systematically for all the possible free (bound) sets. Since we know that the existence of loops of zeros and loops of ones for some free set increases the possibility of a decomposition, we shall first find a large loop of zeros or large loop of ones, or a loop that can be completed to a loop of one of these types. Next we shall test the decomposition possibility for other loops created for the free set of the variables from this loop. If no decomposition exists for this free (bound) set, the decomposition for the next large loop of the above types is investigated.

Example 1.5.2

We will solve the example from p. 171 in [?].

$f(v, w, x, y, z) = \sum(3, 4, 5, 7, 9, 15, 17, 21, 22, 28, 29) + \sum_{\emptyset}(10, 13, 14, 18, 23, 27, 30, 31)$

The K-map for this function is shown in Fig. 1.15.

c \ def	000	001	011	010	110	111	101	100
0	1	-	1	0	1	0	-	0
1	0	1	0	-	-	-	0	1

c \ def	000	001	011	010	110	111	101	100
0	-	-	1	0	1	-	-	0
1	0	1	0	-	0	1	-	1

Figure 1.14: Loops $\bar{a}\bar{b}$ and $a\bar{b}$ from Fig. 15 redrawn to another form

vw \ xyz	000	001	011	010	110	111	101	100
00	0	0	1	0	-	1	1	1
01	0	1	0	-	-	1	-	0
11	0	0	-	0	-	-	1	1
10	0	1	0	-	1	-	1	0

Figure 1.15: Function for Example 7.2

vw \ xyz	000	001	011	010	110	111	101	100
00	0	0	1	0	-	1	1	1
01	0	1	0	-	-	1	-	0
11	0	0	-	0	-	-	1	1
10	0	1	0	-	1	-	1	0

Figure 1.16: A loop $x\ z$ as a loop that can be completed to 1's is found in the function from Fig. 17

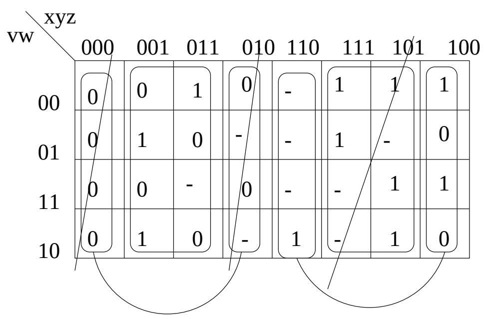


Figure 1.17: The remaining loops are found for function from Fig. 18

First we try to find large loops that can be completed to loops of all zeros or to loops of all ones. A loop $x\ z$ is found as a loop that can be completed to ones - see Fig. 1.16. This suggests selecting the set $\{x, z\}$ as a free set. By examining the other loops for this set we find that loop $\overline{x}\ \overline{z}$ can be completed to all zeros.

Now the remaining loops are found and compared (Fig. 1.17). As we see, assuming that loop $x\ \overline{z}$ corresponds to G, we notice that the loop $\overline{x}\ z$ will correspond to $\overline{G}$. The completion process is shown in Fig. 1.18a. The realization of function $G(v, w, y)$ is shown in Fig. 1.18d. Operator $\diamond$ denotes the merging (common minimal completion) operation of functions. Operator $\neg$ negates the function that follows it.

Therefore the decomposition for free set $\{x, z\}$ is found, as shown in Fig. 1.18a. Fig. 1.18b, c presents two methods for realization of function $f(x, z, G)$. The final circuit is presented in Fig. 1.18e.

1.5.2 Non-disjoint Ashenhurst Decomposition of Incompletely Specified Boolean Functions.

See section 12.4, Example 1.4.4.

1.5.3 Non-disjoint Ashenhurst Decomposition of Incompletely Specified Multiple-Valued Functions.

To determine whether a function is decomposable we may use MV extension of the Ashenhurst theorem. Ashenhurst theorem for binary functions states that a function is decomposable if the column multiplicity of its K-map is less then or equal to two. Multiple valued logic function extension of this theorem states that column multiplicity must be less than or equal to n, where n is the number of values the function can take (for binary function $n = 2$).

Example 1.5.3

Multiple-valued function (three possible values: 0, 1, 2) which is defined in Fig. 1.19 is Ashenhurst decomposable because the column multiplicity is equal to three (00,02,12 ; 10,01,21 ; 20,11,22).

Incompletely specified (with "don't cares") MV function however, requires MV don't care definition.

In the case of binary function, DC in the function output means that the output is either 0 or 1. For MV function there are two ways of generalizing the concept of DC. One is, that *any* possible output value of the function may be taken (as for binary function). The other is that any subset of more than one value may be taken. For instance, in three-valued logic, the logic values are 0, 1, and 2, and the generalized don't cares are:

- {0,1} - function may take either 0 or 1 value

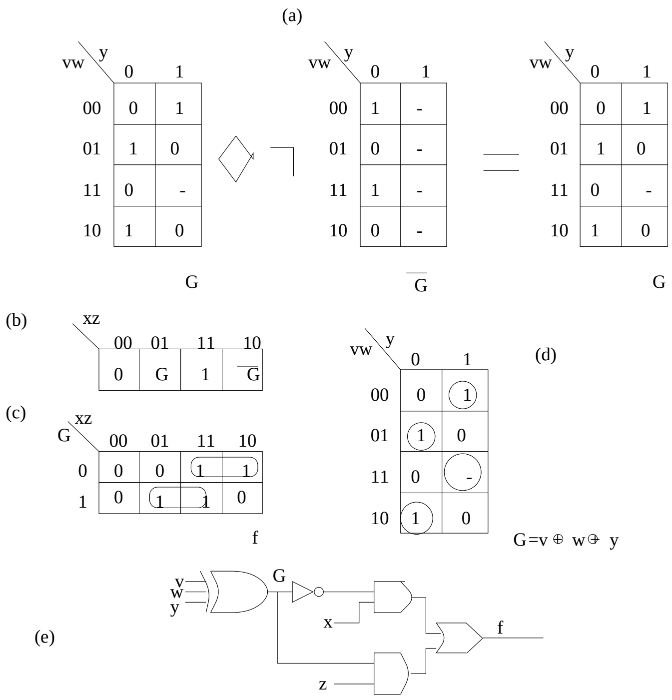


Figure 1.18: The decomposition for free set $\{x, z\}$

a \ bc	00	10	20	01	11	21	02	12	22
0	0	1	2	1	2	1	0	0	2
1	0	0	2	0	2	0	0	0	2
2	2	2	1	2	1	2	2	2	1

y

a, b, c, y = {0, 1, 2}

Figure 1.19: A multiple-valued function that is Ashenhurst decomposable in Example 7.3

ab \ bc	00	10	20	01	11	21	02	12	22
00	0			1			0		
10	0			0			0		
20	2			2			2		
01		1			2			0	
11		0			2			0	
21		2			1			2	
02			2			1			2
12			2			0			2
22			1			2			1

y

a, b, c, y = {0, 1, 2}

Figure 1.20: A Repeated Variable Map for function from Example 7.3 with repeated variable b

- {1,2} - function may take either 1 or 2 value
- {0,2} - function may take either 0 or 2 value
- {0,1,2} - function may take any out of 0, 1, 2 values

As an example, let us again consider the function from Fig. 1.19 with DC of type {0, 1} in the field 022 (a=0, b=2, c=2). Since this field can only take 0 or 1 value the function in not decomposable (column multiplicity is equal to four). However, if we put DC of any other type (containing 2) in the same field, the function will become decomposable.

Non-disjoint Ashenhurst decomposition is the one having non disjoint bound (B) and free (A) sets. If we choose $A = \{a, b\}$ and $B = \{b, c\}$ for the function from Fig. 1.19 we will come out with K-map in Fig. 1.20. The empty fields are DC fields of type {0, 1, 2}, which means that they may be replaced by 0, 1 or 2 value. Since the disjoint decomposition exists, the non-disjoint decompositions exist as well. A non-disjoint decomposition, however, is usually used only to decompose functions which are not disjoint decomposable.

ab \ cde	000	001	011	010	110	111	101	100
00	-	0	0	0	0	-	1	1
01	1	1	-	-	0	0	0	-
11	1	1	1	1	-	-	0	0
10	-	-	0	-	0	1	-	1

Figure 1.21: A Karnaugh map to Example 8.1

1.6 Introduction to Analytical Methods for Column Minimization for Incompletely Specified Single-Output Boolean Functions.

The analytical method presented below serves to explain the computer algorithms from the next sections. Our goal is to find a formal method to check if the completion to G and $\overline{G}$ exists for a given free set A.

Definition 1.6.1 *By a loop pair we will understand a pair:*

$$[loop, respective \quad loop \quad function] = [loop, f(loop)]$$

Definition 1.6.2 *Two functions are called the* compatible functions *when they can be completed to the same function.*

Definition 1.6.3 *Two loop pairs $p_1 = [l_1, f(l_1)]$ and $p_2 = [l_2, f(l_2)]$ are called* compatible loop pairs *when $f(l_1)$ and $f(l_2)$ are compatible functions.*

Definition 1.6.4 *The operation of merging, with two functions f_1 and f_2 as arguments, produces the least defined function $f = f_1 \diamond f_2$ that completes both of them by replacing dont-cares with zeros or ones.*

This is done on a cell-by-cell basis: If a cell has the same value in both functions, this value is retained, don't care and cell will give the value of this cell: $1 \diamond 1 = 1$, $0 \diamond 0 = 0$, $1 \diamond - = 1$, $0 \diamond - = 0$. If the respective cells have values 1 and 0, the merging cannot be done (we denote this by $1 \diamond 0 = \epsilon$) and we say that functions f_1 and f_2 are *not compatible.*

Definition 1.6.5 *Two loop pairs $p_1 = [l_1, f(l_1)]$ and $p_2 = [l_2, f(l_2)]$ are* merged *to a loop pair $[\{l_1, l_2\}, f(l_1) \diamond f(l_2)]$ if $f(l_1)$ and $f(l_2)$ are compatible functions.*

The reason to introduce loop pairs is to allow performing operations on the respective loop functions but keeping track of what loops are the merged functions coming from.

Let us first observe that the relation of *loop pair compatibility* is not an equivalence relation.

Example 1.6.1

Let us consider a function given in a Karnaugh map of Fig. 1.21, assuming $B = \{c, d, e\}$.

The bound set $B = \{c, d, e\}$, so the loop functions correspond to the columns of the map. After removing the loop $\overline{c}\ \overline{d}\ \overline{e}$ of 1's and the loop $c\ d\ \overline{e}$ of 0's, the *loop pair Compatibility Graph* of Fig. 1.22 is created with the remaining loop pairs as nodes.

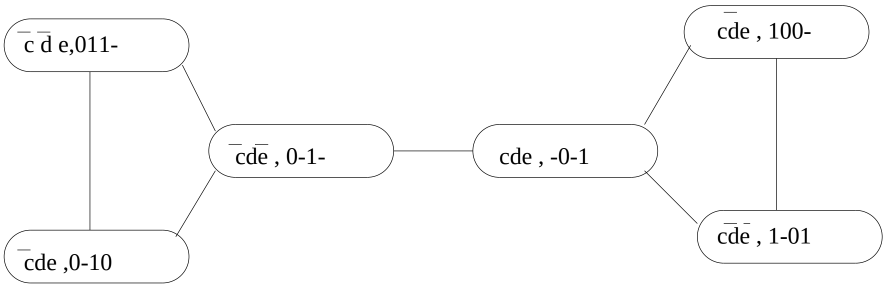


Figure 1.22: A loop compatibility graph to Example 8.1

Let us observe, that one can create a Compatibility Graph for the nodes found from the free set and for nodes found from the bound set. We will call them, the *Free Set Compatibility Graph*, and the *Bound Set Compatibility Graph*, respectively.

The compatibility of two loop pairs in the graph is denoted by an edge between the corresponding nodes. Let us for instance compare loop functions for columns 011 and 010 (loops $\overline{c}\ d\ e$ and $\overline{c}\ d\ \overline{e}$). The loop function for a column will be denoted in the current section, for the sake of explanation, as the string of symbols 0, 1, and - , in the same order, as they are in the column from the top to the bottom. The computer representation of loop functions will be discussed in the next section, when the computer program will be presented. The loop function for column 011 (loop $\overline{c}\ d\ e$) is 0-10, and the loop function for column 010 is 0-1-. These loop functions are compatible. This is denoted in cube calculus as: 0-10 $\diamond$ 0-1- = 0-10. By $\diamond$ we denote the merging (common minimal completion) operation of functions represented as arrays of cubes. The respective loop pairs are then compatible as well:

[$\overline{c}\ d\ e$, 0-10] [$\overline{c}\ d\ \overline{e}$, 0-1-] = [$\overline{c}\ d\ e$, $\overline{c}\ d\ \overline{e}$, 0-10].

Compatibility of loop pairs is denoted as an edge between nodes

[$\overline{c}\ d\ e$, $f(\overline{c}\ d\ e)$] and [$\overline{c}\ d\ \overline{e}$, $f(\overline{c}\ d\ \overline{e})$].

In our notation, the nodes: [$\overline{c}\ d\ e$, 0-10] and [$\overline{c}\ d\ \overline{e}, 0-1-$] (see Fig. 1.22).

Similarly, loop pairs [$\overline{c}\ d\ \overline{e}$, 0-1-] and [$c\ d\ e$, -0-1] are compatible, since 0-1- $\diamond$ -0-1 = 0011. However loop pairs [$\overline{c}\ d\ e$, 0-10] and [$c\ d\ e$, -0-1] are not compatible since 0-10 $\diamond$ -0-1 = 001ϵ (by ϵ we denote the incompatible 0 and 1 in the lowest row of the K-map).

Therefore, the relation of compatibility is not transitive and hence it is not an equivalence relation.

Let us observe, that for a completely specified function the relation of compatibility is an equivalence relation, which makes the whole problem much easier.

Using the method shown above one can create the loop pair compatibility graph from Fig. 1.22. The next stage is to find *a partition of this graph to two maximum cliques.* A *clique* of a graph is a subgraph with any two nodes linked by an edge (a complete graph). *The maximum clique* is a clique with the maximum number of nodes (a clique that is not included in another clique). If after merging the loop functions from the first clique we would create function G then the merging from the another clique should be $\overline{G}$, otherwise there is no decomposition. The Ashenhurst decomposition of the function does not exist in one of the following two cases:

- there is more than two cliques in the minimal partition in row (column) compatibility graph.
- $\diamond_{c_i \in C_1} c_i \neq \overline{\diamond_{c_j \in C_2} c_j}$ in column (row) compatibility graph.

  i.e. the two mergings from the cliques C_1 and C_2 can not be completed to functions G and $\overline{G}$ being the mutual negations.

The fact that the relation of compatibility is not an equivalence relation makes the search for maximum cliques more difficult.

We can create a simple sequential clique-growing algorithm in which the two cliques are extended step-by-step, starting from any loop pair. For instance, by selecting first the loop pair [$\overline{c}\ \overline{d}\ e$, 011-] and next the pair [$\overline{c}\ d\ e$, 0-10] we create a merged loop pair [$\overline{c}\ \overline{d}\ e$, $\overline{c}\ d\ e$, 0110]. Now [$\overline{c}\ d\ \overline{e}$, 0-1-] is selected and we join it to this loop pair: [$\overline{c}\ \overline{d}\ e$, $\overline{c}\ d\ e$, $\overline{c}\ \overline{d}\ e$, 0110]. Now we select [$c\ d\ e$, -0-1]. This cannot be merged with the previous merged loop pair, so is becomes a beginning of the new merged loop pair.

Next we select [$c\ \overline{d}\ \overline{e}$, 1-01], creating the merged loop pair [$c\ d\ e$, $c\ \overline{d}\ \overline{e}$, 1001]. When next the pair [$c\ \overline{d}\ e$, 100-] is selected it is joined to the last merged loop pair, creating a pair: [$c\ d\ e$, $c\ \overline{d}\ \overline{e}$, $c\ \overline{d}\ e$, 1001]. After scanning all loop pairs two merged loop pairs: [$\overline{c}\ \overline{d}\ e$, $\overline{c}\ d\ e$, $\overline{c}\ \overline{d}\ e$, 0110] and [$c\ d\ e$, $c\ \overline{d}\ \overline{e}$, $c\ \overline{d}\ e$, 1001] have then been created. They are mutual negations, so the respective decomposition exists.

Let us however observe, that this method would not work for another order of selection. By selecting first the loop pair [$\overline{c}\ d\ \overline{e}$, 0-1-] and next the pair [$c\ d\ e$, -0-1] we would create the pair [$\overline{c}\ d\ \overline{e}$, $c\ d\ e$, 0011].

Now we select loop pair [$\overline{c}\ \overline{d}\ e$, 011-] and we see that the loop functions, 011-, from this pair, and 0011 from the pair [$\overline{c}\ d\ \overline{e}$, $c\ d\ e$, 0011] cannot be mutual negations and cannot be merged either. For this order of loop pairs selection the decomposition was then not found, even if it exists, as we have found for another order.

The problem of finding a maximum clique partitioning is in general an NP-hard one, but we will propose below an exact and polynomial algorithm that gives very good results for 2-coloring.

1.6.1 Exact Algorithm for Ashenhurst Decomposition.

Existence of Ashenhurst Decomposition can be checked either using Property 1, or Property 2.

Property 1. Ashenhurst Decomposition with bound set B and free set A exists if all row patterns are: ON Pattern, OFF Pattern, F Pattern and $\overline{F}$ Pattern.

Property 2. Ashenhurst Decomposition with bound set B and free set A exists if all column patterns are F Pattern and G Pattern, $G \neq F$. In other words, column multiplicity index $\mu = 2$.

Both these properties can be used to verify the existence of Ashenhurst Decomposition, depending on the sizes of row and column sets of variables. Traditionally, for incompletely specified functions, the Ashenhurst and Curtis decompositions were reduced either to the clique partitioning of the Column Compatibility Graph or the graph coloring of the Column Incompatibility Graph [?, ?, ?, ?, ?, ?, ?, ?]. All these problems are in general NP-hard. However, in case of Ashenhurst decomposition, the problem can be solved by a polynomial algorithm. The following algorithm is based on Property 1.

Algorithm 1.6.1

1. Remove from RVM all rows that correspond to ON/DC, OFF/DC and DC Patterns.
2. Find two rows, r_i and r_j that are incompatible, and remove them from the RVM.
   From remaining rows create the set *Remaining_Rows*.
3. Pair_Counter := 1.
4. Put row r_i to LEFT[Pair_Counter] and row r_j to RIGHT[Pair_Counter].
5. Take next row r_s in set *Remaining_Rows* and remove it from set *Remaining_Rows*.
6. Compare r_s with arrays LEFT and RIGHT.
   a) If there exists a pair (LEFT[k], RIGHT[k]) such that r_s is incompatible with both LEFT[k] and RIGHT[k],
      then exit "No Ashenhurst Decomposition".
   b) Else if
      for all v from 1 to Pair_Counter

r_s is compatible with LEFT[v] and r_s is compatible with RIGHT[v]

then

if RIGHT[Pair_Counter] $\neq \phi$ then

Pair_Counter := Pair_Counter + 1;

put r_s to LEFT[Pair_Counter];

else

LEFT[Pair_Counter] := Combine_Rows(r_s, LEFT[Pair_Counter]);

c) Else Combine(LEFT,RIGHT,r_s).

7. If there are still rows in *Remaining_Rows*, go to 5.
8. Combine all sets LEFT[i] (i=1,...,Pair_Counter) creating set LEFT,
   Combine all sets RIGHT[i] (i=1,...,Pair_Counter) creating set RIGHT.
9. Return pair (LEFT, RIGHT) as the 2-coloring of the Compatibility Graph.

Procedure **Combine_Rows(r_s, r_v)** combines row r_s with row r_v, position by position in a row, using the combining rules:
$symbol_i$:= $symbol_i$ *combine* $symbol_i$, $symbol_j$:= $symbol_j$ *combine* dont'care,
Procedure **Combine(LEFT,RIGHT,r_s)** does the following:

Algorithm 1.6.2

1. Find set of such indices vl=1,...,Pair_Counter that r_s is incompatible with LEFT[vl]
   Combine all their RIGHT[vl] to RIGHT1 and all their LEFT[vl] to LEFT1.
   RIGHT1 := Combine_Rows(r_s, RIGHT1).
2. Find set of such indices vr=1,...,Pair_Counter that r_s is incompatible with RIGHT[vr]
   Combine all their RIGHT[vr] to RIGHT2 and all their LEFT[vr] to LEFT2.
   LEFT2 := Combine_Rows(r_s, LEFT2).
3. RIGHT3 := Combine_Rows(RIGHT1, LEFT2).
   LEFT3 := Combine_Rows(RIGHT2, LEFT1).
4. Remove all rows vl and vr from arrays LEFT and RIGHT, append the combined row RIGHT3 to the end of array RIGHT, append the combined row LEFT3 to the end of array LEFT.

The second method, based on Property 2, can be applied to mv-output functions, and has a very similar algorithm. Algorithm 8.1 is usually more efficient, but can be applied only to binary-output functions.

Example 1.6.2

Given is the graph from Figure 1.23a, that is the same graph as in Example 8.1. Its complement graph, the incompatibility graph is shown in Figure 1.23b. Let us assume that a pair of incompatible nodes initially selected is [{1}, {4}], Figure 1.23c. Now, node 3 is selected, Figure 1.23d. It is compatible with both nodes 1 and 4. Now, node 5 is selected, Figure 1.23e. It is incompatible with both nodes 1 and 3, but compatible with node 4. Thus, nodes 1 and 3 are combined, Figure 1.23f, and next nodes 4

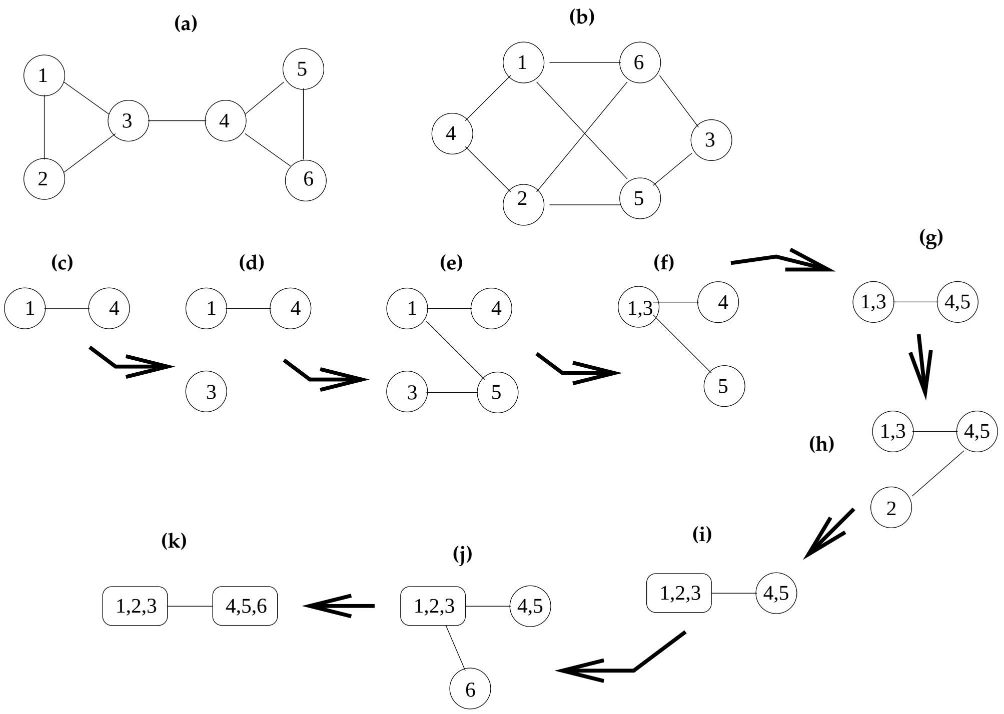


Figure 1.23: Illustration of the 2-coloring algorithm: (a) the compatibility graph G, (b) the corresponding incompatibility graph $\bar{G}$, (c) - (k) stages of coloring

and 5 are combined, Figure 1.23g, leading to graph with a pair $[\{1,3\},\ \{4,5\}]$. Now, node 2 is selected, Figure 1.23h. It is incompatible with nodes {4,5} but compatible with node {1,3}. Nodes 2 and {1,3} are combined, Figure 1.23i, leading to graph with a pair $[\{1,2,3\},\ \{4,5\}]$. Now, node 6 is selected, Figure 1.23j. It is incompatible with nodes {1,2,3} but compatible with node {4,5}. Nodes 6 and {4,5} are combined, Figure 1.23k, leading to graph with a pair $[\{1,2,3\},\ \{4,5,6\}]$. All nodes have been compared and only one pair of incompatible nodes was created. Thus the graph is 2-colorable. Nodes 1,2, and 3 are colored with color A, and nodes 4,5, and 6 are colored with color B. This is the same exact solution as the one found previously.

Let us observe, that, in contrast to standard graph coloring algorithms, the above algorithm does not require creating the graph for coloring. It checks incompatibility and combines the compatible nodes, "on the fly". Therefore, if the graph is not 2-colorable, the decision about this fact can be done after considering only a small portion of nodes. This is a very important property, because during Ashenhurst decomposition in most cases the graphs are not 2-colorable.

Chapter 2

Binary Curtis Decompositions

2.1 Curtis Decomposition

2.1.1 Disjoint Curtis Decomposition.

A decomposition approach based on Graph Coloring, very similar to the one from section 16.1, can be applied to Curtis Decomposition. It will not be repeated it here, since it was presented in detail in [?, ?]. Instead of finding cofactors for free variables, one finds cofactors for bound variables. Next, there is no need to remove cofactors of ones and cofactors of zeros. The graph is colored as previously. Analogously, one can find a Compatibility Graph and find its minimum disjoint covering with maximum cliques.

Let us observe that for both the PUB and Curtis decompositions, we have two pairs of graphs.

For free sets in the PUB Decomposition we have been using the following:

- Free Set Compatibility Graph - optimized by clique partitioning. or clique partitioning of "care columns" with similarity constraints (the *"care columns"* are columns other than the *"don't care columns"*, where the dont't care columns are columns that only don't cares).
- Free Set Incompatibility Graph - optimized by graph coloring, or graph coloring of care columns with similarity constraints.

Similarly, for bound sets in the Curtis Decomposition we have:

- Bound Set Compatibility Graph - optimized by clique partitioning, or clique partitioning of care columns with similarity constraints.
- Bound Set Incompatibility Graph - optimized by graph coloring, or graph coloring of care columns with similarity constraints.

Comparison of PUB Decomposition and Curtis Decomposition for small number of address variables (bound variables in case of Curtis) is shown in Figure 2.1. Indices μ denote values of column multiplicity indices for the bound sets in Curtis Decomposition. Indices ν denote values of row multiplicity indices for the free (address) sets in PUB Decompositions.

We can see that a successful PUB Decomposition creates more functions, but simpler, than the functions created by the Curtis Decomposition. The PUB decomposition is a link between the Curtis Decomposition and the multiplexer-based design and orthogonal expansions.

2.1.2 Non-disjoint Curtis Decomposition.

In non-disjoint Curtis decomposition, bound and free sets overlap i.e., they have common elements. Such arrangement often simplifies the decomposition blocks, or makes the decomposition possible when the disjoint decomposition does not exist.

Example 2.1.1

Let us consider the function f from Example 16.1.2 where bound (B) and free set (A) are defined as follows:

$A = \{a, b, e\}$, (rows); $\qquad B = \{c, d, e\}$, (columns)

K-map of this function is repeated in Fig. 2.2a. Using a compatibility graph, an incompatibility graph, or any method from section 8 we find that the column multiplicity for this K-map is equal to 2 ({000,001,011,010,110,111 ; 101,100}). Therefore, the function is Ashenhurst decomposable (a special case of Curtis decomposition). K-map obtained by filling DCs with 0s and 1s to obtain the

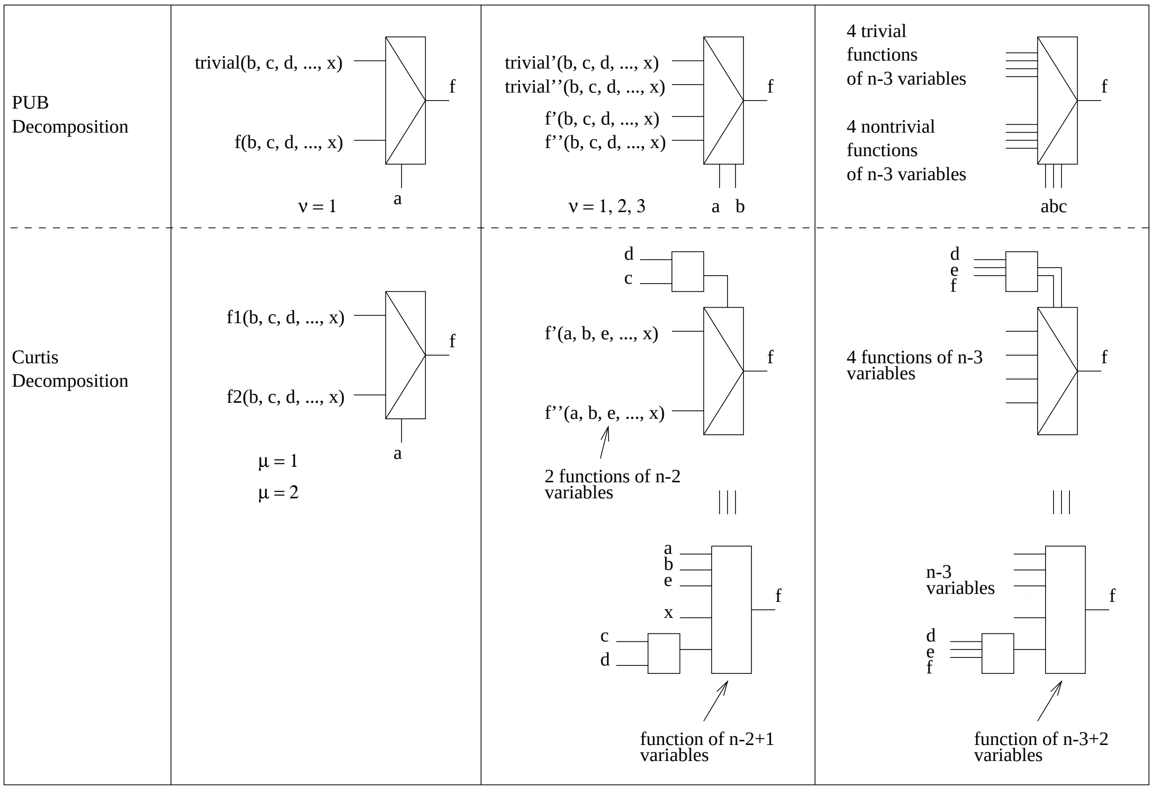


Figure 2.1: Comparison of PUB and Curtis decompositions for small k

abe \ cde	000	001	011	010	110	111	101	100
000	0	-	-	0	0	-	-	-
001	-	-	1	-	-	1	0	-
011	-	0	-	-	-	-	1	-
010	-	-	-	0	0	-	-	-
110	0	-	-	0	-	-	-	1
111	-	1	1	-	-	-	-	-
101	-	-	-	-	-	0	0	-
100	1	-	-	-	1	-	-	0

a)

abe \ cde	000	001	011	010	110	111	101	100
000	0	0	0	0	0	0	0	0
001	1	1	1	1	1	1	0	0
011	0	0	0	0	0	0	1	1
010	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	1	1
111	1	1	1	1	1	1	1	1
101	0	0	0	0	0	0	0	0
100	1	1	1	1	1	1	0	0

b)

Figure 2.2: K-map of function f to Example 10.1

column multiplicity of 2 is shown in Fig. 2.2b. Loops 000, 010 and 101 contain only 0s and loop 111 contains only 1s. If we define the loop function corresponding to $a = 0$, $b = 0$, $e = 1$ as G we will see from Fig. 2.2b that any other row loop function (different from 1s or 0s) is either equal to G or to $\overline{G}$. In this special case, a by-product of the nondisjoint Curtis decomposition is, that the function is PUB-decomposable and Ashenhurst decomposable: $f(a, b, c, d, e) = H[G(a, b, e), c, d, e]$.

2.2 Programs for Curtis Decomposition.

Basic Decompositions in the GUD-2 decomposer are *Curtis Decomposition* and *PUB Decomposition* (GUD-1 has only Curtis). However, when executed with a large value of the multiplicity index μ, they would lead to difficult encoding problems (in case of Curtis), and to the not necessarily minimum circuits (especially that we are never able to perform exhaustive search of sets A, B and C). Therefore, we execute these Basic Decompositions only for small values of the multiplicity index $\mu = 3,...8$.

In this section we will briefly present the algorithmic aspects of the Curtis Decomposition (more in [?] and [?]). We have three variants of the algorithm, called: $Curtis_{color}$, $Curtis_{clique}$, and $Curtis_{concurrent}$, respectively.

Given the bound set $B \cup C$ and the free set $A \cup C$, the stages of the $Curtis_{color}$ decomposition are the following:

1. Find a fast approximate *graph coloring* of the Incompatibility Graph.
2. Use the encoding method that is similar to the input encoding algorithm for function H from [?]. This algorithm is designed to minimize only the logic of function H.
3. Find functions G and H.

The reason to use here graph coloring instead of Clique Partitioning (as in literature) is to dramatically decrease the size of the memory, [?].

Let us assume that the N columns of the covering table correspond to the columns of the RVM, and the rows of the covering table correspond to the maximal cliques. Thus, for strongly unspecified functions, the number of rows is exponential, while the graph has only N * (N-1) / 2 edges.

Well known is a set covering algorithm that makes use of *essential rows, secondary essential rows* and *dominations of rows and columns* [?]. In case of non-cyclic covering tables, this algorithm finds the exact solution without backtracking. We created a similar algorithm for graph coloring.

Node G2 of the graph is dominated by node G1 if the set of incident nodes of G1 includes (properly or not) the set of incident nodes of node G2. In such case, any color of node G1 can be also applied to node G2. Thus, this fact can be stored, and the node G2 can be removed from the graph, together with all its incident edges. This leads to a new graph, that can possibly have new dominated nodes, and so on, until the graph is reduced to a complete graph, for which every node is colored with a different color. When there are no dominated nodes in the graph (a counterpart of a cyclic set covering), a coloring choice is done as in [?]. This can lead to dominated nodes, and the node dominations are propagated and removals are done as presented above, until a new branching choice is necessary. There is a perfect analogy of this coloring approach with methods of solving both the cyclic and non-cyclic covering problems, but our approach is faster and does not require creating large covering tables. Moreover, it was found experimentally that most of the column incompatibility problems are non-cyclic.

Example 2.2.1

We will illustrate how to convert the covering problem (in this case, a SOP minimization) to the coloring problem using a Kmap of a non-cyclic function f, Figure 2.3a. Numbers denote the true minterms. All other cells are false minterms. Obviously in this case, after sharping (*sharping* B from A means applying the cube calculus operation $A\#B$) of its essential primes $\overline{a}\overline{b}\overline{c}$ and $ac\overline{d}$, the secondary essential primes $\overline{a}bd$ and bcd are created. After sharping these secondary essential primes no true minterms remain, so the exact solution was found for a non-cyclic function f without backtracking. The Incompatibility Graph corresponding to this map is shown in Figure 2.3b. The stages of coloring the graph are shown in Figure 2.3b - e. In Figure 2.3b node 2 has neighbors 4,5,6,7, and node 1 has neighbors 3,4,5,6,7. Therefore, node 1 dominates node 2, and 2 is removed from the graph, leading to the graph from Fig. 3c. Now node 6 has neighbors 1,3,4 and node 7 has neighbors 1,3,4,5. Thus node 7 dominates node

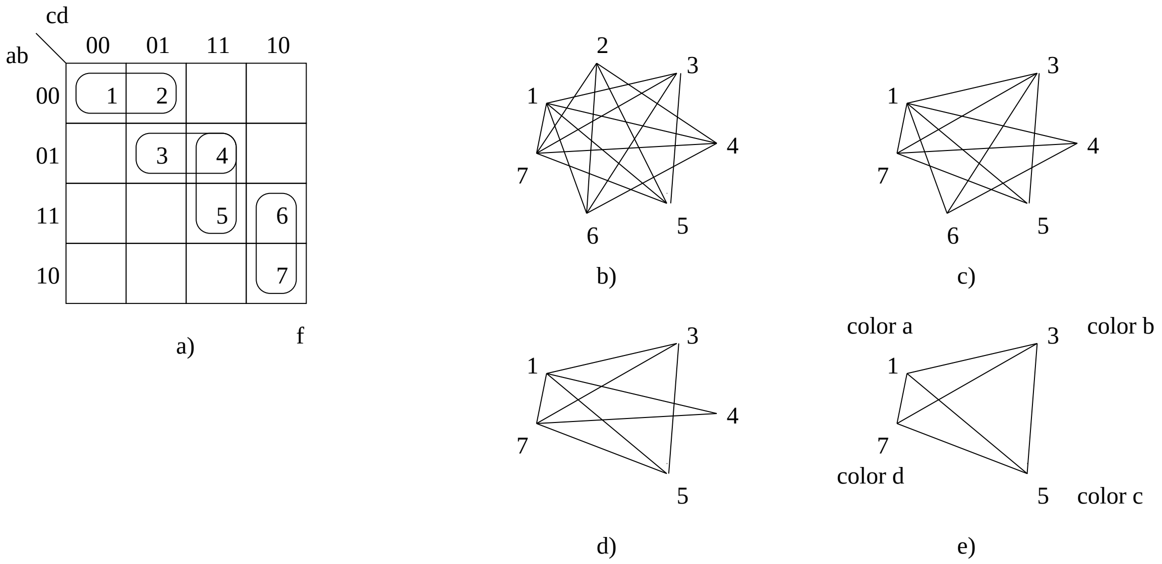


Figure 2.3: Proper coloring of a non-cyclic graph: (a) a non-cyclic covering problem formulated as SOP minimization, (b) - (e) stages of coloring of the graph corresponding to the covering problem from (a)

6 and node 6 is removed. This leads to the graph from Figure 2.3d. Now node 3 has neighbors 1,5,7 and node 4 has neighbors 1,7. So, node 3 dominates node 4 and node 4 is removed. Now, Figure 2.3e, the graph is a complete graph. Its nodes are colored with different colors, as shown. With respect to dominations, node 2 is colored with the same color as node 1, which is *color a*, node 4 is colored with *color b*, and node 6 is colored with *color d*.

In some covering problems, like in SOP minimization, a group of nodes colored with the same color are not necessarily all compatible, and group compatibility must be checked [?]. It can be proven, however, that in the Column Compatibility problem for both binary and mv cases, if columns in a set are pairwise compatible the set of all these columns is compatible as well. This is, however, no longer true for the generalized don't cares - see section ??.2.

In the second variant of Curtis Decomposition, $Curtis_{clique}$, given the bound and free sets, the stages of the decomposition are the following. First a fast *clique covering* of the Column Compatibility Graph is found. The nodes of this graph correspond to the columns of the original RVM. Note, that this is the Clique Covering (not Clique Partitioning as elsewhere), so the cliques in the solution overlap, and a column may be included in more than one clique. Next the encoding algorithm is used that is similar to the approach for concurrent state and input minimization and assignment from [?]. This approach can be summarized as follows. For every node of the graph, a set of symbols of all maximum cliques that cover this node is created. In a Kmap of function G with inputs from the bound set, every cell corresponds to a column of the RVM, and to a node of the graph. Every cell includes the **set** of clique symbols, selected in Clique Covering, that cover the corresponding node. This map represents then a **a symbolic output relation for function G.** The choice of one of these symbols is done for every cell in such a way that the two following goals are achieved concurrently for the logic of function G: 1) the good output encoding of G is found to simplify functions corresponding to the new binary variables between G and H. 2) after the encoding, the symbolic output relation changes to a standard Boolean relation. One wants to find such encoding that the relation can be next minimized to as simple Boolean functions as possible. In particular, we want these functions to have as many output don't cares as possible: for instance, codes 00 and 01 are combined to 0-. This is one of the algorithms in GUD-2.

The third variant, $Curtis_{concurrent}$, assumes the (overlapping) bound and free sets as well. It uses the method that is similar to the algorithm for concurrent state minimization and state assignment for state machines from [?]. It is based on the analogy that states of the machine can be compared to columns in the decomposition RVM. This way, the algorithm for the concurrent state minimization and state assignment for a state machine is translated to an algorithm for concurrent column minimization and column encoding problem - combining of the columns and encoding them is done in parallel, by finding the set $\{\tau_i$: i=1,...k$\}$ of partitions of the set of columns, such that in every block in partition product $\prod_{i=1,\ldots k} \tau_i$ every two elements (columns) included in the block are compatible. This is explained in more detail in [?].

2.3 Modern Explanation of Ashenhurst-Curtis Decomposition.

In this section we will present the fundamental Ashenhurst-Curtis Approach, and illustrate it with various function representations in order to be able to compare all the existing approaches to decomposition.

For instance, an incompletely specified binary single-output function can be represented as a table with don't cares in some of cells, as an array of cubes, as a ternary BDD (a ternary BDD is a BDD with three types of terminal nodes; 0, 1 and X (representing a don't care)), or as a pair of standard BDDs, one for ON and one for OFF set.

Similarly, a multi-output incompletely specified binary function can be represented as a multi-output table with don't cares, as a multi-output array of cubes, as a shared ternary BDD, or as a shared BDD of ON-OFF pairs of standard BDDs for each component function.

For Machine Learning applications, which have a large number of don't cares (DC), the best representation should deal with function ON and OFF sets represented in some way (cubes, BDDs, etc.).

2.3.1 Basic Data Formats and Definitions

Suppose that one intends to decompose an incompletely specified function consisting of twenty-five inputs and twenty outputs into several smaller logic blocks. The function is given in Espresso format:

```
.i  25
.o  20
.ilb  i1 i2 i3 i4 i5 i6 i7 i8 i9 i10 i11 i12 i13 i14 i15
i16 i17 i18 i19 i20 i21 i22 i23 i24 i25
.ob  o1 o2 o3 o4 o5 o6 o7 o8 o9 o10 o11 o12 o13 o14 o15
o16 o17 o18 o19 o20
.type  fr
10-01-010101-01-01010-10-   10-10010-1010-01-10-
1-11-111-1--1100000-01-1-   01-01-00101010-----1
00000001-01010101-11-0110   01-0-1-010101-1101-0
..
..
00100101010101010-------1   1-110101001---010101
.end
```

Espresso Format is a two-level description of a Boolean function. It is a character matrix with keywords embedded in the input to specify the size of the matrix and the logical format of the input function. In the above file:

```
.i 25
```

specifies the number of input variables (25).

```
.o 20
```

specifies the number of function outputs (20).

```
.ilb i1 i2 ...... i25
```

specifies the names of input variables.

The left matrix of the above file corresponds to the input cube array, `i1` is the name of the input variable corresponding to the first column of the input cube array, `i2` to the second column, and so forth.

```
.ob o1 o2 ...... o20
```

specifies the names of function outputs.

The right matrix of the above file corresponds to the output cube array, `o1` is the name of the output variable corresponding to the first column of the output cube array, `o2` to the second column, and so forth.

o1 is the name of the function output corresponding to the first column of the output array (right matrix of the above file), o2 to the second column, and so forth.

```
.type  fr
```

sets the logical interpretation of the character matrix of output array. Symbol `fr` specifies that a 1 in the output array means that the corresponding cube in the input cube array belongs to the ON set. A 0 in the output array means that the corresponding cube in the input cube array belongs to the OFF set. The symbol '-' or ' " in the output array means that the corresponding cube in the input cube has no meaning for the value of this function. DC set may be computed as the complement of the union of the ON set and the OFF set.

```
.end
```

marks the end of the input logic.

With respect to the algorithms used in the program, DC cubes are not needed for the output function minimization. This decreases the memory demand and makes algorithm much more efficient (especially when the number of DC cubes is large).

Fundamental Definitions

A *Cube* is a compact expression of a set of minterms. For example, minterms 11010 and 11000 can be expressed as a cube 110-0. '-' means it takes the value of both 0 and 1.
If the output of a cube is 1, it is called the ON cube.
If the output of a cube is 0, it is called the OFF cube.
If the output of a cube is - (don't care. It can be either 0 or 1), it is called the DC cube.

The ON set is the collection of all ON cubes. The OFF set is the collection of all OFF cubes. The DC set is the collection of all DC cubes. Cube, ON cube, OFF cube, DC cube, ON set, OFF set and DC set are showed in Figure 2.4.

The *Cube Calculus* is a set of operations applied to cubes and arrays (lists) of cubes. In our description, we will use the Intersection operation, which can be illustrated using maps, arrays of cubes, or BDDs.
Figure 2.5 shows the rules of Intersection operation. The ϵ is the result of the Intersection of 0 with 1 or 1 with 0. x or X have the same meaning as the "-".
From Figure 2.5, the rules are:
$0 \cap 0 = 0$,

		Cube ↓			
ON	cube →	10110-	1		
		1–000	1	← ON	set
		111010	1		
OFF	cube →	0-1001	0		
		01101-	0	← OFF	set
		0-1001	0		
DC	cube →	01010-	-		
		0100-1	-	← DC	set
		–0101	-		

Figure 2.4: Cubes and cube sets

$\bullet$	0	1	x
0	0	ε	0
1	ε	1	1
x	0	1	x

Figure 2.5: Intersection operation

Decomposition

Figure 2.6: Decomposition

$0 \cap 1 = \epsilon$,
$0 \cap \mathrm{x} = 0$,
$1 \cap 0 = \epsilon$,
$1 \cap 1 = 1$,
$1 \cap \mathrm{x} = 1$,
$\mathrm{x} \cap 0 = 0$,
$\mathrm{x} \cap 1 = 1$,
and $\mathrm{x} \cap \mathrm{x} = \mathrm{x}$.

Figure 2.6 shows the general decomposition scheme. Boolean decomposition uses Boolean representation.

For example, a bit-by-bit intersection on cubes can be illustrated as follows:

```
1010xx
100x1x
------
10ϵ01x
```

The *Decomposition Chart* [?, ?, ?] is a chart that is similar to the Karnaugh map with the only difference being that the column and row indexes of the decomposition chart are in the straight binary order, while that of the Karnaugh map are in the Gray code order. Figure 2.7b shows an example of a decomposition chart.

The corresponding Karnaugh map is shown in Figure 2.7a. The column of the chart is denoted as a vector of its successive minterms. For example, column 1 in Figure 2.7b is denoted as a vector [1, 1, 0, 1]. Because there is no essential difference between the Karnaugh map and the decomposition chart, Karnaugh maps will be used instead of decomposition charts for illustration.

The *Bound Set* is a set of variables forming the columns of the decomposition chart. In Figure 2.7b, { c, d, e } is a bound set.

The *Free Set* is a set of variables forming the rows of the decomposition chart. In Figure 2.7b, { a, b } is a free set.

ab \ cde	000	001	011	010	110	111	101	100
00	x	1	1	0	1	0	0	1
01	x	1	x	1	x	x	1	1
11	x	1	x	x	x	x	x	0
10	x	0	0	1	0	1	1	0
Column	0	1	3	2	6	7	5	4

f

(a)

ab \ cde	000	001	010	011	100	101	110	111
00	x	1	0	1	1	0	1	0
01	x	1	1	x	1	1	x	x
10	x	0	1	0	0	1	0	1
11	x	1	x	x	0	x	x	x
Column	0	1	2	3	4	5	6	7

f

(b)

Figure 2.7: Karnaugh map and decomposition chart

The *Column Multiplicity* denoted by $\mu(A \mid B)$, is the number of different columns in a decomposition chart. In $\mu(A \mid B)$, A stands for the free set, B stands for the bound set. For example, in Figure 2.7b, $A = \{a, b\}, B = \{c, d, e\}$ and $\mu(A \mid B) = \mu(ab \mid cde) = 3$.

If two horizontally corresponding cells in two columns of the decomposition chart are (0,0), (1,1), (0,x), (1,x), (x,0), (x,1) or (x,x), these two cells are called *compatible*. If all the corresponding cells in two columns are compatible, these two columns are called *compatible*. Otherwise, they are called *incompatible*. In Figure 2.7b columns 1 ([1, 1, 0, 1]) and 6 ([1, x, 0, x]) are compatible, while columns 5 ([0, 1, 1, x]) and 6 ([1, x, 0, x]) are incompatible. In this explanation, we will use maps, Cube Calculus, or BDDs, to test whether two columns (called also *column functions*, *column cofactors* or *K-map loops*) are compatible or not. The formula [?] to test the compatibility of two columns (columns i and j) is:

[ON(i) $\cap$ OFF(j) = 0] $\wedge$ [ON(j) $\cap$ OFF(i) = 0] $\Rightarrow$ i-th and j-th column are compatible

The 0 stands for a zero function (0 Boolean constant). The formula states that if the Intersection of the ON function of column i and the OFF function of column j is zero, and the Intersection of the ON function of column j and the OFF function of column i is zero as well, these two columns are compatible. Otherwise, they are incompatible. Let us observe, that this condition does not specify how functions ON and OFF are realized. For instance, ON and OFF are represented as cube arrays in TRADE.

The *Incompatibility Graph* is a graph which illustrates the incompatibility relationship among columns of the decomposition chart. Each node in the incompatibility graph corresponds to a column in the decomposition chart. If two columns are incompatible in the Incompatibility Graph, there is an edge between the corresponding nodes. If they are compatible, there is no edge.
Figure 2.8 shows an Incompatibility Graph corresponding to the decomposition chart in Figure 2.7b.

The *Compatibility Graph* is a graph which illustrates the compatibility relationship among columns of the decomposition chart. Because two columns can be either compatible or incompatible, the compatibility graph and the incompatibility graph are mutual complements. It means that the sets of *edges* of these graphs are disjoint and the union of the sets of edges creates a full graph. The column minimization problem has been reduced in the past to one of the following:

- incompatibility graph coloring,
- incompatibility graph maximum independent set partitioning.

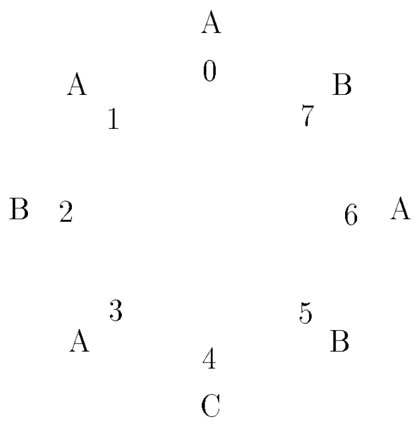


Figure 2.8: Example of an incompatibility graph

- compatibility graph maximum clique partitioning.
- compatibility graph maximum clique covering.

All these problems are mathematically equivalent, but can be solved with heuristic approaches that will perform better or worse on particular categories of graphs (sparse, dense).

In Figure 2.8, the number in each node (denoted by a circle) is the column number. The letter beside the circle is the color assigned to the node (column) after graph coloring.

2.3.2 Decomposition of the Incompletely Specified Functions.

In this section, the functional Boolean decomposition of incompletely specified functions will be presented. The basic ideas follow [?, ?, ?, ?] and the general approach based on graph coloring is patterned after [?, ?, ?].

Curtis has described the decomposition of completely specified functions in [?]. He proved the *fundamental theorem*:

$$\mu(A|B) \leq 2^k \Leftrightarrow f(B,A) = F(\phi_1(B), \phi_2(B), \ldots, \phi_k(B), A)$$

This formula states that if the column multiplicity $\mu(A \mid B)$ (under the partition of the bound set B and free set A) is less than 2^k, then the function $f(B,A)$ can be decomposed into the form:

$$f(B,A) = F(\phi_1(B), \phi_2(B), \ldots, \phi_k(B), A)$$

The graphical representation of this theorem is shown in Figure 2.9.

From Figure 2.9 we observe that, after decomposition, the big block f is broken into several smaller sub-blocks ϕ_1, ϕ_2, ..., ϕ_k and F.

The essential problem of the decomposition of incompletely specified function is how to assign DC outputs as 0 or 1 to minimize the column multiplicity. Because the number of colors in a properly colored incompatibility graph is the same as the number of different columns (column multiplicity) in a decomposition chart [?], the problem of finding the smallest column multiplicity can be transformed into that of performing the proper graph coloring to find the smallest number of colors. We use the following criterion:

Assume n to be the expected number of output variables from the blocks with bound set as inputs, and n be less than the number of variables in the bound set. If the column multiplicity is equal to or less

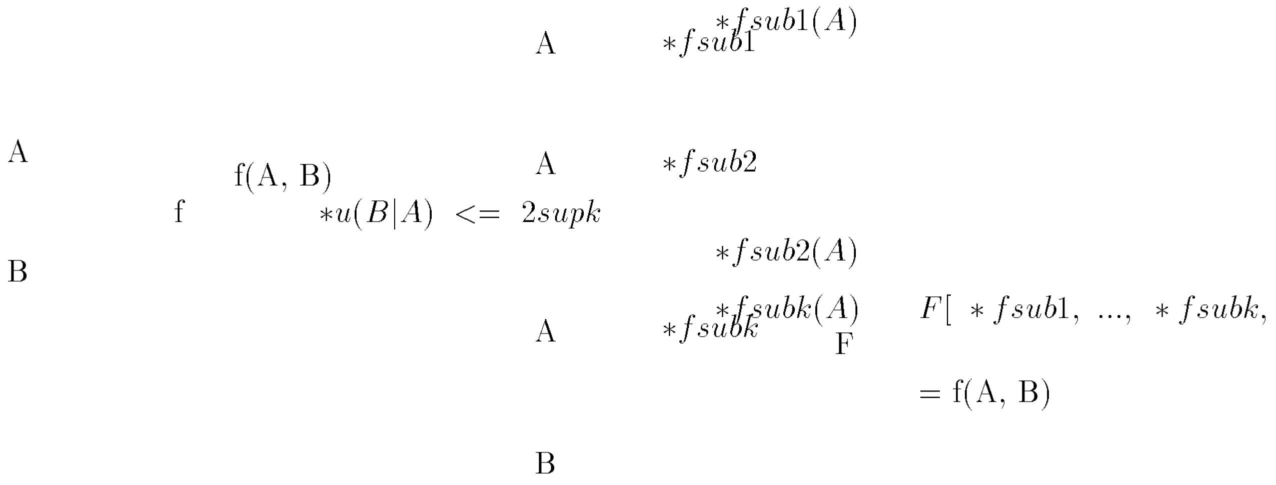


Figure 2.9: Curtis decomposition

than 2^k, and k is less than or equal to n, the decomposition is *successful* (or the function is *decomposable*) for this bound set under the expected value of n. Otherwise, the function is *non-decomposable* for this bound set under the expected value of n.

After a successful decomposition, the number of input variables of each sub-function (decomposed blocks, like ϕ_1, ϕ_2, ..., ϕ_k *and* F in Figure 2.9) is decreased, and the complexity of each sub-function is decreased as well. This will be illustrated with an example.

Figure 2.10a is the Karnaugh map of function f with don't care outputs. For instance, one may intend to decompose the function f into several sub-functions (denoted by L, M and N in Figure 2.10c) with the input variables of each sub-function less than or equal to four.

According to the rules presented above, the incompatibility graph is created as shown in Figure 2.11.

After graph coloring, three colors, A, B, and C, are obtained. Which means - $\mu = 3$. These colors group nodes as A = {0, 1, 3, 6}, B = {2, 5, 7} and C = {4}. The columns with the same color are combined horizontally by the rules: (0, 0) → 0, (0, x) → 0, (x, 0) → 0, (1, 1) → 1, (1, x) → 1, (x, 1) → 1 and (x, x) → x.

For example, columns 0, 1, 3 and 6 in Figure 2.10a are combined and replaced by a new vector [1, 1, 1, 0] as shown in the final don't care assignment in Figure 2.12. In the above example, we have chosen the variables a and b as the free set and variables c, d and e as the bound set. This partition results in a successful decomposition in the sense of the column multiplicity less than or equal to three. In Figure 2.10c, x and y are the encoded outputs of the bound set, two variables are enough for three different columns ($3 \leq 2^2 = 4$). The encoding of Bound set will be discussed in the next section.

There has been certain criticizm of AC model. It starts from an observation that in the AC decomposition the block F which is the result of decomposition has fewer input variables then the initial function block f. This property, however, is not true for most of the practical circuit realizations, for instance the SOP realization, which has the number of primes in the minimal cover (the second level) that is larger than the number of input variables (the first level). And there are many other practical circuits that have the same property. Because of that, the application of the AC decomposition model is questioned for general circuit design. However, this criticism does not apply to the most general non-disjoint decomposition model, where the number of blocks in intermediate levels can grow, because they use different overlapping subsets of intermediate variables.

ab \ cde	000	001	011	010	110	111	101	100
00	x	1	1	0	1	0	0	1
01	x	1	x	1	x	x	1	1
11	x	1	x	x	x	x	x	0
10	x	0	0	1	0	1	1	0
Column	0	1	3	2	6	7	5	4

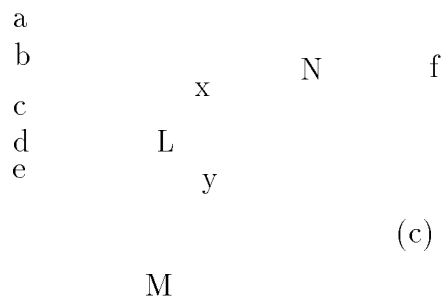
f

(a)

ab \ cde	000	001	010	011	100	101	110	111
00	x	1	0	1	1	0	1	0
01	x	1	1	x	1	1	x	x
10	x	0	1	0	0	1	0	1
11	x	1	x	x	0	x	x	x
Column	0	1	2	3	4	5	6	7

f

(b)



(c)

Figure 2.10: Karnaugh map, decomposition chart and the expected decomposition

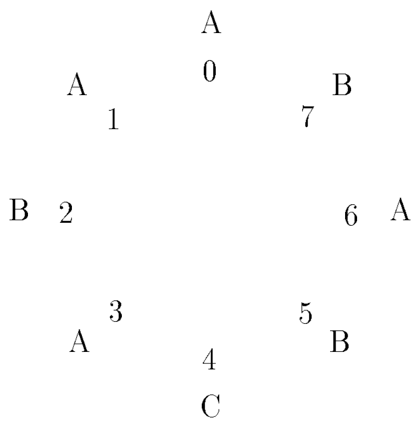


Figure 2.11: Incompatibility graph

ab \ cde	000	001	011	010	110	111	101	100	
00	1	1	1	0	1	0	0	1	
01	1	1	1	1	1	1	1	1	
11	1	1	1	x	1	x	x	0	
10	0	0	0	1	0	1	1	0	f
Column	0	1	3	2	6	7	5	4	
Color	A	A	A	B	A	B	B	C	

Figure 2.12: Final don't care assignment

	B	C
A	A-B 2	A-C 1
B		B-C 2

Figure 2.13: Similarity Factor Table

2.3.3 Bound Set Encoding

There are many methods [?, ?] to implement the decomposed blocks (blocks L, M and N in Figure 2.10c). Here we introduce an algorithm to encode the bound set that aims at simplifying the block N. The encoding algorithm assigns adjacent codes (Gray code) to the similar columns. This increases the number of large cubes in the block N. The similarity (or difference) between two columns is measured by the so-called *Difference Factor*. The more similar the two columns, the lower the value of the Difference Factor. The *Difference Factor* is the number of minterms in which the two columns are not identical. The Difference Factor between the i-th and j-th columns is:

$$\textit{Difference Factor} = \text{minterm\_size}(\text{ON(i)} \cap \text{OFF(j)}) + \text{minterm\_size}(\text{OFF(i)} \cap \text{ON(j)})$$

In the above formula, "minterm_size()" calculates the number of minterms. "ON(i) $\cap$ OFF(j)" is the Intersection of ON function of the i-th column with the OFF function of the j-th column. "OFF(i) $\cap$ ON(j)" is the Intersection of OFF function of the i-th column with the ON function of the j-th column. The more similar the two functions, the smaller the value of the Difference Factor. It becomes a zero for identical functions.

In this example, bound set $\{c,\ d,\ e\}$ forms eight columns as shown in Figure 2.10a. After graph coloring, three different columns were found. These three columns are: [1, 1, 1, 0] corresponding to color A, [0, 1, x, 1] corresponding to color B and [1, 1, 0, 0] corresponding to color C as shown in Figure 2.12. We introduce two new variables x and y to encode the bound set $\{c,\ d,\ e\}$. First let us calculate the Difference Factors. The Difference Factor between columns corresponding to color A and B has a value of 2. The value of the Difference Factor between columns corresponding to color A and C is 1. And the value of the Difference Factor between columns corresponding to color B and C is 2.

A Table of Difference Factors is created as shown in Figure 2.13.

Because the Difference Factor between columns corresponding to color A and C is smaller (with a

c \ de	00	01	11	10	
0	0	0	0	1	
1	0	1	1	0	x

(a) Block L

c \ de	00	01	11	10	
0	1	1	1	1	
1	0	1	1	1	y

(b) Block M

ab \ xy	00	01	11	10	
00	1	1	0	x	
01	1	1	1	x	
11	0	1	x	x	
10	0	0	1	x	f
Color	C	A	B		

(c) Block N

ab \ xy	00	01	11	10	
00	1	0	1	x	
01	1	1	1	x	
11	1	x	0	x	
10	0	1	0	x	f
Color	A	B	C		

(d) Block N

Figure 2.14: Decomposed Karnaugh maps

value of 1), these two columns are put in adjacent positions, as shown in Figure 2.14c. Let us code the column corresponding to color C as 00, the column corresponding to color A as 01 and the column corresponding to color B as 11 as shown in Figure 2.14c, which is the Karnaugh map of the block N. Color A has the code 01, which means that x is equal to 0 and y is equal to 1 for all columns with the color A. These columns are 000, 001, 011 and 110 in Figure 2.12, therefore the cells 000, 001, 011 and 110 of the Karnaugh map in Figure 2.14a, which is the Karnaugh map of the block L, are filled with 0 because x is equal to 0. The same cells in Figure 2.14b, which is the Karnaugh map of the block M, are filled with 1 because y is equal to 1. The same way, color B has the code 11, which means that both x and y are equal to 1. Columns 010, 111, and 101 correspond to color B, therefore the cells 010, 111 and 101 of the Karnaugh maps in both Figure 2.14a and (b) are filled with 1. Color C has the code 00, both x and y are equal to 0, column 100 correspond to color C, the cell 100 of the Karnaugh maps in both Figure 2.14a and (b) are filled with 0.

Two variables can encode up to four columns ($2^2 = 4$). There are only three columns, corresponding to color A, B and C, that need to be encoded in our example. We fill the remaining column (column 10 in Figure 2.14c) with don't cares (*DC column*). The existence of this newly introduced DC column will further simplify the block N. This example shows that even if the input function is completely specified, the algorithm may introduce DCs in the middle of the process, which is very useful for the simplification of the later stages.

Figure 2.14d shows a Karnaugh map of an alternative implementation for the function f, which uses the natural order of the colors. Clearly, the Karnaugh map in Figure 2.14c is simpler than that in Figure 2.14d.

The pseudo-code for bound set encoding is shown in Figure 2.15. The BLIFF format of the result is as follows:

```
.model example
.inputs a b c d e
.outputs f
```

```
bound_set_encoding( )
{
create Similarity Factor Table;
sort Similarity Factor Table in increasing order;

l_c = one column of the column pair at position 0 of the queue;
mark l_c as used;
r_c = another column of the column pair at position 0 of the queue;
mark r_c as used;
put l_c and r_c in line; *l_c at left, r_c at right *
c_n = 2;

while (c_n < column_multiplicity)
  for (i = 1; i < (column_multiplicity*(column_multiplicity - 1))/2; i++)
    if ((c_i = one of the pair at position i) == l_c)
      {
      mark c_i as used;
      put c_i at the left of l_c;
      l_c = c_i;
      c_n++;
      break;
      }
    else if ((c_i = one of the pair at position i) == r_c)
      {
      mark c_i as used;
      put c_i at the right of r_c;
      r_c = c_i;
      c_n++;
      break;
    }
}
```

Figure 2.15: Pseudo-code of bound set encoding

```
.names c d e x
1-1 1
010 1
.names c d e y
0-- 1
--1 1
-1- 1
.names a b x y f
0-0- 1
-1-1 1
1-1- 1
.end
```

BLIF Format is a multi-level description of the Boolean network. Each node in this representation has a single output. Therefore, each net (or signal) has only a single driver, and one can therefore name either the signal or the gate which drives the signal without ambiguity.

```
.model example 16
```

specifies the name of the model (example).

```
.inputs a b c d e
```

gives the name of the input variables (a, b, c, d, e).

```
.outputs f
```

gives the name of the output of the function (f).

```
.names c d e x
```

with the following ON set describes the logic of a node (sub-block L in Figure 2.10c). The input variables to this node are c, d, e, and the output variable is x.

```
.names c d e x
```

with the following ON set describes the logic of a node (sub-block M in Figure 2.10c). The input variables to this node are c, d, e, and the output variable is y.

```
.names a b x y f
```

with the following ON set describes the logic of a node (sub-block N in Figure 2.10c). The input variables to this node are a, b, x, y, and the output variable is f.

```
.end
```

marks the end of the model.

There are four fundamental problems to be solved in a Boolean decomposition of an incompletely specified function:

- How to chose the bound set to minimize the column multiplicity?
- How to minimize the column multiplicity for a given bound set ?
- How to encode the functions?
- How to transform a non-decomposable function into a decomposable one?

These questions will be discussed in more detail in next sections.

Variable Partitioning is the separation of the input variables into two sets, the bound set and the free set. Each partition corresponds to an individual decomposition chart which is going to be used to calculate the column multiplicity. In order to find the decomposition that corresponds to the smallest column multiplicity, one needs to go through all possible decomposition charts. If there are total m input variables and n variables in the bound set, the number of all possible partitions is $\binom{m}{n}$.

For example, if m = 64 and n = 5, then $\binom{64}{5} = 7,624,512$.

If the time required to calculate the column multiplicity of a decomposition chart is 0.01 second, one would need more than 20 hours to complete all calculations. This 20 hours will be repeated thousands of times to get the decomposition done. Therefore, it is impractical to try all possible partitions to find the best one.

There are basically three approaches to the Variable Partitioning problem:

- a fast method - find few good partitions using a powerful heuristic,
- perform a heuristic search for a reasonably good subset of all partitions,
- check all partitions to find the exact solution.

Chapter 3

Multi-valued Decompositions

3.1 Decomposition of Multiple-Valued Logic Functions and Information Systems.

Decompositions of Multiple-Valued Functions have been discussed by:

- Thelliez [?] (only for ternary logic).
- Walliuzzaman and Vranesic [?].
- Fang and Wojcik [?].
- Luba, Mochocki and Rybnik [?, ?, ?].
- Abugharrbieh and Lee [?, ?].

Similarly to the binary case, there are two types of decompositions for multiple-valued functions: the AC-like decompositions, and the Dietmeyer-like decompositions.

One approach to decomposition of multiple-valued logic functions is to extend the AC binary approach. This approach is represented by Walliuzzaman and Vranesic, [?], and Thelliez, [?], for ternary logic. Especially the partition-based approach of Luba allows him to create an algorithm that is a straightforward adaptation of his former binary algorithms. Another approach is represented by Fang and Wojcik, [?]. Their paper is a very good reference for some non-FD approach to Boolean Decomposition, but it is not of an interest to this paper, because it is so different in principle.

Papers by Luba et al on decomposing multiple-valued functions are connected to "machine learning" [?, ?, ?, ?]. On one hand, their approach to decomposition is similar to both classical Curtis, and Walliuzzaman's or Abugharrbieh's and Lee's approaches to mv decomposition. The difference is that they apply the partition theory. The sufficient condition in this approach leads to the calculation of the minimal cover of maximum compatible classes. Some new results are obtained by using the method described by Selvaraj in his Ph.D. Thesis [?]. The new idea is to combine the parallel and serial decompositions in a single program. These ideas can be extented to multiple-valued logic.

Many problems related to decision making are of the following nature: Given a decision table - find all minimal decision algorithms associated with the table. This problem can also be seen as learning from examples, or, as its is often called, decision rules generation from examples - Grzymala-Busse, [?, ?, ?]. These problems have been investigated from a number points of view: one of them is whether the whole set of attributes is always necessary to define a given partition of an universe and the other concerns the simplification of decision tables, namely the reduction of condition attributes in a decision table.

The concept of information system decomposition can be treated as an aid in analyzing the existing system [?] or as a base for design a new one [?, ?]. Quality of decomposition depends mainly on the analyst's experience. Some authors try to provide a formal approach to decomposition [?]. Decompositional approach similar in essence to the one proposed earlier by Armstrong [?] was presented by Luba, Mochocki and Rybnik in 1993. Avoiding data redundancy is an important problem in the implementation of of information systems. The problem is usually overcome by devising simpler rules and removing redundant rules, as well as minimizing the number of attributes. The importance of decomposition is emphasized by Courtois [?]: "Decomposition has long been recognized as a powerful tool for the analysis of large and complex systems".

From the technical point of view, the decomposition phase for information systems is exactly the same as in the multiple-valued decomposition with no constraint on the number of values of the output from the predecessor block.

Partition Based decomposition is just a representation of Switching Functions used in decomposition, so it can be applied to mv generalizations of any of the previously described types of decomposition, thus creating:

- mv Curtis (a generalization of mv Ashenhurst),
- mv PUB (another generalization of mv Ashenhurst),
- mv strong decomposition - for quaternary gates,
- mv generalization of XBOOLE decomposition for AND, OR and EXOR gates. In this generalization, these gates become MIN, MAX and MOD_M gates in M-valued logic, respectively.

3.2 Fundamental Formalisms for Partition-Based Decomposition.

Below we will introduce a formalism that will be used to represent all above decompositions using partitions, and next BDD-realized, double-coded partitions.

Let x_i be a multiple-valued variable, and $C_i = \{0, 1, ..., c_i - 1\}$ be a set of values that it may assume.

Definition 3.2.1 *A Multiple-Valued input, binary output function with n input, m output variables is defined as a mapping:*

$$F(x_1, ..., x_n) : C_1 \times C_2 \times \ldots \times C_n \longrightarrow D^m \tag{3.1}$$

where $D = \{0, 1, -\}$ represents the binary value of the function. The value "-" (a don't care) at one of the outputs means that the value is unspecified, and a value of 0 or 1 will be accepted to realize this part of the function.

Let $D_j = \{-, 0, 1, ..., d_j - 1\}$ be a set of values that output variable j can take.

Definition 3.2.2 *A Multiple-Valued input, Multiple-Valued output function with n input, m output variables is defined as a mapping:*

$$F(x_1, ..., x_n) : C_1 \times C_2 \times \ldots \times C_n \longrightarrow D_1 \times D_2 \times \ldots \times D_m \tag{3.2}$$

Every element of the domain $C_1 \times C_2 \times \ldots \times C_n$ is called *a minterm* or a *sample.* A listing of minterms with the value of the function is called a truth table. Truth tables do not include minterms with the function value not specified for all outputs. Set of minterms for which the function value is unspecified is called a DC-set (Don't Care-set). Functions with non empty DC-set are called *incompletely specified.*

Let us observe that in case of binary logic, a concept of an output don't care in output O_j can be understood that output O_j is either 0 or 1. Let us observe also, that for mv-logic there are two ways of generalizing the concept of don't care. One is, that *all* possible values are taken, as above. The other is that any subset of more than one values is taken. For instance, in three-valued logic, the logic values are 0, 1, and 2, and the generalized don't cares are:

- {0,1},
- {1,2},
- {0,2},
- {0,1,2} = -.

This leads to a concept of a Generalized MV function, that is located between a mv function and a mv relation.

Let $D_j = \{-, 0, 1, ..., d_j - 1\}$ be a set of values that its output variable j can take and

Definition 3.2.3 *A Generalized Multiple-Valued input, Multiple-Valued output function with n input, m output variables is defined as a mapping:*

$$F(x_1, ..., x_n) : C_1 \times C_2 \times \ldots \times C_n \longrightarrow 2^{D_1} \times 2^{D}_2 \times \ldots \times 2^{D_m} \tag{3.3}$$

For the sake of clarity a truth table may be viewed as a four-tuple $T = (M, A, X, Y)$, where

- M - is a non-empty, finite set of minterms,
- A - is a finite set of arguments i.e. input and output variables; $A = X \cup Y$, where X is the set of input variables and Y is the set of output variables, $X \cap Y = \emptyset$. Moreover

- a - is a function which maps the argument $a \in A$ into their values for every minterm $v \in M$, i.e.:

$$a : M \longrightarrow V_a \tag{3.4}$$

where V_a is a domain (set of possible values) of argument a. The set of values of output variables $V_y = \{0, 1, -\}$.

In general, any pair of minterms in a specification table of Multiple-Valued Function may have identical values for some number of input variables. A convenient way to reflect such similarities can be introduced with the help of the so called *indiscernibility relation* [?].

Definition 3.2.4 *The indiscernibility relation, denoted by IND, is associated with any subset of input variables as follows:*

Let $B \subseteq X; m1, m2 \in M$,
$(m1, m2) \in IND(B)$ if and only if $x(m_1) = x(m_2)$ for all $x \in B$.

This means, that $(m_1, m_2) \in IND(B)$ if the values of the arguments belonging to B are identical for both m_1 and m_2. Minterms m_1 and m_2 are said to be indiscernible by arguments from B.

The indiscernibility relation is an equivalence relation on M and

$$IND(B) = \bigcap_{x \in B} IND(x) \tag{3.5}$$

Thus, the relation IND partitions M into equivalence classes $M/IND(B)$. Such partitions are of primary importance in logic synthesis [?]. To simplify, we shall denote partition $M/IND(B)$ by $P(B)$ and call such a partition as an input partition generated by set B. Then, the formula equivalent to those of (1) may be written as

$$P(B) = \prod_{x \in B} P(x) \tag{3.6}$$

where $\prod$ denotes the product of partitions.

Definition 3.2.5 *Two output vectors of a function, p and q, are said to be* consistent *if their respective entries, which are defined, are equal, i.e.*

$\forall i \in \{1, ..., m\}$ (m is the number of output variables) $(p_i = q_i) \vee (p_i = -) \vee (q_i = -)$.
Consistency relation of the output vectors is denoted by $p \sim q$.

In general, every pair of minterms in the specification table may have consistent output values for some output variables y_i. Therefore the output-consistency relation, denoted by CON, can be related to every subset B of output variables. This relation can be formally defined as follows:

Let $B \in Y$ and $p, q \in M$. Minterms $p, q \in CON(B)$ iff $y(p) \sim y(q)$ for every $y \in B$, where $a_1 a_2$ means that a_1 and a_2 are equal if defined.

If every pair of minterms in a set is consistent, then the set constitutes a consistence class. The classes which are not the subsets of any other consistent class are called Maximal Consistent Classes (MCCs).

The CON relation is not an equivalence relation on the set M, as the consistence classes of the set M can be overlapping. However, a unique set of maximal consistence classes of minterms exists for every given CON relation. Thus, the same notation as defined earlier for the inputs can also be used for the output consistence classes, i.e. $P_F(B)$, where the index F differentiates the IND and CON relations.

	X_1	X_2	X_3	X_4	X_5	X_6	Y_1	Y_2	Y_3
1	0	0	0	0	0	0	1	1	0
2	0	0	1	1	0	0	1	1	0
3	1	2	2	0	1	1	0	1	1
4	0	1	1	0	0	1	0	1	1
5	0	1	0	2	0	1	1	0	1
6	1	2	2	3	2	0	0	1	1
7	1	2	2	2	0	1	1	1	0
8	0	0	1	1	0	1	1	0	1
9	0	1	0	3	2	0	0	1	0
10	2	2	2	3	2	0	0	1	0

Table 3.1: A Truth Table to Example 13.1

When $B = Y$, the CON relation is denoted as P_F and is called the output partition of the function F. As the blocks of output partition P_F are conjunct, the *consistence classes* of their CON relation are called an *r-partition* on the set M.

Conventions used in denoting r-partitions and their typical operators are the same as in the case of partitions i.e. an r-partition on a set M may be viewed as a collection of non-disjoint subsets of M, where the set union is M. Thus r-partition concepts are simple extensions of the partition algebra [?], with which reader's familiarity is assumed.

Especially the relation less than or equal to holds between two r-partitions Π_1 and Π_2 ($\Pi_1 \leq \Pi_2$) iff for every block of Π_1, in short denoted by $B_i(\Pi_1)$, there exists a $B_j(\Pi_2)$ such that $B_i(\Pi_1) \subseteq B_j(\Pi_2)$.

If Π_1 and Π_2 are partitions, this definition reduces to the conventional ordering relation between two partitions.

This points out the main difference between the completely and incompletely specified Boolean functions. While the equivalence classes of partitions in completely specified functions consist of disjoint subsets, the subsets of consistent minterms for partially specified functions may be overlapping. This is the reason for generalizing the typical partition description.

To present a Boolean function F, i.e., functional dependence between outputs Y and inputs X, usually described by formula $Y = F(X)$, the table specification should be consistent.

A logic specification table is consistent iff for every pair of row vectors $r_1 = (x_1, y_1)$, $r_2 = (x_2, y_2)$, the equality $x_1 = x_2$ implies $y_1 \sim y_2$ (i.e. for every $m_1, m_2, m_1 = m_2$ implies $F(m_1) \sim F(m_2)$.

Example 3.2.1

Consider a multiple-valued function F given in Table 3.1.

Using the notations already defined for T:

- $M = \{1, ..., 10\}$,
- $X = \{x_1, ..., x_6\}$,
- $Y = \{y_1, y_2, y_3\}$,

and $V_{x_1} = \{0, 1, 2\}, V_{x_4} = \{0, 1, 2\}, V_{x_6} = \{0, 1\}$. Example IND relations for $B_1 = \{x_1\}$ and $B_2 = \{x_2, x_3\}$ can be written as :

$$P(B_1) = \{\{1, 2, 4, 5, 8, 9\}, \{3, 6, 7\}, \{10\}\}$$

$$P(B_2) = \{\{1\}, \{2, 8\}, \{3, 6, 7, 10\}, \{4\}, \{5, 9\}\}$$

	X_1	X_2	X_3	X_4	X_5	X_6	Y_1	Y_2	Y_3
1	0	0	0	0	0	0	1	1	0
2	0	0	1	1	0	0	1	1	0
3	1	2	2	0	1	1	0	1	-
4	0	1	1	0	0	1	0	1	1
5	0	1	0	2	0	1	1	0	1
6	1	2	2	3	2	0	0	1	1
7	1	2	2	2	0	1	-	1	0
8	0	0	1	1	0	1	1	0	1
9	0	1	0	3	2	0	0	1	0
10	2	2	2	3	2	0	0	1	0

Table 3.2: A second variant, with standard don't cares in outputs, of the Truth Table to Example 13.1

	X_0	X_1	X_2	Y_0	Y_1
0	1	0	0	0	1
1	1	0	1	0	1
2	-	1	-	0	-
3	0	1	-	0	0
4	-	1	1	0	-
5	-	1	0	0	0
6	0	0	-	1	0
7	0	-	-	-	0

Table 3.3: A Truth Table to Example 13.2

Proceeding in the same way for the output-consistency relation CON, we obtain the following r-partitions:

$$P_F(y_1) = \{\{1,2,5,7,8\},\{3,4,6,9,10\}\}$$

$$P_F(y_2) = \{\{1,2,3,4,6,7,9,10\},\{5,8\}\}$$

$$P_F = \{\{1,2,7\};\ \{3,4,6\};\ \{5,8\};\ \{9,10\}\}$$

On the other hand, if the respective output vectors are as it is shown in Table 3.2, then the corresponding r-partition P_F can be written:

$$P_F = \{\{1,2,7\};\ \{3,4,6\};\ \{5,8\};\ \{3,7,9,10\}\}$$

Definition 3.2.6 *Two output vectors of a generalized function, p and q, are said to be* consistent *if their respective entries, which are defined, are equal, i.e.* $(\forall i \in \{1,...,m\})$ *[* $p_i \cap q_i \neq \emptyset$ *] (m is the number of output variables)*

Consistency relation of the output vectors for generalized functions is denoted by $p \sim q$, the same symbol as one used before for the consistency relation of output vectors for functions.

Example 3.2.2

Let us analyze one more example (Table 3.3).

Some of the indiscernibility relations for input variables are:

The partition of input variable X_0 is the following:

X0 \ X1X2	00	01	11	10
0	10 -0	10 -0	00 0-	00 0- -0
1	01	01	00 0-	0-

Figure 3.1: The Karnaugh Map to Example 13.2

$$P(X_0) = \{\{0,1,2,4,5\};\ \{2,3,4,5,6,7\}\};$$

The partition of input variables X_1 and X_2 is the following:

$$P(X_1\ X_2) = \{\{0,6,7\};\ \{1,6,7\};\ \{2,3,5,7\};\ \{2,3,4,7\}\}$$

The partition of output signals is:

$$P_F = P(Y_0\ Y_1) = \{\{2,3,4,5,7\};\ \{0,1,2,4\};\ \{6,7\}\}$$

This is called the consistency relation of output signal.

As we can see from P_F, its sets are non-disjoint, we call this relation as a rough-partition (r-partition).

The corresponding Karnaugh map is shown in Figure 3.1.

Sometimes this notation can lead to troubles. For example: if we have cube -01 with output 01 and cube 10- with output 10, since this two cubes intersect, there is a conflict. We should watch for this in the situation of having DC sets both in input and output. This must be taken care of by the Reader program that will read some files taken from other sources.

3.3 Cube Diagram Bundles to Represent Discrete Mappings.

In this section we will present a new representation of Boolean and multiple-valued functions, called **Cube Diagram Bundles.** This representation is especially good for functions that are both very strongly unspecified and multiple-valued. The disjoint cubes of the original function as well as the original variables are re-encoded with a smaller number of new variables, so that several smaller BDDs are used to represent the function. This representation allows to efficiently implement algorithms based on the Cube Calculus and calculus of Rough Partitions.

As an application of this new general-purpose representation method we discuss in subsequent sections a generalized, goal-oriented multi-level decomposition, that makes also special use of two-input gates and respective decompositions. This unified approach includes the decompositions of Ashenhurst, Curtis, Steinbach et al, Luba et al, and Perkowski et al as special cases.

The presented representation can be applied to binary and multiple-valued functions, both completely specified, incompletely specified, and with generalized multiple-valued don't cares. Other applications of the new representation are presented briefly in section ??.

Boolean and multiple-valued functions that include very many don't cares are becoming increasingly important in several areas of applications [?]. The new representation and several synthesis methods based on it will be very useful in the minimization of such functions, and, as we believe, also in general-purpose logic synthesis.

In principle, two essentially different representation methods for Boolean functions have been successfully used in logic synthesis software: Cube Calculus (CC), and Decision Diagrams (DDs). Similarly, for multiple-valued logic, two representation methods have been proposed: Multiple-Valued Cube Calculus (Positional Notation), and Multiple-Valued Decision Diagrams. Various methods have been also proposed to extend these representations for standard input and output don't cares of multiple-valued functions.

All these representations are being continuously improved with time, and several variants of them have been invented and proved superior in some applications. For instance, XBOOLE system of Bochmann and Steinbach [?] introduces Ternary Vector Lists (TVLs), a variant of Cube Calculus with disjoint cubes, new method of encoding symbols of cubes in machine words, and new operations. They demonstrate the superiority of this new system on some applications. Similarly, Functional Decision Diagrams (FDDs), Kronecker Decision Diagrams (KDDs), Algebraic Decision Diagrams (ADDs), Moment Decision Diagrams (MDDs), and other Decision Diagrams (DDs) have been introduced and shown superior to the well-known Binary Decision Diagrams (BDDs) in several applications. Hartmanis and Stearns introduced the so-called *set systems* in the sixties and used them for decomposition [?]. Kohavi uses them under the name of *covers* and Luba et al call them *Rough Partitions*. Luba et al [?] implemented a functional decomposer in which Rough Partitions are used as a basic data structure for representation and manipulation of Boolean and multiple-valued functions. Truth Table Permutations to create BDDs are recently investigated [?, ?].

Cube calculus representation seems superior in problems where the synthesized circuit has a limited number of levels. Such problems include: Sum-of-Products (SOP) synthesis, Exclusive-Or-Sum-of-Products (ESOP) synthesis, state assignment for PLA realization, Conditional Decoder (CDEC) networks synthesis [?], Three level AND/NOT Networks with True inputs (TANT) networks synthesis [?], and other.

On the contrary, the Decision Diagrams are superior for general-purpose Boolean function manipulation, tautology, technology mapping, and verification. They are also becoming a general tool in many areas, also outside circuit design, that deal with discrete functions.

This section introduces a new represention of binary and multiple-valued functions. More generally - a representation for discrete mappings and for some restricted class of discrete relations. We call this new representation the *Cube Diagram Bundles (CDB)*. *Cube* - because they operate on cubes as atomic representations, *Diagrams* - because they use Decision Diagrams (of any kind) to represent sets.

Bundles - because several diagrams and other data are used together to specify a function or a set of functions.

CDBs are related to four representations known from literature:

- Cube Calculus [?],
- Decision Diagrams [?],
- Rough-Partitions [?],
- and Boolean Relations [?].

This new representation is general and can be applied to both binary and multiple-valued functions **in the same way** (multiple-valued functions can also represent finite state machines). The CBD representation allows to add and remove variables (i.e., inputs and functions) *during* the synthesis process, which would be difficult or inefficient using other representations. All operations are reduced in CBDs to set-theoretical operations on decision diagrams that constitute their part.

Let us observe, that the meaning of a representation in an algorithm is two-fold. First, it allows to compress data - the switching functions - so that the algorithm becomes tractable in time or in space. Secondly, any representation introduces certain bias for function processing, making some algorithms particularly suited for some representations, and less for the other ones. For instance, it is very easy to find a supercube of two cubes in Cube Calculus, but it is more difficult to accomplish this using the representation of Rough Partitions. While finding prime implicants, cubes are naturally suited to search for prime implicants from the smallest to the largest cubes, while the rough partitions are better suited to generate primes from the largest to the smaller ones. Rough partitions and BDDs are better than Cube Calculus to find cofactors, and so on.

We believe that it is not possible to create a single represention that will be good for all applications, and the progress of various representation methods in past years seems to support this opinion. Therefore, here we concentrate on an area that has not found sufficient interest until very recently, but one that in our opinion will be quickly gaining in importance: namely, the binary and multiple-valued, very strongly unspecified functions. We will call this class *SUF - Strongly Unspecified Functions.* Such functions occur in Machine Learning (ML) [?, ?], Knowledge Discovery in Databases (KDD), Artificial Intelligence (AI), and also in some problems of circuit design, such as realization of cellular automata. One can observe that many well known problems in logic synthesis can be also converted to binary SUF functions: for instance every multi-output function can be converted to a single output binary SUF. SUF functions are manipulated while solving some decision problems and Boolean equations. Also, every multiple-valued input function can be converted to a binary SUF. Multiple-Valued SUF occur in ML and KDD areas. Since state machines can be represented as Multiple-Valued SUF, we believe that there exist applications where state machines with very many don't cares are processed. Multiple-valued functions occur also in many problems of logic synthesis (such as for instance encoding for multiple-level logic), and are thus not necessarily related to designing circuits that have multiple-valued signals. We believe, therefore, that the concepts that we introduce here (mainly to use them in Machine Learning applications), will find in future more applications in circuit design problems as well.

One can think about a discrete function as a two-dimensional table - see Table 3.4, in which the (enumerated) rows correspond to the elements of the domain (the minterms) or to certain groups of elements of the domain (the cubes). The columns of the table correspond to the input and output variables. Sometimes, they correspond also to intermediate (auxiliary) variables, which can be (temporarily) treated as input or output variables. Let us observe, that such table is in a sense realized in cube calculus: where the cubes correspond to the rows. The disadvantages of cube calculus include however:

- some large multilevel functions, such as an EXOR of many variables, produce too many cubes after flattening, so that their cube arrays cannot be created,

	a	b	c	d	f
0	0	0	0	0	0,1
1	0	1	0	0	1,2
2	1	1	0	0	0
3	0	0	0	1	0,3
4	1	1	0	1	0,3
5	1	0	0	1	0,4
6	1	0	0	0	0,3
7	0	0	1	1	1,3
8	0	1	1	1	0,1
9	0	0	1	0	2,3
10	1	0	1	0	1,4
11	0	1	1	0	2,3

Table 3.4: A Table of a Multiple-Valued-Output Function with Generalized Don't Cares.

- it is difficult to add and remove input and output variables to the cubes dynamically in the synthesis process,
- in strongly unspecified functions we would need relatively few but very long cubes,
- column-based operations are global, and therefore slow.

As far as we know, Rough Partitions (r-partitions, or RP) have been for the first time used in a program by Luba [?]. This representation stores r-partitions $\pi(v_i)$ for all input and output variables v_i as lists of ordered lists. Each upper level list represents an r-partition $\pi(v_i)$ for variable v_i, and lower level lists correspond to the blocks of this partition. A block of partition $\pi(v_i)$ includes *numbers* of rows of the table that have the same value VAL in the column corresponding to variable v_i. For instance, in ternary logic there are three blocks that correspond to values $VAL = 0$, $VAL = 1$, and $VAL = 2$, respectively. All operations are next performed on these r-partitions using r-partition operations that extend the classical Partition Calculus operations of product, sum and relation $\leq$ of Hartmanis and Stearns. Blocks included in other blocks are removed. The origination of a block is lost (by the origination of a block we mean which values of input variables this block comes from). This makes some operations in this representation not possible, and some other not efficient. Let us observe, that RP are not preserving all information about the function, they are a an abstraction useful for decomposition, but cannot be used as a general representation such as cubes of BDDs. This is an auxiliary data structure and table is still necessary to reconstruct the function.

Our other source of inspiration while creating this representation was the concept of the Generalized Don't Cares. In binary logic, a single-output function F has two values: F^0 and F^1, and there exists one don't care $F^{\{0,1\}}$ that corresponds to a choice of any of these two values. Analogously, in a three-valued logic, function F has three values: F^0, F^1, F^2 and there exist the following combinations of values: $F^{\{0,1\}}$, $F^{\{0,2\}}$, $F^{\{1,2\}}$, and $F^{\{0,1,2\}}$. The last one corresponds to a classical don't care, and will be called a standard don't care in the text. The other three combinations, however, have been not discussed before. We will define them all as the *generalized don't cares*. Similarly, the concept of generalized don't cares can be applied to k-ary logic for any value of k. This concept has applications for instance in Machine Learning and Knowledge Discovery from Databases. It is also close to Boolean Relations and MV Relations introduced by Brayton [189]. In this section, a multi-valued, multi-output function F with generalized don't cares will be referred to as a function.

In CDB representation function F is represented as a record of:

1. A pointer to a list $Var(F)$ of primary input variables on which the function depends. The variables are sorted lexicographically.

2. A pointer to a list $Inp(F)$ of vectors of primary input columns. The vectors are in the same order as the input variables. Each vector has as many positions as the corresponding variable has values, and the positions are sorted starting from 0 to k-1, where k is the number of values. Each position of the vector is a pointer to a DD. These are called "input value DDs."

3. A pointer to a list $Out(F)$ of vectors of output columns. This list is analogical to the $Inp(F)$ list. The DDs in the vectors in this list are called "output value DDs." The "input value DDs" and the "output value DDs" are called "value DDs".

(An example of a CDB representing a function F from Table 3.4 will be given below).

The representation of Luba has been used only for decomposition, and CDBs are a general-purpose function representation designed for speed and data compression - there are then several differences of CDBs and the representation of Luba. For efficiency of operations, the CDB of F stores also the list of primary input variables on which F depends (some of them can be still vacuous). It stores vectors of function values, and not rough partitions. This means, inclusion operations on blocks are not performed, and we keep track on the origin of each block - what value of the variable it corresponds to. In case of cofactors, we store then the cofactor functions, and not their equivalence classes. CDBs represent functions with generalized don't cares, while Luba represents only classical don't cares (this is reflected in the way how DDs for output variables are created in CDBs). In essence, rough partitions can be also used to represent generalized don't cares, For example, the rough partition

$$\{\overline{0,1,2}; \overline{1,2,2}; \overline{2,3,4}\} = \{\{0,1,2\}; \{1,2,3\}; \{2,3,4\}\}$$

represents the function:

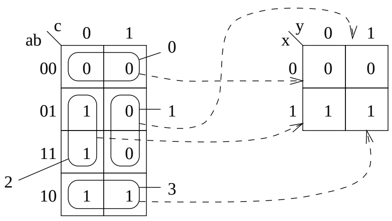
cube_number	output
0	{1}
1	{1,2}
2	{1,2,3}
3	{2,3}
4	{3}

Another difference of CDBs and RPs is that the rows (their numbers) correspond in Luba's approach to minterms or arbitrary cubes, while they correspond to disjoint cubes in CDBs. All sets are represented as ordered lists in RP and as Decision Diagrams in CDBs. Currently we use standard BDDs, but any kind of decision diagrams can be used. Because the sets are represented as DDs, CDBs introduce new variables to realize these DDs with. They are called the *secondary input variables.* The number of these variables is usually much smaller than the number of primary input variables, and the complete freedom of encoding rows with these new variables allows to minimize the size of all BDDs. This property is totally missing in Rough Partitions and exists only in [?, ?]. While the authors from [?, ?] solve it as a truth-table permutation problem, we solve it as a cube encoding problem, which is more general.

Example 3.3.1

The first example illustrates a table and encoding for a binary-input, binary-output completely specified function. A Kmap with primary input variables a, b, and c is shown in Fig. 3.2a. This table has four disjoint cubes. Two are OFF cubes, encoded by 0 and 1, and two are ON cubes, encoded by 2 and 3. As the results of encoding of primary cubes with secondary input variables, x and y, the new map shown in Fig. 3.2b is created. Figure 3.2a,b shows clearly how cubes of the first map are mapped (encoded) to the minterms of the secondary map. The table for the function from Fig. 3.2a is shown in Fig. 3.2c, and the encodings of its rows to secondary input variables is shown in Fig. 3.2d.

The function is specified as the following CDB.



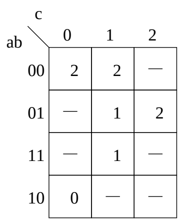
a) b)

row (cube)	input variables a	b	c	output variable f
0	0	0	-	0
1	-	1	1	0
2	-	1	0	1
3	1	0	-	1

c)

x	y
0	0
0	1
1	0
1	1

d)



e)

row	a	b	c	f
0	1	0	0	0
1	-	1	1	1
2	0	1	2	2
3	0	0	0,1	2

f)

x	y
0	0
0	1
1	0
1	1

g)

Figure 3.2: Maps and encodings for functions from Example 3.1 and Example 3.3, (a) Function from Example 3.1 with primary input variables, (b) secondary space with secondary input variables x and y, (c) table of function f, (d) encoding of primary cubes, (e) Function from Example 3.3 with primary input variables, (f) table of this function, (g) encoding of primary cubes

$Var(F)$ = {a,b,c}.
$Inp(F)$ = { [pointer to BDD for {0,1,2}, pointer to BDD for {1,2,3}], ;;;; for variable a.
[pointer to BDD for {0,3}, pointer to BDD for {1,2}], ;;;; for variable b.
[pointer to BDD for {0,2,3}, pointer to BDD for {0,1,3}] } ;;;; for variable c.
$Out(F)$ = { [pointer to BDD for {0,1}, pointer to BDD for {2,3}] } ;;;; for variable f.

Example 3.3.2

The second example, see Table 3.4, presents a table for a binary-input, 5-valued-output incompletely specified function with generalized don't cares. The map for this function is in Fig. 68. In this case:

$Out(F)$ = { [pointer to BDD for {0,2,3,4,5,6,8}, pointer to BDD for {0,1,7,8,10}, pointer to BDD for {1,9,11}, pointer to BDD for {3,4,6,7,9,11}, pointer to BDD for {5,10}] } ;;;;;; for 5-valued output variable f.

Example 3.3.3

The third example presents a table, Fig. 3.2f, for an incompletely specified function with standard don't cares from Fig. 3.2e. It has two binary input variables, a and b; a ternary input variable c, and a 3-valued-output. The encoding of primary cubes with secondary input variables x and y is shown in Fig. 3.2g.

The function is specified as the following CDB.

$Var(F)$ = {a,b,c}.
$Inp(F)$ = { [pointer to BDD for {1,2,3}, pointer to BDD for {0,1}], ;;;; for binary variable a.

[pointer to BDD for {0,3}, pointer to BDD for {1,2}], ;;;; for binary variable b.
[pointer to BDD for {0,3}, pointer to BDD for {1,3}, pointer to BDD for {2}] }
;;;; for ternary variable c.

$Out(F)$ = { [pointer to BDD for {0}, pointer to BDD for {1}, pointer to BDD for {2,3}] }
;;;; for ternary variable f.

Let us observe, that standard don't care positions are not stored in tables and CDBs, but generalized don't care positions are stored.

The following points about CDBs are important to note:

1. The definition of a Binary Cube Diagram Bundles and Multiple-valued Cube Diagram Bundles (mvCDBs) are **exactly the same.** Therefore, the same manipulations can be applied to them.

2. All Boolean operations on CDBs can be easily realized as set-theoretical operations on corresponding DDs. Therefore, part of our research plan is to test which of the known DDs are the best ones to represent the sets that occur in logic decomposition.

3. The important concept of a *cofactor* is calculated, that also uses only the set-theoretical operations. The cofactor CF of F with respect to cube C is calculated as follows:

   CF := DD(F) $\wedge$ DD(C), $\qquad Var(CF) := Var(F)$ - $Var(C)$.

   Let us observe that since the cofactor CF calulated with respect to cube C do not depend on variables from cube C, we execute the second operation that removes variables from C from the set of arguments of cofactor CF, so it will be known for the next operations on what primary variable (at most) the BDD depends. It can be noted that this operation is unique to CDBs.

4. The concepts of derivative, differential, minimum, maximum, k-differential, k-minimum, and k-maximum of a function [?] are also realized. Since all these operations are based on the cofactor operator and the set (Boolean, multi-valued) operations, they can be easily realized with respect to points 2 and 3 above.

5. A CDB represents a set of cubes. Each true minterm in DD(F) is an ON cube in function F on primary variables, and each false minterm in DD(F) is an OFF cube.

6. In CDBs, there is no difference in the representation of primary input variables, auxiliary variables and output variables. This property is not unique to CDBs, and exists for example also for Boolean expressions, but it does not exist in cubes and in Decision Diagrams. This makes that CDBs become useful in several new applications. For instance, CDBs allow to represent easily functions that are defined on some new intermediate functions (for instance two-input Boolean operators) on the input variables of the function. One can just add new "columns" representing these new functions to the CBD. This means, new sets of "value DDs" are added for auxiliary variables. For instance, to check a separability of a function to unate functions, one just introduces new input variables, like $A_1 = \overline{A}$. Similarly, one can create new variables $A_2 = a \oplus b$, $A_3 = a \cdot b$, $A_4 = a + b$ by adding them to the CDB. This property allows also to realize algorithms that operate on output functions as on variables, to use auxiliary functions for synthesis, and to re-use the existing functions in the synthesis process. This can be for instance used in spectral methods that build EXOR-based preprocessors to multi-input, multi-output functions.

7. There exist one more variant of CDBs, that we call the *encoded CDBs*, or *ECDBs*. For instance, when there are four values of variable v_i, the standard CDB would create four DDs for this variable. However, the ECDB would create only two "encoded" DDs. This obviously saves space. Operations on ECDBs are very similar to those on CDBs, and they will be discussed in more detail in one of our forthcoming reports.

The input data are being read incrementally in the form of disjoint cubes of primary variables (rows of the "table"). During this process, the primary cubes are being heuristically *encoded* as minterms in the new space of secondary variables. The goal of this encoding is to simplify all DDs of the CDB. This is done in such a way that false minterms being encodings of all primary OFF cubes, are grouped near cell 0...0 (the minimum minterm in the space of new variables). Similarly, true minterms of new space being encodings of all primary ON cubes, are grouped near cell 1...1 (the maximum minterm in the new space). In addition, the larger the cube, the closer it should be located to the minimum or the maximum cell, respectively. If a cube is an ON cube is some and an OFF cube in some other output functions it is located closer to cell 0...0 or 1...1, depending in how many functions it is ON or OFF - the predominantly ON cubes are placed near to 1...1, and so on. Moreover, the cube encoding algorithm attempts to fill those cubes in the new space that are of smaller Hamming distances with either the minimum or the maximum minterm, in such a way that as many as possible of these cubes become completely filled with the same types of minterms. These all attempts are done for all value DDs of the CDB in parallel. This problem will be also presented in one of our forthcoming reports.

3.4 Basic Theorem.

Let F be a multiple-valued function representing functional dependency $Y = F(X)$, where X is the set of multiple-valued input variables and Y is the set of binary output variables. Let $X = A \cup B$, $A \cap B = \emptyset$ and $C \subseteq A$. C is a so-called set of shared (or, overlapping) variables.

We say that there is a functional decomposition of F iff

$$F = H(A, G(B, C)) = H(A, Z) \tag{3.7}$$

where G and H denote functional dependencies: $G(B, C) = Z$ and $H(A, Z) = Y$ and Z is the set of two-valued variables. If in addition, $C = \emptyset$, then H is called a simple disjoint decomposition of F.

In other words we try to find functions G and H, such that G depends on variables in $B \cup C$, whereas H depends on variables in A and variables in Z, where Z is the set of binary outputs of G. The outputs of the function H are consistent with those of function F. So the function F can be implemented as a multi-level PLA structure, but in contrast to the method proposed by Ciesielski [?], the circuits implementing components G and H, in general, are circuits with multiple-valued input, two-valued output variables. Moreover as the decomposition process can be applied iteratively, the first level may contain an arbitrary number of PLAs (components G) and the second level contains a single PLA. The structure of the decomposed circuit is not limited to that of PLAs and standard decoders as in [?], however the PLAs with decoders are the most suitable devices for implementing MVL circuits. So the method is not confined to PLA synthesis and can be considered as a general function decomposition approach.

Particularly the proposed decomposition process can be applied to circuits following the assignment of the input variables to the decoders which additionally reduces the area of the final circuit. With the proposed structure, a multilevel circuit can be realized as the serially connected components G and H, where the outputs of G form intermediate variables. Thus we will refer to this process as to a *serial decomposition.*

The following theorem [?] states the sufficient condition for the existence of a serial decomposition.

Theorem 3.4.1 *Function G and H represent a serial decomposition of function F , i.e. $F = H(A, G(B, C))$, if there exists a partition $\Pi_G \geq P(B \cup C)$ such that*

$$P(A) \cdot \Pi_G \leq P_F \tag{3.8}$$

where all the partitions are over the set of cubes and the number of two-valued output variables of component Π_G is equal to $[\log_2 L(\Pi_G)]$, here $L(\Pi)$ denotes the number of blocks of partition Π, and $[x]$ denotes the smallest integer equal to or large than x.

Here $[\log_2 L(\Pi_G)]$ gives us the number of output signals from function G.

It is evident that partition Π_G represents component G, and the product of partitions $P(A)$ and Π_G corresponds to H. The truth tables of the resulting components can thus be obtained from these partitions. This is trivial for cubes being minterms, but is not trivial for cubes being not minterms. Some cubes may belong to many blocks of Π_G, thus leading to the generalized don't cares. For instance, cube C_i that belongs to blocks B_3 and B_4 creates a generalized don't care with values 3,4. In the realization, any of these values can be selected, but a smarter approach is possible in order to improve the encoding of G. This problem will be presented in more detail in section ??, and especially in our forthcoming reports. It can also be shown that this theorem is true regardless the initial set of cubes.

Note, that we are not concerned here with the method of finding partition Π_G. It will be explained in subsequent sections.

Example 3.4.1

X_2	X_3	X_4	X_5	G
0	0	0	0	0
0	1	1	0	0
2	2	0	1	0
1	1	0	0	1
1	0	2	0	0
2	2	3	2	0
2	2	2	0	1
1	0	3	2	1

Table 3.5: A Truth Table of function G

X_1	X_6	G	Y_1	Y_2	Y_3
0	0	0	1	1	0
1	1	0	0	1	1
0	1	1	0	1	1
0	1	0	1	0	1
1	0	0	0	1	1
1	1	1	1	1	0
0	0	1	0	1	0
2	0	0	0	1	0

Table 3.6: A Truth Table of output functions

Let us decompose the function F of Table 3.1. For $A = \{x_1, x_6\}$, $B = \{x_2, x_3, x_4, x_5\}$, $C = \emptyset$, we have

$$P(A) = \{\{1,2,9\};\ \{3,7\};\ \{4,5,8\};\ \{6\};\ \{10\}\}$$

$$P(B) = \{\{1\};\ \{2,8\};\ \{3\};\ \{4\};\ \{5\};\ \{6,10\};\ \{7\};\ \{9\}\}$$

Consider

$$\Pi_G = \{\{1,2,3,5,6,8,10\};\ \{4,7,9\}\}$$

It can be easily verified that since $P(A) \cdot \Pi_G \leq P_F$, function F is decomposable as:
$F = H(x_1, x_6, G(x_2, x_3, x_4, x_5))$,
where G is a one-output function of four variables, and P_F is the output partition of function F.

The truth tables of components G and H can be obtained from partitions $P(A)$, Π_G, and P_F. Encoding the blocks of Π_G respectively as 0 and 1, we immediately obtain the truth table of function G; it is presented in Table 3.5. The truth table of function H can be derived by reencoding input vectors of F using an intermediate variable g. The truth table obtained in this way is shown in Table 3.6.

Example 3.4.2 For another example, in Table 5, we have:

$$P(B) = P(X_1X_2) = \{\{0,6,7\};\ \{1,6,7\};\ \{2,3,5,7\};\ \{2,3,4,7\}\};$$

$$P(A) = P(X_0) = \{\{0,1,2,4,5\};\ \{2,3,4,5,6,7\}\};$$

if

$$\Pi_G = \{\{0,1,6,7\};\ \{2,3,4,5,7\}\};$$

then

$$P(B) < \Pi_G$$

$$P(A) \cdot \Pi_G = \{\{0,1\};\ \{6,7\};\ \{2,3,4,5,7\}\} < P_F = \{\{2,3,4,5,7\};\ \{0,1,2,4\};\ \{6,7\}\}$$

Therefore this is a feasible decomposition with boundset $B = \{X_1, X_2\}$ and free set $A = \{X_0\}$. The number of output signals from G is $[\log_2 2] = 1$.

The questions are:

1. How to find boundset B, C?
2. How to find Π_G?

Definition 3.4.1 *Let B be a subset of the set of inputs X, an input partition generated by set B is denoted as:*

$$P(B) = \prod_{x \in B} P(x)$$

where $\prod$ denotes the product of partitions. $P(x)$ is a partition for variable x.

Definition 3.4.2 *Let B be a subset of outputs set Y , the output-consistency relation associated with B , i.e. $P_F(B)$, is called a $rough - partition$ (r-partition).*

Maximal Compatible Classes

To find the corresponding set of inputs for component G, we have to find $P(B \cup C)$, such that there exists $\Pi_G \geq P(B \cup C)$ that satisfies Theorem 15.1. To solve this problem, consider a subset of primary inputs, $B \cup C$, and the q-block partition $P(B \cup C) = (B_1, B_2, \ldots, B_q)$ generated by this subset. Then we use a relation of compatibility of partition blocks to verify whether $P(B \cup C)$ is suitable for the decomposition or not.

Definition 3.4.3 *Compatibility relation: Two blocks B_i and $B_j \in P(B \cup C)$ are compatible iff partition $P_{ij}(B \cup C)$ obtained from partition $P(B \cup C)$ by merging blocks B_i and B_j into a single block satisfies $P(A) \cdot P_{ij}(B \cup C) \leq P_F$.*

Example 3.4.3

For example, in Table 5, repeated in Table 3.7, we have:

$$P(X_1\ X_2) = P(B) = \{\{0,6,7\};\ \{1,6,7\};\ \{2,3,5,7\};\ \{2,3,4,7\}\} = \{B_1, B_2, B_3, B_4\};$$

If we merge B_1 and B_2 together, we get:

$$P_{12}(B) = \{\{0,1,6,7\};\ \{2,3,5,7\};\ \{2,3,4,7\}\};$$

$$P(A) \cdot P_{12}(B) = \{\{0,1\};\ \{6,7\};\ \{2,3,5,7\};\ \{2,3,4,7\}\} < P_F$$

therefore B_1 and B_2 are compatible, denoted as $B_1 \sim B_2$.

The same way we can check the compatibility relation of other blocks in $P(B)$.

	X_0	X_1	X_2	Y_0	Y_1	g
0	1	0	0	0	1	0
1	1	0	1	0	1	0
2	-	1	-	0	-	1
3	0	1	-	0	0	1
4	-	1	1	0	-	1
5	-	1	0	0	0	1
6	0	0	-	1	0	0
7	0	-	-	-	0	0

Table 3.7: A Repeated Table for Example 15.3

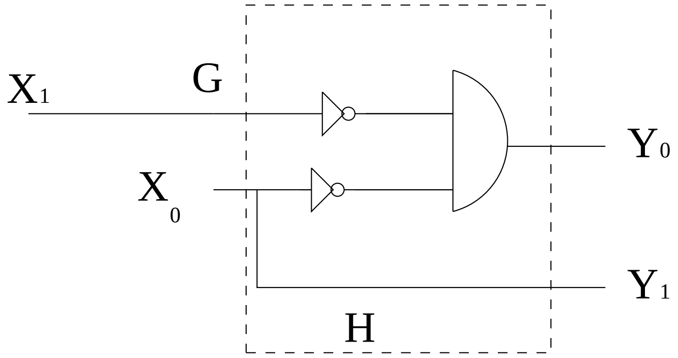


Figure 3.3: Decomposition of function f from Example 15.3

The result is: $B_1 \sim B_2$, $B_3 \sim B_4$.

A compatible class is called Maximal Compatible Class (MCC) if and only if it can not be properly covered by any other compatible class.

Thus we have MCC1 $= \{B_1, B_2\}$, MCC2$=\{B_3, B_4\}$. From this we can find the minimal cover, i.e.$\{\{B_1, B_2\}, \{B_3, B_4\}\}$, and $\Pi_G = \{\{0, 1, 6, 7\};\ \{2, 3, 4, 5, 7\}\}$. We can easily see that Π_G corresponds to a 1-output function G. Therefore we have $F = (X_0, G(X_1, X_2))$.

Encoding of Compatible Classes

As the truth table of H is partially constructed of G block outputs and the G blocks output are nothing but the codes allotted during the encoding process, it is very important to find right code to reduce the number of logic blocks needed to implement a truth table.

It is easy to find functions G and H from Table ?? (Old Table 8). $g = X_1$, $Y_0 = \bar{X_1}\bar{X_0}$, $Y_1 = X_0$. The circuit of the decomposition is shown in Figure 3.3. (OLD FIG.11.3) As we see, X_2 from the bound set is, in essence, a vacuous variable. After the decomposition, all vacuous variables should be removed from the inputs of all blocks. Note that a variable X_i can be vacuous in block B_r but not in block B_s.

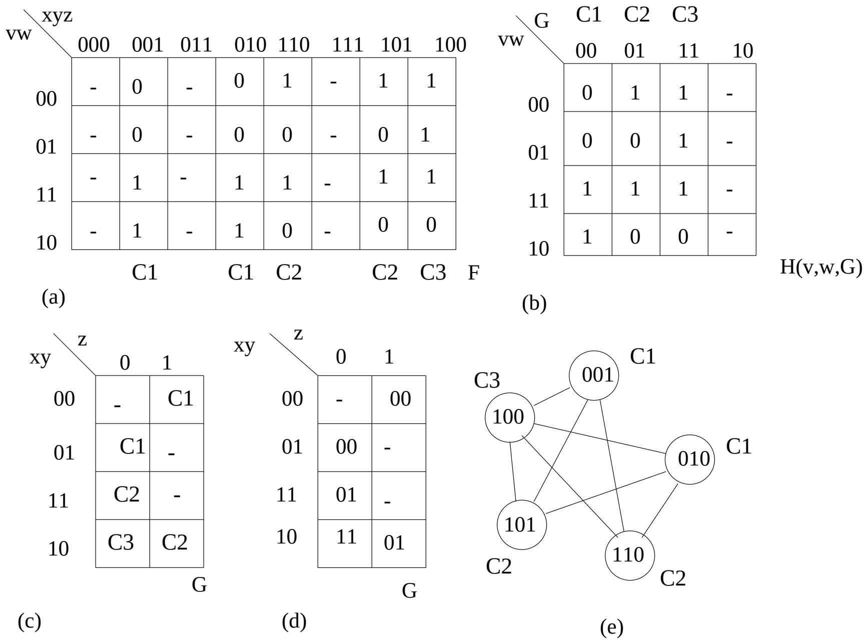


Figure 3.4: Example of treating columns of don't cares: (a) Function $F(x,y,z,v,w)$, (b) Function H with a column of don't cares, (c) Function G with symbolic values, (d) Function G with binary values, (e) The incompatibility graph

3.5 Approaches to Column Compatibility Problem.

It should be one of the goals of this research to develop an understanding how different decompositions, search strategies and constraints affect the final value of the DFC, the generalization abilities of the algorithm, and its learning time.

The quality of the attained column minimization seems to be of a crucial importance here.

The column minimization problem has been reduced in the past to one of these:

- incompatibility graph coloring,
- incompatibility graph maximum independent set partitioning.
- compatibility graph maximum clique partitioning.
- compatibility graph maximum clique covering.

All these problems are mathematically equivalent, but can be solved with heuristic approaches that will perform better or worse on particular categories of graphs.

The *Compatibility Graph* is a graph which illustrates the compatibility relationship among columns of the decomposition chart. Because two columns can be either compatible or not compatible, the compatibility graph and the incompatibility graph are the mutual complements. It means, the sets of *edges* of these graphs are disjoint and the union of the sets of edges creates a full graph.

In Figure 3.4, the number in each node (denoted by a circle) is the column number. The letter beside the circle is the color assigned to the node (column) after the graph coloring. The Set Covering Problem will be discussed in section 3.6. The Graph Coloring Problem will be discussed in section 3.7.

The problem to minimize the number of columns has been discussed by all authors of decomposition programs. Some of them discuss it as the set covering of a compatibility graph with nodes corresponding to subfunctions of cofactors, some other authors represent it as a graph coloring of an incompatibility graph that is a complement of the above compatibility graph. These formulations are mathematically, but not neccessarily heuristically, equivalent.

It seems, however, that there is something wrong with both above formulations. At least, the columns of don't cares should be separated from coloring and covering and next encoded as don't cares. See Fig. 3.4. Columns of don't cares are **not** used as nodes in the coloring graph. This way, more don't cares is preserved for both function G and function H.

3.5.1 Troubles with the Column Compatiblity Problem.

This problem is not known from the literature and was observed in WL. Let us assume that we have 100 input variables and 99.99999999% don't cares. This means, most columns are columns of don't cares. The problem was observed for the parity function with about 100 variables. For parity there is obviously no variable partitioning problem and all the experienced troubles to find the obvious decomposable minimum solution of a negated EXOR are because of the bad column grouping method.

The question is - "how to group columns to avoid these troubles?"

The following ideas should be considered:

1. Do not combine columns of only don't cares with other columns. See Fig. 3.4.
2. Solve column minimization together with the column assignment. For instance, in essence we do not need to minimize absolutely the column multiplicity, only to get it below $k = 2^r$, so if r =3, 8 is as good as 5.
3. Another approach - weighted covering, combine only columns that have 0's corresponding to 0's and 1's corresponding to 1's, and not 0's corresponding to -'s and 1's corresponding to -'s.
4. Use greedy algorithm that selects the largest clique, remove it, and iterate. Similarly to the "disjoint star" covering algorithm of Michalski.
5. Combine step-by-step the most similar columns. Take into account the number of agreements (0 and -, and 1 and - are not treated as agreements).

These subjects will be covered in detail in the forthcoming chapters.

3.6 The Set Covering Approach to the Column Compatibility Problem

3.6.1 The compatibility relation.

As it is seen from Theorem ?? (Old Theorem 15.1), the main task of decomposition is to verify if a subset of input variables for function G which, when applied as a subfunction to function H will generate the final function F. In other words, the goal is to find paritition $P(B)$, such that there exists $\Pi_G \geq P(B)$ that satisfies condition (10) in Theorem ?? (Old Theorem 15.1). To solve this problem, consider a subset of input variables, B, and an m-block partition $P(B) = \{B_1; B_2; ...; B_m\}$ generated by this subset of variables.

A relation of compatibility of partition blocks will be used to verify whether or not partition $P(B)$ is suitable for serial decomposition.

Definition 3.6.1 *Two blocks $B_i, B_j \in P(B)$ are* **compatible** *if and only if partition obtained from partition $P(B)$ by merging blocks B_i and B_j into a single block B_{ij} satisfies condition (10) in Theorem ?? (OLD Theorem 15.1), i.e., if and only if:*

$$P(A) \cdot P_{ij}(B) \leq P_F \tag{3.9}$$

Definition 3.6.2 *A subset of n partition blocks, $B = \{B_{i_1}, B_{i_2}, ..., B_{i_n}\}$, where $B_{i_j} \in P(B)$, is* **a class of compatible blocks for partition** $P(B)$ *if and only if all blocks in B are pairwise compatible.*

Definition 3.6.3 *A compatible class is called a* **Maximal Compatible Class** *(MCC) if and only if it cannot be properly covered by any other compatible class.*

The set $M = \{MCC_1, ..., MCC_r\}$ of all Maximal Compatible Classes can be thus formed from the set of all compatible pairs (B_i, B_j), which in this case can be interpreted as arcs of a compatibility graph $G = (B, COM)$, where elements of B represent its vertices, COM represents the compatibility relation and where two vertices are connected by an arc iff B_i and B_j are compatible, i.e. $(B_i, B_j) \in COM$. In such a formulation the procedure for computing the MCCs can be summarized as follows.

Procedure COM to find the Compatibility Relation.

Let S_j be the set containing all the blocks B_i for which B_j and B_i are compatible and $i < j$.

1. A compatible list (CC-list) is initiated with one set containing the first block as its only element.
2. If S_j is an empty set, a new class consisting of one block B_j is added to the CC-list before moving to the next S. Since block B_j is in conflict with blocks B_1 to B_{j-1}, it is placed in a one element set.
3. If S_j is not empty, its intersection with every member CC of the current CC-list, $S_j \cap$ CC, is calculated. If the intersection is empty, the sets are not changed. Otherwise a new class is created by adding to the intersection the one-element set B_j.

Example 3.6.1

For the function of Example ?? (old Example 13.1), and the same sets A, B as in Example ?? (OLD EXample 15.1), let us denote the blocks of $P(B)$ as B_1,...,B_8 respectively, i.e. $P(B) = \{B_1, ..., B_8\}$, where B_1 = {1}, B_2 = {2,8}, B_3 = {3}, B_4 = {4}, B_5 = {5}, B_6 = {6,10}, B_7 = {7}, B_8 = {9}.

Let us check if B_2 = {2,8} and B_4 = {4} are compatible. We have:

$$P_{24}(B) = \{\{1\};\ \{2,4,8\};\ \{3\};\ \{5\};\ \{6,10\};\ \{7\};\ \{9\}\}.$$

As

$$P(A)\ \cdot\ P_{24}(B)\ = \{\{1\};\ \{2\};\ \{9\};\ \{3\};\ \{7\};\ \{4,8\};\ \{5\};\ \{6\};\ \{10\}\}$$

does not satisfy

$$P(A)\ \cdot\ P_{24}(B)\ \le\ P_F\ (for\ P_F,\ see\ Example\ 13.1),$$

B_2 and B_4 are not compatible.
For $B_2 = \{2,8\}$ and $B_5 = \{5\}$, we obtain

$$P_{25}(B) = \{\{1\};\ \{2,5,8\};\ \{3\};\ \{4\};\ \{6,10\};\ \{7\};\ \{9\}\}$$

and

$$P(A)\ \cdot\ P_{25}(B) = \{\{1\};\ \{2\};\ \{9\};\ \{3\};\ \{7\};\ \{4\};\ \{5,8\};\ \{6\};\ \{10\}\}\ \le\ P_F$$

Thus, B_2 and B_5 are compatible. In a similar way we check the compatibility of each pair of blocks in partition $P(B)$ finding the following relation:

$$\begin{aligned}COM = \{&(B_1,B_2),\ (B_1,B_3),\ (B_1,B_4),\ (B_1,B_5),\ (B_1,B_6),\ (B_1,B_7),\ (B_2,B_3),\ (B_2,B_5),\ (B_2,B_6),\\ &(B_2,B_7),\ (B_3,B_4),\ (B_3,B_5),\ (B_3,B_6),\ (B_3,B_8),\ (B_4,B_6),\ (B_4,B_7),\ (B_4,B_8),\ (B_5,B_6),\ (B_5,B_7),\\ &(B_5,B_8),\ (B_6,B_7),\ (B_6,B_8),\ (B_7,B_8)\}.\end{aligned}$$

The Compatibility Graph drawn from this relation is shown in Figure 3.5. There is an edge in the graph for each element of relation COM.

Hence, assuming that the blocks of P_G are denoted by their indexes we obtain the S_j sets and the corresponding MCCs (for the sake of simplicity we write S_j sets and CC-lists by indexes of B_j):

$$\begin{array}{ll}
S_1 = \emptyset & CC_1 = \{1\}\\
S_2 = \{1\} & CC_2 = \{1,2\}\\
S_3 = \{1,2\} & CC_3 = \{1,2,3\}\\
S_4 = \{1,3\} & CC_4 = \{1,3,4\},\{1,2,3\}\\
S_5 = \{1,2,3\} & CC_5 = \{1,2,3,5\},\{1,3,4\}\\
S_6 = \{1,2,3,4,5\} & CC_6 = \{1,2,3,5,6\},\{1,3,4,6\}\\
S_7 = \{1,2,4,5,6\} & CC_7 = \{1,2,5,6,7\},\{1,4,6,7\},\{1,2,3,5,6\},\{1,3,4,6\}\\
S_8 = \{3,4,5,6,7\} & CC_8 = \{5,6,7,8\},\{4,6,7,8\},\{3,5,6,8\},\\
 & \qquad\{3,4,6,8\},\{1,2,5,6,7\},\{1,4,6,7\},\{1,2,3,5,6\},\{1,3,4,6\}
\end{array}$$

As the last CC-list represents the Maximal Compatible Classes we conclude:

$MCC_1 = \{B_5, B_6, B_7, B_8\}$,

$MCC_2 = \{B_4, B_6, B_7, B_8\}$,

$MCC_3 = \{B_3, B_5, B_6, B_8\}$,

$MCC_4 = \{B_3, B_4, B_6, B_8\}$,

$MCC_5 = \{B_1, B_2, B_5, B_6, B_7\}$,

$MCC_6 = \{B_1, B_4, B_6, B_7\}$,

$MCC_7 = \{B_1, B_2, B_3, B_5, B_6\}$,

$MCC_8 = \{B_1, B_3, B_4, B_6\}$.

The next step of calculating the Π_G is a process of selecting a subset of MCCs that cover the set of all blocks of P_G i.e. $B = \{B_1, ..., B_m\}$. This procedure is based on the following observation. As for each $MCC_i \in M$, where $MCC_i = \{B_{i_1}, B_{i_2}, ..., B_{i_k}\}$, a partition:

$$\Pi = \{\{B_{i_1}, B_{i_2}, ..., B_{i_k}\}, \{B_{i_{k+1}}\}, ..., \{B_{i_m}\}\}$$

satisfies the inequality $P(A) \cdot \Pi \leq P_F$, then the partition Π_G satisfying the same inequality i.e. $P(A) \cdot \Pi_G \leq P_F$ and having the minimum number of blocks can be found by solving the following cover problem:

$$\bigcup MCC_j \ = \ B \ and \ k \ = min.$$

The minimal k ensures the minimum number of blocks of partition Π_G. In other words we try to find a subset of MMCs such that their union results in the set B.

The Clique Partitioning of the Compatibility Graph is shown in Figure 3.5b.

We will describe a selection MIN(M) = $\{MCC_{i_1}, MCC_{i_2}, ..., MCC_{i_r}\}$ of all MCCs in the form of the binary matrix M for which an element m_{ij} ($i = 1, ..., r = CARD(B)$, $j = 1, ..., t = CARD(M)$) is defined as follows:

$$m_{ij} = \begin{cases} 1 & , \quad if \ x_j \in C_i \\ 0 & , \quad otherwise \end{cases}$$

Thus, the M matrix is a 0-1 matrix determined by the MMCs and the problem transforms to the classical Boolean matrix covering problem, often called the Unate Covering problem [?, ?], where the goal is to select an optimal set L of MCCs corresponding to columns of M. Here a "column covering" L means that every row of M contains a "1" in some column which appears in L. More precisely, a column cover of binary matrix is defined as a set L of columns such that for every i:

$$\sum_{j \in L} m_{ij} \ \geq \ 1.$$

Cover L of M is in one-to-one correspondence with the selected subset of MCCs such that their union forms the set B.

The MIN(M) represents partition $\Pi_G = \{_1, ..., _k\}$ in the following way:

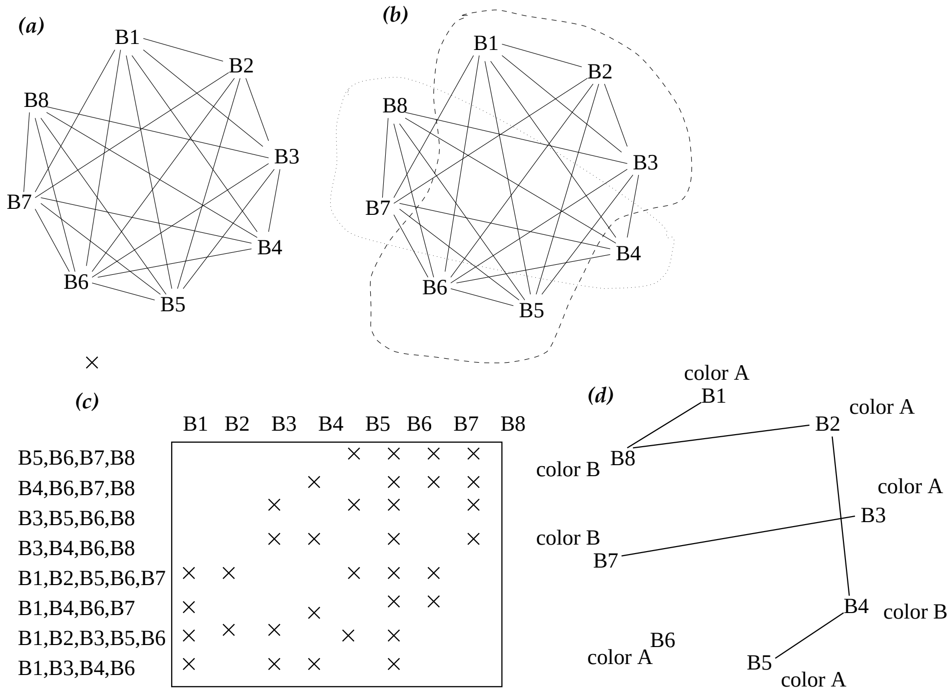
$$\beta = \{B_{i_1}, B_{i_2}, ..., B_{i_q}\}$$

is a block of Π_G if and only if $\beta \subseteq$ MCC and there is no β' such that $\beta \cap \beta' \neq \emptyset$. Thus blocks of Π_G can be created from MCC by eliminating the repeated elements of B in the minimal cover MIN(M). The final Π_G is a result of the union of objects forming a set of blocks included in any single block of Π_G.

Example 3.6.2

For the MCCs from Example ?? (OLD Example 18.1), the covering table is shown in Figure 3.5c. One of the minimal disjoint covers (clique partitions) is [MCC_7, MCC_2 - $\{B_6\}$] = $\{\{B_1, B_2, B_3, B_5, B_6\}$, $\{B_4, B_7, B_8\}\}$. The Π_G partition corresponding to this cover is: $\Pi_G = \{\{1, 2, 3, 5, 6, 8, 10\};\ \{4, 7, 9\}\}$, as it was assumed in Example ?? (OLD Example 15.1).

Example 3.6.3



	B1	B2	B3	B4	B5	B6	B7	B8
B5,B6,B7,B8					×	×	×	×
B4,B6,B7,B8				×		×	×	×
B3,B5,B6,B8			×		×	×		×
B3,B4,B6,B8			×	×		×		×
B1,B2,B5,B6,B7	×	×			×	×	×	
B1,B4,B6,B7	×			×		×	×	
B1,B2,B3,B5,B6	×	×	×		×	×		
B1,B3,B4,B6	×		×	×		×		

Figure 3.5: Comparison of techniques to solve the Column Minimization Problem: (a) The Compatibility Graph, (b) The clique partitioning of the Compatibility Graph, (c) The covering table with maximum cliques from the Compatibility Graph as rows and its nodes as columns, (d) The graph coloring formulation of the same problem using the Incompatibility Graph being the complement of the Compatibility Graph

The coloring formulation of the same problem, using the Incompatibility Graph, is shown in Figure 3.5d. As we see, the same solution can be found by making groups of nodes colored with the same color. It can be seen that the graph from Figure 3.5d has no cycles, and is thus 2-colorable. Thus, the existence of the Ashenhurst decomposition can be easily determined by a fast algorithm looking for the existence of cycles in the graph.

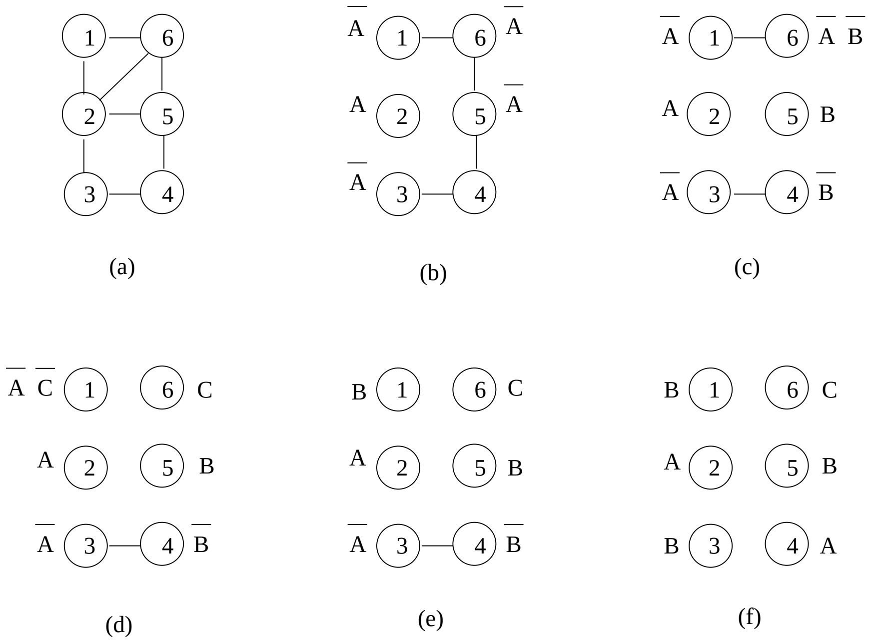


Figure 3.6: The incompatibility graph for coloring

3.7 Graph Coloring Approach to the Column Compatibility Problem.

The graph coloring approach to solve the Column Compatibility Problem was implemented in TRADE program. The problem of finding the smallest column multiplicity was reduced to the one of performing graph coloring with the minimum number of colors. The *Graph Coloring Problem*, [?, ?, ?, ?] defined as a coloring of nodes of a graph with colors in which every two nodes that are linked by an edge are assigned different colors. The *minimum graph coloring* is one with the minimum number of colors. The number of colors in the minimum coloring of graph G is called the *chromatic number of graph G*.

Graph coloring is an NP-hard problem. There has been a substantial research on it in order to find the algorithms for a quasi-optimum solution with the fastest possible speed. The method presented here is a fast graph coloring method. This method has been programmed and tested on many examples, it resulted in excellent colorings. The method found exact colorings for graphs in [?] and even found better coloring than the one claimed to be the minimal one in this book. The method is called the *Color Influence Method*.

The main idea of this method is to evaluate the influence of the color assignment to a node over the entire graph, and chose the color which results in a minimum influence. The *minimum influence* means that an assignment of a color to a node will produce a minimum increase of the number of color-in-bar's. The *color-in-bar's* (color restrictions) are the colors that the node cannot be assigned with. They are denoted by $\overline{A}$, $\overline{B}$..., $\overline{A}$ $\overline{B}$, $\overline{A}$ $\overline{C}$, ... as in Figure 3.6. After each assignment of color to a node, this node and its edges are removed, and the complexity of the graph is decreased. This is a greedy method with global evaluation. The next example is used to illustrate this method.

Figure 3.6a shows a graph that needs to be colored. The algorithm starts with the node that has the most number of edges. That is node 2, and color A is assigned to it. This color assignment results in the color restriction that nodes 1, 3, 5 and 6 cannot be assigned with color A. We denote this restriction near those nodes by writing the color-in-bar $\overline{A}$'s. We remove all the corresponding edges, as shown in Figure 3.6b. Then, the node node is colored that has the most highest number of colors-in-bar's. If there is more than one node with the same number of color-in-bar's, the node is chosen that has the highest number of edges. If there is still more than one node, evaluate the influence of each color assignment, and assign the node with a color which results in a minimum influence. If a node can be assigned with more than one color, the evaluation of the influence of each color assignment is also required. According to the rules stated above, nodes 5 and 6 are selected because they have the same number of colors-in-bar's and the same number of edges. Assigning color B to node 6 will result in a restriction $\overline{A}\,\overline{B}$ on node 1 and a restriction $\overline{A}\,\overline{B}$ on node 5. While assigning color B to node 5 will result in a restriction $\overline{A}\,\overline{B}$ on node 6 and a restriction $\overline{B}$ on node 4. Because one $\overline{A}\,\overline{B}$ restriction and one $\overline{B}$ restriction result in less influence (a total number of 3) than two $\overline{A}\,\overline{B}$ restrictions (a total number of 4), assigning a color to node 5 produces less influence than assigning a color to node 6. Node 5 is selected, and color B is assigned to it, as shown in Figure 3.6c. The same way, color C is assigned to node 6, as shown in Figure 3.6d. Next, color B is assigned to node 1, as shown in Figure 3.6e. Nodes 3 and 4 are in the same condition, and have the same influence on the graph. If color B were assigned to node 3, color A or color C must be assigned to node 4. The final color assignment is shown in Figure 3.6f.

The above algorithm was incorporated into a program, named *COLOR*, and was run on a networked SUN 4/670MP Workstation. The program was tested on graphs with different number of nodes (N = 100 $\rightarrow$ 1000) and different edge percentages (P = 10% $\rightarrow$ 90%). The maximum number of edges in a graph is:

$\frac{N(N-1)}{2}$

where N is the number of nodes in the graph. The edge percentage (P) is simply the percentage of this maximum number of edges. Edges in the graph are randomly generated.

By statistic analysis, it is found that the time (T) is proportional to the number of nodes (N) in a polynomial form $T = O(N^{2.5})$.

The graph coloring problem is then solved (approximately) in a polynomial time, and not in an exponential time. For small graphs, we are able to verify that the algorithm gives the minimum solutions. It is then hoped that it gives reasonably good results for larger graphs as well, but this claim cannot be verified, since the exact minimal optimizer is not available. The pseudo-code for graph coloring is shown in Figure 3.7.

```
graph_coloring()
{
sort nodes in decreasing order by the number of edges;

color first node;
remove it;
mark restrictions;
remove edges;

while ((cib_set = largest_number_color_in_bar(node_queue)) != empty)
  {
  e_set = largest_number_edge(cib_set);

  if (|e_set| > 1)
    f_n = minimum_influence(e_set);
   else
    f_n = e_set;

    color f_n;
    remove f_n;
    mark restrictions;
    remove edges;
  }
}
```

Figure 3.7: Pseudo-code of graph coloring

Chapter 4

Representation of Functions for Decomposition

a \ bc	00	01	11	10
0	0	1	3	2
	4	5	7	6

Figure 4.1: Standard enumeration of cells of a 3-variable Boolean function

4.1 BDD Representation for Integer Sets Representing Blocks of Partitions

There exists three kinds of operations in partition-based decompositions,

- partition product,
- partition sum,
- relation not-greater-than on partitions.

These partition operations use set operations:

- set intersection,
- set union,
- set inclusion.

These operations can be easily realized in the existing BDD packages, thus making it possible to use BDDs to represent set operations.

Example 4.1.1

For instance, in Example ??(OLD Example13.2), we have:

$$P(X_0) = \{\{0, 1, 2, 4, 5\};\ \{2, 3, 4, 5, 6, 7\}\}$$

Using the following encoding method:

- 0 $---$ 10;
- 1 $---$ 01;
- DC $---$ 11;

and the encoding of numbers of rows of function from Table 3 with secondary variables as in Figure 4.1 we obtain the Karnaugh maps as in Figure 4.2, (OLD Figure 43):

The function from Fig. ?? (OLD FIG42) is: $P_1(X_0) = \bar{b} + \bar{a}\bar{c}$. The function from Fig. ?? (OLD Fig43) is: $P_2(X_0) = a + b$. Their respective BDDs are shown in Figure 4.4.

Using this method, we can use BDDs to represent sets. Therefore for a partition, there exists corresponding BDDs to represent the sets corresponding to blocks of this partition.

Set Operations.

a \ bc	00	01	11	10
0	1	1	0	1
1	1	1	0	0

Figure 4.2: Karnaugh map for $P_1(X_0)$ for Example 14.1

a \ bc	00	01	11	10
0	0	0	1	1
1	1	1	1	1

Figure 4.3: Karnaugh map for $P_2(X_0)$ for Example 14.1

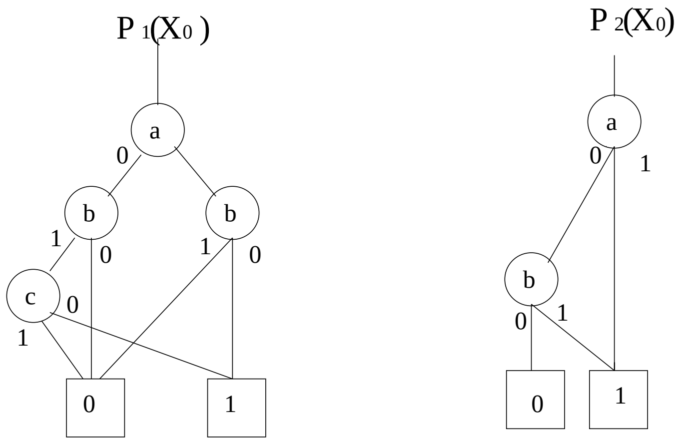


Figure 4.4: The BDDs for functions $P_1(X_0)$ and $P_2(X_0)$

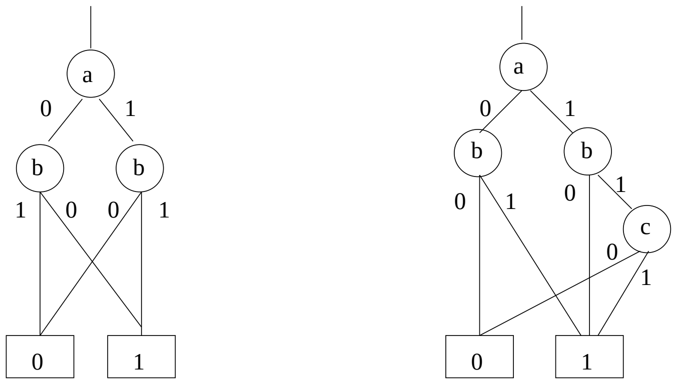


Figure 4.5: Partitions Π_1 and Pi_2 to Example ?? (OLD EXample 14.1): $\Pi_1 = \bar{a}\bar{b}+ab,\ \Pi_2 = a\bar{b}+bc+\bar{a}b$

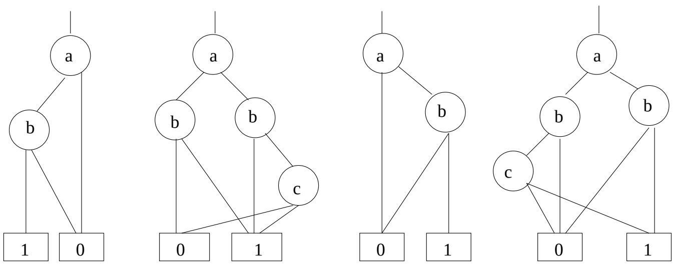


Figure 4.6: Partitions to Example 14.1: $P_1\Pi_1 = \bar{a}\bar{b}, \quad P_2\Pi_2 = a\bar{b} + bc + \bar{a}b, \quad P_2\Pi_1 = ab, \quad P_1\Pi_2 = a\bar{b} + \bar{a}b\bar{c}$

1) Set intersection:

Set intersection is equivalent to do AND operation on BDDs which is an operation in BDD package.

For example:

$$\Pi_G = \{\{0,1,6,7\};\ \{2,3,4,5,7\}\}$$

Let Π_1 and Π_2 represent BDDs, respectively (see Fig. 4.5).

Then with a BDD package, we can get $P_1\Pi_1, P_1\Pi_2, P_2\Pi_1, P_2\Pi_2$ which correspond to the four sets $\{\{0,1\};\ \{6,7\};\ \{2,4,5\};\ \{2,3,4,5,7\}\}$ after set intersection of $P(X_0)$ and Π_G. (see Fig. 4.6).

2) Set Union:

Set Union is equivalent to the OR operation on BDDs which could be realized with a BDD package.

For example: $P(B) = \{\{0,6,7\};\ \{1,6,7\};\ \{2,3,5,7\};\ \{2,3,4,7\}\}$, when merging B_1 and B_2 together, the BDD for the new set {0,1,6,7} is the same as the BDD obtained from OR operation on the two BDDs of {0,6,7} and {1,6,7}.

3) Relation less than or equal to.

If set $B_1 \subseteq B_2$, then $B_1 \# B_2 = \emptyset <=> B_1 \bigcap \bar{B_2} = \emptyset$. With this method, it is easy to check if one set is properly included in, or improperly included (equal to), another set, because we know BDD package can do AND operation, and we can realize negation in BDDs just by exchanging the position of leaf nodes 0 and 1.

4.2 Representation of Functions.

In this and following sections, we will present partial problems of decomposition. The subsequent four sections will discuss the four most important issues:

1. Representation of problems.
2. Variable partitioning.
3. Column minimization.
4. Sub-function encoding.

As we observed in previous sections, many decomposition programs were successful due mainly to some new approaches to problem representation: cube calculus, Reed-Muller forms, Binary Decision Diagrams, Walsh Transforms, or partition calculus based representations. We hope that these were not the last words in representation and that among many new and interesting new general representations of Boolean functions, there are some that may be used to create superior programs for Boolean Decomposition.

Because we believe that representation is THE most important aspect of successful decomposers, we will devote more attention in this section to the representation problem. Below, we will present the main research results in sections corresponding to various representations that have been already investigated for some other applications. Because Boolean methods are usually slower than algebraic methods, an effective representation and efficient manipulation of functions is the key to the success of these methods.

For representing Boolean functions, the following methods have been used in decomposition programs:

1. Karnaugh maps: Ashenhurst, Curtis, Ross et al, Torkelson, Luba.
2. Cubes: Hwang and Owen, (IEEE Transactions on CAD, May 1992), Karp [?], Perkowski/Brown [?], Wan/Perkowski [?], Bibilo [?], Steinbach [?] (a variant called TVL - ternary vector lists that has a different coding of symbols 0, 1, and X and uses disjoint cubes).
3. Canonical Positive Polarity Reed-Muller Forms: Shen and Kellar, Trachtenberg and Varma.
4. Walsh Transforms: Karpovsky, Trachtenberg and Varma, Stankovic.
5. OBDDs: Chang and Marek-Sadowska [?], Lai, Pedram, and Vrudhula [?], and Sasao [?].
6. Edge-Valued Binary Decision Diagrams (EVBDDs) by Lai, Pedram, and Vrudhula [?].
7. Partitions of minterms: Luba, Selvaraj.
8. Partitions of cubes: Luba [?], Selvaraj [?, ?].

The following representations of binary functions are close to the above and were shown to have some advantages in other problems, so they remain possible prospective candidates:

1. Canonical AND/EXOR and Orthogonal forms, other than the Canonical Positive Polarity Reed-Muller Forms.
2. Spectral Transforms: all AND/EXOR transforms, Arithmetical Transform, Adding Transform, Orthogonal Binary Transforms, Orthogonal Multiple-valued Transforms.
3. Decision Diagrams: Zero-Suppressed, Moment, Functional, Kronecker, Orthogonal, Multiple-Valued.

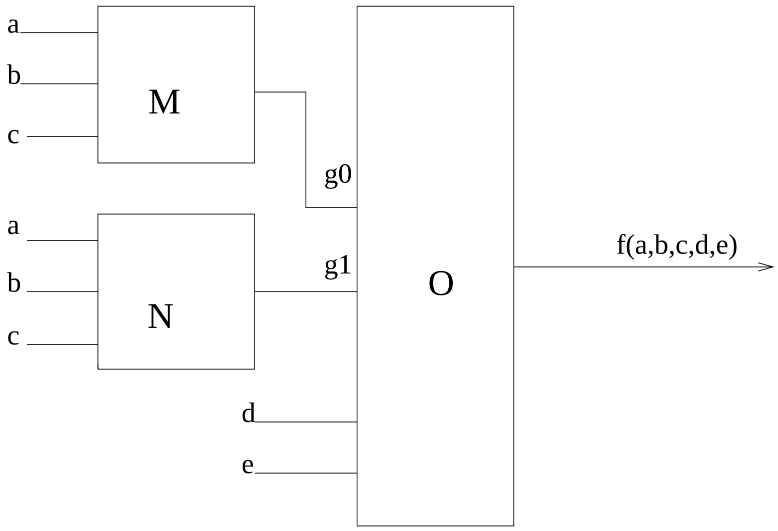


Figure 4.7: A schematic diagram of the decomposition

4. Array of Cubes representing an ESOP.

For representing multiple-valued functions the following methods have been used:

1. Maps, expressions Walliuzzaman and Vranesic, [?].
2. Expressions, Wojcik and Fang, [?].
3. Maps, Abugharbieh, [?].
4. partitioned minterms: Luba, [?].
5. multiple-valued cubes and minterms: Sasao 1989, [?].
6. Multiple-Valued diagrams [?].

Single output functions have been discussed in: Ashenhurst, Curtis, Ross, Steinbach. Multi-output functions have been presented in: Luba [?], Pedram [?], He and Torkelson [?], and Karp [?].

In the following subsections we will present the most prospective representations.

4.2.1 Various Types of Decision Diagrams

Reduced, ordered Binary Decision Diagrams (BDDs) provide a compact and canonical representation of Boolean functions. These data structures are used in the decomposition program by Lai, Pedram and Vrudhula [?]. When the function is incompletely specified, there are two ways to represent it using BDDs. The first is to generalize the BDD to 'Ternary DDs' that have three terminal nodes, standard nodes 0 and 1, and a '*' node corresponding to the don't care value of the function. This method was applied by Lai, Pedram and Vrudhula [?].

The other method, as far as we know not used in any program yet, would be to represent functions 'q' and 'r' of XBOOLE or functions ON and OFF of TRADE, each with one standard BDDs. In case of multi-output functions a shared OBDD can be used, or any other shared diagram (Functional, Zero-suppressed, with negated edges, Kronecker, etc). This would allow to use standard DD packages and 're-use' all (or most) ideas of XBOOLE and TRADE, which propose two different approaches, that can be now totally unified on an algebraic level.

Pedram et al use also their new data structure, called Edge-Valued Binary Decision Diagrams (EVBDDs) [?]. EVBDDs provide a representation of Boolean functions over the integer domain and have been shown to be useful for verification of arithmetic functions. Another similar and even more powerful decision diagrams have been recently created by Brayant and Chen, 1994, and called Binary Moment Diagrams. These topics are the areas of recent research.

4.2.2 Approach of Lai, Pedram and Vrudhula based on Binary Decision Diagrams.

Lai, Pedram and Vrudhula proposed in DAC '93 one of the most successful approaches to AC decomposition [?]. Their approach is applicable to both disjunctive and non disjunctive decompositions, both completely and incompletely specified Boolean functions, single-output and multi-output.

Their general theory uses a new algorithm, based on both BDD and EVBDD representations (two variants), for generating the set of all bound variables that make the function decomposable.

A function $f(x_0,\ldots,x_{n-1})$ is said to be decomposable under bound set $\{x_0,\ldots,x_{i-1}\}$ where $0 < i < n$ if f can be transformed to $f(g_0(x_0,\ldots,x_{i-1}),\ldots,g_{j-1}(x_0,\ldots,xi-1),x_{i-k},\ldots,x_{n-1})$. Then, if $k=0$, the function is said to be disjunctively decomposable, otherwise it is non disjunctively decomposable.

Consider a simple example of a function $f(a,b,c,d,e)$. Let $bound\_set = \{a,b,c\}$ and $free\_set = \{d,e\}$. Here $k=0$ hence a disjunctive decomposition is applied. Figure 4.7 shows a schematic diagram of this decomposition.

If two or less than two g functions exist for the bound set (a,b,c), the decomposition exists. Else, there is no decomposition. Ordered Binary Decision Diagrams (OBDDs) are used to represent functions.

Notations

- The left edge of the BDD represents the true edge or 1
- v represents function $f(x_0,...x_{n-1})$ and BDD rooted by node v

Disjunctive Decomposition on BDDs

In this section disjunctive decomposition with BDD's used as the method of representation will be explained and the case of completely specified functions is presented.

Example 4.2.1

Let $f = x_0x_1x_2x_3 + x_0x_1x_2\overline{x}_3\overline{x}_4 + x_0x_1\overline{x}_2\overline{x}_4 + x_0\overline{x}_1x_2\overline{x}_4 + x_0\overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_0x_1x_2\overline{x}_4 + \overline{x}_0x_1\overline{x}_2\overline{x}_3 + \overline{x}_0\overline{x}_1x_2x_3 + \overline{x}_0\overline{x}_1x_2\overline{x}_3\overline{x}_4 + \overline{x}_0\overline{x}_1\overline{x}_2\overline{x}_3$. The corresponding decomposition chart is as shown in Fig 4.8.

Looking at Figure 4.8 we can see that there are 3 distinct columns. Hence we know that there is a simple disjunctive decomposition with the chosen bound and free sets. But how can this be found by looking at the BDD?

The Pedram's method is this:
Draw the BDD with such a variable ordering that the variables from the Bound Set are on the top of the BDD, and then they are followed by the variables from the Free Set. Figure 4.9 shows the BDD of our example.

Figure 4.10 shows how the columns of the decomposition chart correspond to paths along the BDD. If one looks at path 101 along the BDD it results in $\overline{x}4$ which is the same as column 101. This illustrates the correspondence of the decomposition chart and the BDD. Figure 4.11 explains what a cut set is.

x3x4 \ x0x1x2	000	001	010	011	100	101	110	111
00	1	1	1	1	1	1	1	1
01	1	0	1	0	1	0	0	0
10	0	1	0	1	0	1	1	1
11	0	1	0	0	0	0	0	1

Figure 4.8: A Decomposition Chart to Example 21.2

One important assumption of Pedram's method is that the variable ordering of the BDD must correspond to the bound set. If it does not correspond, a rotate operation must be used to move the bound variables to the top.

Now a complete example will illustrate how to perform a disjunctive decomposition on BDDs.

Example 4.2.2 Let us assume a Boolean function of 5 variables f(a,b,c,d,e). Let {a,b} be the bound set and {c,d,e} be the free set. Assuming a simple disjunctive decomposition with two g functions, the decomposition would look like shown in Figure 4.12.

M and N are called the predecessor blocks and O is called the successor block. The entire process is described in Figure 4.13, 4.14 and 4.15.

Finding predecessor and successor blocks.

The steps involved in finding predecessor and successor blocks are the following:

STEP 1: Construct a BDD.

STEP 2: Find the best cut of the BDD.

STEP 3: Encode each node in the cut set, i.e *g,h,i*

Let g=10, h=01, i=00. Since we have used two bits for encoding we have two g functions $g0$ and $g1$ as shown in the Figures 4.14, 4.15 and 4.16.

At this point we know that the successor function has inputs of $g0, g1, x3, x4$. Hence the original cut set of the BDD will remain the same, and variables g0 and g1 will stand on the top of the BDD. This is shown in Figure 4.16.

Both Ashenhurst and Curtis decompositions are easily found from the value of the cut. This technique is next extended to a non disjunctive decomposition, and to multiple-output functions in a standard way.

Pedram's method was also applied to incompletely specified functions. It is interesting how well is this method suited for decomposition of multi-input, multi-output functions with a large number of don't cares? Figure 4.17 shows an example of an incompletely specified function, and Figure 4.18 shows the corresponding 'ternary' BDD with 3 terminal nodes - 1,0 and X.

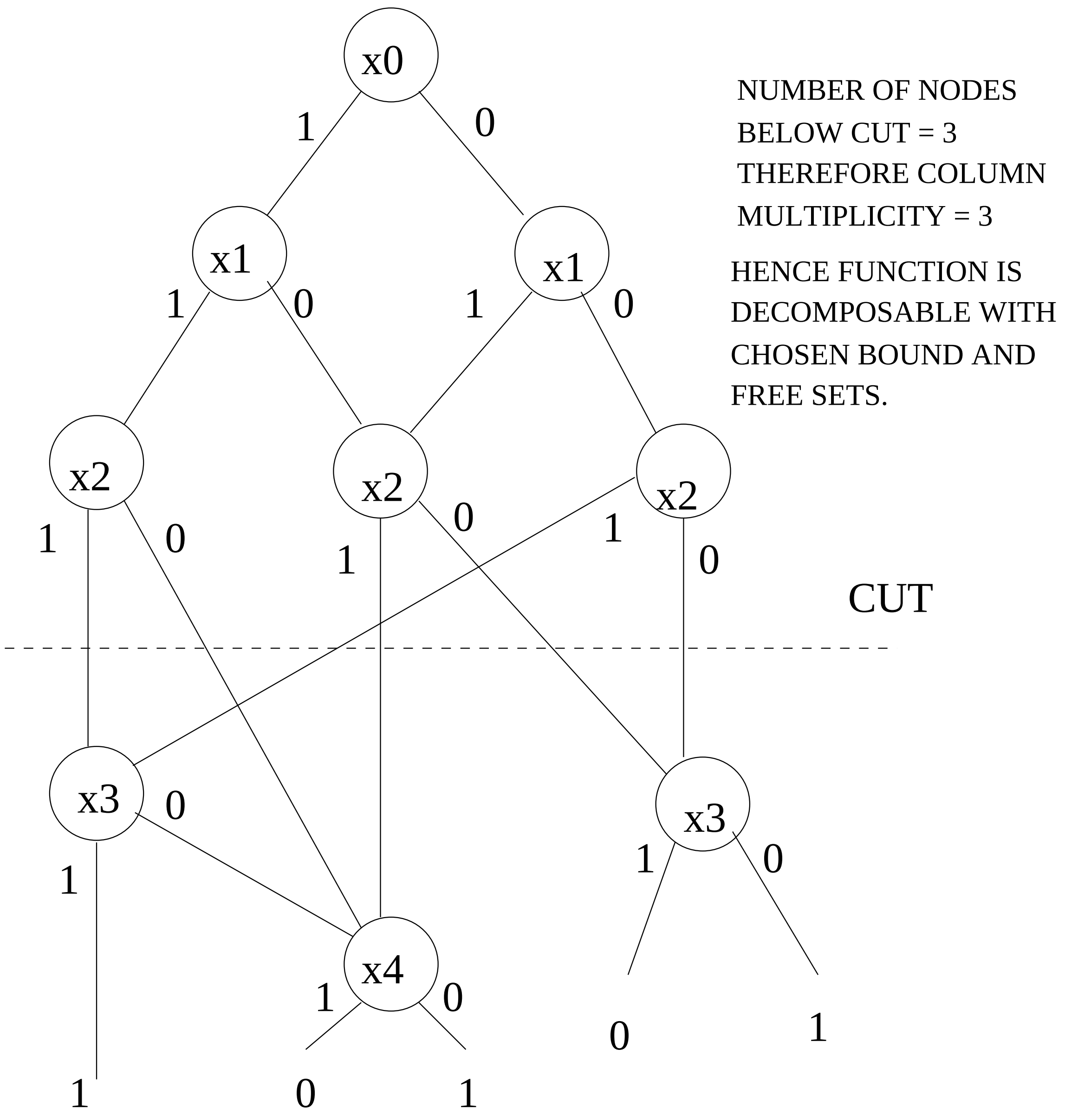


Figure 4.9: BDD of the decomposition chart

x3x4 \ x0x1x2	000	001	010	011	100	101	110	111
00	1	1	1	1	1	1	1	1
01	1	0	1	0	1	0	0	0
10	0	1	0	1	0	1	1	1
11	0	1	0	0	0	0	0	1

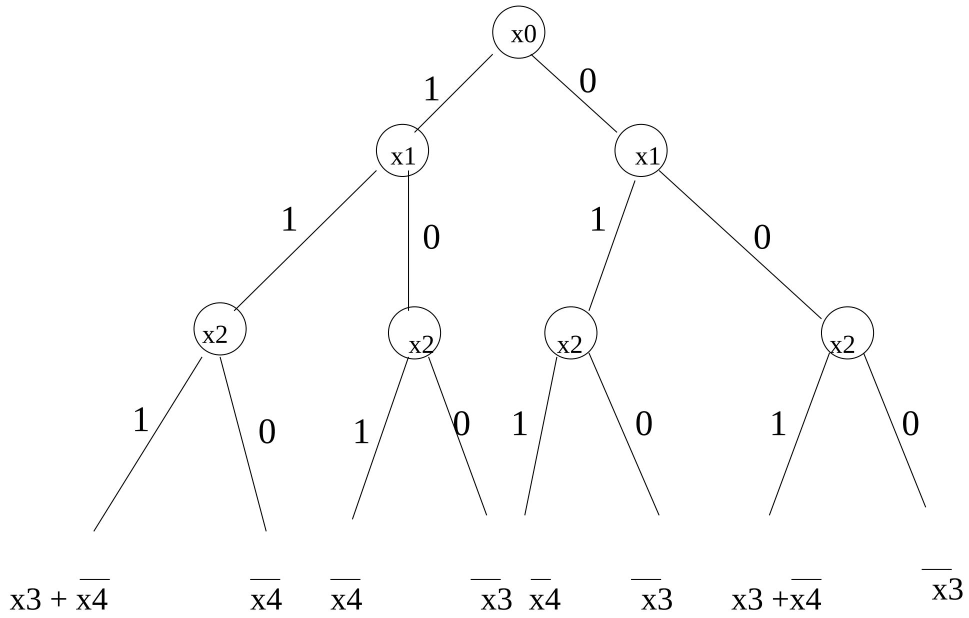


Figure 4.10: Correspondence between columns of the decomposition chart and BDD

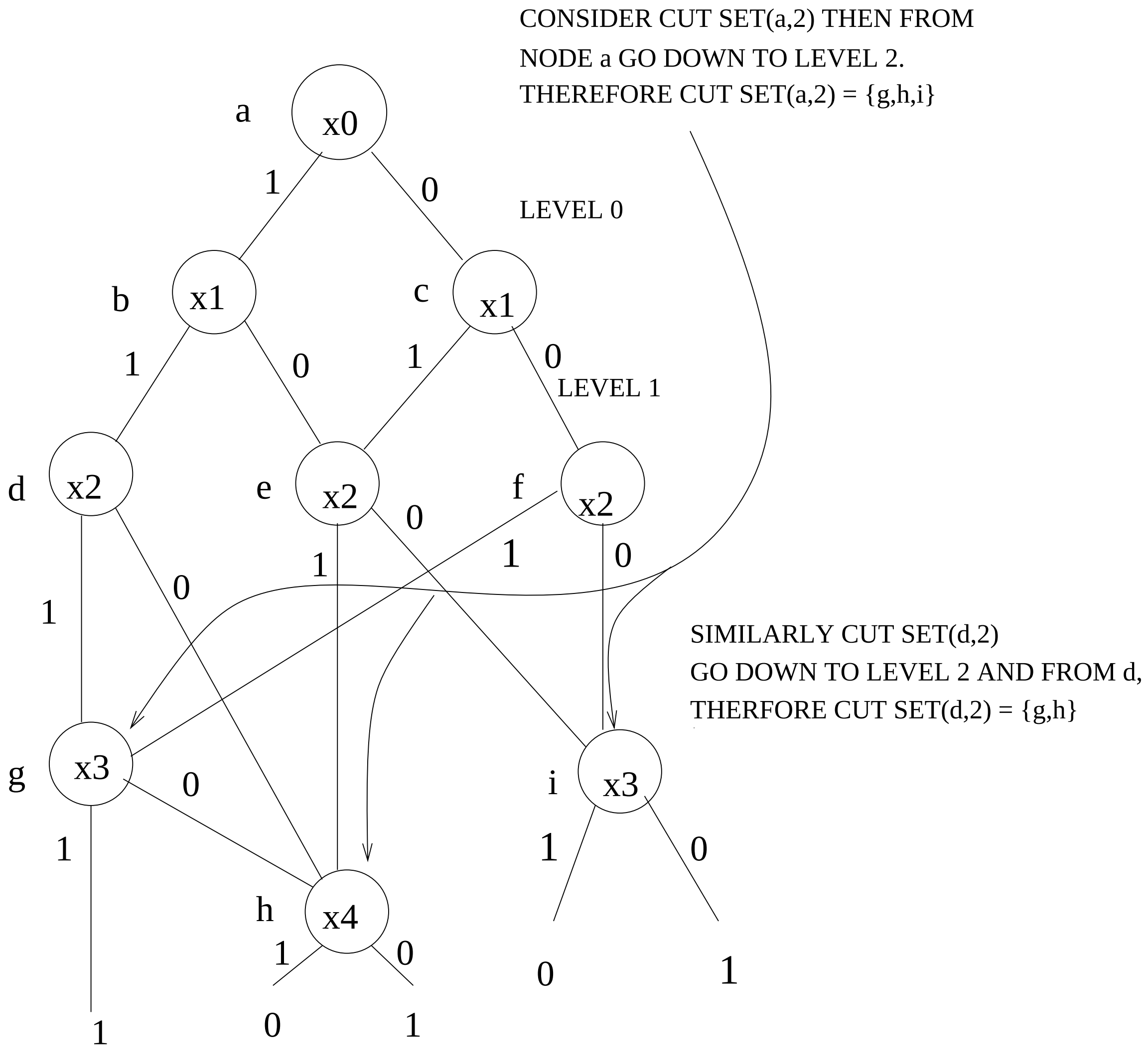


Figure 4.11: Explanation of a cut set

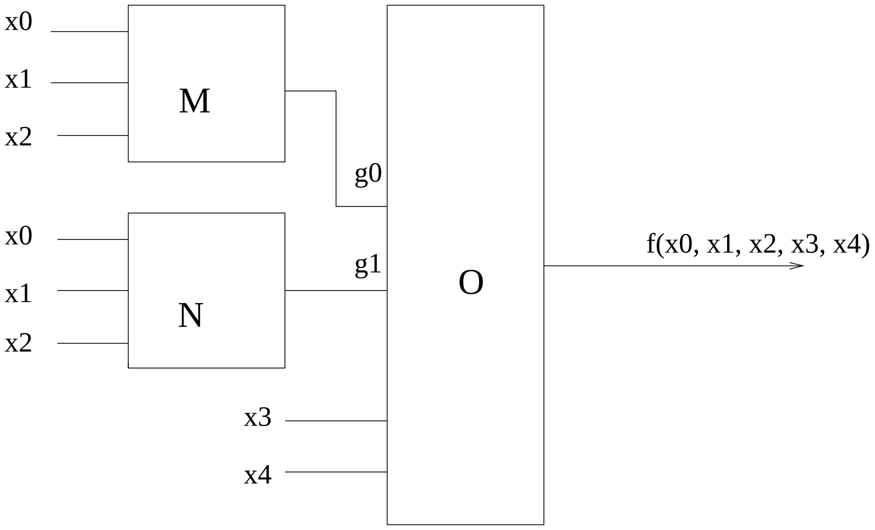


Figure 4.12: Structure of the decomposition with two g functions

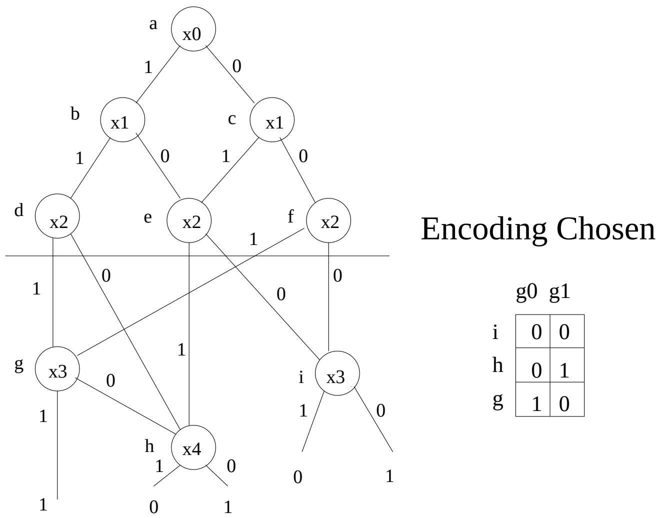


Figure 4.13: A BDD Example

so now we draw the BDD for g0 by making the nodes
g=1 h=0 i=0

a x0

1 0 combining nodes we get

b x1 c x1

1 0 1 0

d x2 e x2 f x2

1 0 1 0 1 0

x0

1 0 0 1 0

g h i

x1 x1

reduced g0 function 1 0 1 0

x2

1 0

1 0 0

Figure 4.14: Finding the predecessor block

Similarly we get the g1 function by encoding

i=0 h=1 g=0

a x0

1 0

b x1 c x1

1 0 1 0

d x2 e x2 f x2

1 0 1 0 1 0

0 1 0

g h i

combining nodes we get

x0

1 0

x1 x1

1 0 1 0

x2 x2

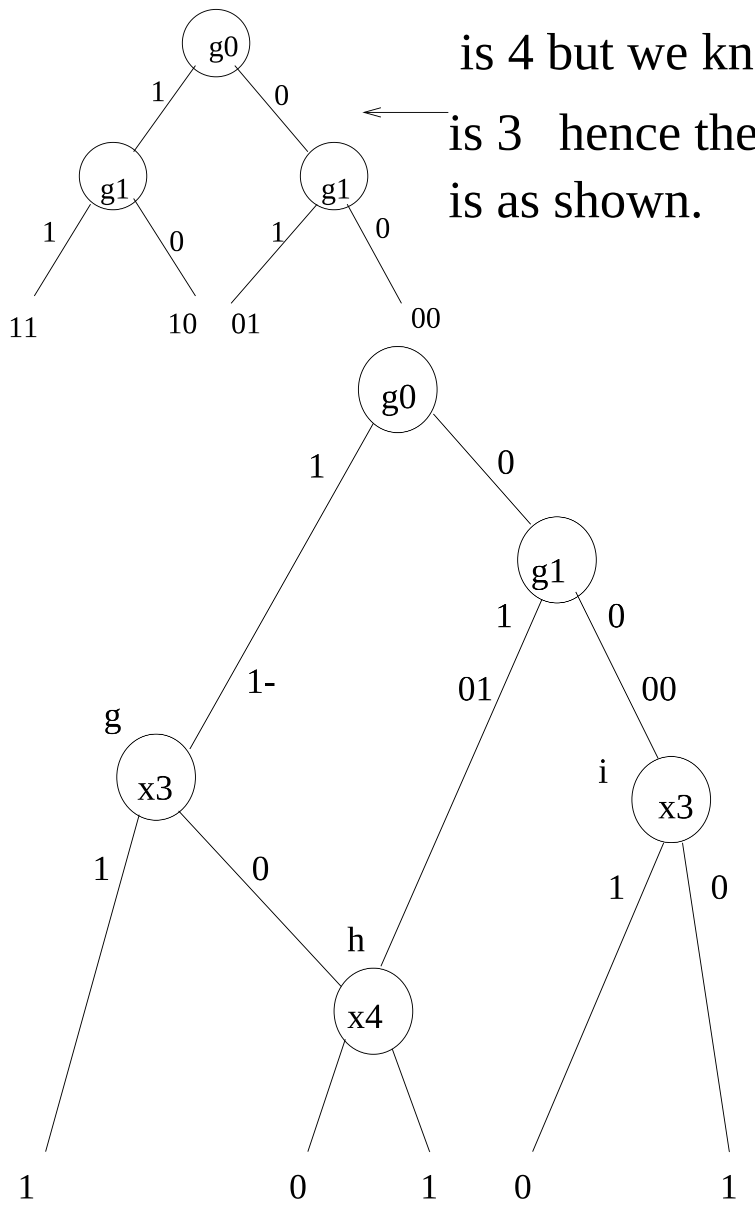
1 0 1 0

0 1 0

reduced g1 function

Figure 4.15: Finding the predecessor block

hence for this column multiplicity is 4 but we know that multiplicity is 3 hence the encoding chosen is as shown.



hence we get the BDD of our sucessor function

Figure 4.16: Finding the successor block

ab \ cde	000	001	010	011	100	101	110	111
00	x	1	0	1	1	0	1	0
01	x	1	1	x	1	1	x	x
10	x	0	1	0	0	1	0	1
11	x	1	x	x	0	x	x	x

Figure 4.17: Decomposition chart of an incompletely specified function

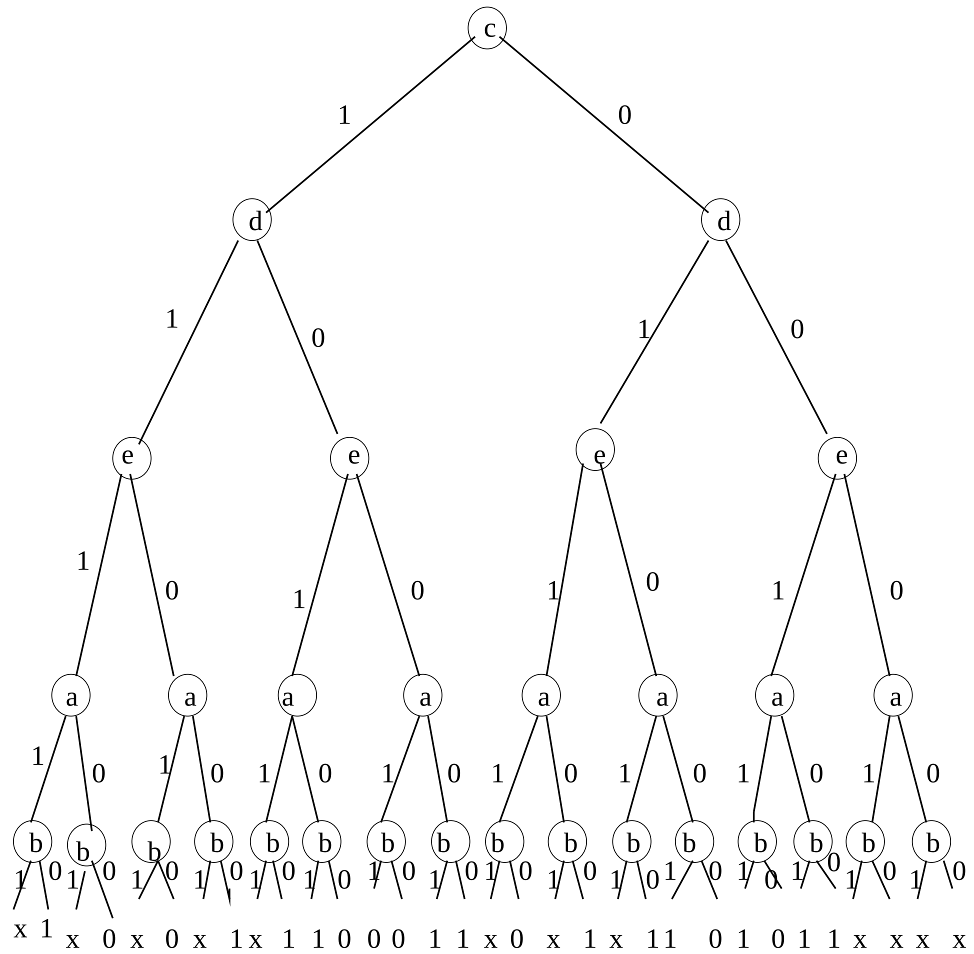


Figure 4.18: BDD with 3 nodes for an incompletely specified function

It can be seen from this BDD that what is needed is some kind of technique to combine compatible nodes to the smallest number of nodes. For instance, familiar graph coloring or set covering methods can be used, so that the nodes of the BDD will be combined, just like combining columns in the Karnaugh map so as to reduce the column multiplicity. One possibility would be to create an Incompatibility Graph and apply graph coloring, another possibility is to create Compatibility Graph, maximum cliques, and solve the coloring table. One more approach would be to create a new method performing 'column minimization' in a smart heuristic way directly on the BDD.

When functions are incompletely specified, an analogous difficulty occurs as in Roth-Karp and Perkowski-Brown-Wan algorithms. The authors propose a set covering algorithm, but do not describe it in detail, so it is perhaps a standard one from MIS tools. The encoding method is also not presented or even mentioned.

While Roth-Karp use NP-hard problem, and Perkowski use NP-hard graph-coloring problem, Lai et al are faced with performing analogous task on BDDs. Since BDD can represent only a completely specified function, they first generalize the BDD to a graph that has three types of terminal vertices: 0, 1, and DC, to represent the constant function 'don't care'.

Next, they compute the cut-set as before. Since each node in cut-set may represent an incompletely specified function, they need to compute the compatibility between any two nodes in the cut-set so that a minimal k can be found. The determination of compatibility between two BDD nodes, or the compatibility between their corresponding functions, is carried out by algorithm *is-compatible*. After this step, the construction of computability graph and the computation of minimum clique cover is the same as in the Roth-Karp algorithm.

From the point of view of the size of functions that can be handled, the Pedram's method is the top decomposition method in the world today. They are able to solve all decomposable forms for functions such as vg2 which has 25 inputs and 8 outputs, or even e64 that has 65 inputs and 65 outputs. These are clearly one of the largest examples of AC decomposition ever achieved. Perhaps, THE largest. However, for the predecessor and successor functions there is no encoding scheme used so the random encoding chosen is not guaranteed to be a good encoding.

Concluding, one can state that the only innovative idea of their paper is the BDD representation and the concept of cut-set in BDD, as well as using variable sifting techniques to find the good bound sets.

Hence Pedram's method can be improved by combining Luba's method, Steinbach's method, or any other method that proposes new decompositions, conditions, algorithms or heuristics. Also, if some encoding method is used it will greatly improve the quality of decomposed blocks.

4.2.3 New Classes of AND/EXOR Decision Diagrams.

The Direct Acyclic Graph (DAG) based representation and decision diagrams which are more general than BDDs have been recently created and applied. While Hurst in 1985 [?] only mentioned a similar approach, recently both a theory and several synthesis tools have been developed in our group, [?, ?, ?, ?], [?, ?, ?, ?, ?, ?, ?, ?], and also by Sasao [?] Steinbach [?], Rosenstiel [?, ?], and Marek-Sadowska [?, ?].

Perkowski et al [?, ?, ?] formulated the new class of Decision Diagrams, called *Kronecker Decision Diagrams (KDD)*. These Decision Diagrams are the generalization of the popular Binary Decision Diagrams (Bryant, 1986) and Functional Decision Diagrams, Kebschull, Schubert, and Rosenstiel, [?], and are more compact than both of the former Decision Diagrams. The method is much more efficient than FDDs, and can be applied to very large functions given in multi-level net-lists.

In addition, KDDs, similar to BDDs and FDDs, provide a canonical representation of functions and can be applied in many areas. Currently, BDDs have been used in many applications in logic synthesis, verification, testing, modeling and simulation. KDDs, while being more compact, can also be utilized in many of such applications and thus can provide a major improvement over the current techniques in these areas. They can also drastically cut on the number of nodes in the Decision Diagram for very large functions that up to now have not been able to be represented by BDDs.

For any Decision Diagram to prove to be useful, the compactness of the representation has to be compared with the ease of construction and manipulation. A package for representation and manipulation of functions has been developed [?, ?], which shows the compactness of the KDDs together with ease of manipulation and construction. It has been shown that for the standard, hard, benchmark examples, KDDs are on the average 35% more compact than Binary Decision Diagrams, with some reductions of up to 75% being observed. The minimization scheme is based on the state of the art minimization schemes for BDDs, namely, dynamic variable ordering with sifting algorithm (Rudell, IWLS 1993, [?, ?]). Here the sifting is performed for both the order of variables as well as the type of decompositions.

Furthermore, a class of functions was presented for which both BDD and FDD representations are exponential in size but KDD is of polynomial size [?]. This property together with the canonicity and the ease of construction and manipulation distinguishes the major significance of the KDDs. Although, thanks to their canonicity, the KDD seem to be the most prospective of the introduced by our group diagrams as a general purpose representation of Boolean Functions, we found that other diagrams are best for mapping to fine-grain FPGAs. Recently Ho and Perkowski developed a concept of Free Kronecker Decision Diagrams [?]. They showed experimentally that it gives very good results on mapping to Atmel 6000 series FPGAs. The difference between the Kronecker Functional Decision Diagrams and the Free Kronecker Decision Diagrams (FKDD) is that in the former diagrams all nodes at certain level have the same variable and the same (one of three) expansion type. We generalized this concept to *'Pseudo-Kronecker Decision Diagrams'* where in each level we have still one variable, but all three types of expansions are possible in nodes. This has been generalized even further, to FKDDs, where there can be many different variables in a level. The tree or a graph is thus no longer 'ordered'. Although it is more difficult to create a program for generating this type of diagrams, they reduce significantly the number of nodes, and result in better mappings to FPGAs. It is possible to make these free diagrams canonical, implement operations on them, and treat them as the general-purpose function representations.

In another development, Perkowski et al observed that the totally symmetric Boolean functions can be realized in totally symmetric and regular decision diagrams that have only local connections. They call such flat and regular diagrams, the *trellis diagrams*. A question can be now asked: 'can the functions that are not symmetric be realized in trellis diagrams?' Interestingly, the answer is 'yes', if the variables of the diagram are repeated. This leads them to the concept of 'diagrams with repeated variables'. The current work is on the fundamental problem to create such diagrams for a given Boolean function, namely, what should be the (possibly repeated) order of input variables, so that the total number of variables (and the lattice area) will be minimized.

Finally, new classes of diagrams, called canonical *'Boolean Ternary Kronecker Decision Diagrams (BKTDD)'* have been created, that can theoretically be better than all other diagrams [?]. For selecting the expansion type, one has 1 choice in BDDs and original FDDs, 2 choices in modified FDDs, 3 choices in KDDs, and 12 choices in BKTDDs. There is no danger of loosing good choices in BKTDDs, since the three standard expansions of the KDD are still available in them. Therefore, the exact BKTDD is always not worse than the exact KDD.

Some of the introduced families, such as the KDDs, include all types of nodes from the BDDs and FDDs. It is then obvious that KDD diagrams are always more compact. The important questions are only: 'how much percent decrease in the number of nodes can be obtained by constructing KDDs for industrial benchmark logic functions? Can one represent with the KDDs some large functions than are not able to be represented by BDDs?' The numerical results are very good for some benchmarks, especially for incompletely specified and arithmetical functions. For the functions with a large number of input variables our algorithm can still be significantly improved.

KDD diagrams allow also to represent some especially constructed large functions that can not be represented by BDDs and FDDs. An example of such a function is given in (Becker, 1993). This is one of the areas of the current research.

4.2.4 Canonical Fixed Polarity Reed-Muller Forms

As presented in one of earlier sections, a very successful approaches to decomposition by Shen and Kellar [?, ?, ?] and Trachtenberg and Varma, 1989 [?] were based on Positive Polarity (Canonical) Reed-Muller Forms. There exists a family of more general forms, called Canonical Fixed Polarity Reed-Muller Forms (FPRMs), that are, however, very close to Positive Polarity RM Forms. They find several applications in logic synthesis, and allow for more compact function representation.

They were used as internal representation of Boolean functions in design automation systems; they were used as a convenient model to discuss some important algorithms, they were used as an aid in studying function properties and function classification (Marek-Sadowska, 1994 [?]); and finally they were useful to generate efficiently other types of EXOR Circuits, such as ESOPs, or Generalized Reed-Muller Forms.

FPRM forms can be also relatively efficiently realized in cellular arrays. Their realizations are similar to PLAs and are based on the concept of regularity to combine the synthesis and physical placement and routing stages. In many applications the AND/EXOR realizations of the circuits require less layout area than their AND/OR counterparts [?, ?, ?, ?]. The AND/EXOR PLAs often require fewer products than AND/OR PLAs (Sasao and Besslich, 1990 [?]). The major advantages of this logic stem from the information processing capabilities of EXOR gate.

For the specific AND/EXOR realizations, efficient heuristic algorithms for Fixed Polarity Reed-Muller canonical forms have been created [?, ?]. Fast techniques for identification of a minimal realization of Boolean functions in fixed polarity AND/XOR canonical forms were introduced. This scheme is then used in a Generalized RM canonical form minimization technique. Identification of a minimal realization would reduce the number of cells needed in the regular array implementation. Fast algorithm allows to approximately minimize functions larger than in the methods from the literature.

Many of the theorems and properties found in [?, ?] for FPRM forms are of general nature and can find applications in EXOR Logic Synthesis outside the topic of FPRM minimization.

4.2.5 Generalized Reed-Muller Forms.

The Generalized Reed-Muller Forms (GRM) forms are another canonical AND/EXOR forms that are more powerful than FPRMEs. As shown experimentally in [?, ?] and recently by Tsutomu Sasao, the good quality solutions, or exact solutions, for GRM forms are closer to the minimal ESOPs, than any other canonical forms investigated in the literature.

In papers by Csanky, Perkowski et al [?, ?, ?] a theory of such forms has been developed, as well as an efficient heuristic algorithm based on it. These papers suggest also an exhaustive exact method for GRMs. In [?] several useful concepts of general nature have been introduced, that have been next used by Marek-Sadowska [?]. For instance, the prime cubes, that have some similarity to prime implicants in SOP minimization. Interestingly, the concepts introduced by us have been recently found useful by other researchers for other applications. For instance, T. Sasao uses them to create an exact algorithm for GRM minimization [?], and Malgorzata Marek-Sadowska uses them for classification of Boolean functions and matching [?].

Although Sasao's algorithm, being an exact one, creates not worse and often better solutions than the algorithm from [?], the older algorithm remains still useful, since it is faster and can solve larger functions in an approximate way.

These forms can be used both as a representation and also in some special types of decompositions and investigations of symmetry and other properties useful in decomposition.

4.2.6 Canonical AND/EXOR Multiple-Valued Input, Binary Output Forms

The ideas of minimal Fixed Polarity RM forms were expanded to Multi-Valued Input Kronecker Reed-Muller (MIKRM) forms. While the FPRM forms are based on two expansions: Positive Davio for variables of positive polarity, and Negative Davio for variables of negative polarity, the binary Kronecker Reed-Muller (KRM) forms use three expansions: Positive Davio, Negative Davio, and Shannon. Therefore, there is 3^n KRM forms for a function of n variables. Each binary variable in a KRM form can have one of three polarities; to each polarity there corresponds a non-singular (orthogonal) binary array. The concept of polarity can be further extended for multiple-valued logic. For a multiple-valued variable there are as many polarities as corresponding orthogonal matrices. Each polarity corresponds to an expansion and an universal cell. These concepts lead to the canonical Decision Diagram and the Flat Multi-valued KRM representations of multiple-valued input functions. These diagrams and representations can find similar applications to the previously mentioned AND/EXOR forms.

4.2.7 Approximate Minimization of ESOPs

Approximate Minimization of ESOPs is a very important problem to which much research has been devoted over the last 30 years. It has such a relation to the AND/EXOR PLAs as the classical two-level SOP minimization problem has to the AND/OR PLAs. Similarly to SOPs, creating a quasi-minimal ESOP can become the first stage of a multi-level minimization, and can be also used as a general-purpose representation of Boolean functions.

The first efficient program of this type was called Exorcism [?]. Exorcism was based on application of iterative-improvement of two, introduced by us, cube calculus operations on EXOR-Array cubes, called *primary crosslink* and *secondary crosslink.* This approach has been generalized to handle don't cares, and to multiple-valued input functions, making it thus the first multiple-valued ESOP minimizer [?]. In 1990 Tsutomu Sasao created a similar program, called EXMIN, which was at that time in all respects superior to Exorcism, except for the handling of don't cares [?]. In efforts to further improve Exorcism, more cube reshaping and cube combining operations have been added to it, and the distance of the cubes to which the operations can be applied has been limited. New measure on cubes, called cube difference, which was used to evaluate prospective application of cube operations, has been also introduced. Further experimentation with large examples led to the invention of a new cube calculus operation, called *exorlink.* This operation allowed for very efficient realization in software, and it also includes all the well-known operations of both Exorcism and EXMIN2 (in the meantime Tsutomu Sasao improved EXMIN and renamed it EXMIN2), plus many other new cube operations. Based on the notions of difference and distance of cubes, the authors were able to execute only some particular cases of exorlink operation to selected subsets of cubes, decreasing thus substantially the search space. Finally, since 1993 Exorcism is able to obtain better results than EXMIN2. Moreover, the new version, Exorcism-mv-2 can assume any number of values for each input variable, and these numbers can be different for any particular variable. The program has also no inherent constraints on the size of the function. In addition, it improves much the handling of the incompletely specified functions, which property is missing in EXMIN2. Exorcism-mv-2 was compared with the incompletely-specified function ESOP minimizer created by J. Saul, and very good results were obtained on benchmarks of incompletely specified functions taken from industrial circuits (MCNC, ISCAS). Those circuits had from about 10% to 80% of don't cares. However, working recently in Wright Labs of Wright-Patterson Air Force Base on applying Exorcism-mv-2 to Machine Learning applications, the author found that its performance on 'learning' benchmarks, although better than some of the popularly used 'learning' programs such as C4.5, is still unsatisfactory on larger examples. Analyzing these cases, the author came to the conclusion, that the inferior behavior of Exorcism-mv-2 is caused by the extremely high percent of don't cares that is typical for Machine Learning benchmarks, and can be as high as 99.99999%. We have done some small improvements to Exorcism-mv-2 which improved its operation on many examples, but comparing its operation to a Boolean Ashenhurst/Curtis Decomposer we feel, that much further improvements are possible. This opens up a new general research area, namely designing Boolean and multiple-valued minimization algorithms for *strongly unspecified functions* typical for Machine Learning, data-base, image recognition and AI applications.

In conclusion, Exorcism-mv-2 is currently the *best heuristic ESOP minimizer available*, especially for the case of multi-output strongly unspecified Boolean and multiple-valued input functions. In addition, the author sees clearly now, how both its speed and the quality of the results can be further improved. Exorcism can be used both to create ESOPs as a representation for decomposition, and to minimize the non-decomposable blocks in AC decomposition. This is one of the current research areas.

4.2.8 Orthogonal Transforms, Fundamental Theorem and Its Applications

Paper [?] presents a theorem that forms a foundation to all well-known and all possible new canonical circuits with EXOR output gates. Let M be a $2^n \times 2^n$ binary matrix with columns corresponding to minterms and rows corresponding to a family of Boolean functions of n variables. $M[i,j] = 1$ means that the Boolean function of row i includes the minterm corresponding to column j. If the rows are linearly independent with respect to a bit-by-bit EXOR operation, then the family is called *'orthogonal family of Boolean functions'*. The functions from the family are called *'orthogonal'* functions. The theorem states that for any orthogonal family of 2^n Boolean functions f_i of n variables represented as a $2^n \times 2^n$ matrix M, there exists a canonical three-level realization:

$$F = f_0 \;\cdot\; S_0 \oplus ... \oplus f_{2^n-1} \;\cdot\; S_{2^n-1}$$

where functions f_i are the given orthogonal functions, and coefficients S_i are determined by multiplying matrix M^{-1} by the vector of minterms FV of function F.

Each such canonical expansion creates also an universal cell that can be used in multi-level trees. Analogously as in case of BDDs and KDDs, such trees can be factorized to Directed Acyclic Graphs (DAGs), so that the concept of generalized functional decision diagrams, called Orthogonal Decision Diagrams can be created. Such diagrams include all the known canonical decision diagrams from the literature as their special cases [?].

The generalization of these concepts for multi-output incompletely specified functions F has been also created. These concepts can be used to create efficient decision diagrams and programs to minimize non-decomposable blocks in AC decompositions.

4.2.9 Logic with Multiple-Valued Inputs and Multiple-Valued Outputs

An efficient algorithm for ternary Reed-Muller expansions under fixed polarities is presented in [?]. It requires less computation than all the previous algorithms, including those by Chen and Wu, Yang, Green and Harking. The algorithm, which is based on a recursive matrix, can be easily extended to any higher radix, demonstrating that perhaps many other of the described here methods, created by the author, which were developed for binary case or multiple-valued input, binary output, can also be extended to multiple-valued-input, multiple-valued-output logic ('truly multiple-valued'). The algorithm can be applied to both truly multiple-valued circuit logic hardware implementation, or to multi-level realization of binary encoding of this logic, leading thus to circuits with relatively high EXOR component, plus AND and OR gates. All such decision diagrams and canonical forms may be used as representations in decomposition of mv functions.

f f

(a) (b)

f

(c)

Figure 4.19: Cube arrangement

4.3 The Variable Partitioning Problem.

In this section we will present the Variable Partitioning approach applied in TRADE. Good heuristics for variable selection (partitioning) is also in [?].

In this section a method operating on cubes will be presented. It is called the *Pair Weighting Method*. This heuristic method will produce no more than four 'good' partitions which are to be used for decomposition. The basic idea of this method is to arrange cubes (minterms) in the Karnaugh map in such a way that they become more concentrated in either certain columns or rows, like the arrangements in Figure 4.19a and (b), but not like that in Figure 4.19c. For a given function, the number of minterms is fixed, the number of cubes after minimization is fixed as well. There are six minterms in Figure 4.19a. Under that partition, the minterms are concentrated in two columns. In Figure 4.19b it is the same function. Under that partition, the minterms are concentrated in two rows. In Figure 4.19c it is the same function again, but the minterms are diverged across the Karnaugh map. Clearly, the column multiplicities under the partitions shown in Figure 4.19a and (b) are less than that in Figure 4.19c.

How can we put more cubes in certain columns or rows? We first analyze the relative positions between two cubes in the Karnaugh map. There are four possible relative positions between two cubes as shown in Figure 4.20. The rectangles stand for cubes. The shaded areas from both cubes are the possible parts that can be put into the same columns or rows in the Karnaugh map. The question then arises, under what partition do the shaded areas from both cubes reside in the same columns or rows? The next example is used to show how to find that partition.

The ON set of function f in Figure 4.21 is:

```
        abcd
cube1   0-0-
cube2   1110
```

The remaining cubes belong to the OFF set.

The column multiplicity is three under this partition ($ab \mid cd$, bound set cd, free set ab). The Intersection of the two cubes results in the '*Intersection Cube*' cube0.

no e (a) one e (b) more than one e (c) all e (d)

Figure 4.20: Relative positions between two cubes

ab \ cd	00	01	11	10	
00	1	1			cube1
01	1	1			
11				1	cube2
10					

f

Figure 4.21: Karnaugh map of function f

```
          abcd
cube1     0-0-
cube2     1110
---------------
cube0     e1e0
```

Because there are two ϵ's, these two cubes have the relative position as shown in Figure 4.20c. The comparisons of cube0 with cube1 and the cube0 with cube2 result in the Partition Cubes C1 and C2, respectively:

```
          abcd                      abcd
cube0     e1e0            cube0     e1e0
cube1     0-0-            cube2     1110
--------------            --------------
C1        NNNN            C2        NYNY
```

The *Partition Cube* is formed as follows: If the corresponding bits of the two cubes are the same, there is a Y in the Partition Cube. Otherwise N. Next, according to the Partition Cubes one attempts to form partitions that would put as many shaded areas from both cubes into the same columns or rows as possible. The Partition Cubes are ignored if they have the value of all Y's or all N's. Therefore, C1 is ignored. Group variables corresponding to Y in C2 into a group, group b, d, forming a partition $ac \mid bd$. Under this partition, part of cube1 and part of cube2 reside in column 10 ($b = 1$, $d = 0$) of the Karnaugh map.

ac \ bd	00	01	11	10
00	1	1	1	1
01				
11				1
10				

f

(a)

bd \ ac	00	01	11	10
00	1			
01	1			
11	1			
10	1		1	

f

(b)

Figure 4.22: Karnaugh maps for the 'best' partition

ab \ cde	000	001	011	010	110	111	101	100
00	0	1	0	0	0	0	1	0
01	0	0	1	0	0	1	0	0
11	0	0	0	0	0	1	0	0
10	0	0	0	0	0	0	1	0

f

Figure 4.23: Variable partitioning example

These are minterms 0010 and 1110 as shaded in Figure 4.22a. This is just what we wanted: both cubes have a part in one column. We form another partition $bd \mid ac$ by grouping variables corresponding to N in C2 (a, c) in one group. Under this partition, part of cube1 and part of cube2 reside in row 10 $(b = 1, d = 0)$ of the Karnaugh map. They are minterms 1000 and 1011 (shaded as well) in Figure 4.22b. Again, this is what we aimed at: both cubes have a part in one row.

Clearly, the bound set $\{b, d\}$ is a better partition which produces a column multiplicity of two, while bound set *a, c* results in a column multiplicity of three. In summary, first all Partition Cubes are calculated for each pair of variables. Each Partition Cube forms a partition. Number of appearances of each pair is counted, and according to this number the final partitions are formed.

Now a more complex example will be used to explain the detailed procedure of variable partitioning.

Figure 4.23 shows the Karnaugh map of function f with the column multiplicity $\mu(ab|cde)$ of five. Its Espresso format input file is as follows:

```
.i 5
.o 1
.ilb a b c d e
.ob f
.type fr
00-01 1
-0101 1
01-11 1
-1111 1
----0 0
1-0-- 0
-0-1- 0
-1-0- 0
```

```
.end
```

'Variable partitioning example'

(1) Calculate Intersection Cubes. For ON set, they are:

```
00-01     00-01     00-01     -0101     -0101     01-11
-0101     01-11     -1111     01-11     -1111     -1111
-----     -----     -----     -----     -----     -----
00101     0e-e1     0e1e1     0e1e1     -e1e1     01111
```

For OFF set, they are:

```
----0     ----0     ----0     1-0--     1-0--     -0-1-
1-0--     -0-1-     -1-0-     -0-1-     -1-0-     -1-0-
-----     -----     -----     -----     -----     -----
1-0-0     -0-10     -1-00     1001-     1100-     -e-e-
```

(2) Calculate Partition Cubes. For ON set, they are:

```
00101     00101     0e-e1     0e-e1     0e1e1     0e1e1
00-01     -0101     00-01     01-11     00-01     -1111
-----     -----     -----     -----     -----     -----
YYNYY     NYYYY     YNYNY     YNYNY     YNNNY     NNYNY

0e1e1     0e1e1     -e1e1     -e1e1     01111     01111
-0101     01-11     -0101     -1111     01-11     -1111
-----     -----     -----     -----     -----     -----
NNYNY     YNNNY     YNYNY     YNYNY     YYNYY     NYYYY
```

For OFF set, they are:

```
1-0-0     1-0-0     -0-10     -0-10     -1-00     -1-00
----0     1-0--     ----0     -0-1-     ----0     -1-0-
-----     -----     -----     -----     -----     -----
NYNYY     YYYYN     YNYNY     YYYYN     YNYNY     YYYYN

1001-     1001-     1100-     1100-     -e-e-     -e-e-
1-0--     -0-1-     1-0--     -1-0-     -0-1-     -1-0-
-----     -----     -----     -----     -----     -----
YNYNY     NYNYY     YNYNY     NYNYY     YNYNY     YNYNY
```

(3) Calculate Relationship Factors.

1. Create four triangle tables as shown in Figure 4.24, call them ON-Y Table, ON-N Table, OFF-Y Table and OFF-N Table.

2. Initiate all their cells to zero. The value in each cell represents a weighted *Relationship Factor*. between the variables corresponding to the row and column labels of the table. Later on, we will sort the Relationship Factor Table. In order to keep the correct correspondence between the value and the variable pair it represents, we attach two more storage units to each cell to store the two variables that the Relationship Factor corresponds to. So the Relationship Factor Table is, in fact, a 3-tuple list. For example, the ON-Y Table in Figure 4.24 is a list: $\{(a,b,2),(a,c,4),(a,d,2),(a,e,8),(b,c,2),(b,d,4),(b,e,4),(c,d,2),(c,e,8),(d,e,4)\}$.

	b	c	d	e
a	a-b 2	a-c 4	a-d 2	a-e 8
	b	b-c 2	b-d 4	b-e 4
		c	c-d 2	c-e 8
			d	d-e 4

ON-Y Table

	b	c	d	e
a	a-b 2	a-c 0	a-d 2	a-e 0
	b	b-c 2	b-d 8	b-e 0
		c	c-d 2	c-e 0
			d	d-e 0

ON-N Table

	b	c	d	e
a	a-b 3	a-c 9	a-d 3	a-e 6
	b	b-c 3	b-d 6	b-e 3
		c	c-d 3	c-e 6
			d	d-e 3

OFF-Y Table

	b	c	d	e
a	a-b 0	a-c 4	a-d 0	a-e 0
	b	b-c 0	b-d 5	b-e 0
		c	c-d 0	c-e 0
			d	d-e 0

OFF-N Table

Figure 4.24: ON-Y, ON-N, OFF-Y and OFF-N tables

3. The Partition Cubes obtained in the second step are weighted in the following way:

- Group variables corresponding to Y's in the Partition Cube.
- Select a pair of variables in the group and increase the value of the corresponding cell of the ON-Y Tables by 1.
- Group variables corresponding to N's in the Partition Cube,
- Select a pair of variables in the group and increase the value of the corresponding cell of the ON-N Tables by 1.
- Execute the calculations for all pairs in the group.
- Perform the same operations for the OFF-Y and OFF-N Tables.

The formula of Relationship Factor is:

$$RelationshipFactor \ = \ sum \ ((pair\ between\ variables\ i\ and\ j)\ ?\ 1 \ : \ 0)$$

The summation is over all Partition Cubes and all pairs of variables in the Partition Cubes. '(pair between variables i and j)? 1 : 0' in the equation means that if variables i and j are a pair, take the value 1. Otherwise 0.

For example, if a Partition Cube C1 from the ON set is

```
     abcdefg

C1 = YNYNNYY
```

the cells *a-c, a-f, a-g, c-f, c-g* and *f-g* of the ON-Y Table and the cells *b-d, b-e* and *d-e* of the ON-N Table will be increased by 1.

According to above rules, we fill the ON-Y, ON-N, OFF-Y and OFF-N Tables as shown in Figure 4.24.

'ON-Y, ON-N, OFF-Y and OFF-N Tables'

(4) Form the 'best' partitions.

The Relationship Factor Table is sorted in decreasing order by the value of the Relationship Factors. The variable pairs with largest values in the ON-Y Table are collected until the required number of variables for the bound set is reached. Both cells *a-e* and *c-e* have the values of 8, so these two pairs are selected and the bound set $\{a, c, e\}$ is formed. For the ON-N Table, cell *b-d* has the largest value of 8. All the remaining are with the same value of 2. Cell *a-b* has the value of 2 and it shares the common variable b with set $\{b, d\}$. Set $\{a, b\}$ is then added to set $\{d\}$ creating a bound set $\{a, b, d\}$. The same operations are performed for the OFF-Y and OFF-N Tables to obtain another two bound sets $\{a, c, e\}$ and $\{b, d, e\}$.

The Karnaugh maps under these four partitions are shown in Figure 4.25. In fact, we have only three Karnaugh maps because there are only three different partitions out of these four partitions. Partition $\{bd \mid ace\}$ and $\{ac \mid bde\}$ result in a column multiplicity of two. While partition $\{ce \mid abd\}$ results in a column multiplicity of three.

We have checked that only partitions $\{bd \mid ace\}$ and $\{ac \mid bde\}$ can produce the minimum column multiplicity of two. There are ten $(\binom{5}{3} = 10)$ possible partitions out of these five-input function.

The pseudo-code for variable partitioning is shown in Figure 4.26.

bd \ ace	000	001	011	010	110	111	101	100
00		1	1			1		
01								
11		1	1			1		
10								

f

ac \ bde	000	001	011	010	110	111	101	100
00		1				1		
01		1				1		
11		1				1		
10								

f

ce \ abd	000	001	011	010	110	111	101	100
00								
01	1		1					
11	1		1			1		1
10								

f

Figure 4.25: Partitions after variable partitioning

```
variable_partitioning()
{
repeat for ON-Y, ON-N, OFF-Y and OFF-N tables
  {
  create Relationship Factor Table;
  sort Relationship Factor Table in decreasing order;
  bound_set = first pair in the queue;

  while (|bound_set| < maximum_bound_set_number)
    bound_set = bound_set U pair in the next position;
  }
}
```

Figure 4.26: Pseudo-code of variable partitioning

4.4 The Column Minimization Problem.

The problem to minimize the number of columns is discussed by all authors. Some of them discuss it as the set covering of a compatibility graph with nodes corresponding to sub functions of cofactors, some other authors represent it as graph coloring of an incompatibility graph that is a complement of the compatibility graph. These formulations are mathematically equivalent.

It seems, however, that there is something fundamentally wrong in both above formulations from the point of view of decomposing strongly unspecified functions for ML applications. We will discuss it in the next subsection.

4.4.1 Troubles with the Column Compatibility Problem.

This problem is not known from the literature and was observed in Wright Labs. Let us assume that we have 100 input variables and 99.99999999% don't cares. This means, most columns are columns of don't cares.

Such problem was observed for the parity function with 96 variables. There is no partitioning problem here and all the troubles to find a decomposable minimum solution of negated EXORs are due to bad column grouping. The question is: 'how to group columns to avoid these troubles?' The following ideas should be considered:

1. Do not combine columns of only don't cares with other columns.
2. Solve the column minimization problem together with the column encoding problem. For instance, in essence, we do not need to absolutely minimize the column multiplicity, only to get it below $k = 2^r$ (r = bound set cardinality - 1), so if maximum number of $r = 3$, 8 is as good as 5.
3. another approach - weighted covering, combine only columns that have 0's corresponding to 0's and 1's corresponding to 1's, and not 0's corresponding to -'s and 1's corresponding to -'s.
4. Use greedy algorithm that selects the largest clique, remove it, and iterate. Similarly to the 'disjoint star' covering algorithm of Michalski.
5. Combine step-by-step the most similar columns. Do not treat '-' as similar. Use the weights for:

   1) number of agreements.

   2) probability.

   3) probability that identical values are matched.

This is an area of our active current research. The goal is a reformulation of the fundamental Column Minimization Problem.

4.4.2 Graph Coloring

The graph coloring approach was implemented in TRADE program. The problem of finding the smallest column multiplicity was reduced to the one of performing proper graph coloring with the minimum number of colors. *Graph coloring*, [?, ?] is one in which every two nodes linked by an edge are assigned different colors. *Minimum graph coloring* is one with the minimum number of colors.

Graph coloring is an NP-hard problem. There has been a substantial research in graph coloring in order to find the algorithms for a quasi-optimum solution with the fastest possible speed. The method presented here is a fast graph coloring method. This method has been programmed and tested on many examples, it resulted in excellent colorings. The method found exact colorings for graphs in [?] and even found better coloring than that that was claimed to be minimal in the book. The method is called the *Color Influence Method.*

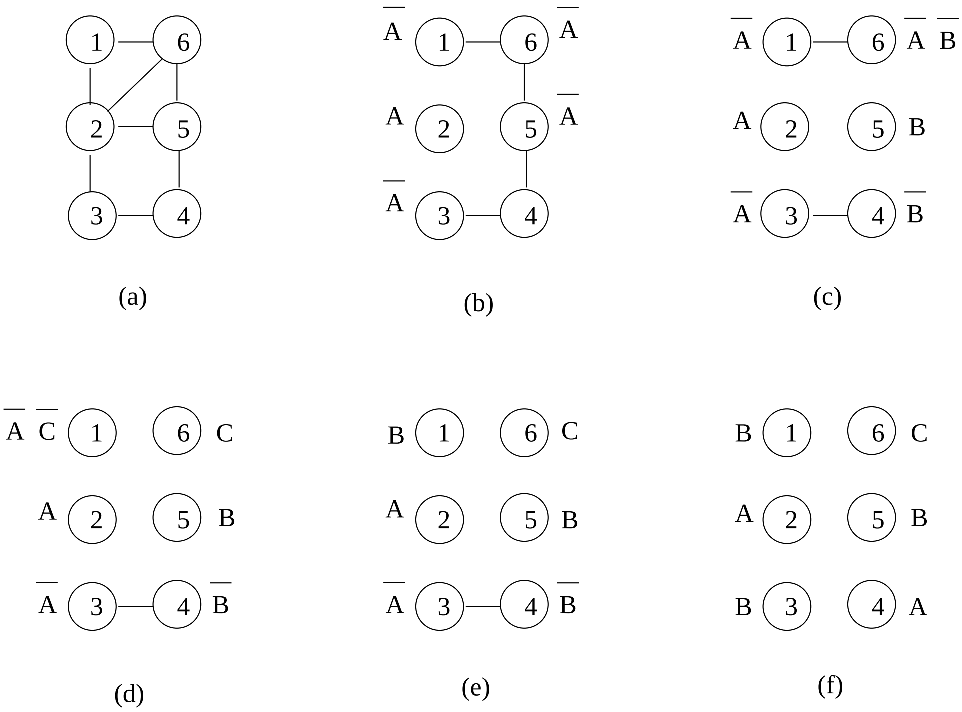


Figure 4.27: Graph coloring using Color Influence Method

The main idea of this method is to evaluate the influence of the color assignment to a node over the entire graph, and chose the color which results in a minimum influence. The *minimum influence* means that the color assignment to a node will produce a minimum increase of color-in-bar's.

The *color-in-bar's* (restrictions) are the colors that the node cannot be assigned with, which are denoted by $\overline{A}, \overline{B}, \ldots, \overline{A}\ \overline{B}, \overline{A}\ \overline{C}, \ldots$ as in Figure 4.27.

After each color assignment to a node, the complexity of the graph is decreased. This is a greedy method with global evaluation. The next example is used to illustrate this method.

Figure 4.27a shows a graph that need to be colored. Start from the node with the greater number of edges, that is node 2, assign color A to it. This color assignment results in that nodes 1, 3, 5 and 6 cannot be assigned with color A, denote this restriction on those nodes by color-in-bar $\overline{A}$'s, and remove all corresponding edges as shown in Figure 4.27b. Then, color the node with the most number of color-in-bar's. If there are more than one node with the same number of color-in-bar's, chose the node with the most number of edges. If there are still more than one node, evaluate the influence of each color assignment, and assign the node with a color which results in a minimum influence. If a node can be assigned with more than one color, the evaluation of the influence of each color assignment is also required. According to the rules stated above, nodes 5 and 6 are selected because they have the same number of color-in-bar's and the same number of edges. Assigning color B to node 6 will result in a restriction $\overline{A}\ \overline{B}$ on node 1 and a restriction $\overline{A}\ \overline{B}$ on node 5. While assigning color B to node 5 will result in a restriction $\overline{A}\ \overline{B}$ on node 6 and a restriction $\overline{B}$ on node 4. Because one $\overline{A}\ \overline{B}$ restriction and one $\overline{B}$ restriction result in less influence than two $\overline{A}\ \overline{B}$ restrictions, assigning a color to node 5 produces less influence than assigning a color to node 6. Node 5 is selected, color B is assigned to it, as shown in Figure 4.27c. The same way, color C is assigned to node 6, as shown in Figure 4.27d, and color B to node 1, as shown in Figure 4.27e. Nodes 3 and 4 are in the same condition, and have the same influence on the graph. If color B were assigned to node 3, color A or color C must be assigned to node 4. The final color assignment is shown in Figure 4.27f.

```
graph_coloring()
{
sort nodes in decreasing order by the number of edges;

color first node;
remove it;
mark restrictions;
remove edges;

while ((cib_set = largest_number_color_in_bar(node_queue)) != empty)
  {
  e_set = largest_number_edge(cib_set);

  if (|e_set| > 1)
    f_n = minimum_influence(e_set);
   else
    f_n = e_set;

    color f_n;
    remove f_n;
    mark restrictions;
    remove edges;
  }
}
```

Figure 4.28: Pseudo-code of graph coloring

The above algorithm was incorporated into a program, named *COLOR*, and was run on a networked SUN 4/670MP workstation. The program was tested on graphs with different number of nodes (N = 100 $\rightarrow$ 1000) and different edge percentages (P = 10% $\rightarrow$ 90%). The maximum number of edges in a graph is $\frac{N(N-1)}{2}$, N is the number of nodes in the graph. The edge percentage (P) is simply the percentage of the maximum number of edges. Edges in the graph are randomly generated.

By statistic analysis, it is found that the time (T) is proportional to the number of nodes (N) in a polynomial form $T \sim N^{2.5}$.

The graph coloring problem is then solved (approximately) in a polynomial time, and not in an exponential time. For small graphs, we are able to verify that the algorithm gives the minimum solutions. It is then hoped that it gives reasonably good results for larger graphs as well, but this claim cannot be verified, since the exact minimal optimizer is not yet available (we are working on it). The pseudo-code for graph coloring is shown in Figure 4.28.

4.5 Detection of Symmetry.

Detection of symmetry helps to simplify the variable partitioning problem. This was used in binary decomposition programs, but not in multiple-valued decomposers.

4.5.1 Approach of Stankovic and Moraga to Detection of Symmetry.

Stankovic and Moraga in [?] presented an approach to symmetry detection of mv functions.

In general, the problem of realization of a multiple-valued function is a relatively difficult task. Let us however first observe, that the examples of mv functions are naturally simpler than binary functions for the same tasks. For instance, assume that there is 5 different values of variable and we have 6 variables. In binary case we need 3 bits for a variable, times 6 = 18 binary variables. In the mv case we need just six 5-valued variables.

In some cases, when the given mv function has some particular properties, for example, symmetry or partial symmetry with respect to a given pair of variables, this problem is greatly simplified.

Further, let us observe that if we have symmetric variables and we use them to create mv variables, the symmetry is preserved. If we have symmetric mv variables, it is not necessarily reflected in respective binary variables that encode them, so symmetry of mv variables is a more powerful property, and will perhaps contribute more to the synthesis of strongly unspecified functions.

Decomposition of an mv function in recognition of its symmetries will lead to significant simplifications of the synthesis procedure.

In practice, one of the advantages of knowing that a given function is symmetric or partially symmetric is the potential economy offered by the realization of such a function using a smaller number of components. A similar statement applies to the functional decomposition, in which case the synthesis of an n-variable function reduces to the synthesis of two or more auxiliary functions with fewer variables. In both cases, the time for performing the synthesis procedure is reduced as well.

The problem of symmetry detection and functional decomposition on a given pair of variables may be studied both in the original domain and in the spectral domain. The existing results derived in the original domain are primarily based on some algebraic transformations using tabular representation of mv functions representing the extensions of the corresponding techniques by Ashenhurst/Curtis or by the graph coloring techniques of Muzio [?].

In the spectral domain, methods for symmetry detection and functional decomposition are based on relationships between spectral coefficients [?], [?], [?], [?], [?], [?], [?, ?], [?], [?], [?], [?], and [?],

In functional domain, the problem of using symmetry in decomposition was discussed in [?, ?, ?, ?, ?, ?].

The efficiency of spectral methods is mainly supported by Fast Fourier transforms on groups (the Walsh transform for Boolean and the Chrestenson transform for mv functions). For an n-variable mv function there are $(n/2)$ different pairs of variables. Therefore, to detect all possible symmetries of this function (or to examine the possibility of the decomposition on all pairs of variables), we need to perform a great number of checks. In the practical applications this represents a tedious task related to high requirements on processor time and memory. This fact considerably reduces advantages of this method.

4.5.2 Use of Symmetry in Decomposition.

It is well known that the number of possible partitions for a function with n variables is 2^n, which makes it impractical to search all partitions for larger functions. However, if variables x_1 and x_2 are partially symmetric, then if a bound set $D \cup \{x_1\}$ has been considered, the bound set $D \cup \{x_2\}$ does not have to be considered. The more partial symmetries exist, the less bound sets need to be considered. These ideas lead to the improved decomposition algorithms for functions with partial and total symmetries.

Recently, Malgorzata Marek-Sadowska created a new efficient method of classifying Boolean functions [?]. This method can be used to design much more powerful decomposers, since it takes into account more relations on the variables, such as:

- nonequivalence symmetry,
- equivalence symmetry,
- mixed symmetries,
- skew-nonequivalence symmetry,
- other

In a similar but earlier study, Radomir Stankovic and Nebojsa Denic [?], introduced the same 8 symmetry classes as Sadowska, but used the apparatus of Gibbs derivatives instead of FPRMs.

Nobody has done the study of using symmetry of multiple valued functions in decomposition, but there are interesting papers by Butler and Schueller [?], Epstein [?], Epstein, Miller and Muzio [?], [?], Stojmenovic, Miyakawa and Tosic [?], Mukhophadyay [?], Muzio [?], Muzio, Miller, and Epstein [?].

Muzio [?] has studied the realization of general multiple-valued symmetric functions from fundamental symmetric functions. Specifically, any arbitrary symmetric function can be realized as the disjunction of fundamental symmetric functions. Some of these functions may be combined, improving the efficiency of the algorithm. The objective of the design is to realize a given function with as few combinations as possible. The problem discussed by Muzio was the worst case. That is, certain functions will require the maximum number of combinations, and it is desired to find this maximum number. Muzio computed the maximum for 3- and 4- valued systems and conjectured the values for general m-valued systems.

Butler [?] solved the general problem and showed that as in the 3- and 4- valued cases, the ratio of the maximum number of components to the total number of fundamental symmetric functions approaches one-half as the number of variables n increases. This has an analog in binary sum-of-products expressions. That is, the binary function with the greater number of product terms (the EXOR function) requires exactly one half of the total number of minterms.

If we apply the tree search paradigm in the space of variables for partitioning [?], then it would be relatively easy to apply standard operator equivalence techniques to remove descriptors from nodes. For instance, if variables a and b are symmetric, one of them is removed as a descriptor. Let us observe that this technique can be used dynamically, to the newly created subfunctions, and the goal of previous decompositions would be to increase the symmetries in the subsequent stages. This way a subset of all bound sets would be tested to obtain the exact decomposition (the decompositions that minimize the value of the multiplicity index μ).

4.5.3 Kim's and Dietmeyer Approach to Symmetric Decomposition.

Kim and Dietmeyer [?], [?] observed that current systems for multilevel synthesis from PLA-like macros have a difficulty with symmetric functions. Designs of totally symmetric functions generated from descriptions that lack global network structure, have on average more than twice as many literals as the best designs, while the designs of non symmetric functions have on average 20% more literals than the best designs.

Classical disjoint decomposition, which is generally considered for small networks, is particularly suitable for symmetric function synthesis.

The Boolean PLA decomposers of Devadas [?, ?], and Ciesielski [?, ?], overcome the two most important drawbacks of algebraic factorization methods: too much dependency on given expressions and the sequential generation of subfunctions. However, they have no special treatment for symmetric functions.

Kim and Dietmeyer developed a simple but effective heuristic method for synthesizing symmetric functions that detects and takes advantages of symmetry and is based upon classical disjoint decomposition theory. The program of Kim and Dietmeyer produced almost always the best designs known to these authors.

Taking advantage of symmetry in multilevel synthesis requires detection of symmetry sets of multiple-output function F. Extensive work on the detection of symmetry sets using the array format was reported by [?].

Detecting of symmetry in the arrays of F may be performed by sequentially testing the arrays of each f_i The detection of symmetry pairs forms the basis for detecting larger symmetry sets. Detecting symmetry pairs is described by the following theorem:

Theorem 4.5.1 *(Kim and Dietmeyer).*
$f = (ON, DC, OFF)$ is symmetric with respect to set $L = \{x_i, x_j\}$ if and only if $ON_{x_i\ \overline{x_j}} \wedge OFF_{\overline{x_i}\ x_j} = 0$ and $ON_{\overline{x_i}\ x_j} \wedge OFF_{x_i\ \overline{x_j}} = 0$

If OFF-conditions are not provided, tests based on the following corollary are usually preferred because computing the complement often takes more time than other operations.

Corrolary 4.5.1 *(Kim and Dietmeyer).*
$f = (ON, DC, OFF)$ is symmetric with respect to set $L = \{x_i, x_j\}$ if and only if
$[\ ON_{x_i\ \overline{x_j}} \subseteq ON_{\overline{x_i}\ x_j} \vee DC_{\overline{x_i}\ x_j}\ ]$ and $[\ ON_{\overline{x_i}\ x_j} \subseteq ON_{x_i\ \overline{x_j}} \vee DC_{x_i\ \overline{x_j}}\ ]$

Theorem 4.5.2 *(Kim and Dietmeyer).*
If f is a completely specified function, symmetry is an equivalence relation on the variables of symmetry. Thus if $\{a, b\}$ and $\{b, c\}$ are symmetry pairs, $\{a, c\}$ is also a symmetry pair, and $\{a, b, c\}$ is a symmetry set. If L is a symmetry set and $\{a, b\}$ is a symmetry pair where a from L and b not from L then $L \cup \{b\}$ is also a symmetry set.

However if f is not completely specified, detection is not easy. Symmetry is not even a compatible relation in general. The three-element symmetry set is a special case among larger symmetry sets, and the following theorem shows how in may be detected.

Theorem 4.5.3 *(Kim and Dietmeyer).*
$f = (ON, DC, OFF)$ is symmetric with respect to $\{a, b, c\}$ if and only if f is pairwise symmetric with respect to $\{a, b\}$, $\{b, c\}$ and $\{a, c\}$.

Dietmeyer introduced a new operator to find largest symmetry sets in an relatively efficient way.

It is well-known that if f is totally symmetric then the output of f depends only on the weights of the input vectors. This can be carried over to multi-level multi-output functions: if F is symmetric with respect to L, then every single-output function f_i in F depends only on the L-part weights of input vectors. This weight dependency suggests the disjoint decomposition.

$$F(X) = F'(W, X - L) \tag{4.1}$$

where $X = \{x_1, x_2, \ldots\}$ is the set of input variables and $W = \{w_1, w_2, \ldots\}$ is the set of intermediate variable names assigned to the outputs $w_i(L)$ of $W(L)$, which is a multiple-output function totally symmetric on L. The variables of L are called *decomposition variables* and the function $W(L)$ will be called *recoder*.

Iterative decomposition using their heuristic recoding techniques is efficient for totally symmetric functions. The authors write that expanding this approach for partially symmetric functions is an open problem, but they believe that many of their ideas will also work for partially symmetric functions. They want to select the best recoder type based on the analysis of functions to be decomposed. Let us hope that this approach will also be easily expandable for multiple-valued functions.

4.6 Maitra Cascades

There are two types of Maitra cascades. The first one is a redundant cascade where one or more inputs are driven by the same Boolean variable. The second type is a non-redundant cascade in which every input is connected to a distinct Boolean variable.

The properties of Boolean functions which make them Maitra-realizable have been discussed in both spectral and function domains. The spectral approach seems to be an efficient method for testing and realizing any given Boolean function because the spectral coefficients provide global information about the function, which is more useful than the Boolean representation of the function.

The map [?], and the algebraic [?] approaches require the function to be in the minimum sum-of-product Boolean expression, which is a disadvantage.

The matrix approach of Sklansky [?] is better than the map and algebraic techniques. When the spectral approach is compared with the matrix method, it is observed that the procedure structure in spectral domain is more convenient for computer implementation, and it is not restricted by the number of true minterms or the number of input variables. This is then much better than in [?].

Let us observe that in general, the spectral approaches to various types of Boolean decompositions, not only AC one, but also pre- and post-processor decompositions, become now available thanks to the efficient Clarke's algorithm [?] to find a Walsh spectrum. Such algorithms were not useful practically before, since no efficient method to calculate the entire or partial spectra were available. The only disadvantage is that a method such as one by Clarke may be quite complex to implement.

Not much has been published about multiple-valued logic cascades, especially for incompletely specified functions.

4.6.1 Cascades and their Generalizations

Let us recall the 96-input EXOR problem. If one would assume that the corresponding circuit is Maitra-cascade realizable, then more efficient decomposition algorithm can be found. For each input variable one of three possible operations, AND, OR and EXOR is assumed, and the function is folded with respect to this operation. If no folding is possible, another operation is tried, or the program backtracks. If the circuit is Maitra-cascade realizable, this algorithm gives the solution, regardless on the number of inputs and percent of the don't cares. Although this algorithm is exponential with the number of input variables, it is still more efficient than the general AC decomposition. This is the reason why we are interested in using some ideas based on Maitra cascades in the framework of the AC decomposition.

We plan:

1. Develop an algorithm to create Maitra Cascades, [?], for strongly unspecified functions. For such functions, because of the high percent of don't cares, there may be solutions for most real life benchmark functions. Prove properties of the algorithm; specifically, we will have to prove that if such a cascade exists, the algorithm is able to find it.

2. Generalize the concept of Maitra cascade with the above in mind.

3. Use the function folding approach of Perkowski to solve the simpler EXOR problem for strongly unspecified function of 96 variables (10,000 samples).

4. This is similar to the problem of removing vacuous variables, removing all vacuous variables may be not good, still removing such variables should remain as an option. Use a method similar to above to reduce the number of variables. If found, the totally vacuous variables should be definitely removed.

4.6.2 Spectral Approaches to Maitra Cascades

Spectral methods have been also used for synthesis of cascades in the paper by Asaad and Bennett [?]. Asaad and Bennett define properties of Boolean networks that are realizable in non-redundant cascades. They use Walsh transform and give an algorithm for realizing a cascade based on these properties. Although their method is only for completely specified functions, it can be adapted to incompletely specified functions using techniques from Falkowski et al. [?, ?].

Synthesis of Maitra cascades by means of spectral techniques is also discussed by Stankovic, Tosic and Nikolic, [?].

By using the same approach as Mukhopadhyay and Stone [?], M. Stankovic et al give, in the spectral domain, a procedure that can be used to test an arbitrary function for cascade realizability and to generate a Maitra cascade for a function if one exists. The procedure determines one or more cells at the bottom of a cascade realization of the function and the residual function that drives this portion of the function. The procedure is applied iteratively to the successive residual functions, until a trivial residual function is found, since the applied method is based on the partial decomposition of the function. In the spectral domain, this decomposition is uniquely defined by the values of the determined pairs of spectral coefficients. A spectrum of the residual function is determined by applying the spectral translation to the spectral coefficients of the given function. These results permit to formulate a procedure that is suitable for computer processing. However, no experimental results are given in the paper.

Classification of Boolean functions by spectral means is also discussed in [?].

4.7 Binary Decision Diagrams versus Decomposition.

A point that is rarely made (besides the papers of Tsutomu Sasao, and Massud Pedram et al) and which we found very important, is the direct relation of the decision diagrams, networks of multiplexers, and AC decompositions. The cut in an ordered decision diagram (with a root node at the top) separates variables into two distinct, non-overlapping groups: variables that are above the cut, and those that are below the cut. The variables that are above the cut correspond to the free variables in AC decomposition, and using another terminology, to address variables of the output multiplexer. The variables that are below the cut correspond to the bound set variables in AC decomposition, and in other terminology, to the input variables that drive the data-inputs of the output multiplexer.

This relation is very useful for AC Decomposition for the following reasons:

1. There is a big body of sophisticated methods and algorithms for efficient variable ordering in Decision Diagrams. All those methods can be used either directly or indirectly. A Direct Method is to find the order of variables for partitioning as it were a decision diagram, but actually without implementing a diagram as a data structure in memory. An order of variables a, b, c, d,... corresponds to a sequence of sets: a, a,b, a,b,c, a,b,c,d... These sets can be either sets of bound variables, or sets of free variables. This is the approach used by Haomin Wu and Marek Perkowski [?, ?], and by Craig Files [?].

   Quinlan and other authors who work on decision diagrams are using entropy for variable ordering. Is it better than other methods for variable ordering? There are several other authors that use entropy for variable grouping and ordering, but there are no papers that would compare these approaches. Moreover, entropy based methods are used in general systems to separate variables to overlapping subsets, and thus such methods can be used in non-disjoint decompositions. There are no papers with this observation, not any experimental comparisons. This is one of our areas of research.

   Another and simpler approach is to use the BDD ordering methods in the decomposition just by applying a BDD program from SIS or other package to find a good diagram. Then one can create a sequence of variables in this diagram and look for the best cut in this order. We do not know of anybody using this approach.

   Yet another approach is to create some special decision diagram and apply the above technique. This is the approach applied by both Massud Pedram et al and Tsutomu Sasao.

2. Similarly, the methods that have been created for networks of multiplexers are equally applicable. For instance, Schaefer and Perkowski use the method developed by Almaini et al to create Kronecker Decision Diagrams. This approach has powerful advantage from the point of view of Pattern Theory, since it is well-suited to the strongly unspecified functions. Analogously, this approach can be applied to AC decomposition as well.

3. There is recently much research about new kinds of decision diagrams that have a decreased node count [?, ?, ?, ?, ?, ?, ?, ?, ?] and have been created for strongly unspecified functions. Some of them [?, ?] have been created for machine learning applications, this research can be again used to find a good variable sequence for AC decomposition.

4. Because decision diagrams are just factorized decision trees, the good variable ordering for a tree is usually a good order for a corresponding decision diagram. Therefore the results of the research on finding the variable order in programs such as C4.5 may be useful as well for finding good variables sequence.

5. There exist variable ordering algorithms [?] which use a combination of many methods to find a good order. These methods can be used to combine the best heuristics of the above methods in

the case of tie break of other methods (this method in a very simplified variant was also used by Craig Files [?]).

6. An interesting point in variable partitioning is that both the variant with increasing step-by-step the free set of variables, and the variant of increasing the bound set of variables have been tested in an older versions of Flash. These two algorithms correspond to two variable ordering methods applied in the decision diagrams. While most researchers find the order of the variables by building the diagram from the root to leaves [?], some other build the graph from the leaves to the root. This is an approach applied by both H. Wu and M. Perkowski and R. Kohavi. There is one more approach, the variable sifting algorithm, which gives currently the best results for BDDs but is the most complex from the programming point of view. This approach has been created by Richard Rudell in his BDD package [?] and by Rolf Drechsler in his KDD package [?]. The disadvantage of using this approach directly, i.e. on a completed diagram, might be obtaining poor quality results in case of the completely specified diagrams (one with terminal nodes: 0, and 1) were originally created. A solution to this problem would be to create from scratch an incompletely specified diagram of a kind described by Massud Pedram. Such diagram has three types of terminal nodes: 0, 1 and DC). However, as far as we know, there are no algorithms in the literature yet to implement the sifting algorithm of Rudell, or another order modifying method to this kind of diagrams.

The advantage of these methods is that they are all in the exact mainstream of logic synthesis research and industrial development, so many people work on them and there is much progress in this area recently. Re-using their results in the less popular and "crowded" area of decomposition seems to me like a practical and good idea, of not "reinventing a wheel".

4.8 Approaches to Graph Coloring in Literature

As explained in the introduction, our general plan is to investigate relations between graph coloring and Column Minimization. With this plan in mind, we have started our investigations in algorithms for the Graph Coloring and Maximum Clique problems. It is interesting, that although the number of papers on these topics is very large, there exist actually very few papers (and all of them relatively new) that deal with practical exact and high quality heuristic algorithms. In this section we will report those results from the point of view of using them in the research outlined.

Given is graph G with set of nodes N. Let us denote $card(N) = nn$. The *(Proper) Graph Coloring* of graph G is to find a color for each node of G such that every two adjacent (i.e. linked by an edge) nodes will have different colors. The set of all colors is denoted by C. The minimum cardinality of colors to color graph G is denoted by $\kappa(G)$, and called its *chromatic number*.

Graph Coloring has been proven to be an NP-complete problem Why is the coloring problem so hard? Is it "harder" than some other NP-hard problems used in CAD?

Although for some NP-hard problems good approximate algorithms exist, coloring is different: none of the algorithms found in literature does a good job for coloring large (n ¿ 1000) graphs [?, ?] See also the cited above theoretical result of Johnson.

We will review the exact algorithms first. The algorithms for finding chromatic number in an exact way are given in Hammer and Rudeanu [?], Christofides [?, ?], Persin [?], Wang [?], and Brelaz [?]. They were applied to graphs with not more than 100 (check it) nodes. Some algorithms, such as Maghoute's (cited after [?], use the method of finding all maximum independent sets and next covering nodes with independent sets. This method is totally analogous to the classical Quine-McCluskey algorithm for SOP minimization, so using this method to find exact graph colorings could take advantage of high-quality modern Boolean minimizers. (such as Espresso-Signature).

We are interested in other Graph Coloring methods based on backtracking, which may be will be more efficient than our graph-coloring algoritms.

Most of the approximate graph coloring algorithms belong to the greedy category. Greedy algorithm colors nodes one by one in some prespecified order not coloring neighbors with the same color.

All other algorithms use one of the following heuristics [?, ?]:

A. A node of high degree is harder to color than a node of low degree.

B. Nodes adjacent to the same set of nodes are colored with the same color.

C. Try to color as many nodes as possible with every color already used.

D. Color with the smallest color available.

There are two kinds of Graph Coloring Algorithms: Sequential (Preset) and Recursive (Adaptive).

4.8.1 The Sequential (Preset) Algorithms

These algorithms decide the order of coloring of nodes before the coloring.

1. The LARGEST_FIRST algorithm of Welsh and Powell [?] orders nodes by descending degree. Then colors them greedily.

2. The SMALLEST_LAST algorithm of Matula, Marble and Isaacson [?] selects the node of the lowest degree to color last, removes it and repeats the process. Then colors greedily beginning with the last node removed.

3. Algorithm of Williams [?] orders nodes in descending degree by considering neighborhood degree sums, and so on, instead of the degrees.

4. Syslo (1989) comments on the relations between sequential coloring and the Welsh-Powell (1967) upper bound for the chromatic number of a graph. He suggests that new ordering may lead to improved results. His book (1983) [?] has section 4.1 devoted entirely to coloring algorithms and extensive bibliography.

4.8.2 The Recursive Sequential (Adaptive) Algorithms

1. The recursive algorithms can change order recursively during coloring. They use heuristics A,B,C,D dynamically.

2. The algorithm of Johri and Matula.

3. The algorithm by Tehrani [?].

4. The algorithm of Leighton [?].

5. The algorithm of Wood [?] defines a similarity measure on pairs of nodes and next examines nodes in pairs in order of decreasing similarity, sometimes coloring and sometines not, depending on degrees of the pair and whether or not one of the two is already colored.

6. The algorithm of Johnson [?] selects nodes of low degree and constructs large independent sets.

7. The algorithm by Brelaz, called DSATUR, uses heuristics A and B, [?]. The color-degree of node v is the number of colors used so far on vertices adjacent to v. DSATUR repeatedly chooses a node of largest color-degree, and assigns to it the smallest possible color. If two nodes have equal color-degrees, the one adjacent to most uncolored nodes is colored first. DSATUR colored graphs with 100 nodes and gave solutions worse than few nodes. By some authors it is treated as the best algorithm for small graphs [?, ?]

8. The really interesting Boolean functions produce graphs of 1000 nodes. According to [?] all those algorithms give poor results on graphs with 1000 nodes.

9. An interesting hypothesis is given in [?]. The authors suggest that the "failure in coloring large graphs may be due, at least partially, to the way in which κ is determined in large graphs".

4.9 The Maximum Clique Problem

The maximum clique of graph G is the maximum independent set of graph $\overline{G}$, the complement of graph G. Both of these problems can be reduced to linear integer programming [?].

There are several important papers on maximum cliques in our references. Several algorithms have been written, computer programs implemented, and even compared for this problem, all of them more efficient than the classical old approaches referenced by most authors. There are especially many good papers from former Soviet Union that include deep mathematical theory for Graph Coloring. Below we will provide bf only partial literature on the subject. (I have more positions but I have no time to find them all now.) Some algorithms are better for planar, triangle-free, bipartie, etc graphs. There are many mathematical results and timing/complexity comparisons. Since a maximum clique for graph G is a maximum independent set for the complement graph of G, then the results on the maximum independent sets should be utilized as well.

There exist the well known relations among maximum clique generation, clique partitioning, clique covering and coloring of a complement graph, that are used by several authors to find good solutions to this or very similar problems.

In our opinion the area of maximum clique generation is very competitive, even more than decomposition. Therefore the results of this research may prove very useful in creating efficient decomposers.

There are, basically, two groups of methods. Some methods start from small cliques and grow them, other methods start from large groups and remove nodes until cliques are found. It seems that it depends on the type of graph which method is better. Perhaps for graphs of Boolean functions it is better to start from small cliques and grow them instead of starting from large groups and shrinking them. In any case, which particular method is better, depends perhaps on the density, type and other properties of the graph. What is the experimental evidence of the properties of graphs that are created for decomposition? Do they have any particular properties that make this algorithm better than the well known algorithms?

Important algorithms for maximum cliques are in: [?, ?, ?, ?, ?, ?, ?, ?, ?, ?]. Paper [?] is a very important paper, also, chapter 9 in [?] is worth reading.

Many algorithms for proper graph coloring and related problems exist in the literature, [?, ?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?], [?, ?, ?].

Part II

APPLICATIONS OF DECOMPOSITION

Chapter 5

Decomposition for FPGAs

5.1 FPGA synthesis.

A renewed interest in AC decomposition in recent years is caused by the introduction of Look-up Table Field Programmable Gate Arrays (FPGAs) by Xilinx in 1986, and other companies in succeeding years (AT&T, Actel, Motorola, Algotronix). When it was found that the adaptation of earlier algebraic methods didn't not work sufficiently well for the Lookup Table model, researchers switched to Boolean decomposition. Currently, in most of the systems, the decomposition is only an auxiliary process (MIS-PGA of Murgai and Brayton [?, ?], HYDRA of Filo, Yang, Mailhot, and DeMicheli, [?]).

However, with the arrival of systems such as TRADE [?], as well as the systems of Pedram [?], Steinbach [?], and Luba [?], that are *primarily* based on the decomposition, the situation is becoming to change drastically. Even the U.C. Berkeley group of Professor Brayton, who invented the algebraic factorization approach and was for a long time its most stubborn adherent, is recently devoting more attention to Boolean decomposition.

5.1.1 MIS-PGA and MIS-PGA(new).

In [?] R. Murgai et al presented improved algorithms for Table Look Up Architectures which were included in the well-known program MIS-PGA(new). It uses several techniques, and Roth-Karp decomposition is only one of them. They also use other decomposition techniques such as: co-factoring, and AND-OR decomposition.

MIS-PGA(new) [?] starts from a tree-like network. First, it applies a variety of decomposition methods to decompose the input network into a feasible network. The *feasible network* is a network in which the number of inputs of each node is limited. For Xilinx architecture, this input number is up to five. Compared with its predecessor *MIS-PGA*, [?], more decomposition techniques have been incorporated. The decomposition methods that *MIS-PGA(new)* employs include *cube packing* which works well for functions with more or less *mutually orthogonal cubes*. *Roth-Karp decomposition* is suitable for *symmetric functions* but doesn't work with non-decomposable functions. *AND-OR decomposition* can break an infeasible node into several feasible nodes. The generated feasible node is either an inverter, a two-input AND or a two-input OR gate. *Cofactoring decomposition* uses the concept of Shannon expansion to expand the network into a feasible network, in which all nodes have up to three input variables. *decomp -d* (there is no special name for this decomposition in [?]) partitions the cubes of the input network into a set of cubes having disjoint variable support, and creates a node for each partition of cubes and a node which is the OR of all these partitions. The resulting subnetworks may not be feasible, and neither are those from the *kernel extraction decomposition*. Other decomposition techniques are used to make the network feasible. After decomposition, *MIS-PGA(new)* uses a *maxflow algorithm* to generate all possible *super-nodes* and solves the *binate covering problem* to minimize the cardinality of the super-node set which covers the entire network. Finally, by solving the *maximum matching problem*, it merges all possible nodes into the FG mode CLBs.

An infeasible node is one that cannot be realized by a single Configurable Logic Block (CLB). A feasible function can be realized by a single CLB. The decomposition phase starts from a possibly infeasible network and creates a feasible network.

In *kernel extraction,* the kernels are extracted from an infeasible node. The node and the kernel are then recursively decomposed. In the RK decomposition, a bound set X of cardinality m is chosen from the fan-ins of the infeasible node n. The rest of the inputs of n form the free set, Y.

Then the decomposition of n is of the form:

$$f(X,Y) = g(G_1(X), G_2(X)......G_t(X), Y)$$

.

where $G_1, ..., G_t$ are feasible functions.

If $t + |Y| > m$, g needs to be decomposed further ($|Y|$ denotes the cardinality of Y). To get the best possible disjoint decomposition, all choices of bound sets have to be tried.

Murgai's et al implementation does not follow the original Roth and Karp papers. They do not search all possible choices of bound sets, but select the first one. This is the form in which RK decomposition has become most popular. The results may be then highly non-optimal, but the method is fast. They just observe that if the function is symmetric then all partitions are equally good, so their scheme produced excellent results on some of the benchmarks like 9symm1, 9sym, rd84, and rd73. The recent improvements to this approach are in [?].

5.1.2 Other Lookup-Table FPGA Mappers

Chortle-crf [?] by Francis et al starts from a *Directed Acyclic Graph* (DAG). It first divides the DAG into a *forest of trees*. Then, by using the *dynamic programming approach*, it carries out technology mapping on each tree to find the *minimum cost circuit.* Several techniques are used, such as *two-level decomposition* which uses a *bin packing algorithm*, *multi-level decomposition*, *exploiting reconvergent paths* and *replication of logic at fanout nodes*. These make the *Chortle-crf* get a significant improvement over its predecessor *Chortle* [?] in both the quality of solutions and the running time.

X-map [?] by Karplus converts BLIFF format into an *if-then-else* DAG, which is a network with the number of inputs of each node less than or equal to three. Then it goes through a *marking* and *reduction processes* to minimize the network. Finally, a simple merging algorithm is applied to merge all possible nodes into the FG mode CLBs.

VISMAP [?] by Woo introduces the concept of invisible edges. The *invisible edge* is an edge which doesn't appear in the resulting network after mapping. It starts from a feasible network (in DAG format), partitions this network into several subgraphs of a reasonable size and goes through a *pre-processing* and a *main processing* step to determine the invisible edges to reduce each subgraph. A merge algorithm is used to merge all possible nodes into the FG mode CLBs.

Hydra [?] by Filo et al is similar to MIS-PGA: it does disjoint decomposition followed by a node minimization phase. The main difference is that both phases are driven by the fact that Xilinx CLB may realize two outputs also. Hydra puts more attention on the FG mode CLBs.

The main objective of the above FPGA technology mapping approaches is to minimize the area. There are several other approaches: *MIS-PGA(d)* by Murgai et al, [?], *Chortle-d* [?] by Francis, [?], and *DAG-MAP* [?] by Cong, which aim at the delay optimization.

The FPGA mapping approaches mentioned above consist, in general, of four major steps:

- graph construction,
- decomposition,
- reduction,
- packing.

In the *graph construction* step, the very first step, a special kind of network-like graph or a set of subgraphs is created. The graph (or network) can be feasible or infeasible. Several specific FPGA mapping techniques are applied to it. In the *decomposition* step, the most important step in the mapping process, a variety of decomposition methods are applied to transform an infeasible network into a feasible one. During the process, the decomposition algorithms try to minimize the number of nodes in the decomposed network as well as the number of input variables per node. In the *Reduction* step, generally more computationally expensive, some covering algorithms are applied in order to find a set of minimum number of CLBs which can cover the entire network. In the *Packing* step, according to the specific FPGA architecture, some algorithms are used to merge the possible nodes to further decrease the area. Most of the operations used in the above four steps are local operations. The dynamic programming algorithm ensures the local operation to traverse across the network. The program is recursively invoked until a satisfactory result is reached.

It is not clear which, if any, of the above methods have some advantages to DFC minimization for very strongly unspecified functions typical for machine learning.

Figure 5.1: Local transformation

5.2 General Decomposition with EXOR Transformations

The known decomposition methods are passive in the sense that they only test whether a function is decomposable or not. If it is, the decomposition is carried out. But what do we do if the function is not decomposable?

The method presented in TRADE is called a *Local Transformation Method.* This method can transform a function which is non decomposable (in the sense of Curtis) into several decomposable ones.

The basic idea of this method is to transform some columns in the original Karnaugh map to make them identical to some other columns in order to decrease the column multiplicity. It is carried out in three steps:

1. The output values of the Conflict Cubes (the Conflict Cubes are the cubes that make the two columns different) in one column are complemented in order to make the two columns identical, creating the Modified Karnaugh Map.
2. The so-called Modifying Karnaugh Map is created from the initial map and the Modified Map.
3. EXOR operation on the function described by the Modified Karnaugh Map and the function described by the Modifying Karnaugh Map is executed, creating the original function.

A *Modified Cube* is a name for the Conflict Cube which outputs have been complemented, both cubes are identical. An attempt is made in the first step to make the Modified Cubes as large as possible, and at the same time keep the number of the Modified Cubes as small as possible.

Like the cube -1 in columns 000 and 001 in Figure 5.1a, it makes these two columns different. The formula for calculating the Conflict Cubes between i-th and j-th columns is:

Conflict Cube Set = (ON(i) $\sqcap$ OFF(j)) $\sqcup$ (ON(j) $\sqcap$ OFF(i))

where:
ON(i) and OFF(i) are the ON and OFF sets of i-th column.
ON(j) and OFF(j) are the ON and OFF sets of j-th column.

The formula states that the Conflict Cubes between the i-th and j-th column are the union of the intersection of the ON set of i-th column with the OFF set of j-th column and the intersection of the ON set of j-th column with the OFF set of i-th column.

The following example is used to illustrate this method and above terminologies. Figure 5.1a is an original Karnaugh map with the bond set $\{c, d, e\}$ and a column multiplicity of four. We intend to decompose this function into two sub functions, and each of them has a column multiplicity of no more than two.

We perform the local transformation to decrease the column multiplicity. First, the output values of cubes -10-1 and -1101 in Figure 5.1a are complemented. The resulting cubes are called *Modified Cubes* (they are the same as the Conflict Cubes). After modification, the resultant Karnaugh map is called the *Modified Karnaugh Map (Modified Function)* f_{ed} as shown in Figure 5.1b. Now the column multiplicity of the Modified Karnaugh Map is two.

Next, because the output value of the Modified Cubes have been complemented, the compensation must be made. To achieve this, another Karnaugh map is created with the positions corresponding to the Modified Cubes in the original Karnaugh map set to 1's and the others set to 0's. This is shown in Figure 5.1c. It is called the *Modifying Karnaugh Map* (*Modifying Function* f_{ing}). Notice that in this case the column multiplicity of the Modifying Karnaugh Map is two as well. If there are DC cubes in the original Karnaugh map, we keep them unchanged in both the Modified and Modifying Karnaugh Maps, because they will make both of these maps as amenable as possible for further minimization.

Finally, the EXOR operation of the function described by the Modified Karnaugh Map and the function described by the Modifying Karnaugh Map results in the function described by the original Karnaugh map. That is:

$$f = f_{ed} \oplus f_{ing}$$

The presented method changes a non decomposable function (in sense of the column multiplicity less than or equal to two) into two decomposable functions (with the column multiplicities of both equal to two). The method is called local transformation, but this local transformation is based on the global view of the entire function to make both Modified and Modifying Karnaugh Maps more simple.

The application of local transformation to the general implementation of FPGA mapping is shown in Figure 5.2. If a function is non decomposable, it will be transformed into two functions, a Modified and a Modifying function. If the Modified or Modifying function is non decomposable (in a more restricted condition), the local transformation should be applied again. Next, the function f from Figure 5.1a is used again as an example to show the detailed procedure of local transformation.

In Figure 5.1a there are only four different columns as shown in Figure 5.3. We denote these four columns by letter A, B, C and D, and call them (letter) column A, column B, column C and column D. Column A (letter column) includes three real columns: columns 000, 110 and 100 in the original Karnaugh map. Column B includes one column, column 001. Column C includes two columns: columns 011 and 101. Column D includes two columns: columns 010 and 111. "Number" in Figure 5.3 is the number of real columns that a letter column contains.

Create the Modification Factor Table as shown in Figure 5.4 and initiate all its cells to zero. The value in each cell is a weighted Modification Factor of the column pair corresponding to the row and column labels of the table. Cell $A \rightarrow B$ stores the Modification Factor of complementing the output value of all Conflict Cubes in the column A in order to make columns A and B identical. Cell $B \rightarrow A$ stores the Modification Factor of complementing the output value of all Conflict Cubes in the column B in order to make columns B and A identical.

Later on, the Modification Factor Table is sorted. In order to keep the correct correspondence between the value and the column pair it represents, two more storage units are attached to each cell to store the two columns that the Modification Factor corresponds to. The Modification Factor Table is, therefore, a list of 3-tuples. For example, the Modification Factor Table in Figure 5.4 is a list: $\{(A, B, 12)$, $(A, C, 24)$, $(A, D, 15)$, $(B, A, 4)$, $(B, C, 5)$, $(B, D, 8)$, $(C, A, 16)$, $(C, B, 10)$, $(C, D, 8)$, $(D, A, 10)$, $(D, B, 16)$, $(D, C, 8)\}$.

The Modification Factors are calculated in the following way:

- If the Conflict Cube is a minterm, the value of the corresponding cell in the table is increased by the number of input variables.
- If the Conflict Cube consists of two minterms, the value of the corresponding cell in the table is increased by the number of input variables minus one. This is because a larger cube can simplify to a greater extent both the Modified and Modifying Karnaugh Maps.
- If the Conflict Cube is composed of four minterms, the value of the corresponding cell in the table is increased by the number of input variables minus two, and so forth. That is:

$$ModificationFactor = \sum(number\ of\ input\ variables - number\ of\ DCs\ in\ the\ Conflict\ Cube)$$

The summation is over all Conflict Cubes in the modified column.

CLBs

f_{ed}

CLBs

f_{ed}

f

f

f_{ing}

CLBs

f_{ing}

Figure 5.2: Application of local transformation to FPGA mapping

ab				
00	1	1	0	0
01	1	0	0	1
11	0	1	1	0
10	0	0	0	0
Column	A	B	C	D
Number	3	1	2	2

Figure 5.3: four different columns

	A	B	C	D
A		A(-¿B 12	A(-¿C 24	A(-¿D 15
B	B(-¿A 4		B(-¿C 5	B(-¿D 8
C	C(-¿A 16	C(-¿B 10		C(-¿D 8
D	D(-¿A 10	D(-¿B 16	D(-¿C 8	

Figure 5.4: Modification Factor Table

Let's fill the Modification Factor Table in Figure 5.4 now. The Conflict Cube between column A and B is the cube -1. If we change column A to make it identical to column B, we need to complement the output value of the Conflict Cube (cube -1 in column A). That is to change the output value of minterm 01 of column A in Figure 5.3 from 1 to 0 and the output value of minterm 11 from 0 to 1. Column A will have the vector [1, 0, 1, 0] after complementation of the Conflict Cube. This vector is the same as that of the column B. Cube -1 has one - (dash), the Modification Factor would be the number of input variables minus 1, that is four. Furthermore, column A includes three columns, columns 000, 110 and 100. All these three columns need to be modified. Therefore, the Modification Factor must be multiplied by three, and the final Modification Factor is twelve, as shown in cell $A \rightarrow B$ of the Modification Factor Table in Figure 5.4. Another way to make columns A and B identical is to complement the output value of the Conflict Cube, cube -1 in column B. That is to change the output value of minterm 01 of column B in Figure 5.3 from 0 to 1 and the output value of minterm 11 from 1 to 0, so both columns have the same vector [1, 1, 0, 0] after the modification. Column B includes only one column which is column 001, the Modification Factor is four as shown in Figure 5.4. Using the same reasoning the Modification Factor Table is completed.

The Modification Factor Table is sorted in an increased order of the Modification Factors. The columns which have smaller values of the Modification Factors are now modified until the required column multiplicity is reached. Cell B $\rightarrow$ A is selected because it has the smallest value of four. Complement the output value of Conflict Cube, cube -1 in column 001 of the Karnaugh map in Figure 5.1(a). The vector of column 001 is changed from [1, 0, 1, 0] to [1, 1, 0, 0]. At the same time, set the output value of cube -1001 of the Modifying Karnaugh in Figure 5.1(c) to 1. After modifying column B, the column multiplicity of the Modified Karnaugh Map is reduced to three. Our aim is to reduce it to two, so another modification will be carried out. The next smaller Modification Factor is five in cell B $\rightarrow$ C. But we cannot change column B to column C because we have changed column B to column A. The next smaller value is eight in cell $C \rightarrow D$. Complement the output value of Conflict Cubes, cube -1 in both columns 011 and 101 of the Karnaugh map in Figure 5.1(a), The vectors of the columns 011 and 101 are changed from [0, 0, 1, 0] to [0, 1, 0, 0]. At the same time, set the output value of cubes -1011 and -1101 of the Modifying Karnaugh in Figure 5.1(c) to 1. After this modification, the column multiplicity of the Modified Karnaugh Map is reduced to two.

The function f with a column multiplicity of four has been transformed into two functions: f_{ed} and f_{ing}, both of them have a column multiplicity of two. The relation between them is:

$$f = f_{ed} \oplus f_{ing}$$

The pseudo-code for local transformation is shown in Figure 5.5.

The above technique has been incorporated into TRADE. The basic steps of *TRADE* are as follows:

1. Read in the input file written in *Espresso* ".type fr" format
2. Select an output. Perform partition analysis (the number of bond set variables is fixed to five) to obtain the "best" partitions and Additional Partitions. *Additional Partitions* are the partitions whose bond sets consist of the variables that are the input variables of some CLBs in the CLB Pool, and these variables are also in the range of the input variables of the current decomposition. The *CLB Pool* is a list of all CLBs previously generated by the program. For example, if the input variables of the current decomposition are *a, b, c, d, e, f, g, h, i,* and in the CLB Pool there are three CLBs with the input variables of each: $\{a, b, c, d, e\}, \{c, d, g, h, i\}$ and $\{a, h, k, l, m\}$ respectively. Then the variable sets $\{a, b, c, d, e\}$ and $\{c, d, g, h, i\}$ can be used as the additional partitions, while *a, h, k, l, m* can not be used because there are no variables *k, l* and *m* in the current decomposition.
3. Execute decompositions using the "best" and Additional Partitions. Encode the bond set and try to use as many CLBs from the CLB Pool as possible. The way to efficiently reuse CLBs

```
local_transformation()
{
create Modification Factor Table;
sort Modification Factor Table in increasing order;
modify column at the beginning of the queue;
change Modified Karnaugh Map;
fill Modifying Karnaugh Map;

while (column_multiplicity > required_column_multiplicity)
  {
  modify column at the next position
  change Modified Karnaugh Map;
  fill Modifying Karnaugh Map;
  }
}
```

Figure 5.5: Pseudo-code of local transformation

Before
A After
x
B
CLBs y
z

C

Figure 5.6: Before and after decomposition

will be discussed later in this chapter. Graph coloring technique is applied at this stage to get a quasi-optimum don't care assignment. Select a partition which results in the smallest Cover Ratio. The *Cover Ratio* is a ratio of the number of newly created CLBs over the difference of the input variables before and after decomposition, that is:

$$CoverRatio = \frac{number\_of\_newly\_created\ CLBs}{inputs\_before\_decomposition - inputs\_after\_decomposition}$$

For example, in Figure 5.6, if CLB x and y exist in the CLB Pool, CLB z is a newly created one, then Cover Ratio = 1/(8 - 6) = 0.5.

If the function is non decomposable, perform the local transformation to make it decomposable.

4. Repeat steps 2 to 3 for the blocks left (block C in Figure 5.6) until all decomposed blocks are with five or less inputs.

5. Repeat steps 2 to 4 for all Modifying Functions which were created by local transformations.

6. Repeat steps 2 to 5 for all outputs.

Recently, there is work on incorporating many improvements to a new version of TRADE, to provide it with the possibility of obtaining better solutions with larger search space, and, in general, investigating trade-offs between the quality of the solution and the speed of obtaining it.

Code	Color 3	2	1	0	
					x
					y

(a)

Code	Color 3	2	1	0	
	1	1	0	0	x
	0	1	1	0	y

(b)

Code	Color 3	2	1	0	
	0	0	1	1	x
	1	0	1	0	y

(c)

Figure 5.7: CLB reusing concept

5.3 CLB Reusing

In the previous section, we have presented the encoding algorithm for the bond set, but we didn't mention the possibility that some CLBs in the CLB Pool might be reused. In TRADE, each previously generated CLB is recorded in a 32-bit long word, called a *CLB frame*. Because there can be up to five inputs to each CLB, thirty two bits are required to store all possible combinations of the inputs ($2^5 = 32$). For example, the logic of CLB h:

```
.names a b c d e h
10-01 1
11101 1
01-0- 1
```

is stored in the program as:

```
31                             0  Minterm position
00100000001000100011001100000000  CLB h
```

Each position in the above frame corresponds to a minterm. For example, minterm 11101 resides in the position 29. If a minterm belongs to the ON set, there is a 1 in the corresponding position of the frame. Otherwise 0. By proper coding it is possible to reuse CLBs in the CLB Pool.

The basic idea of CLB Reusing is originated from the following facts:

Suppose that the column multiplicity of a Karnaugh map is four (it is the same as saying that there are four colors). We would need two ($4 \leq 2^2 = 4$) variables (x and y as shown in Figure 5.7a) to encode the bond set.

We can code variables xy as 00, 01, 11, 10 as shown in Figure 5.7b. There are $4! = 24$ possible ways to code xy. But, if in the CLB Pool there is a CLB that can implement the pattern 0011 (call it x, this is the pattern for coding as shown in Figure 5.7c, not the internal storage frame). If we code y as 1010 (y is a newly created CLB) as shown in Figure 5.7c, we have realized the coding of xy by using one CLB in the CLB Pool and a newly created one. The y is coded in such a way that no two xy Codes are identical. Next example is used to show the detailed procedure of CLB Reusing algorithm. Suppose that we have a function with seven input variables a, b, c, d, e, f, g as shown in Figure 5.8.

After graph coloring, six colors are obtained as shown in Figure 5.9. In Figure 5.9, columns 0, 1, 3, 4, 7, 9, 12, 14, 15 and 17 belong to color 0, columns 18, 19, 22, 23, 25, 26 and 27 belong to color 1, and so forth. A careful reader will find that columns 20 and 29 are missing. They are DC columns and can be put in any color position.

Because there are six colors (column multiplicity is six as well), we need three CLBs ($3 > \log_2 6 = 2.585$) to encode the bond set. Suppose that in the CLB Pool there are three CLBs which have the same input variables as that in the current bond set, they are CLBs w, x and y.

?

a
b
c
d
e

x

CLB Pool

y

z

f
g

Figure 5.8: CLB reusing example

```
00010000101010010010100000001011   w
01001110111011101111011110011011   x
10011110110011000000000001100100   y
```

We try to use as many CLBs in the CLB Pool as possible. If the CLBs in the CLB Pool can fully cover the columns in some colors, they might be reused. The term "fully cover the columns in some colors" means that in the CLB frame all positions corresponding to the columns included in some colors must be exactly 1, while all the other positions must be 0. We test if the CLBs w, x and y can fully cover the columns in some colors in Figure 5.9. It is found that the CLB x can fully cover the colors 0, 1 and 3, because in the CLB x frame all positions correspond to the columns included in the color 0, 1 and 3 are 1, all the others are 0. CLB x has the pattern 001011 as shown in Figure 5.10. The same test results in that CLB y can fully cover the colors 1 and 2, and it has the pattern 000110 as shown in Figure 5.10. While CLB w can cover the columns 0, 1 and 3 in the color 0, but it cannot cover the columns 4, 7, 9, 12, 14, 15 and 17 in the color 0, this is not a full cover. Therefore CLBs x and y can be reused, but CLB w can not be reused. We add a new CLB (CLB z) and code it in such a way that each color has a different Code bit. The procedure for coding the newly added CLB is as follows:

> Start from the color position 0 in Figure 5.10. CLB y and CLB x give the bits 0 and 1 (Code 01. Code is read vertically, not horizontally), respectively. Put a 0 at the color position 0 of CLB z as shown in Figure 5.10. In the color position 1, the corresponding bits from CLB y and CLB x are 1 and 1 (Code 11) respectively. Because Code 11 is different from Code 01 in the color position 0, put a 0 again at the color position 1 of CLB z. The same way, put a 0 at the color position 2 of CLB z. In the color position 3, the Code is 01, it is the same as that in the color position 0. Increase the Code put in the color position 0 of CLB z (that is 0) by 1, this leads to a 1. So put a 1 at the color position 3 of CLB z. Perform the same operation for the color positions 4 and 5.

abcde 00000 00001 10000

fg

00

01

11

10

Color 0 0 0 2 2 0 2 0 0 3 0 0 3 4 0 3 4 1 1 1 3 2 x 2 x 3 1 1 1 1 0 5

Color

5	4	3	2	1	0	
16	11	8	2	18	0	Column
	24	10	5	19	1	
		13	6	22	3	
		21	28	23	4	
		30	31	25	7	
				26	9	
				27	12	
					14	
					15	
					17	

Figure 5.9: After graph coloring

	Color 5	4	3	2	1	0	
Code	1	0	1	0	0	0	CLB z
	0	0	0	1	1	0	CLB y
	0	0	1	0	1	1	CLB x
	16	11	8	2	18	0	Column
		24	10	5	19	1	
			13	6	22	3	
			21	28	23	4	
			30	31	25	7	
					26	9	
					27	12	
						14	
						15	
						17	

Figure 5.10: The final coding

The final coding of CLB z is 101000 as shown in Figure 5.10.

The newly generated CLB z has the internal storage frame as:

```
01000000001000010010010100000000    z
```

It is formed according to the pattern 101000 of CLB z. The color position 3 is 1 in the pattern of CLB z. Columns 8, 10, 13, 21 and 30 correspond to this color, so in the CLB z frame, the positions 8, 10, 13, 21 and 30 are filled with 1. The same operation is performed for the color position 5, and the internal storage frame of CLB z is obtained. By proper coding, we have reused two CLBs in the CLB Pool. The pseudo-code for CLB reusing is shown in Figure 5.11.

```
CLB_reusing()
{
collect reusable CLBs;
fully cover testing;

for (i = 0; i < total_color; i++)
  {
  if (reused_CLB_code_repeat(i) == true)
    code(i)++;

  new_CLB_code(i) = code(i);
  }
}
```

Figure 5.11: Pseudo-code of CLB reusing

Chapter 6

Decomposition for submicron VLSI layout

Ben Drucker, Jeske, Burns oba, GUD, Luba, Jozwiak.

6.1 Is DFC a good measure of decomposition quality?

Is DFC-controlled Decomposition Search a Good Measure for Machine Learning? How is Flash compared to other methods? If one will use another cost function than DFC, how the learning quality would be affected?

Complexity minimization of a circuit by a decomposition controlled by DFC is in our opinion not a good method for some categories of functions, especially multi-output symmetric functions. This is because it will not create the circuit with actual minimum of DFC, or especially any other circuit complexity measure involving connection "complexity", like counting literals. Flash is not solving the problem because it can end up with large lookup table blocks which are practically non-realizable.

An example of weakness of Flash in this respect is a set of all symmetric functions of 5 variables. In standard symmetric realization it requires 1+2+3+4+5 demultiplexers and 1+2+3+4 2-input OR gates. Since some functions of 5 variables are non-decomposable, LOOKUP Table realization would require at least two lookup table of 5 inputs each, a total of DFC = 32 * 2. May be, however, this problem does not exist asymptotically, i.e. for large functions.

Another related problem is efficient realization of recorder functions in case the total symmetry of a subset of variables was used in the decomposition. The problem is well known from the literature, for instance it is discussed by Kim and Dietmeyer [?].

It is believed [?] that DFC calculated in terms of 2-input gates is close to the number of literals. This belief should be verified on examples.

Chapter 7

Decomposition for Machine Learning and Data Mining

7.1 Decomposition of Decision Tables.

The proposed functional decomposition procedure can also be effectively used for reduction of space requirements in many problems related to data representation in Machine Learning [?, ?].

Let F be a mapping function

$$F : c_1 \times \ldots \times c_n \longrightarrow D \tag{7.1}$$

representing functional dependency $D = F(C)$, where C is the set of condition attributes and D is the set of decision attributes.

Let A, B be the subsets of C such that $C = A \cup B$ and $A \cap B = \emptyset$. We say that there is a functional decomposition of F iff the original mapping F can be replaced with two mapping functions G and H such that

- G: $b_1 \times \ldots \times b_t \longrightarrow L$
- H: $a_1 \times \ldots \times a_t \times L \longrightarrow D$
- i.e. $F(C) = H(A, G(B)) = H(A, g)$, where G and H denote functional dependencies:
- $G(B) = g$ and $H(A, g) = D$.

The final decision D is achieved trough a two-step process whereby an instance is mapped to an intermediate label using one subset of attributes, and then mapped to the final label using a disjoint subset of attributes.

The problem is, therefore, quite similar to that of decomposition of multiple-valued functions. The only difference lies in fact that the set of binary outputs is now replaced by a multiple valued decision attribute. Thus a simple counterpart of Theorem 15.1 for a case of decision tables can be stated as follows:

Theorem 7.1.1 *Functions G and H represent a functional decomposition of function F i.e. $F = H(A, G(B))$ iff there exists a partition $\Pi_G \geq P(B)$ such that*

$$P(A) \cdot \Pi_G \leq P(D) \tag{7.2}$$

where all the partitions are over the set of objects and the number of values of component G is equal to the number of blocks (i.e. equivalence classes) of partition Π_G.

The structure of the decomposed function is shown in the Figure 7.1. The procedure of making a final decision is as follows: an intermediate decision is made on the basis of the attributes' subset B and then taking into consideration both the intermediate decision and the attributes subset A, the goal decision is made, which is equal to the corresponding value of the function F.

In the theorem the partition Π_G represents mapping function G, and the product of partitions $P(A)$ and Π_G corresponds to H. The component decision tables of the resulting mappings can be obtained from these partitions.

The decomposition gain comes from the fact that in some cases the table representation size of the function generated using decomposition will be much smaller than the representation of a single unified function.

The same example can be used to explain how the two-step decomposition process may be a better way to capture structure in the data represented by decision tables. The function F given previously in the form of a truth table now is rewritten in the form of a decision table (Table 7.1).

Example 7.1.1

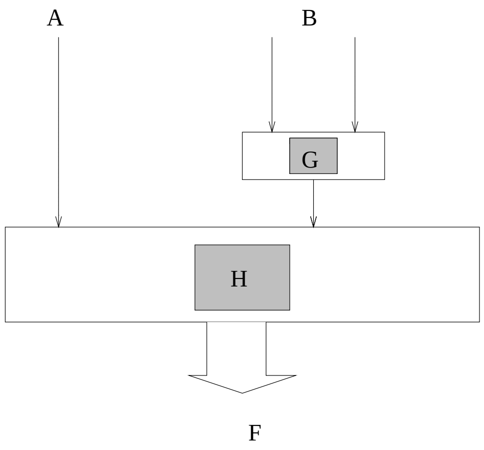


Figure 7.1: Block Diagram of the Decomposition of Decision Tables

	a_1	a_2	a_3	a_4	a_5	a_6	d
1	0	0	0	0	0	0	1
2	0	0	1	1	0	0	1
3	1	2	2	0	1	1	2
4	0	1	1	0	0	1	2
5	0	1	0	2	0	1	3
6	1	2	2	3	2	0	2
7	1	2	2	2	0	1	1
8	0	0	1	1	0	1	3
9	0	1	0	3	2	0	4
10	2	2	2	3	2	0	4

Table 7.1: A Truth Table to Example 16.1

	a_1	a_2	a_3	G
1	0	0	0	1
2	0	0	1	1
3	1	2	2	1
4	0	1	1	1
5	0	1	0	2
6	2	2	2	2

Table 7.2: A Truth Table of function G

	a_4	a_5	a_6	G	d
1	0	0	0	1	1
2	1	0	0	1	1
3	0	1	1	1	2
4	0	0	1	1	2
5	2	0	1	2	3
6	3	2	0	1	2
7	2	0	1	1	1
8	1	0	1	1	3
9	3	2	0	2	4

Table 7.3: A Truth Table of function H

Let us decompose the function F of Table 7.1 taking into consideration the following sets of attributes: $A = \{a_4, a_5, a_6\}$, $B = \{a_1, a_2, a_3\}$. Representing them in the form of partitions

$$P(A) = \{\{1\};\ \{2\};\ \{3\};\ \{4\};\ \{5, 7\};\ \{6, 9, 10\};\ \{8\}\}$$

$$P(B) = \{\{1\};\ \{2, 8\};\ \{3, 6, 7\};\ \{4\};\ \{5, 9\};\ \{10\}\}$$

and assuming the following partition to be the partition $\Pi_G = \{\{1, 2, 3, 4, 6, 7, 8\};\ \{5, 9, 10\}\}$, it can be easily verified that $P(A) \cdot \Pi_G \leq P_D$, where $P_D = \{\{1, 2, 7\};\ \{3, 4, 6\};\ \{5, 8\};\ \{9, 10\}\}$.

Thus the original function F can be replaced with two mapping functions G and H such that

$$F = H(a_4, a_5, a_6, G(a_1, a_2, a_3))$$

The decision tables corresponding to the G and H can be obtained from partitions $P(A)$, Π_G, and P_D. Encoding the blocks of Π_G respectively as 1 and 2, we immediately obtain the table of function G; it is presented in Table 10. The table of function H can be derived by reencoding the input vectors of F using an intermediate variable g. The table obtained in this way is shown in Table 11.

Function G can be further simplified (using a SOP or another minimizer) to a form:

$$G = \{\ \ if\ \ [{a_1}^2\ or\ ({a_2}^1\ and\ {a_3}^0)]\ \ then\ 2,\ \ else\ 1\ \ \}$$

Similarly function H can be simplified, these problems are presented in [?].

7.2 Special Restricted Classes of Decomposition.

In this section we will present various special kinds of decomposition that assume some special structures of circuits. All these decompositions, however, are close to FD and can be relatively easily added to a general-purpose FD decomposer.

7.2.1 The Research of Butler and Bender.

For more than twenty years, there has been an interest in fanout-free networks (trees), where each gate has a fanout of one. This interest has been motivated by the relative ease with which tests can be generated for such networks, and also by some technological restrictions.

In particular, Hayes [?] considered fanout-free networks of AND, OR and NOT gates; Chakrabarti and Kolp [?], Butler and Breeding [?] and Maruoka and Honda [?, ?] have considered networks of AND, OR, EXOR, and NOT gates. Kodandapani and Seth [?] have considered networks which also include the majority function. A special case of a fanout-free network, a cascade, has been discussed by Maitra [?] and Mukhopadhyay [?]. Cascades are fanout-free networks in which each gate connects to at least one net input.

One way to measure the relative merits of various classes of circuits is to compare the number of n-variable functions realized by a class. However, as shown by Butler and Bender, this can be deceptive, since the relative number of functions realized by two networks for small n may be quite different than that for a large n. Let ${F_T}^{CM}$ (n) be the number of n-variable functions realized by networks whose topology T is a cascade (C) or unrestricted fanout free (FF), and whose component modules (CM) are two-input EXORs (E), AND gates (A), OR gates (O), three-input majority gates (M) and inverters (N).

The particular networks of interest discussed by Butler and Bender are:

- unate cascades of Mukhopadhyay [?]: ${F_C}^{AON}(n)$.
- Maitra cascades [?]: ${F_C}^{AOEN}(n)$.
- fanout free #1 [?], [?] : $F_{FF}^{AON}(n)$.
- fanout free #2 [?] : $F_{FF}^{AOMN}(n)$.
- fanout free #3 [?, ?, ?, ?] : $F_{FF}^{AOEN}(n)$.
- fanout free #4 [?] : $F_{FF}^{AOEMN}(n)$.

For all but one of the networks listed (not published for ${F_C}^{AON}$ (n)) recursive relations have been derived and have been evaluated at least for $n = 7$. However, computation time and the large values involved preclude computer evaluation much beyond this.

Consider now the relative number of functions for the various networks. The following can be determined:

1. The fraction of n-variable fanout-free functions which are cascade realizable becomes arbitrarily close to 0 as n approaches infinity for two cases, i.e.

$$ {F_C}^{AON}(n) = o(F_{FF}^{AON}(n)) $$

and

$$ {F_C}^{AOEN}(n) = o(F_{FF}^{AOEN}(n)) $$

.

2.

$$F_{FF}^{AOMN}(n) = o(F_{FF}^{AOEN}(n))$$

showing that the EXOR in combination with ANDs, ORs, and NOTs produces more functions than the majority gate. This is an interesting fact in view of the fact that $F_{FF}^{EN}(n) = 2$ for all n, while $F_{FF}^{MN}(n) \rightarrow \infty$ as $n \rightarrow \infty$.

3. The number of strictly fanout-free functions $F_{FF}^{AON}(n)$ represents a vanishingly small fraction of the number of functions realized by AOEN nets, verifying a conjecture by Kodandapani and Seth [?].

4. The relative number of functions for large n can be quite different than for small n.

$$F_C{}^{AOEN}(n) = o(F_{FF}^{AON}(n))$$

and

$$F_C{}^{AOEN}(n) = o(F_{FF}^{AOMN}(n))$$

However, for small n a different situation exists. For $2 \leq n \leq 6$ the AOEN cascades realize more functions than either the AON or AOMN fanout-free networks.

The asymptotic approximations for these networks suggest that the asymptotic behavior of cascades and fanout-free networks with *general* component module sets have the same characteristics of the specific examples shown here. If this is the case, the addition of a distinct module to the module set in general enlarges the function set by an arbitrarily large amount, as the number of variables approaches the infinity.

7.2.2 Bender's and Butler's Enumeration

In 1979 Intern. Symp. on Multiple-Valued Logic Bender and Butler derived a recursion relation is derived for the number of functions realized by fanout-free networks of multi-valued gates. An asymptotic approximation to this relation shows that when the number of logic levels grows, the number of functions realized is nearly independent of the network structure and is equal to $c\ L^n$, where n is the number of inputs and c and L are parameters that depend on the number of logic levels. Fanout-free networks are shown which realize any function of the input variables.

7.2.3 Maruoka's and Honda's networks of flexible cells.

Maruoka and Honda discuss logical networks of flexible cells [?]. The flexible network is a network composed of flexible multi-valued cells that can be adjusted to perform any function. The network is assumed to have a fixed-interconnection pattern and to have a fixed-input variable assignment. The following question is answered: "What is the necessary and sufficient range of the flexibility of the cells' functions to obtain all the output functions the circuit could realize?" Furthermore, the relation between the cells that realize the same output function is derived and the number of different realizable output functions is counted.

7.2.4 Universal Modules.

Yau and Tang [?] proposed several tree-type Universal Logic Modules (ULMs) that are based on Shannon Decomposition Theorem. The disadvantage of the realization of arbitrary functions using the universal logic modules of Yau and Tang is that, in general, it results in large trees of modules.

Although better pin bounds exist for other modules [?], [?], the arbitrary n-variable functions are easy to realize by constructing trees of k-variable modules $k \leq n$. Unfortunately, the number of these modules used in such a tree depends on the order of Shannon decompositions. In [?] Yau and Tang pose the problem of minimizing the number of modules in such trees.

Cheung and Ehrich investigated chain decompositions which, when they exist, reduce the number of modules [?]. A decomposition theorem and algorithm are given in [?] for reducing the size of such trees, using what is called the "chain decomposition". The basic assumption of the authors was that good realizations are obtained by minimizing the fan-out at each stage of the decomposition. Although the paper deals with three-variable IULM3 decompositions, their theorem can be readily generalized to ULM-k for $k > 3$. The module realizes any n-variable function by successive expansions of $k-1$ variables for ULM-k when $k < n$ each residue function is again expanded.

The following theorems are used.

Lemma 7.2.1 *(Curtis).*
If $f = F(N_1(A,B),C) = G(N_2(A,C),B)$ *then* $f = K_1(A) * K_2(B) * K_3(C)$
*where * is OR, AND, or EXOR operator.*

Lemma 7.2.2 *(Curtis).*
If $f = F(N_1(A,B),C,D) = G(N_2(A,C),B,D)$ *then* $f = H(L,D)$
where $L = K_1(A) * K_2(B) * K_3(C)$ *and * is AND, OR, or EXOR operator.*

Theorem 7.2.1 *(Curtis).*
Let f be a non-degenerate n-variable function. If f has two chain ULM-3 (CULM3) realizations obtained by expanding with respect to two different pairs of variables, then the expansion with respect to either pair of variables will yield identical CULM3 structures.

The length of the CULM3 realization depends on the residue function rather than the choice of the expansion variable pairs. As a result, in determining the CULM3 realizability of the function, if there exists such a pair of variables, one does not have to look for other pairs, since the decomposition theorem indicates that all realizations will yield the same ULM3 structure. A similar proof has been established when IULMM-k are used for $3 < k < n$ [?].

The algorithm of Cheung and Ehrich is the following. A function is first tested for single variable decompositions. Then tests for various types of two-variable decompositions are applied to the function successively. The algorithm is non-exhaustive to these tests for a chain ULM3 realization. Applying this algorithm enables to determine the realizability of ULM3 chain decompositions. However, it provides no information about how to carry out the realization efficiently when there is no chain decomposition at a particular stage. There is no guarantee that an initial fan-out of the tree realization might not ultimately result in a better realization. However, there seems to be no procedure, short of exhaustion, for determining when this is the case.

7.2.5 Irredundant Tree Networks of Chakrabarti and Kolp

The paper by Chakrabarti and Kolp [?] is concerned with the synthesis of irrendundant tree networks with two-input single-output flexible cells. An algorithm is developed which tests whether a given Boolean function is tree realizable. If it is tree realizable, the best tree is generated which realizes the function. It is shown that for each realizable function there exist a non-trivial unique partition from which a best tree can be constructed. Finally the number of functions realizable by irredundant trees is determined. The best tree is the one having the minimum number of levels.

Definition 7.2.1 *(Tree-realizable function) (Chakrabarti and Kolp).*
Any two variable function is tree-realizable. A Boolean function f of n variables ($n > 2$) is said to be tree realizable iff there exists a two-block partition of f
$P = \{B_1, B_2\}$
such that
$f(X) = F(K_1(B_1), K_2(B_2))$
and K_1 and K_2 are tree-realizable.

Theorem 7.2.2 *(Chakrabarti and Kolp).*
Let P be a simple partition of f (i.e. $f(X) = F(\Psi(V), V')$. Then f is tree realizable iff F and Ψ are tree-realizable.

Theorem 7.2.3 *(Chakrabarti and Kolp).*
If f is tree-realizable, then there exists a nontrivial unique partition A of f, and A and the Boolean operation $*$ are unique.

Theorem 7.2.4 *(Chakrabarti and Kolp).*
Let f be a given tree-realizable function which is expressed by its unique form as:

$$f(X) = F_1(B_1) * F_2(B_2) * \ldots * F_m(B_m)$$

Let also $F(Y) = y_1 * y_2 * \ldots * y_m$ and $d(y_i) = l(F_i)$. Then $d(F(Y)) = l(f)$.

7.3 Decomposition of Multiple-Valued Logic Functions.

The decomposition of Multiple-Valued Functions has been discussed by:

- Thelliez, [?] (only for ternary logic).
- Walliuzzaman and Vranesic, [?].
- Fang and Wojcik [?].
- Luba, Mochocki and Rybnik [?, ?, ?]
- Abugharrbieh and Lee, [?, ?]

Similarly to the binary case, there are two types of decomposition for multiple-valued functions:

1. AC-like decomposition,
2. Dietmeyer-like decomposition.

The first approach is to extend the AC binary approach. This approach is represented by Walliuzzaman and Vranesic, [?], and Thelliez, [?], for ternary logic. Especially the partition-based approach of Luba allows him to create an algorithm that is a straightforward adaptation of his former binary algorithms.

The second approach is represented by Fang and Wojcik, [?]. Their paper is a very good reference for some non-FD approach to Boolean Decomposition. We will call this approach a *compositional approach to multiple-valued logic synthesis.*

Both the above approaches have their characteristic advantages and disadvantages. The first approach may require generating of all partitions. Next, function representation for each partition must be examined to determine if it possesses the specific decomposition property that is being considered. Since only a very small number of functions will have a specific property [?], if the function does not have the desired property, the analysis does not yield a design. Hence the designer will have to consider alternative decompositions or select some other approaches to design the function. Finally, in multiple-valued systems, the entire classical decomposition approach is considerably more complex than for binary systems because of the associated combinatorial explosion (this is not necessarily true in the case of Luba's decomposition, but there are no comparisons of binary versus multiple-valued decomposers yet).

The disadvantage of the second method is its inherent strong bias. Its advantage may be that it will appear easy to convert it to the Abductive Networks developed by ABTECH Corporation [?, ?].

7.3.1 The Compositional Approach of Fang and Wojcik

The method of Fang and Wojcik follows the approach of Dietmeyer [?], but it is different in several key respects.

In the paper by Feng and Wojcik, the complex problem of identifying whether or not the function is decomposable is greatly simplified, because the decomposition is considered only in terms of available components. The components may consist of a set of building blocks or a single universal logic module. All functions can be decomposed using this approach. In the worst case, a canonical form of the function is generated, leading to a tree structure to implement the function.

The basic idea is to develop a systematic procedure which emphasizes function decomposition with modularity. They apply a top-down approach which seeks to identify common sub-functions. The decomposition of the target function into sub-functions is considered in terms of available components. The components must constitute a functionally complete set. The set can be a collection of building blocks, or a single universal building block such as an m-variable multiplexer, or an n-variable PLA. The technique of Dietmeyer was also applied to multiple-valued logic by Antoni Michalski in his 1980 Ph.D. in Poland [?].

The technique of Dietmeyer is based on using building blocks that are determined *after* the search of partition matrices for a classic decomposition property. Since the decomposition may not exist, the set of building blocks will not be found.

While in AC approach a function is rewritten in terms of new variables (functions), Wojcik and Feng do not attempt to rewrite the function in terms of sub-functions which are next used as new variables in the logic equation of the function. Rather, they identify which of the available components used to implement the sub-functions can be interconnected to represent the target function.

In AC decomposition a binary function:

$$f(x_1, x_2, x_3, x_4) = x_1x_3 + x_2x_3 + x_1x_4 + x_2x_4$$

is rewritten to:

$$f(x_1, x_2, x_3, x_4) = F(A(x_1, x_2), B(x_3, x_4))$$

where:

$$F(A, B) = A \ B$$

$$A(x_1, x_2) = x_1 + x_2$$

$$B(x_3, x_4) = x_3 \ + \ x_4$$

In the approach of Feng and Wojcik for this example, if a module composed of a two-input OR gate is used as the available component, then the function may be rewritten as:

$$F(x_3, x_4) = x_3G(x_1, x_2) + x_4G(x_1, x_2) = G(x_1, x_2)(x_3 + x_4)$$

where:

$$G(x_1, x_2) = x_1 + x_2$$

is a sub-function.

The approach uses a predefined collection of modules that are used to design an arbitrary function. The approach is applicable to any radix and indeed can be applied to binary functions as well.

Feng and Wojcik developed the synthesis method along with several techniques to minimize the number of components used in the design. The objective of the minimization was to reduce the number of building blocks used from a predefined component collection, but not to minimize the number of basic logic components from which the building blocks are composed. These techniques do not guarantee an optimal solution, namely the least number of the building blocks. Furthermore, optimality is a function of many criteria, such as the number of modules, the complexity of module interconnections, the number of logic levels in the design, etc.

Further work is needed to study the interaction among the optimization techniques and their ability to produce an optimal design. It is also necessary to develop other techniques that will contribute to achieving the optimization that is desired. Although some work on these kinds of compositional methods has been done in the past, recently there have been no papers applying such ideas in frameworks of modern representations and search methods.

It is this our opinion, that the approach of Feng and Wojcik would give very good results if implemented with BDDs or KDDs (This kind of techniques have not been tried with DDs yet).

7.3.2 Tokmen's Approach to Disjoint Decomposability of Multi-valued functions by Spectral Methods

.

The decomposition methods based on spectral analysis were investigated by Tokmen [?] and more recently by Poswig [?]. It was shown that if the Walsh spectrum of a function is evaluated using an FFT-like algorithm, the execution time of a program can be reduced significantly [?]. Other good methods can be found in [?, ?, ?]. However as the number of variables increases, the execution time still grows and it is not sure what are the practical limits of using these methods to decomposition.

The method of Tokmen, [?], is defined as follows. Given is an m-valued function. The method makes use of the given function $f(X)$ and of the functions (gates) with which it is desired to realize the decomposition. The method involves the solution to a set of m^k simultaneous equations each with m unknowns for the m-valued case. The decomposition is possible if each and all sets of simultaneous equations have unique solutions. The method may be applied to functions of arbitrary size and is implementation independent.

Further, if it is found that the decomposition using the desired gates is possible, then the detection and proof of this decomposition fully defines the spectrum and hence functional relationships of the next level (reminder) function h. The detection of such simple disjoint decomposability of a given multi-valued function may be considered as the first stage in the design of combinational multi-valued logic circuits. Tokmen refers to a paper by Fricke [?], and a book by Karpovsky [?].

7.3.3 Waliuzzaman's Approach to Multi-Valued Functions Decomposition.

The approach is presented in [?]. The standard Curtis-like approach of investigating all possible decompositions to find the best one becomes prohibitively time consuming for systems with $m > 4$. The break-count procedure presented by Walliuzzamam allows for easy selection of decompositions which lead to simple implementation.

Theorem 7.3.1 *(Walliuzzaman).*
A switching function $f(X)$; $X = x_1, x_2, \ldots, x_n$ has a simple disjunctive decomposition given by $F(Y, h(Z))$, $Y = y_1, y_2, \ldots, y_{n-s}$, $Z = z+1, z_2, \ldots, z_s$ if and only if its partition matrix $M(f, Y|Z)$ has column multiplicity $\mu \leq m$.

The proof is obvious, since in such case each column type is encoded with one logic value of the output of the predecessor function.

Theorem 7.3.2 *(Walliuzzaman).*
If $f(X) = f(A, B, C)$ can be expressed as $f(A, B, C) = F(h(A, B), C) = G(N(A), B, C)$ then $f(A, B, C) = F(K(N(A), B), C)$ where $K(N, B)$ is a uniquely determinable function of N and B for which $K(N(A), B) = h(A, B)$.

Theorem 7.3.3 *(Walliuzzaman).*
If $f(A, B, C) = F(h(A), B, C) = G(N(B), A, C)$ then $f(A, B, C) = H(h(A), N(B), C)$ where $H(h, N, C)$ is a uniquely determinable function of h, N and C.

These theorems are just modifications of well-known Ashenhurst and Curtis theorems to mv logic. Walliuzzaman's paper refers to papers by Arango, Santos, and Chacur [?], and Merril [?]. It seems to me that presently this paper is only of historic value since more powerful and general methods have been described in later years.

7.3.4 Multiple-Valued Decomposition by Abugharrbieh and Lee

The objective of Abugharrbieh's and Lee's research [?, ?] was to generalize the Shen et al [?, ?, ?] binary function decomposition algorithm to m-valued functions with $m > 2$. They obtained the necessary condition for the decomposability for m-valued functions and used it in generating candidate bound sets. A fast method for testing the necessary condition using partial partition tables (PP tables) was also created, whereby the decomposability of the function can be determined for a randomly chosen function in $(Nm)^3$ time.

A fast algorithm for disjoint decomposition that does not require exponential time, proposed by Shen and Kellar [?, ?, ?], applies Jacobian operator to switching functions to test if they satisfy the necessary condition. This operator requires switching function to be given as an algebraic expression in a canonical form using AND, EXOR, and NOT operators. Shen and Kellar showed that a randomly chosen function of n variables has a very small probability of being decomposable and the time needed to conduct such a decomposition test is $O(n^3)$.

Abugharrbieh and Lee developed an algorithm that is capable of testing m-valued functions for $m > 2$. They generalized the necessary condition developed by Shen et al for the multi-valued logic. Instead of evaluating the Jacobian operator, a different method is used to test the necessary condition for decomposability. Furthermore the method operates on functions which may be given either by truth tables or algebraic expressions. They showed that for n-variable m-valued randomly chosen functions the execution time is $O(nm)^3$ which is considerably less than in other approaches to mv decomposition: [?, ?]. In the case where a function has a single disjunctive decomposition, the execution time is $O(n^3m^n)$.

Theorem 7.3.4 *(Abugharrbieh and Lee).*
An m-valued function $f(X)$ has a simple disjunctive decomposition with a bound set X_1 and a free set X_2 if and only if the column multiplicity μ (the number of distinct columns) of $T(X_1 : X_2)$ is less than or equal to m.

Theorem 7.3.4 is the same as the Theorem 1 of Walliuzzaman.

Theorem 7.3.5 *(Abugharrbieh and Lee).*
Let $f(X) = g(h(X_1), X_2)$ be a decomposable m-valued function with n variables. If $\{x_i, x_j\}$ is included in X_1 and x_k is a member of X_2, $i \neq j$, then all the PP tables $T(x_i, x_j : x_k, e)$, $0 \leq e \leq m^{n-3} - 1$ have column multiplicity $\mu \leq m$.

This indicates a necessary condition for a function $f(X)$ to be decomposable.

Theorem 7.3.6 *(Abugharrbieh and Lee).*
Let $f(X)$ be an m-valued function with n variables. If x_1, x_2 and x_k are members of X, $i \neq j \neq k$ and at least one PP table $T(x_o, x_j : x_k, e)$, where $0 \leq e \leq m^{n-3} - 1$, has column multiplicity $\mu > m$ then if $f(X) = g(h(X_1), X_2)$ such that x_i, x_j is included in X_1 then x_k is a member of X_1.

Theorem 7.3.7 *(Abugharrbieh and Lee).*
Let $f(X)$ be an m-valued function with n variables. If B is the set of all AMB sets then any CB set can be obtained by a finite union of elements of B.

In lieu of the Jacobian operator, partial tables were used in testing of the necessary condition for decomposability. The method is easy to implement.

7.3.5 Luba's et al Approach to Multiple-Valued Decomposition

Some new papers by Luba et al are on decomposing multiple-valued functions. They are related to Machine Learning, [?, ?]. Their approach to decomposition is similar to both classical Curtis and Walliuzzaman approaches. The difference is in that they apply the partition theory. The sufficient condition in this approach leads to the calculation of minimal cover of the maximum compatible classes.

Luba does not write in his papers how he selects the "bound set". In general, he does not devote much attention to the algorithmic search aspects. Some new results are obtained by using the method described by Selvaraj in his Ph.D. thesis [?]. The new idea is to combine the parallel and serial decompositions in a single program. In an unpublished paper, Piotr Sapiecha in Poland is dealing with "Simulated Annealing" algorithm to calculate the so-called reducts. In Luba's understanding they are nothing but the minimal sets of non-vacuous variables that Luba uses in his approach.

The really innovative asset of Luba's approach is his partition-based representation that allows for relatively easy determination of several properties. Most, if not all, of his theorems are restatements of known theorems to his notation, but the method seems to have a large potential for incompletely specified functions. It is still not clear which, if any, of his theorems and methods are really new. For instance, his products of partitions have close relation to cofactors, his covering of the compatibility graph corresponds to the coloring of the incompatibility graph in TRADE or to the set covering of Pedram's approach, and so on.

7.4 Luba's et al Approach to Multiple-Valued Decomposition.

Some new papers by Luba et al are on decomposing multiple-valued functions and are connected to "machine learning" [?][?]. On one hand, their approach to decomposition is similar to both classical Curtis and Walliuzzaman approaches. The difference is that they apply the partition theory. The sufficient condition in this approach leads to the calculation of minimal cover of the maximum compatible classes.

Luba does not write in his papers how he selects the "bound set". In general, he does not devote much attention to the algoritmic search aspects. Some new results are obtained by using the method described by Selvaraj in his Ph. D. Thesis [?]. The new idea is to combine the parallel and serial decompositions in a single program. In an unpublished paper, Piotr Sapiecha in Poland is dealing with "Simulated Annealing" algorithm to calculate the so-called reducts. In Luba's understanding they are nothing but the minimal sets of variables that Luba uses in his approach.

Based on my undertstanding, the really innovative asset of Luba's approach is his partition-based representation that allows for relatively easy determination of several properties. Most, if not all, of his theorems are restatements of known theorems to his notation, but the method seems to have a large potential for incompletely specified functions. It is still not clear which, if any, of his theorems and methods are really new. For instance, his products of partitions are the cofactors, his covering of the compatibility graph is the coloring of the incompatibility graph in TRADE or set covering of Pedram's approach, and so on.

7.5 Decomposition of Information Systems.

Many problems related to decision making are of the following nature: Given a decision table - find all minimal decision algorithms associated with the table. This problem can also be seen as learning from examples, or, as its is often called, decision rules generation from examples - Grzymala-Busse, [?] [?] [?]. These problems have been investigated from a number points of view: one of them is whether the whole set of attributes is always necessary to define a given partition of an universe and the other concerns the simplification of decision tables, namely the reduction of condition attributes in a decision table.

The concept of information system decomposition can be treated as an aid in analyzing the existing system [?] or as a base for design a new one [?, ?]. Quality of decomposition depends mainly on analyst's experience. Some authors try to provide a formal approach to decomposition [?]. Decompositional approach similar in essence to one proposed earlier by Armstrong [?] was presented by Luba, Mochocki and Rybnik in 1993. Avoiding data redundancy is an important problem in the implementation of of information systems. The problem is usually overcome by devising simpler rules and removing redundant rules, as well as minimizing the number of attributes.

The importance of decomposition is emphasized by Courtois [?]: "Decomposition has long been recognized as a powerful tool for the analysis of large and complex systems".

¿From the technical point of view, the decomposition phase is exactly the same as in the multiple-valued decomposition with no constraint on the number of values of the output from the predecessor block, so it will be not further discussed.

7.6 Decomposition of Fuzzy Logic Functions.

Decomposition of fuzzy logic functions was first discused by Abraham Kandel in [?]. Later he found errors in his paper which, after removing, maked his method completely impractical. Therefore, in the next paper by Francioni and Kandel, [?], they propose a much improved method for disjoint decomposition.

The disadvantages of the new method include:

1. lack of intuitive explanation how their decomposition rules have been created makes it difficult to improve the algorithm,

2. the algorithm is strictly tabular, and no computer implementation exists.

3. the algorithm is quite complex and requires another program to minimize fuzzy functions.

The non-disjoint decomposition of fuzzy logic functions remains an open problem. Francioni and Kandel mention interesting examples of applications of fuzzy logic decomposition, including medical diagnosis, hardware realization of fuzzy logic circuits, pattern recognition, and document retrieval systems.

7.7 The Decomposition of Continuous Functions.

The only paper about generalizing the AC decomposition to continuous functions is one by Ross et al. [?]. However, the method of orthogonal decision diagrams for continuous veriables, developed by Pierzchala, Perkowski and Grygiel, should be also easily adaptable to AC decomposition, because of the close link of orthogonal and AC decompositions [?].

Perkowski, Pierzchala and Grygiel developed methods to realize Galois Field operations for small fields in analog/mixed *Field Programmable Analog Array - FPAA*. Paper [?] gives the example of realization of GF(4) operations. This way, Galois Field type of expansions can be realized in hardware in an analogous way to Galois(2) expansions, i.e. AND/EXOR canonical forms. The regular array structure of the FPAA allows also to realize structures based on Orthogonal Expansions that are not Galois Expansions. The structure of the corresponding circuit resembles a PLA as well, and has columns corresponding to orthogonal functions over GF(2^n), where the functions are realized by cascades of Galois Addition and Galois Multiplication gates, and pass gates. Each column is Galois-multiplied by a constant, and next these values are Galois-added in rows. Thus, the column is a generalization of an AND-term of a PLA, and a row is a generalization of an OR-term. (Interestingly, a very similar array structure allows also to realize fuzzy logic circuits). Further, the trellis expansions of functions can be also generalized to continuous logic, where obviously variable repetition is required in a general case. Several other generalizations to other algebras, including fuzzy logic, as well as their realizations in regular array structures have been also described in another not yet published paper. This constitutes a fruitful area of our current research.

In another paper, [?], they observe that the general decision diagrams, universal cells, regular structures and decompositions are so general, that they apply not only to binary logic, but also to multiple-valued logic, Galois Fields and other algebras (fields and some rings). They are also applicable to continuous, i.e. analog logic, for instance fuzzy logic. This helped to create the concept of universal analog-digital "tissue" which would allow to realize a very wide category of logics and other formal systems in hardware. A practical realization of these efforts is the concept of Field Programmable Analog (Mixed) Array which is for analog (mixed) circuits what FPGA is for binary circuits [?].

Perkowski and Hong [?] and Luba and Lasocki [?] presented a case of decomposing a function with various numbers of values in each variable. The recent research of the author is on the expansion of these methods to include also continuous and fuzzy variables. It is done using generalized decision diagrams that have nodes corresponding to binary, multiple-valued, continuous and fuzzy variables.

7.8 Decomposition of Information Systems.

Many problems related to decision making are of the following nature: Given a decision table - find all minimal decision algorithms associated with the table. This problem can also be seen as learning from examples, or, as it is often called, decision rules generation from examples - Grzymala-Busse, [?] [?] [?]. These problems have been investigated from a number of points of view: one of them is whether the whole set of attributes is always necessary to define a given partition of an universe, and the other concerns the simplification of decision tables, namely the reduction of condition attributes in a decision table.

The concept of information system decomposition can be treated as an aid in analyzing the existing system [?] or as a base for designing a new one [?, ?]. Quality of decomposition depends mainly on the analyst's experience. Some authors try to provide a formal approach to decomposition [?]. Decomposition approach similar in essence to one proposed earlier by Armstrong [?] was presented by Luba, Mochocki and Rybnik in 1993. Avoiding data redundancy is an important problem in the implementation of information systems. The problem is usually overcome by devising simpler rules and removing redundant rules, as well as minimizing the number of attributes.

The importance of decomposition in general system design is emphasized by Courtois [?]: "Decomposition has long been recognized as a powerful tool for the analysis of large and complex systems".

From the technical point of view, the decomposition phase is exactly the same as in the multiple-valued decomposition with no constraint on the number of values of the output from the predecessor block, so it will be not further discussed.

7.9 The Generalized Decomposition.

Although Pattern Theory (PT) group interests are only in combinational logic decomposition, from the mathematical point of view the decomposition of Finite State Machines is a close issue. Historically, the decomposition methods have been applied to state machine design for many years, starting with works of Hartmanis and Stearns in 1960's and 1970's. Hartmanis and Stearns created a very general and computationally convenient mathematical apparatus, called *"Partition Algebra"*, and proved powerful theorems about parallel, cascade and other decompositions of finite state machines. It was, however, observed that these methods are inefficient for practical state machines, and rarely a good decomposition exists for a random machine. In general, the criticism and the reception of these methods in industry were initially similar to that of the reception of AC decomposition in case of combinational circuits. However, there has been recently an increased interest in state machine decomposition methods in companies such as Philips, Siemens, and Synopsys and in top research universities. Moreover, there exists a feeling that new decompositions and techniques can be developed that will be useful for decomposing both switching functions and state machines [?, ?].

The partition algebra has been used by Luba et al for combinational decomposition, leading to a totally new approach and practical results of a very good quality. Recently, a new generalized theory of decomposition of both state machines and combinational logic has been developed by Jozwiak et al. [?, ?]. It is yet uncertain to us, whether the theory of Jozwiak does indeed bring new practical decompositions in case of combinational circuits (other than a common presentation framework with sequential circuits). However, it definitely allows to create much more general decompositions of sequential machines by considering more decomposition types, including some quite complex new types that are able to perform decomposition ("splitting") of internal states.

It is possible to extend the current model of Pattern Theory Group to state machines. This would be the simplest and most natural extension of the current model, because the state machines include the combinational logic, and there is no popular model of computability model that would be located in the hierarchy between the two. If ever the Pattern Theory group would decide to make a state machine generation their most basic model of algorithm Machine Learning then the classical model of state machine decomposition would be is a good candidate. Especially the "Generalized Decompositions" of Jozwiak are good candidates because of their similarity to the current decomposition model investigated by the Pattern Theory group, and with respect to the use of the generalization of Luba's partition-based decomposition model at PSU.

7.10 Multi-Valued Functional Decomposition as a Machine Learning Method

In the past few years, several authors have presented methods of using functional decomposition as applied to machine learning. These authors explore the ideas of functional decomposition, but left the concepts of machine learning to the papers that they reference. In general, they never fully explain why a logic synthesis method should be applied to machine learning. This paper explores and presents the basic concepts of machine learning, and how some concepts match nicely with multi-valued logic synthesis, while others pose great difficulties. The main reason for using multi-valued synthesis is that many problems are naturally multi-valued (i.e., values taken from a discrete set). Thus, mapping the problem directly to a multi-valued set of inputs and outputs is much more natural than encoding the problem into a binary form. The paper also shows that any multi-valued logic synthesis method could be applied to the machine learning problem. But, this paper focuses on multi-valued functional decomposition because of its generality of minimizing a given data set.

7.11 Introduction

This paper explores functional decomposition as a machine learning method. Functional decomposition of binary functions was first proposed by Ashenhurst in the 1950s [?] as a method of Boolean multi-level logic minimization. While this process has been known for many years, up until recently it could not be utilized because of the large complexity computation procedures that are required. Thus, very little was written about the practical aspects of decomposition from the 1950s until the late 1980s. In the cases that something was written, the papers were structured around decomposing small functions to binary circuits in various technologies and normally didn't present many extensions to the original Ashenhurst decomposition method. Lately, though, decomposition has become quite popular because of its application to logic synthesis for Field Programmable Gate Arrays (FPGAs) [?]. Also, note the early work in the field of multi-valued functional decomposition [?, ?].

The relationship between machine learning and logic synthesis is based on the similarities between realizing circuits with low complexity (smaller size) and the natural assumption of Occam's Razor [?]. In logic synthesis, the result of synthesis is a circuit that is designed with respect to a minimal number of gates, levels, inputs, literals, or product terms. In the case of machine learning, the goal is to effectively generalize concepts from the data [?]. This is done by using Occam's Razor, which is a principle in machine learning that states that if several theories explain all the facts then the simplest theory is the best. Thus, logic synthesis and machine learning are not all that independent, in that they are both attempting to reduce the description complexity of a given system specified by its partial behavior samples.

The central idea in machine learning is to learn or gain information from a given data set. This can be a very difficult task since practical databases are usually very large. To help handle this difficulty, the assumptions of Occam's Razor and introducing explicit domain knowledge are used to reduce the search space. Machine learning is not an exact methodology, but is an attempt of *learning* based on heuristics and probabilities. This attempt of *learning* is becoming more and more important as computers provide relatively inexpensive means to collect and store data. Thus, traditional methods, such as manual data analysis, are insufficient to fully evaluate a given data set. Instead, a new method called Knowledge Discovery in Databases (KDD) is being used to analyze data using analytical tools from statistics, pattern recognition, and artificial intelligence.

It is unfortunate that concepts that are already known in the logic synthesis community had to be re-learned by the machine learning community. In fact, several papers were written on rule generation systems that are based on circuit design techniques [?, ?]. The bridge between the two communities, called *pattern theory* [?], is an approach to finding algorithms for both the logic synthesis community and the machine learning community. The idea is to analyze the behavior of these algorithms using Computational Learning Theory (COLT). COLT ascertains how much information is needed for the algorithm to be within some probabilistic measure of correctness. The findings so far show that *pattern* *theory* algorithms, based on logic synthesis, are very promising in finding solutions for machine learning. However, the computational complexity of these algorithms still remains an outstanding problem. This is where heuristics in multi-valued functional decomposition or other multi-valued logic synthesis techniques are needed.

This paper is structured as follows: in Section two, a background of machine learning is given. Section three presents some of the difficulties in mapping machine learning problems into functional decomposition. Section four explains how decomposition can be used for finding solutions in machine learning. This is done by exploring previous Boolean decomposition results that were compared to machine learning approaches and other logic synthesis results on machine learning problems. While noting that decomposition of multi-valued decomposition should do much better than the Boolean decomposition techniques. In Section five, conclusions are raised about the use of functional decomposition in machine learning.

7.12 Machine Learning Background

Within the structure of this paper, the term *machine learning* is used to describe the process of applying expert knowledge of a particular problem domain to solve problems within the domain. This is also known as an *expert system* or a computer program that automates the process of *learning* or gaining information from a given data set. What makes up an expert system, is a process called *inductive learning* [?] which uses empirical evidence, in the form of examples, to derive rules for the given data. Systems that use *automated inductive inference* (in the machine learning paradigm) use rules to classify data. Rules are operations on variables, for instance, the average function, addition, subtraction, etc. Finding rules on variables is a process of discovering patterns and relationships that exist between variable combinations [?]. In logic synthesis terminology this means that *learning* is done by evaluating all the known outputs (or *cares*) of a given function to help in the determination of all the *don't cares* (*don't knows* in the machine learning paradigm) in the output of the given function. Thus, by treating *examples* as *cares* and considering functions with many *don't cares*, machine learning problem can be directly converted to a logic synthesis problem. While several papers have been written on this topic [?, ?, ?, ?, ?, ?, ?], one of the first papers to deal with this topic was [?].

Given that induction is a method of extrapolating samples of a function, the problem becomes very complex without making some reasonable assumptions. The assumption that is generally made is the one of a low complexity, as in Occam's Razor. This assumption has been applied and theoretically analyzed in several papers, including [?, ?]. Plus, Occam-based learning theory has been extended to actual implementations. One such implementation is Abductory Inference Mechanism (AIM[1]) [?] which creates networks of polynomials. The resulting network is the network with the least complexity based on the size of a given polynomial network and how well the network maps to the given data. Another implementation, C4.5 [?], is based on decision trees, where the resulting network is determined by comparing the sizes of decisions trees and selecting the least complex, or the smallest decision tree. The decision tree implementation used in C4.5 can be thought of as a network of multi-valued multiplexors, where the least complex solution is the smallest network of these gates. Note, that both of these implementations can handle data with noise, continuous variables, and conflicting data. In Section 7.13, aspects of *learning* and their corresponding types of data are explained in more detail, but note that even though functional decomposition was initially created for discrete Boolean values, it has been used in solving problems with all of these types of data [?, ?, ?, ?].

One drawback for both C4.5 and AIM is their use of a narrow complexity measure to determine the effectiveness of a solution. For instance, neither program finds the parity function to have low complexity, even though parity is highly patterned and can easily be computed [?]. One solution for determining complexity was found by [?] to convert the problem to pattern theory, where the complexity is determined as the problem of finding a pattern. Given that the complexity is based on combinational

[1]Trademark of AbTech Corporation

methods, there are many possible measures of complexity that could be used. Each of these complexities is based on the number of basis functions (gates) necessary to represent that function. The complexity measure selected is a measure proposed by [?]. This method, called Decomposed Function Cardinality (DFC) is the sum of the cardinalities of the component functions in a combinational representation, when that sum has been minimized. Any function can be a component, but its cost goes up with its *cardinality*. The *cardinality* of an n-input, m-output binary function is $2^n * m$. In the Boolean world, an AND-gate will have the same *cardinality* as an XOR-gate.

So why use functional decomposition? Decomposition has the advantage that it is not based on a set of operators or gates (in contrast to all other logic synthesis methods). This is especially notable in the case of multi-valued logic, where the number of operators can increase as the value of the logic grows. Thus, decomposition is not constrained by a technology or a pre-selected single theory. This has wide applications in both logic synthesis and machine learning. After decomposition the sub-blocks are small, so highly sophisticated and/or specialized logic synthesis/rule derivation methods can be used [?]. Since [?], Epstein, Liu, Miller, Muzio [?] and others have found that functional decomposition for a function class can be used to discover optimal operations for the decomposition theory of functions in the class.

7.13 Machine Learning to Logic Synthesis

This section gives an overview of mapping a machine learning problem to a multi-valued functional decomposition problem. While there are many aspects that are the same between the two methodologies, this section will also point out the differences between the two. Also presented are some problems that even the machine learning community has not fully implemented.

7.13.1 Machine Learning Examples

In general, machine learning concepts or examples are used to find a generalization of the original data set. An example or concept is thought of as a *Care Term* by the logic synthesis but in machine learning it has a different meaning. For example, in the case of a cancer database, doctors may take data about a person's age, sex, height, weight, x-rays, etc., and if the person has cancer or not. Now, the number of input data values (variables) that are taken can be enormous while the number of patients that are evaluated in the database can be quite small. For instance, if a doctor asks 20 questions about sex, height, etc., but evaluates only 500 patients, then the number of examples is quite small compared to the number of possible outcomes given the number of questions. In this example, if each question was a yes or no question, then there could possibly be 2^{20} $\approx$1-million combinations given different values of the inputs, while only 500 patients were evaluated. Thus, this is only a small representation of the possible values that exist, which explains the name *example*. Each *example* is only an small piece or example of the actual function.

Mapping from machine learning to logic synthesis can then be done by mapping the examples to care terms. But, the number of *don't cares* in the functions that will be evaluated with logic synthesis is quite large. In fact, it is thought that most machine learning problems have more than 99.9% *don't cares*. Thus, any logic synthesis method that could possibly be used must be able to handle a large number of *don't cares*.

7.13.2 *Don't Cares* in the Inputs Variables

The possibility exists that data collected could have a *don't care* in the input. This is different than a *don't care* in the input for a logic synthesis problem where the *don't care* is treated as covering all possible values for that variable. In the machine learning world, this means that you have a *don't know*. The *don't know* cannot be handled the same as in logic synthesis since it only represents one data value not all possible values. For instance, say one of the questions asked to patients was their age. But, the person did not want to state their age, and instead of guessing at an approximate age, a *don't know* was placed in the category. In terms of logic synthesis, this would mean that the person was all the ages

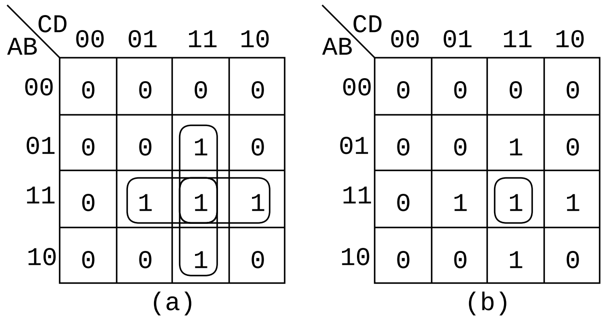


Figure 7.2: *Don't knows* in the input variables

between 0 and 150. While in machine learning this just means that person has only one age between 0 and 150.

Another example of this is shown if Figure 7.2, using 2-valued Karnaugh maps with *don't cares*. Both Karnaugh maps are given the same set of input cubes (a dash indicates a *don't know*, or a *don't care*): $-BCD$, $A-CD$, $AB-D$, and $ABC-$. In Figure 7.2a, the logic synthesis reduction method is shown where all the minterms are placed in essential prime implicants. The result is a minimum circuit with 4 product-terms and 1 sum-term. In Figure 7.2b, the machine learning result is a little bit different. For example, the cube: $-BCD$, is thought to have a *don't know* in it that specifies that either the minterm $ABCD$ exists or the minterm $\overline{A}BCD$ exists. The minimum solution to this problem is $ABCD$.

This problem of input *don't knows* can also be applied using probabilities. Figures 7.3a through 7.3d show the possible combinations when given the cubes: $A-$ and $-B$. Since there are four combinations, then the probability of finding any of the Karnaugh maps in Figures 7.3a-7.3d, is $\frac{1}{4}$. Thus, Figure 7.3e shows the probabilities of finding any given minterm actually being a one. By assuming any probability value in the interval[0,1], it can be calculated for every minterm to be true or false. For example, if the probability value is greater than 0.5 then only the minterm AB is true, resulting in the Figure 7.3f. In the case of a probability value greater than 0.4, the function becomes as shown in Figure 7.3g.

One can see this topic is of great importance to the machine learning community. And, to the knowledge of the authors, there are no published papers on the subject that fully explore this topic. The paper [?] uses relations as an attempt at this problem. But the relations are only used to identify overlapping cubes. For example, given the cubes: $A-C$ and $AB-$, the relation recognizes that the two cubes overlap at ABC. Thus, if $A-C$ has the output value 5, and $AB-$ has the output value 3 then the cube ABC has the output value of 5 or 3.

7.13.3 Retaining MVL synthesis techniques

Since Boolean logic synthesis techniques have been in practice since the 1950s, they are well known, and fairly well understood, compared to multi-valued logic synthesis. That does not mean that they are the future of computing. It also does not mean that machine learning problems should be encoded to Boolean expressions before analysis. The encoding to Boolean expressions may destroy the data either by adding or removing components inherent in the multi-valued-ness of the function or data.

One important aspect of machine learning and expert systems that is not highly publicized is that an expert system will usually perform best by the way the data is given to it by a user. This implies that if the user has some understanding of the data being inputed into the expert system, the user may control the way the expert system handles the data. When the data is encoded to Boolean expressions, the user may loose all intuition of what the data means and how variables can interact with each other.

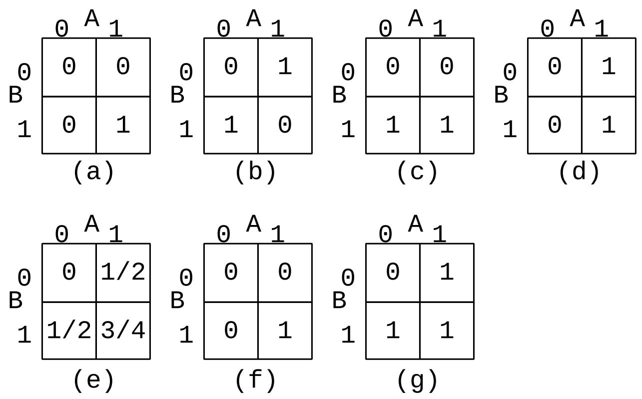


Figure 7.3: Example of using probabilities to calculate a function from cubes $A-$ and $-B$

This is why it is important to leave machine learning problems in their original form, and why it is important to use multi-valued logic synthesis techniques, instead of Boolean techniques.

7.13.4 Noise in the Data

Another concept that comes up is the idea of noise. Noise in machine learning could be caused by many different things. An example of noise could be when the doctors wrote down the data, they could have written the data down in the wrong place, written down the wrong information, etc. This can cause problems in both the machine learning community and the logic synthesis community, in that, how do you detect that the example is actually noisy, and not the correct value. While [?] talks about this concept of stripping out noise, this is a theoretical point of view and may not be the best way of handling noisy data.

There are two types of noise: noise that is in the input, and noise that is generated on the output. Both types of noise cause different problems in determining that noise is actually in the data set. The removal of noise can be done when a certain threshold is hit (parameter set by the user). For instance given two sets with 1000 elements in each, if both sets are equal except for one pairing, then should this pairing be thought of as noise? A threshold could then be used to state that if two sets are, for example, 90% equal, then the two sets are equal. The use of a threshold should be variable since there may not be any noise on some data sets, while other data sets may have a large amount of error. Of course, setting a low threshold may allow sets to be combined that should not be combined.

Another problem that exists is what happens when the algorithm finds noisy data and tries to combine the data. What should the noisy values become; one of the values, an average of the two values, a *don't know*, etc. Handling noise is notably a very deep theoretical issue in machine learning, and needs to be looked into further.

7.13.5 Continuous Data

Machine learning problems also contain continuous data or data that is in the form of real numbers, not natural numbers. This type of data, so far, has shown itself to be very difficult for multi-valued functional decomposition. There have been several attempts at this, though. In [?, ?], the authors map real-valued variables onto natural numbers by discretization. In [?, ?], the data is actually left in the form of real values which causes problems in the decomposition process when trying to combine two values. A way around this is to combine values that are some δ distance apart from another. For example, 2.46 and 2.55 could be combined if δ=0.1. Now the problem is determining the value of the

Table 7.4: Comparing a machine learning approach with logic synthesis approaches [?]

two combined elements: possibly average, one of the two values, a *don't know*, etc. This seems to be a topic that needs further research. But, the initial approach of the authors would be to use another machine learning tool (AIM, C4.5, neural networks) to take care of data in this form as a pre-processing for decomposition, because these tools are based on experience gained in years of experimentation.

This section has shown how data from the machine learning community can be mapped into multi-valued logic synthesis. While some of the data poses certain difficulties for the logic minimization techniques, they are also difficult for the applications in machine learning. Thus, there are a lot of areas that need to be explored on the boundary between multi-valued logic synthesis and machine learning.

7.14 Comparison

Given that there are already Occam-based software programs available in the machine learning community, why would the multi-valued logic synthesis community want to get involved? This section may not fully answer this, but it does show a comparison between the two methods. Among these existing approaches, the logic synthesis approach does much better than the machine learning approach. In our opinion, this is just a beginning, considering that multi-valued logic synthesis should perform much better than their Boolean counterparts. That is, machine learning problems often have multi-valued outputs and inputs. This is the reason for choosing multi-valued functional decomposition; it does not have to put constraints on the operations available in multi-valued systems or can give appropriate operations to achieve optimality in a decomposition theory.

The first example of how well Boolean logic synthesis compares to machine learning concepts is found in [?]. This paper compares C4.5 (the top program in the machine learning community), functional decomposition (using DFC as the measure of complexity) and Espresso, a logic minimization tool distributed by the University of California at Berkeley [?] (an industry standard tool of accepted quality). The data found in [?] is shown in Table 7.4. Each of the 2-valued functions in Table 7.4, using operations AND, OR, EXOR, and COMPLEMENT, are assumed to have eight input variables. Thus, for each of the functions tested, there are several variables that are *vacuous*, i.e., variables that have no effect on the output of function. If these functions were completely specified, determining if the variables are *vacuous* is a trivial issue. But, when the functions are highly unspecified it becomes very difficult to determine which variables are *vacuous* and which variables actually have an effect on the output of the function. In Table 7.4, each method was given a random set of data to train on, ranging from 25 to 250 out of the 256 (2^8) possible cases. Once the method was trained, the entire 256 cases were tested and the number of differences were recorded as errors. This is shown in the section Average Error, where the values can range from 0 to 100, with 0 error being the best. The data found under the heading Number of samples, shows the number of random samples necessary before the learning method obtains a concepts, such that, the error in the result is 0.0. By looking at the averages at the bottom of the tables, notice that decomposition outperforms C4.5 and Espresso in both Average Error and Number of Samples.

In [?], the concept of machine learning was mapped to a Boolean logic synthesis tool that minimizes EXOR of products expressions. This paper showed that this type of synthesis performs as well as C4.5 on functions with a small and medium percentage of *don't cares* but did not do very well with functions with a large number of *don't cares*. Their conclusion is that EXOR synthesis modified for strongly unspecified functions is a viable method that could complement existing machine learning techniques.

Another result of decomposition being applied to machine learning is shown in [?]. This thesis shows many results of their multi-valued decomposition tool, HINT, compared to C4.5 and other machine learning techniques, such as neural networks. In most cases, HINT learned significantly better than C4.5. The interesting achievement here is that HINT was compared with other machine learning programs on data with continuous data, and data that contains noise. The continuous data were discretized in the method introduced by Kononenko et al. [?].

The results from previous researchers show that decomposition and logic synthesis methods can be used in the area of machine learning. In fact, this section showed results that logic synthesis tools outperformed the famous machine learning program C4.5.

7.15 Conclusion

This section showed the basis for using multi-valued logic synthesis in the field of machine learning. While the problems inherent in machine learning are not exactly the same as the problems found in multi-valued logic synthesis there is still a correlation between the two areas. The important aspect of this is that bridging the gap between the two areas could prove beneficial to both communities. There are many theories to explore, especially in the context of manipulating machine learning data.

The reason that multi-valued functional decomposition was the main focus of this paper is because of its generalized behavior. That is, functional decomposition based on heuristics can be used to find a set of rules or used to reduce complexity issues within some data set. Again, because of the complexities inherent in functional decomposition, it is possible that other multi-valued logic synthesis techniques may demonstrate better efficiency. Since there has been little done in the comparison of multi-valued logic synthesis tools, multi-valued decomposition, and machine learning approaches, there is a lot of space for new research.

While there are many issues to work on in this new field of functional decomposition, the results are good enough to compete with the existing "top" machine learning programs. In fact, the data obtained during the research for this paper shows that logic synthesis techniques obtained better results than C4.5 which is a standard machine learning tool.

As the size of databases grows, it is obvious that the need for expert systems and machine learning is going to be necessary to process all the data. This is a tough problem, but by using multi-valued functional decomposition hopefully the problem will be reduced. It is difficult to predict the future, but the possibility exists that machine learning based tools may be used in circuit design as well.

7.16 An Error Reducing Approach to Machine Learning Using Multi-Valued Functional Decomposition

This paper considers minimization of incompletely specified multi-valued functions using functional decomposition. While functional decomposition was originally created for the minimization of logic circuits, this paper uses the decomposition process for both machine learning and logic synthesis of multi-valued functions. As it turns out, the minimization of logic circuits can be used in the concept of "learning" in machine learning, by reducing the complexity of a given data set. A main difference is that machine learning problems normally have a large number of output don't cares. Thus, the decomposition technique presented in this paper is focused on functions with a large number of don't cares. There have been several papers that have discussed the topic of using multi-valued functional decomposition for functions with a large number of don't cares. The novelty brought with this paper is that the proposed method is structured to reduce the resulting "error" of the functional decomposer, where "error" is a measure of how well a machine learning algorithm approximates the actual, or true function.

7.17 Introduction

In the 1950s, Ashenhurst presented a Boolean multi-level logic minimization method called functional decomposition [?]. From the 1950s until the late 1980s, not much was done in the area of creating efficient programs for functional decomposition primarily because of the large computational procedures that are required in the decomposition process. In the late 1980s, functional decomposition was re-introduced as an application to the synthesis of Field Programmable Gate Arrays (FPGAs) [?].

Since then decomposition has been applied to many aspects of Boolean and multi-valued logic synthesis. Several papers written on decomposition of multi-valued networks have shown an interest in using their decomposers in the field of machine learning [?, ?, ?, ?]. Each of these papers had a slight twist to the decomposition process and the type of data that they were able to handle. The motivation was always the same: given some measure of complexity, find the solution with the minimum complexity. This is analogous to reducing the size, structure, number of gates, or number of terms in a logic circuit. The first ideas to apply decomposition to machine learning appeared in [?] and the first successful machine learning programs based on logic synthesis were reported in [?].

The main goal of using a multi-valued functional decomposer is to reduce the complexity (smaller circuit size, simpler description) of a given problem. This is analogous to the assumption in machine learning that the simplest formula derived from the data is the best formula. This is thought of as a natural constraint given that our own brains think in terms of reduced complexity. For example, a child might be given a "connect the dot problem", and from dot to dot, the child will draw straight lines. A straight line in this instance is the simplest way of connecting the dots, and very rarely will the child use lines of large complexities (e.g., sinusoids, circles, crossing over the same line).

This paper explores multi-valued functional decomposition as a logic synthesis method and as a machine learning tool. The decomposition algorithms for these two methods are similar, but there are some significant differences. The biggest difference is that most circuit-related multi-valued logic problems are nearly completely specified, while functions in machine learning tend to be 99.9% unspecified in their respective learning domains.

The paper is structured as follows. In Section two, a background of machine learning is given. In Section three, multi-valued decomposition is discussed. In Section four, the concept of building an *error reducing* multi-valued decomposer is presented. In Section five, an algorithm is presented that is based on the concepts of low complexity. In Section six, results of the algorithm are given. Section seven concludes the paper by remarking on the results of the algorithm.

7.18 Machine Learning

The similarities between machine learning and logic synthesis are based on minimizing circuit designs for size, number of gates, or product terms, and is equivalent to the machine learning concept of reducing complexity. In machine learning the idea is to find patterns in the data, such that the data can be partitioned into smaller concepts (data blocks), which correspond to the sub-blocks of the decomposition. The principle of using decomposition in machine learning is to reduce a given function specified by a set of care minterms (*samples* or *examples*) to a composition of smaller functions (concepts). The result is a set of expressions that describe suitable *intermediate concepts*. Each of these *intermediate concepts* can then be decomposed further, leading to expressions that form a more comprehensible description of the *learned* concepts. The advantage of using decomposition to obtain useful *intermediate concepts* is that it leads to a result specified as a hierarchy of compositions. This produces the description of the original function as a hierarchy of sub-functions and variables, which leads to *learning* that is faster, involves smaller error and gives better explanation of the *learned* concepts.

The central idea in machine learning is to *learn* or gain information from a given data set. This is done by using the care terms (*examples*) in the data to set the values that are not given in the data (output *don't cares*). The objective is to set the *don't cares* to values that follow the same patterns as the care terms. In terms of logic synthesis, this setting of *don't cares* is done by creating a network of multi-valued input, multi-valued output blocks by decomposing the original function into multi-leveled blocks. A machine learning algorithm is evaluated on it effectiveness of learning by how it reduces the *error* of the resulting network. *Error* is how well the algorithm in question set *don't care* terms to care terms. This evaluation is done by selecting a *training set*, which is a random sampling of the original *test* function. The result of learning the *training set* is then compared to the *test* function. If the resulting expression has a high error then it does not approximate the *test* function well and, thus, is not a useful way of describing the function.

This can be a very difficult task given that practical databases that machine learning algorithms are used on are usually very large. To help handle this difficulty, the assumption of Occam's Razor and the introduction of explicit domain knowledge are used to reduce the search space [?]. Thus, machine learning is not an exact methodology for solving certain problems, but is an attempt at *learning* based on heuristics and probabilities. The hope is that these *learning* algorithms perform better than using chance, i.e, a resulting *error* that is less than 50%.

To help solve some of the problems in machine learning, methods have been proposed to involve logic synthesis. This includes creating complexity measures that are general enough to be used in minimizing both circuits and concepts.

7.19 Multi-Valued Functional Decomposition

This section presents the basic principles of the decomposition of multi-valued functions. Note that the decomposition of multi-valued functions is just an extension of the decomposition of Boolean functions. See [?, ?, ?] for more information on Boolean based decomposition.

Definition 7.19.1 Given a multiple-valued variable x_i, the set of values that x_i may assume is $C_j = \{0, 1, \ldots, c_{j-1}\}$. Then an n-input, m-output, multi-valued function is defined as the mapping: $f(x_0, x_1, \ldots, x_{n-1}) = C_0 \times C_1 \times \ldots \times C_{n-1} \rightarrow D_0 \times D_1 \times \ldots \times D_{m-1}$, where $D_j = \{0, 1, \ldots, (p-1), -\}$, with p equal to the number of output values represented in the p-valued logic and "-" represents a *don't care* value.

Definition 7.19.2 A function $f(x_0, x_1, \ldots, x_{n-1})$ is decomposable under *bound set* $\{x_0, \ldots, x_{i-1}\}$ and *free set* $\{x_{i-l}, \ldots, x_{n-1}\}$, $0 < i < n, 0 \leq l$ if and only if f can be represented as the composite function $F(G_0(x_0, \ldots, x_{i-1}), \ldots, G_{j-1}(x_0, \ldots, x_{i-1}), x_{i-l}, \ldots, x_{n-1})$, where $0 < j < i - l$. If l equals 0 then f is said to be *disjunctively decomposable*, otherwise, it is known as *non-disjunctively decomposable* [?, ?].

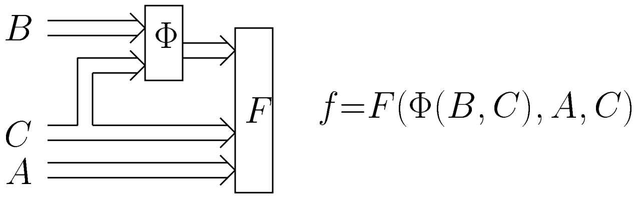


Figure 7.4: Basic Decomposition

The principle idea of the decomposition using the notations from Definition 7.19.2 is shown in Figure 7.4. Given that a function is decomposable under a given bound set, the function may be separated into two newly created functions. This is known as a simple decomposition because one decomposition was done on the function. To fully decompose the function, an iterative process is used. First the function is decomposed, then the sub-functions that are created from the initial decomposition are decomposed. This iterative process continues until a given function cannot be decomposed further under given complexity measures.

Note, that for an n-input function, the number of simple *disjunctive decompositions* is 2^n, while the number of simple *non-disjunctive decompositions* is 3^n. Thus, evaluating all possible partitions is an NP-complete problem when trying to fully decompose a function. Because of the exponential size of simple *non-disjunctive decomposition*, most heuristic methods only attempt at finding *disjunctive decompositions*. This reduces the search complexities of finding partitions in decomposition, but this still does not mean that the problem of finding *non-disjunctive decompositions* becomes trivial, since it is still an exponential search problem.

Definition 7.19.3 Given a k-valued, completely specified function f, with a *bound set* B, and *free set* A, then for the partition $A|B$, a *partition matrix representation of* f is defined as a rectangular array, where the *columns* correspond to the variables in the *bound set*, and the *rows* correspond to the variables in the *free set*.

Using Definition 7.19.2 and 7.19.3 the following is found:
1. The array has $k^{|B|}$ columns and $k^{|A|}$ rows.

2. Given a k-valued function f, with a bound set B, then the corresponding partition matrix has l distinct columns, where l is known as the *Column Multiplicity* of a partition.

3. The *Column Multiplicity* for the function can be reduced if the function is incompletely specified, by finding columns that are *compatible* and combining the two columns by setting *don't care* values. By compatible, for every row, the possible output value-sets of the first column (a number or a *don't care*) intersect the non-empty sets of the corresponding output value-sets of the second column.

4. Thus, to represent f as a composite function in the form: $f = F(G_0(), \dots, G_{j-1}(), x_i, \dots, x_{n-1})$, where each G function has inputs $(x_0, \dots, x_{i-1})$ then $j = \lceil log_k l \rceil$, G functions are needed.

The concept of decomposition is fairly general, and because of this generality it can be applied to any type of logic structures or elements, and takes into account various constraints and requirements.

Of course, this might be a difficult task, but none-the-less, it is possible. Further, multi-valued network decomposition has the advantage that it is not based on any pre-specified set of operators or gates.

7.20 Reducing Error Through Partition Selection

The biggest difference between logic synthesis and machine learning is the difference in the number of care terms in logic synthesis versus the number of care terms in machine learning problems. There is another factor to this that must be noted. The *don't cares* in machine learning problems should actually be thought of as *don't knows*. By *don't knows* we mean that the value is not known and that caution must be used in setting a *don't know* to a value.

With the concept of a *don't know* in mind, this section presents a methodology for selecting partitions for machine learning based decompositions. The formulation for finding partitions and evaluating is based on *don't knows* and the complexity issues found in trying to decompose functions with many variables.

Observation 7.20.1 From the Section 3, given that the function is completely specified, the only way that two columns can be compatible is if for every row, the output value of the first column is equal to the output value of the second column (in the same row). Obviously, if the function is p-valued, completely specified, and each value is equally likely, then for every row the probability of matching one column to another is $\frac{1}{p}$.

From Observation 7.20.1, for a column with B rows, the probability of finding a pair of *compatible* columns is $\left(\frac{1}{p}\right)^B = \frac{1}{p^B}$. The result is that the more rows in a column, the less likely it is to find a *compatibility* between two columns. The problem with Observation 7.20.1 is that it doesn't take *don't cares* into consideration and the fact that machine learning data sets are 99% unspecified.

Observation 7.20.2 Assume the probability of having a *don't care* is $P(X) = 0.99$, then the probability of having any other value in the function is $P(care) = \frac{1-0.99}{p} = \frac{0.01}{p}$, given that the function is p-valued and each value is equally likely. The only way to have an incompatibility between two columns occurs only when, for a given row, the first column's value is some value and the second column's value is a value that is not equal to the first column's value. Thus, the probability of having an incompatibility between two columns is $0.01\frac{p-1}{p}$.

It results from this observation that evaluating small bound sets (small number of columns and a large number of rows in a partition table) should result in a larger probability of incompatible columns. The premise of this paper is that by increasing the probability of incompatible columns we have decreased the amount of *error* in the resulting network. This premise is true because the possibility of combining a *don't care* with an incorrect care term is reduced. This is based on the probability of combining two columns is much smaller, and thus, the probability of combining a *don't care* with a value is also smaller.

For example, given a 3-valued function with 100 inputs (each input is also 3-valued), if the bound set has 2 variables then there are 98^3 rows in partition table. Whereas, if the bound set has 50-variables then there are 50^3 rows in the partition table. From Observation 7.20.2, the probability of having incompatible columns is $P(incomp)$ and the probability of having compatible columns is $1 - P(incomp)$. In comparison, the probability of finding compatible columns with 2-variables, $(1 - P(incomp))^{98^3}$, is much smaller than the probability of finding compatible comlumns with 50-variables, $(1 - P(incomp))^{50^3}$. The results show that it is highly unlikely that two columns will be found to be compatible in the case of 2-variable bound sets. The only way that they will be compatible is if there is some pattern in the data that allows the combination of the two columns. As for the case of the 50-variable bound set, the probability seems like a small enough , but note that there are 50^3 columns for this partition.

Another complexity issue that comes up is determining column compatibility. One of the most promoted methods for finding column compatibilities with many *don't cares* is the use of graph coloring. Since graph coloring is an NP-complete problem (exponential in complexity) itself, then the more columns then the longer the computation time. Thus, it is reasonable to assume that small bound sets with a small number of columns will have a much faster run time than medium-sized (half the variables in the bound set, the other half in the free set) bound sets. In the case of medium or large bound sets, the probability of two columns being incompatible is much smaller than in the case of small bound sets. Hence, the number of compatible columns can be very large. In fact, because there are so many possible coloring in the graph coloring algorithm, selecting compatible columns becomes a random process. Thus, the combination of columns is done randomly with no basis on probabilities. This also shows that having bound sets that are medium or large will result in greater *error*.

What does this mean in terms of complexity. From the Section 3, we know that for an n-input function there are 2^n partitions that can be evaluated or $O(2^n)$. If we restrict the size of partitions, then there are $\binom{n}{2} = \frac{n \times (n-2)}{2}$ partitions that can be evaluated or $O(n^2)$. In the case of medium sized bound sets there are $\binom{n}{n/2} \approx \frac{2^n}{10}$ partitions that can be evaluated or $O(2^n)$. For instance, given that there are 100 input variables, then there are $\approx 1 \times 10^{29}$, 50-variable bound sets. This is a very large search space, much too large to go through all of the partitions with 50 bound set variables. If the bound set is small, then search becomes much smaller. Given 100 input variables there are $\binom{100}{2} = 4950$, 2 variable bound sets. Therefore, based on complexity, it is much easier to evaluate all the small bound sets.

In fact, in terms of logic synthesis (recall the small percent of *don't cares*), the possibility of finding a good decomposition with small bound sets would probably not work. But, from the analysis above, the algorithm presented here is based on the premise that small bound sets are the best way to decompose machine learning problems.

7.21 Algorithm Implementation

This section presents an algorithm that uses multi-valued functional decomposition as a machine learning algorithm. The section is broken down into subsections which describe some of the important aspects of the presented algorithm.

7.21.1 Machine Learning Complexity Measure

The complexity measure selected is a measure proposed in [?]. For more on complexity measures see [?, ?, ?, ?]. This complexity measure, called Decomposed Functional Cardinality (DFC) is the sum of the cardinalities of the component functions in a combinational representation, when the sum has been minimized. Any function is allowed as a component, but its cost goes up with its *cardinality*. The *cardinality* of an n-input, m-output binary function is $2^n * m$. Thus, any n-input gate has the same cardinality as any other n-input gate, regardless of the functionality of that function.

Definition 7.21.1 A function $f(x_0, x_1, \ldots, x_{n-1})$ with k-valued logic and m-outputs, has *cardinality* $k^n * m$. DFC is defined as the sum of the function cardinality of each decomposed partial block. A non-decomposable function is defined as a function that has a DFC$> k^n * m$.

7.21.2 Data Structure of the Decomposer

The representation of a function can sometimes make or break the algorithms that are applied to it. The importance of data structure has been shown in [?] in which a Binary Decision Diagram (BDD) [?, ?, ?] based decomposition proved very efficient in the decomposition of logic functions for FPGAs. The use of Multi-value Decision Diagrams (MDD) was also shown to be effective in the decomposition of multi-valued functions with many *don't cares* [?]. The problem with the MDD package is that it does not allow variables of different valued-ness. Also, it was found that when the function implemented in the MDD has many values, the MDD package acted more like a decision tree than a BDD, that is, there was very little compression within the nodes of the MDD.

Taking the ideas from [?] of mapping a MDD onto a BDD, we created a new MDD package that is built on top of our existing BDD package[2]. By mapping a MDD onto a BDD package, the MDD now has the ability to have variables of different sizes.

7.21.3 Inessential Variables

One of the most important procedures in a machine learning algorithm is the removal of non-essential(*vacuous*) variables from the function. There are actually three different classes of variables as defined in [?]. A variable is either *essential*, *inessential* or *vacuous*. *Essential* and *vacuous* variables are easily defined as variables that either have an impact on the output, or do not have an impact on the output. An *inessential* is a variable that by setting *don't cares* in a certain way, the variable either becomes *essential* or *vacuous*. Because of the structure of the MDD, if a variable is vacuous it is no longer included in the database, so they are automatically removed from the function. For each *inessential* variable, *don't cares* are forced to make the variable vacuous.

To determine which variables can be made vacuous a compatibility graph algorithm is used. Each node in the graph represents an *inessential* variable, and each vertex between two nodes designates that both variables can be forced vacuous at the same time. From this graph a maximum clique algorithm is used to determine the largest clique of *inessential* variables that can be forced to become *vacuous*.

7.21.4 The Decomposition Algorithm

The algorithm is broken down into the following steps:
1. Run graph algorithm to find all *inessential* variable cliques. Evaluate the resulting maximal cliques by calling the decomposer on each of the resulting functions.

2. Decomposer algorithms:
2a. Determine if the current function's DFC is larger than the best DFC found so far. If it is, then return (this is a method of pruning, such that not all functions are decomposed).
2b. Evaluate all two-variable bound sets.
2c. Determine which partitions could lead to a decomposition. This is done by evaluating the column multiplicity of each of the partitions and selecting those with higher ranks. The partitions with the smallest column multiplicity are ranked higher than those with higher column multiplicities.
2d. From the list of possible partitions from 2b, systematically decompose each function f into sub-blocks F and Φ. Recursively call step 2 with F and Φ, separately.

3. Done. Return the function with the best DFC.

7.22 Results

The results obtained demonstrate how the algorithm can reduce the complexity of a given problem while keeping the *error* low. The testing of the algorithm was done on completely specified functions with eight input variables. There are several reasons for using these small functions: the functions are completely specified which makes the comparison of error exact, the smallest DFC of the functions are known, so analysis of the decomposers heuristics can be performed, and many training samples may be run.

The training sets are made from randomly selecting care terms from a given function. The selection is done in steps of 8 in the range [8,248]. At each step, 20 different data sets were found by randomly

[2] See: http://www.ee.pdx.edu/~cfiles/bdd/

selecting P care terms, where P is equal to the current step. The reason for selecting 20 different data sets is to get a good distribution so the output could be fully analyzed.

The test functions that were used come from the Wright Patterson's Pattern Theory Group set of benchmarks[3]. Of course, this is just a measure to see how well a function can handle data which is highly unspecified. But, notice that the minimum number of care terms evaluated is 8 care terms and 248 *don't cares*, which means that the function is 96.875% incompletely specified.

The results are shown in the Figures 7.5, 7.6 and 7.7. For each function the following are shown in the figures: the *error* of the training output to the original KDD function, the DFC that was found on the training function, and the time that it took for the algorithm to find a solution. The title of each graph designates what is actually being displayed on the $Y - axis$ (*error*, DFC, Time), while the $X - axis$ is the number of training samples. For each function, there is some error when the number of training samples is quite small, but as the number of training samples gets larger, the amount of error goes to zero. Also note that the algorithm usually finds the correct function once it finds the DFC of the original function. That is, as the number of training samples grows the algorithm is able to find the original function, and the DFC of the original function. The last thing to note is the time to a solution. We found it quite interesting that the time to find a solution was much longer for small training samples. The reason for this is the number of possibilities that exist with a large number of *don't knows*.

Each of the Figures 7.5, 7.6 and 7.7 was selected out of the 54 possible functions in the benchmark set. These three functions, in general, show the execution of the decomposer. The first two functions are small examples of Knowledge Discovery in Databases, while the third is a character recognition function.

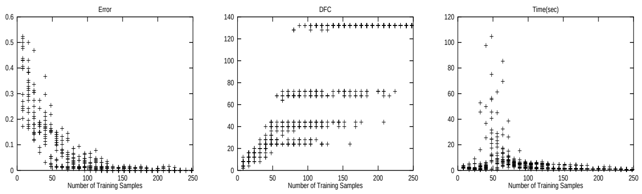


Figure 7.5: Finding the error, DFC, and time of the decomposer on the function KDD5

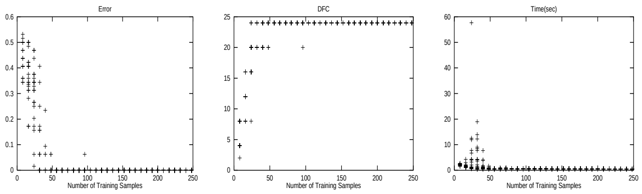


Figure 7.6: Finding the error, DFC, and time of the decomposer on the function KDD10

[3]benchmarks available at http://www.ee.pdx.edu/polo/function/

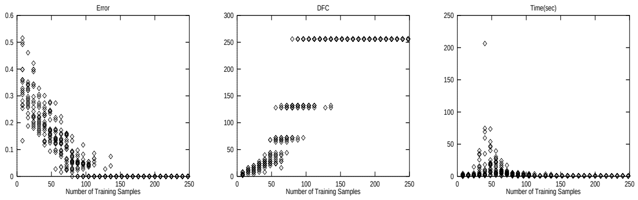


Figure 7.7: Finding the error, DFC, and time of the decomposer on the function CH52f4

7.23 Conclusion

This paper presented a new method for the decomposition of highly unspecified multi-valued output functions. The main goal of this paper was to present a new method of decomposition that is focused on reducing the error in a solution. Finding error is actually a test of an algorithm to see how well it will behave on real-world data. In this paper, the proposed algorithm was tested on a KDD based benchmark test set. From this data analysis, the program did very well on reducing the error between the original function and the solution.

Part III

PROBLEMS IN DECOMPOSITION

Chapter 8

Variable Partitioning

Chapter 9

Graph Creation

Chapter 10

Column Compatibility

Chapter 11

Vacuous Variables

11.1 Dealing with Vacuous Variables

The advantage of removing vacuous variables during decomposition is reducing the size of the problem for the decomposition. This method has been used by Luba et al. Luba presents how to deal with many vacuous variables - he uses and efficient Boolean method adopted from Espresso.

In general the experience from the literature about vacuous variables is mixed or negative. Definitely a naive approach can give bad results, on the other hand some reduction may be very useful. This problem was not much analyzed and should be one of research tasks. Definitely, the totally vacuous variables should be always removed.

11.1.1 Creating Partitions

Algorithm to select partitions in Flash starts with one variable in output block and n-1 variables in input block, this is the same as building BDD from the top. This gives a table with two rows and 2 to n-1 columns.

Another possibility is to start from n-1 variables (or n-2) variables in output block which would correspond to building BDD from the bottom. This approach corresponds to BDD building procedures by Haomin Wu and Ron Kohavi. The method by Craig Files is in essence very similar to the algorithm of Haomin Wu, but the second method is able to deal better with don't cares.

An algorithm presented by Perkowski, Dysko and Falkowski used a learning method with a modifiable weighted evaluation function a'la Samuel Checkers Program. This method can be perhaps further improved by having signature tables instead of linear weighted functions. Such approach was suggested by Armstrong [?].

Another technique close to variable ordering discussed in literature is called K-map folding. The question is "How to combine folding the K-maps with looking for partitions?" Folding has been used especially for incompletely specified functions, so these techniques might offer some new insights to variables' partitioning and ordering.

There is a large body of algorithms and theory of partitioning large and very large graphs which has been developed in the context of VLSI layout - Kernighan, Richie, Fiducia, Mattheyes, Szepieniec. The open question is this - "Can one use such algorithms for Boolean Decomposition?"

Another idea is to add VARIOUS biases to partitioning: symmetry, partial symmetry, other properties used for function classification, for instance as in the research of Malgorzata Marek-Sadowska [?]. Because in decomposition you need heuristics in any case, why don't we use various biases towards symmetry or other types of functions for which synthesis may be easier.

Chapter 12

Encoding

12.1 Approaches to the Input-Output Encoding Problem in Boolean Decomposition

So far, very little has been published in the literature on the encoding problem that occurs in functional (Curtis, Roth-Karp) decomposition. In particular, the encoding techniques used to optimize the decomposition of functions for this problem have tried to optimize only the predecessor or only the successor logic, but not both concurrently.

This report presents several approaches to the input-output "Symbol Encoding Problem" that occurs in the decomposition of Boolean functions. We show new methods for encoding, that attempt to optimize concurrently both the predecessor logic and the successor logic.

A new combined approach to the column minimization and encoding in functional decomposition is also presented, which means, these two problems that have been solved separately until now, are being solved together. We call this a "concurrent approach" to column minimization and column encoding, to distinguish it from "symbol encoding". In this approach the column reduction process is combined in a single search process with the column encoding to find the really minimum decomposition. We call it then also the "column encoding problem". This is similar to the problem of concurrent state minimization and assignment of internal states in a Finite State Machine.

The problem of symbol encoding occurs in many problems related to the design of logic circuits:

1. Assignment of internal states.
2. Assignment of input states.
3. Assignment of output states.
4. Assignment of codes in microprogrammed units [?, ?, ?].
5. Assignment of values to symbolic variables in generalized Sasao decomposition [?].
6. Assignment of codes to the output of encoder in the decomposed realization of FSMs, realized with the two-dimensional minimization [?].
7. Assignment of signals to symbolic variables while designing iterative circuits.
8. Decomposition of logic functions [?, ?, ?, ?].

Encoding is the assignment of binary vectors to different values of some symbolic (i.e. multiple-valued) variable. This variable may be an input state variable, output state variable, or internal state variable. Below is an example of an encoding for a symbolic input to an Arithmetic Logic Unit (ALU). The ALU has two inputs for the operands, one input for the opcode (operator), and an output for the result. The issue here is to encode the opcodes in such a way that the logic is minimized. An example of encoding for this ALU is shown in Fig. 12.1.

Why do we care about Encoding? It has been determined that the selection of the encoding (the mapping to the binary vectors) makes a difference in the minimization or decomposition of the function in question. For instance, the encoding shown above in Fig. 12.1 for the states could be changed to the following encoding from Fig. 12.2.

Notice that the codes have been changed. These codes or other similar ones could yield a simpler function implementation (i.e. fewer logic gates or a smaller value of DFC, see [?, ?] for the definition of DFC). This is the problem that we are trying to solve - "how to come up with an optimal encoding of the symbolic values".

SYMBOLIC	BINARY ENCODING
add	000
sub	001
mul	100
div	101

Figure 12.1: Representations for Symbolic and Binary opcodes.

SYMBOLIC	BINARY
add	100
sub	011
mul	000
div	001

Figure 12.2: Representations for Symbolic and Binary opcodes.

What type of encoding are we interested in here? For this project, we are interested in the *output-input encoding problem.* This problem is slightly unusual in that it is much like the classical *internal state encoding (or, state assignment problem* (i.e. one of input-output encoding), but reversed. To fully understand this problem we will have to explain what is meant by the output-input encoding problem.

In decomposition, we are trying to reduce the complexity of an implementation of a logic function F by dividing it into two logic blocks: block G, called the *predecessor block*, and H, called the *successor block*. In order to accomplish this, the input variables to the function block F (see Fig. 12.3) must be divided into a *free set* of H and a *bound set* of G.

The free set of variables are those that are unmodified and sent directly to the new logic block H. The bound set is the group of variables which will be passed through the logic block G in order to reduce the size and complexity of the implementation. The logic block G has outputs v_i which serve as *intermediate variables* (see Fig. 12.4). Since these are new variables, and we don't really care what values these intermediate variables take on, we need, therefore, to find out what bit vectors these variables should be assigned with. This is our encoding problem. One way of formulating the encoding problem is by imposing certain constraints to be satisfied. We must find a way of simultaneously satisfying the output

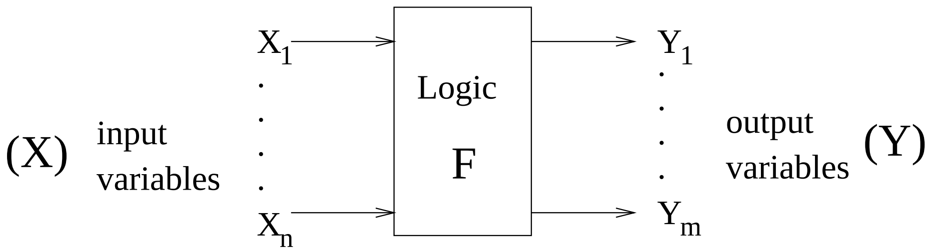


Figure 12.3: Logic function before input variable partitioning.

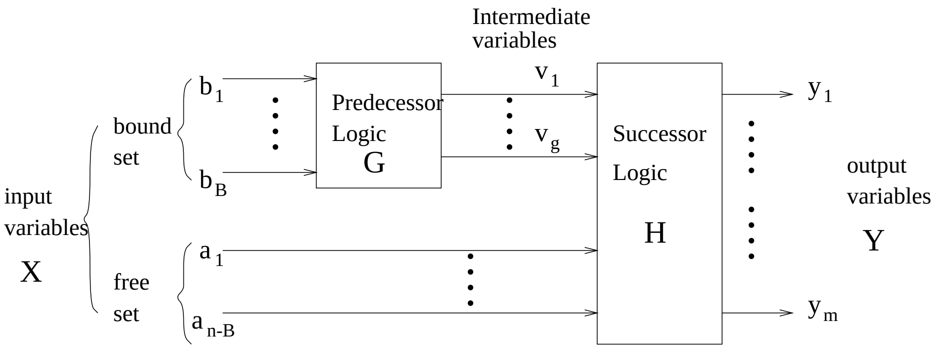


Figure 12.4: Logic function after partitioning of input variables.

and input encoding constraints to optimize both G and H.

It is noted that optimizing both the predecessor and successor logic will undoubtedly make one or both of the logic blocks less optimized. But this is the price for taking the simplification of both logic blocks into account. Our goal is to optimize the total complexity.

Figure 12.5 illustrates the process of the decomposition and the data generated in this process. Original logic function F from Figure 12.3a is decomposed to a predecessor block G and the successor block H from Figure 12.3b. In these figures, $b_1, ..., b_B$ are *bound variables*, $a_1, ..., a_{n-B}$ are *free variables*, and $y_1, ..., y_m$ are *output variables*.

The notation for sets of variables is:

X - the set of input variables,

A - the set of free variables,

B - the set of bound variables,

Y - the set of output variables,

V - the set of intermediate variables.

If sets A and B are disjoint, we will say that the decomposition is disjoint. $\mathbf{S} = \{S_1, ..., S_\mu\}$ is a *set of symbols* between the predecessor and successor blocks. The number of symbols is equal to the multiplicity index μ for given Bound Set $\{b_1, ..., b_B\}$. The symbols can be also treated as the values of a multi-valued variable S. After encoding, these symbols are encoded with *intermediate variables* $v_1, ..., v_g$ (Figure 12.4c). In case that a minimum number of intermediate variables (partitions) is taken, it holds: $g = log_2(\mu)$. The functions realized by these blocks can be represented by tables or array of cubes. We will illustrate them by Karnaugh maps for dydactic reasons. Figure 12.5b (right part) presents the "initial table" with free variables as rows and bound variables as columns. Columns correspond to the cofactors of the bound set. They are denoted by C_i, $i = 1, ..., t$. In each cell of the "initial table" there is a vector of output variables ${y_1}^{\delta_2}, ..., {y_m}^{\delta_m}$, where ${y_j}^{\delta_j} \in \{0,1,X\}$ denotes a value of output variable y_j. Figure 12.5a presents the encoding process with a pair of two tables: the first table, called the "predecessor table", corresponds to the predecessor block with encoded output and the bound variables in the input. The output symbols of G become the input symbols (columns) of the table of function H, called the "successor table". Other inputs to H are the free variables that correspond to rows of the table for H. In each cell of the "output table" there is a vector of output variables ${y_1}^{\delta_1}, ..., {y_m}^{\delta_m}$, where ${y_j}^{v_j} \in \{0,1,X\}$ denotes a value of output variable y_i. The figure shows how the data from the subfunctions G and H are composed together to create the initial function F. As we see, selecting good codes for input symbols S_i will simplify the map of the predecessor function G. It will cause also the successor function H to depend in a simpler way on symbols from S and free variables. Thus, having a partition of input variables, one can evaluate its influence on the complexity of G and H simultaneously.

Fig. 12.5b illustrates Figure 12.5a on a particular example. Fig. 12.5c presents the scheme of decom-

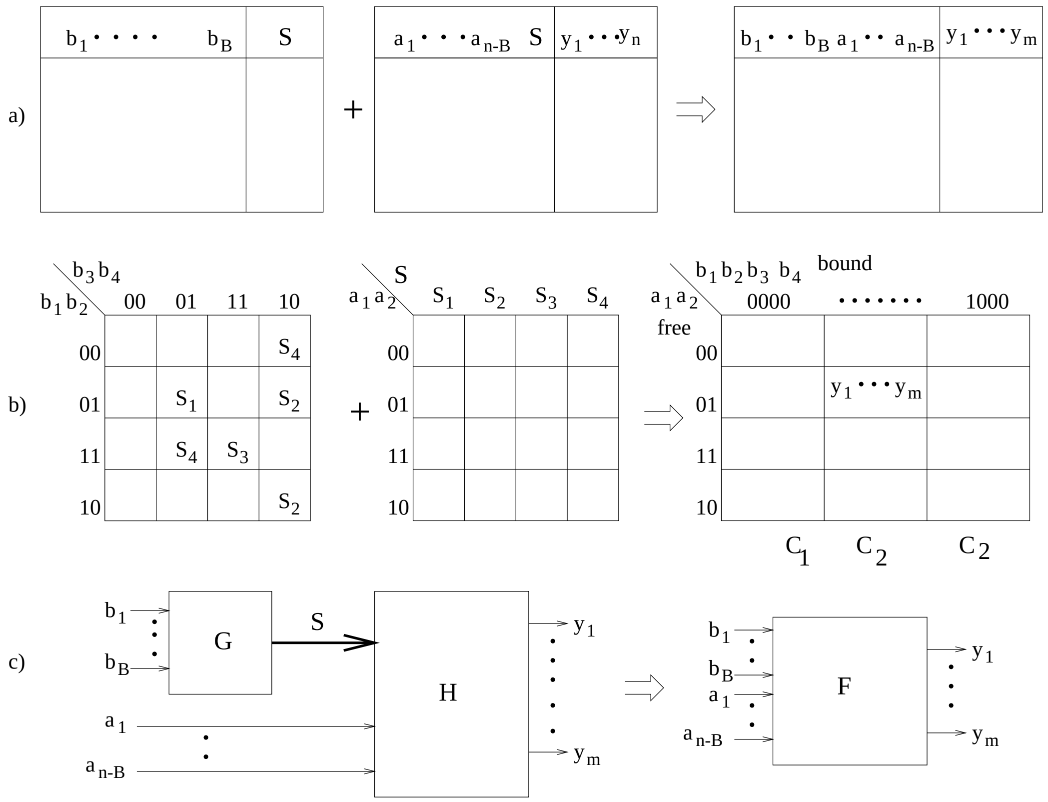


Figure 12.5: The Decomposition Process.

position into blocks corresponding to Figures 12.5a,b.

The encoding process is done to concurrently optimize the logic of both G and H. The assignment and encoding methods from the literature were created for either 2-level realization of logic, or for logic realized with algebraic factorization. Our ideal solution here would be to find an encoding method that would optimize exactly the entire set of logic blocks realized with the decomposition method. This would be, however, very difficult, so only approximate solutions will be proposed.

12.2 Related Previous Work on Logic Encoding and State Assignment for FSMs

Recently, several functional decomposers have been proposed. They include decomposers for Ashenhurst, Steinbach and Curtis (Roth-Karp) decomposition types. It is only the last type that requires the encoding process. The encoding problem is usually solved either by random coding, or by very fast, low quality encoding programs, optimized for the speed of the decomposition process only [?, ?, ?, ?, ?, ?]. Very few of the papers about decomposition discuss the encoding problem and propose efficient approaches to it.

Our functional decomposer, MULTIS, is being created for Machine Learning application, so our interest is particularly in functions with *very many don't cares.* This non-standard type of functions imposes new conditions on the encoding procedure as well. It must be observed, however, that in the decomposition process, when functions of more variables are gradually decomposed into functions of less variables, these smaller functions become more and more specified, since every column minimization process brings reduction in the percent of don't cares. The entire system should then have encoding methods for both: strongly unspecified and slightly unspecified functions.

One of our initial observations was that there is a similarity of the state machine realization problem and the functional decomposition problem. In state machine realization one first minimizes the number of internal states, which in case of incompletely specified machine reduces to the covering/closure problem. Next one finds the state assignment, that is the input-output encoding problem to simplify the excitation logic. Similarly, in functional (Curtis) decomposition one first solves the "column minimization problem" to find the smallest column multiplicity index μ, and next the output-input encoding problem should be solved for the combined compatible columns in order to simplify the logic of the predecessor block and the logic of the successor block. Concluding, many other analogies between the FSM state assignment and encoding for decomposition problems exist, and we hope that they can be used to create good decomposers. We will present some of these analogies in this report.

One of the processes traditionally used to verify and optimize hardware-realized state machines with compatible states and don't cares is *state minimization of FSM.* It was also shown by Pager [?] that the problem of determining the minimal closed table for a compiler which uses Knuth's LR(k) parsing algorithm can be reduced to that of finding the minimal closed cover for an incompletely specified flow table. Such tables have usually high percent of don't cares. We will show, that part of our problem is similar.

The problem of "state assignment of FSM" can be solved during the process of state minimization of an incompletely specified Finite State Machine. One of stages of this process is when the groups of closed and complete compatibles have been already found. As a standard, one just combines states from a compatible group to a single state, selecting randomly when there are many choices as the result of the groups' overlap. However, it can be observed that smart selection of some particular next states from a group of compatible next states can lead to better state assignments at to simpler final machine created using the closed and complete set of compatibles. This *selection problem* was first introduced by Perkowski et al [?]. His method was based on selecting those next states that simplified the state assignment. The states were selected in order to create columns or rows of the same state symbols, equal columns (rows) or some other "simple" areas of the transition table to simplify the assignment of internal states in the subsequent state assignment process. Another approach to this problem, called *mapping*, has been introduced recently by Hachtel et al [?]. This was shown to lead to a very fast minimizer that does not however lead to minimal solutions. Finally, it can be observed that the mapping problem can be reduced to that of minimizing a *Boolean relation* [?], but because such approach seemed inefficient, it has been not yet programmed.

Another idea in FSM design is that of a *concurrent state minimization and state assignment.* The idea was originally proposed for approximate design of asynchronous machines by Hallbauer [?]. It was programmed and was applicable to large machines, but its quality (evaluated on small machines) was poor. The first exact algorithm of this type for synchronous machines was presented in Lee and

Perkowski [?]. Based on partition theory and using the branch-and-bound technique, it minimized the total number of arguments in the excitation functions (each function was separately minimized). Although the program gave high-performance, logic optimized solutions, the computer implementation was limited to 8 $\times$ 8 tables. In the method from [?] the state minimization is a by-product of state assignment of a non-minimal table. The constraint of such approach is that overlapped compatibles cannot be created, which reduces the space of solutions. The state minimization was considered together with state assignment by Kannan and Sarma [?], but they just used separate programs in a sequence. The fastest dedicated method for concurrent state minimization and assignment is due to Hachtel et al [?]. which is based on a special cost function and mapping. In their method the state assignment is a by-product of state minimization in the sense that the mapping problem is solved approximately to anticipate its effects on heuristic encodeability measures. The method is therefore not exact. Another approach to this problem was recently presented by Avedillo and Huertas in [?].

Concluding, it is important to study all concurrent minimization/assignment approaches for FSMs, where both processes will not only be by-products of one another, but the entire space of FSM design would be truly searched in some efficient way. All these methods would lead by analogy to the development of better encoding procedures for functional decomposers as well.

12.3 Basic Definitions

It is assumed that the reader is familiar with the basic terminology from [?].

Definition 12.3.1 *A group of (multi-output) columns* $(C_i, ..., C_j)$ *of function* F *is called the* **compatible group** *(CG) (a* **compatible**, *for short), if and only if, under every row* r *of the Karnaugh map* $(1 \leq r \leq na)$ *the entries* $r, i, ..., r, j$ *corresponding to all the input columns from this group* $(C_i, ..., C_j)$, *are consistent bit by bit. (Symbol* 0 *is consistant with symbol* 0 *and with '-', symbol* 1 *is consistant with symbol* 1 *and with '-').*

Based on Definition 12.3.1 we test the column compatibility conditions. Two input columns (cofactors of bond set) are compatible under the condition that every two corresponding outputs must be either the same or one or both of two bits of outputs are '-'s. This rule must be applied to every two corresponding bits of these two tested outputs.

Definition 12.3.2 *A set of compatible groups of columns is said to* **overlap** *if the same elements, i.e. columns, appear in different groups of this set.*

One popular idea of *column minimization method* is to minimize the overlap. This retains more don't cares after column minimization, which allows for better results during symbol encoding and further decomposition processes. On the contrary, some other methods use the overlap in order to improve the encoding by the process of symbol selection (section 12.6).

Definition 12.3.3 *A set of Compatible Groups of Columns of function* F *is said to satisfy the* **completeness condition,** *if each column (in a table of the initial function* F*) is contained in at least one group of this set.*

Definition 12.3.4 *A* **feasible solution** *to Column Minimization Problem is a set of compatibles that satisfy the completeness condition. In the process, each compatible is uniquely denoted by a symbol* S_i *(for simplification, we will often use capital letters to denote symbols* S_i*).*

From now on, by S_i we will denote the symbol of a group CG of compatible columns.

Definition 12.3.5 *A* **quality** *of a compatible group, CG, is defined by Q, the number of its elements.*

Definition 12.3.6 *A feasible solution that has the lowest cost (the smallest number of CGs), CF, is called the* **exact solution to Column Minimization Problem.**

Having any feasible solution, one can find a set of two tables (predecessor and successor tables), functionally equivalent to the initial table.

If the solution is exact, the number of symbols, which is also the number of columns of function H, is the minimal. The number of symbols is also the number of output states of function G. Comparing to state machines, the symbol is a counterpart of the internal state of the machine.

12.4 The Column Minimization Problem

Standard approach, applied also as one of variants in MULTIS is to first minimize the number of columns (by set covering, or by set partitioning) and replace the compatibles from the cover (or the partition) with the symbols. Next these symbols are encoded. The standard process is then:

1. Column Minimization to groups of columns called symbols.
2. Encoding of symbols.

Column minization is solved by Set Partitioning or Graph Coloring techniques. These approaches produce disjoint columns in each symbol.

Column Minimization can be also solved by the Set Covering or Maximum Clique Algorithms. These approaches produce non-disjoint sets of columns in a symbol.

The *modified* standard process is to follow the Symbol Encoding with the Column Encoding, while the codes for columns are obtained by modification to the codes of symbols.

The modified standard process is:

1. Column Minimization combines columns to groups of columns called symbols.
2. Encoding of symbols with binary vectors.
3. Encoding of columns based on the encoding of symbols: the codes of the columns corresponding to a symbol are modified.

Finally, the new approach to Column Minimization introduced in this report is to modify the Set Covering or Maximum Clique Algorithms. In the modified algorithms some columns obtain higher probabilities to be merged into final groups of compatibles.

In the following subsections we will briefly present the Set Covering and Clique Partitioning problems. Other approaches to this problem can be found in our Survey paper and our Report paper [?].

				Y		
		y1	y2	y3	y4	y5
	x1	1	1		1	
	x2		1	1		
X	x3				1	
	x4		1			1
	x5	1		1		
	x6			1		1

Figure 12.6: A Covering Table

12.4.1 Set Covering

This section will discuss the basic theory of set covering and cliques. It is not the intent of this section to present new research results. Rather, this section will make use of existing algorithms to illustrate the theory of set covering and cliques.

The problem of set covering is an NP-hard one. Oftentimes set covering problems are small in the sense that the covering matrices are small, but they still are difficult to solve because the search space is very large. Thus, set covering applications need to have some smart algorithms and heuristics to limit this search space as much as possible while still obtaining a close-to-optimal solution. The section below will explain the covering problem. The section following, titled *A Set Covering Algorithm*, will demonstrate how set covering has been implemented in algorithms by using the branch-and-bound algorithm as an example.

The set covering problem is based on the following mathematical definitions.

Definition 12.4.1 *Let X be a set and Y a collection of sets. An element y of Y* **covers** *an element x of X, or x is covered by y, if x is a subset of y. A subset Y' of Y covers a subset X' of X, or X' is covered by Y', if X' is a subset of Y'.*

Definition 12.4.2 *Let Cost be a cost function defined on Y, i.e., a function from Y into the set of positive real numbers. For any subset Y' of Y, we define $Cost(Y')$ as the summation of the costs of all of the elements in Y'.*

Definition 12.4.3 Set Covering Problem: *Let X and Y be two sets, and Cost a cost function defined on Y. The set covering problem consists in finding a minimal cost subset of Y that covers X.*

The mathematical definition of set covering, while exact, is not easily understood or visualized. Thus, it is often advantageous to talk about a set covering problem with its covering matrix. The *covering matrix* of a set covering problem is a matrix with rows labeled by elements of X (the *set to be covered*) and columns labeled by elements of Y (the *covering subsets*). A position in the matrix at location $[x,\ y]$, where x is an element of X and y is an element of Y is a 1 if subset y covers x. The set covering problem solved in this form consists of finding the set of columns in the covering matrix that covers all rows in the covering matrix and minimizes the cost. The following example excerpted from [?] will help illustrate the covering matrix approach, see Fig. 12.6.

Assuming that all columns have the same cost, the minimal solutions are $\{y2,\ y3,\ y4\}$ and $\{y3,\ y4,\ y5\}$. Each of these solutions covers every element in X (all rows in the above matrix). Other solutions exist (such as $\{\ y1, y3, y4, y5\}$), but the other solutions have a cost of 4 columns or more while the optimal solutions cover set X using only three columns.

				Y		
		y1	y4	y3	y2	y5
	x3	1				
	x2		1			
X	x5			1		
	x4				1	
	x1					1

Figure 12.7: A Partitioned Covering Table 2

There are a few simple rules that can be used to simplify the covering problem before beginning an in-depth search for the optimal solution. They are: *partitioning, essentiality, dominance,* and *Gimpel's function.*

If the rows and columns of the covering matrix can be permuted in such a fashion that 1's appear only in the diagonal of the covering matrix then the minimal solution of the original problem is the union of all columns that contain a 1. Thus, in Figure 12.7 the minimal solution is $\{y1, y4, y3, y2, y5\}$.

Essentiality involves identifying the columns that must be present in the minimal solution. This involves identifying any columns that contain an element of x that is not present in any other column. Such columns are essential. They are included in the minimal solution and are removed along with any rows they covered.

A row x is *dominated* if covering another row x' would necessarily cover x. A column is dominated if it covers a subset of the elements x covered by another column y' and the $Cost(y) < Cost(y')$. Dominated rows and columns are removed from the covering matrix. The minimal solution to the original problem can now be found by solving the reduced covering matrix for its minimal solution.

Gimpel's reduction is used when two columns $y1, y2$ (each with the same cost) cover a row x. In this case, column $y1$ and all rows covered by $y1$ and $y2$ are removed. Finally some new rows are added to the covering matrix according to the rules of Gimpel's reduction (not presented here.)

After using the above techniques, the covering matrix has likely been reduced to a simpler matrix than was originally formulated. The effect of the reduced size of the covering matrix in solving for the minimal solution is that the reduced matrix will have a considerably smaller search space. This results in greater efficiency in the calculation-intensive search for the optimal solution.

Branch-and-Bound Set Covering Algorithm

The *branch and bound algorithm* consists of splitting a covering matrix $C(X,Y)$ into two covering matrices C_l and C_r. A single column y is picked through some smart heuristics (not presented here). C_l is the covering matrix that is built assuming that y is in the minimal solution and C_r is the covering matrix that is built assuming that y is not in the minimal solution. Now covering matrices C_l and C_r are recursively split again into two new covering matrices by choosing a new column y. When each split occurs, a new y element is bound and included in a solution (not necessarily the minimum solution). The process for the branch and bound algorithm is illustrated below for two iterations:

Assume a covering matrix has been minimized by partitioning, essentiality, dominance, and Gimpel's function. The reduced covering matrix is assumed to have rows $x1, x2, x3,$ and $x4$ and columns $y1, y2, y3,$ and $y4$.

Branch and bound example: assume through some heuristic column $y2$ was chosen to be bound in

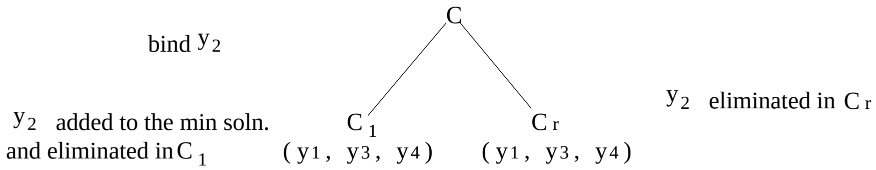


Figure 12.8: Creation of new covering matrices

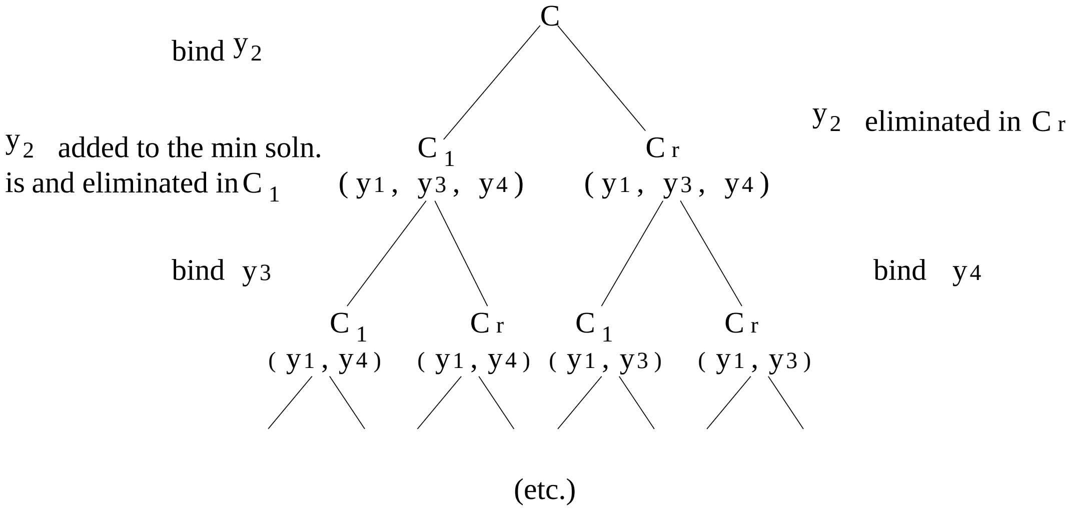


Figure 12.9: Tree Search for Covering Table

the first iteration. Thus, two new covering matrices are created as in Figure 12.8.

Assume that on the second iteration, y_3 is bound in C_l and y_4 is bound in C_r. The resulting tree looks like in Fig. 12.9.

The branch and bound algorithm thus searches the entire search space by intelligently picking one column in a covering matrix to bind and then recursively solving the two smaller covering matrices. The effect is that in each iteration one more column is bound and included in the solution. The included column is then eliminated in the subsequent covering matrices making them simpler to solve. This algorithm, like many other problem decomposition algorithms, reduces the complexity of finding the optimal solution by reducing the size of the search space by one variable at a time.

It should be noted that as it was stated above, the branch and bound algorithm is not very efficient. By halting recursion when a solution becomes less optimal than the currently most minimal solution this algorithm can be improved. There are also a number of additional pruning techniques that have been developed to further reduce the necessary search space although they will not be presented here.

12.4.2 Clique Partitioning and Generalized Clique Partitioning

Definition of Cliques

In the method of cliques, it is desired to express a problem to be solved as a graph. This graph is generated as a number of nodes with edges connecting each node to another node. Each node may be a data point, variable, etc. The edges are formed to represent some sort of relationship between nodes. This relationship between nodes varies by a problem. Take the following graph from Fig. 12.10 for example:

The nodes in the above graph are represented by the numbers 1, 2, 3, 4, and 5. The edges between nodes represent that those nodes have a relation or interconnection with each other. The edges in the

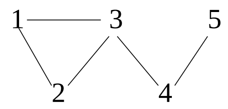


Figure 12.10: Graph to illustrate cliques

above graph are listed below by the numbers of the nodes that they go between:

1,2
1,3
2,3
3,4
4,5

The idea of the clique method is to find a minimum number of subgraphs that entirely describe the relationship between the nodes. This problem is called the *Clique Partitioning Problem* and is mathematically formulated as follows.

Let G be a graph consisting of a finite number of nodes and a set of undirected edges connecting nodes. A nonempty collection C of nodes of G forms a complete subgraph if each node in C is connected to every other node in C. A complete graph C is said to be a clique with respect to G if C is not contained in any other complete graph in G. The clique-partitioning problem is to partition the nodes in G into a number of disjoint clusters such that each node appears in one and only one cluster. Furthermore, each of these clusters itself forms a complete graph [?].

The cliques are found in practice by applying one of a number of algorithms. It should be noted that finding the cliques of a graph G by checking all combinations is an NP-hard problem and thus a smart algorithm is required to efficiently solve the problem. The algorithms often group the nodes according to a cost that is assigned to the edges that connect the nodes.

The generalized clique-partitioning problem is formulated as follows.

- The subgraph in which pairs of nodes have the best profit measure is reduced first.
- Then the pairs of nodes having the next level of profit measure are collected and reduced.
- Repeatedly applying the procedure to the other subgraphs, the process is stopped when a subgraph of a specified category or the original graph G becomes empty, [?].

An example is included in the next subsection that will help clarify the clique method.

Clique Partitioning Algorithm

A basic clique-partitioning algorithm, [?], takes advantage of the neighbor property. Let us consider an undirected graph G with nodes represented by integers. An edge is labelled by the two nodes that it goes between with the lower integer being named first. For example an edge going between nodes 3 and 4 would be called (3,4). Two nodes connected by an edge are called *neighbors*. A common neighbor is a node that is connected by edges to two other nodes which are neighbors. For example in the graph below, Fig. 12.10, nodes 1 and 3 are neighbors. Node 2 is a common neighbor of the pair of nodes 1 and 3 because both nodes 1 and 3 have an edge connecting them to node 2.

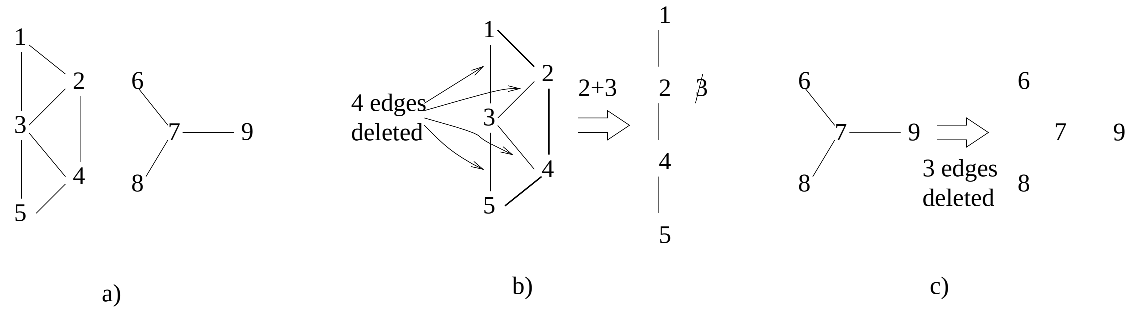


Figure 12.11: Graph 2 to illustrate the concepts of the Clique Partitioning Algorithm

In the neighbor algorithm, two lists are recorded. One is the *Nodes_List* and the other is the *Edges_List*. The list of nodes is a two-dimensional data structure in which each horizontal line represents a clique. At the start of the algorithm, each node occupies its own horizontal line and there are a number of cliques of size 1 equal to the number of nodes. During the algorithm, when nodes are combined to generate cliques of greater size, each horizontal line of the node list will contain more than one node and the number of vertical lines will be reduced as smaller cliques are merged into larger cliques.

The *Edges_List* is a sorted list of all edges in the graph. It is also a two-dimensional data structure. All edges with the same left node are arranged on the same horizontal line with the corresponding right node of each edge sorted in increasing order. For example, in the above graph, the two edges (1,2) and (1,3) exist. In the Edges_List these edges would be recorded as 1 2 3 for left node equal 1 and two right nodes equaling 2 and 3.

Algorithm

1. Scan through the list of edges (*Edges_List*). For each $(i, j), i < j$, compute its number of common neighbors.

2. Pick the edge (p, q) which has the maximum number of common neighbors. Combine the lists of nodes headed by p and q. The smaller one of p and q is used as the head of the resulting clique. Update the list of edges of G (when nodes are combined the edges connecting them are eliminated). If the list of edges is empty, the graph partitioning is completed.

3. Assume p is the head of the resulting clique. Pick an edge which joins node p and other nodes and has the maximum number of common neighbors. Let the edge be (p, r), or (r, p) if r is smaller than p. Save r, or p if r is smaller than p. Update the list of edges in G. If node p (or r if r is smaller than p) no longer appears in *Edges_List* then go to Step 2 and start to collect the next cluster. Otherwise repeat step 3.

For example, consider the following graph from Fig. 12.11a. The stages of execution of the Algorithm are illustrated in Figure 12.11b,c.

The sorted edges of this graph along with their corresponding number of common neighbors and the number of edges that must be deleted to combine the nodes in the edge are shown in Table from Figure 12.12.

It is desired to select the edge that has the most number of common neighbors. If there is a tie, then of those edges, the edge with the fewest number of edges that must be deleted is selected. If this is again a tie then one of the edges is arbitrarily selected. In the above example, edges (2,3) and (3,4) have the same number of common neighbors and the same number of edges that must be deleted. Assume

First Iteration:

Edge	# of Common Neighbors	# of Edges that Must be Deleted to Combine Nodes
(1,2)	1	3
(1,3)	1	4
(2,3)	2	4
(2,4)	1	4
(3,4)	2	4
(3,5)	1	4
(4,5)	1	3
(6,7)	0	3
(7,8)	0	3
(7,9)	0	3

Figure 12.12: Table of Edges

Second Iteration:

Edge	# of Common Neighbors	# of Edges that Must be Deleted to Combine Nodes
(1,2)	0	2
(2,4)	0	3
(4,5)	0	2
(6,7)	0	3
(7,8)	0	3
(7,9)	0	3

Figure 12.13: Second Iteration in Table of Edges

that in the second iteration, nodes 2 and 3 are selected to be combined. This results in the following reduced set of edges from Table from Figure 12.13.

Notice that the edges that are no longer necessary after nodes 2 and 3 are combined are deleted from the edge list. Repeatedly applying this method to the rest of the nodes in the graph results in the following cliques: {1,2,3}, {4,5}, {6,7}, {8}, and {9}.

Concluding, in this section we introduced few methods for set covering and clique partitioning problems. Similar methods have been implemented in MULTIS and we are experimenting with their different variants. The methods differ in their approaches, but basically can be programmed using similar techniques. The set covering algorithm was based on recursively binding progressively more and more sets y until the covering matrices were reduced to a trivial solution (empty). In this algorithm the sets were bound one by one in each iteration in order to generate the minimal solution. In the clique method, the nodes were paired two at a time. This process was continued until the list of edges became empty.

12.5 The Symbol Encoding Problem

This section describes a high-performance algorithm for the Symbol Encoding Problem in Decomposition. (The method finds the encodings only for the already column-minimized initial tables.)

This method can find application whenever a very high quality circuit is looked for, in a given realization technology, but when the function is of a rather small size.

The branch-and-bound optimum algorithm based on multi-valued minimization of the successor function and Boolean minimization of the predecessor function will be presented.

12.5.1 Basic Assumptions of the Symbol Encoding Method

The next design stage after column minimization is the encoding of the symbols. The graph embedding approach [?, ?], the "column-based" approach [?], and the partition theory [?, ?, ?, ?] are the best known state encoding methods that were applied in the past for state machines, and these methods give inspiration to create new algorithms for Symbol Encoding in the case of Functional Decomposition.

It is the conclusion based on the considered by us examples that the classical FSM state encoding assumptions are not directly useful for our encoding problems. Similarly, the recently developed heuristic encoding methods (DeMicheli [?], Mustang of U.C. Berkeley), do not produce the high quality results. It is, therefore, necessary to look for new encoding methods for functional decomposition. The paper [?] had the same basic idea as the method presented in this section, however the proposed algorithms were too slow, mainly because of the inefficient formulation of some search problems. Below we try to improve on the weaknesses of these papers and also, we apply these ideas to the encoding for decomposition.

It is our opinion that the exact minimum solution can be obtained only when the methods based on the partition theory are used. We will, therefore, briefly review some necessary notions. First, it must be mentioned that many of these methods generate the minimum solution assuming the minimization of the number of intermediate variables. In such a case, for K symbols the partitions in a total number of k are selected, where $2^k - 1 < K \leq 2^k$. This leads to the concept of *proper partitions,* defined as the two-block partitions in which the number of symbols in the larger block does not exceed 2^{k-1}. The partitions will be, therefore, selected among the set TP of proper partitions. The final family (set) of partitions, TF, is defined as such a family, that the partition product of all partitions from that family is a zero partion. *The optimum family*, TO, is such a final family that minimizes the value of an assumed cost function.

There are two approaches possible here. The first approach is to assume only the minimum number of partitions, and hence the minimum number of intermediate variables. The argument is that less variables minimizes the DFC of the first-level decomposition. The second approach is to take any number of partitions, usually slightly larger than the minimum, in order to simplify the functions of the decomposed blocks. Although the DFC of the first decomposition may suffer, the functions are simpler, so that the next decompositions of both blocks will be able to recover from this temporary DFC increase, and would produce a final DFC with a smaller value for the total decomposed function. Only experiments will confirm which of these ideas will work statistically better on Machine Learning benchmarks. The family TO in the first approach is selected among the proper partitions, and in the second approach among all the two-block partitions. Let us denote the zero partition by $\overline{0}$: $\overline{0} = \{\overline{1}, \overline{2}, ..., \overline{K}\}$.

In all methods of symbol encoding the partitions are selected to TF in such a way that a unique (one-to-one) encoding is created.

$$\prod_{\tau_i \in TF} \tau_i = \overline{0} \tag{12.1}$$

In both approaches we look for such final family of partitions TF that the cost of encoding is minimized.

The problem of symbol encoding of the column-minimized table can be formulated as follows:

Given are:

- S - the set of symbols of the column-minimized function F; $K = CARD(S)$,
- $T2$ - the set of all proper (first appraoch), or two-block (second appraoch) partitions,
- the inital table of function F.

The set TO to be found is such a subset of the $T2$ set, that both of the following conditions are satisfied:

1.
$$\prod_{\tau_i \in TO} \tau_i = \tau_1 \cdot \tau_2 \cdot \ldots \cdot \tau_s = \overline{0} \tag{12.2}$$

That is, the partition product of the selected partitions for intermediate variables v_1, v_2,..., v_s is a zero partition, which means that a different binary sequence is assigned to each symbol.

2.
$$min(CF_r) = min(\sum_{v_i \in \{v_i, \ldots, v_s\} = V^{TO}} q_r(v_i)^{TO} + \sum_{y_j \in \{y_i, \ldots, y_m\}} q_r(y_j(A, V^{TO}))), \tag{12.3}$$

where:

A is the set of free variables,

V^{TO} is the set of intermediate variables for set of partitions TO,

$\{y_1, ..., y_m\}$ is the set of output signals of function H (the same as for function F).

$q_r(V)^{TO}$ is the real costs of the realization of the τ_i partition, as one of the TO set. The costs are calculated for the respective circuit realization for corresponding to τ_i intermediate variable v_i.

$q_r(y_j)$ is the cost of realization of an output signal y_j, calculated assuming the given partitions for TO.

Let us assume that both functions G and H are minimized here assuming two-level logic realizations created for set TO of partitions. This is called the condition of the total cost function minimization.

The encoding methods discussed in sections ??.6 and refSection6 (OLD 12.5.7 and 12.5.8 apply) apply the heuristic search and the branch-and-bound principle [?]. This prevents us from searching the entire solution space, being in this case the set of all subsets of the set of proper partitions (the first approach) or two-block partitions (the second approach). A few heuristics are applied as a means for controlling the search strategy. Among them is the application of quality functions to evaluate partitions. However, in this approach the heuristics do not lead to losses of the minimum solutions (in section (OLD 12.10) 8 the heuristics are described, that increase search efficiency but can omit the optimum). The quality function and the cost function are discussed in sections (OLD 12.55,6.) 5.4 and 5.5.

12.5.2 Operators and Relations on Cubes

Before we discuss the quality functions (denoted below by QF), used for the symbol encoding and for the joint column minimization and encoding (called also column encoding, see section ?? (old section 7)), we need to define a number of operations on an array of cubes.

The names of the operators and relations arised by reason of their application in this report.

a) a_i (rows), b_i (columns)

∇	0	1
0	0	1
1	1	ϕ

b) a_i (rows), b_i (columns)

$\triangle$	0	1	X
0	0	X	ϕ
1	X	1	ϕ
X	ϕ	ϕ	X

c) a_i (rows), b_i (columns)

$\triangleleft$	0	1
0	0	ε
1	ϕ	1

d)

$\sqsubseteq$	0	1	X
0	ε	ϕ	ε
1	ϕ	ε	ε
X	ϕ	ϕ	ε

Figure 12.14: Tables of operators on cubes: (a) column compatibility, (b) row compatibity, (c) column absorption, (d) row absorption

Definition 12.5.1 The operator of column compatibility , *denoted by* ∇, *is defined for binary cubes* a *and* b *(here* a *and* b *are n-tuples of elements* $a_i, b_i \in \{0,1\}$*), as follows:*

$$a \nabla b = \begin{cases} \emptyset & if(\exists i)[a_i \nabla b_i = \emptyset] \\ c = (c_1, c_2, .., c_n) & in\ the\ opposite\ case, \end{cases}$$

where
$(\forall i = 1, ..., n)[c_i = a_i \nabla b_i]$ *in accordance with the table, Fig. 12.14a.*

Examples:

$$1100 \nabla 0010 = 1110$$

$$0101 \nabla 0001 = \emptyset \quad \text{because} \quad a_4 \cdot b_4 = \emptyset$$

Definition 12.5.2 The operator of cube (row) compatibility, *denoted by* $\triangle$, *is defined for ternary cubes* a, b *(ternary cubes are n-tuples of elements* a_i, b_i $\{0, 1, X\}$*), as follows:*

$$a \triangle b = \begin{cases} \emptyset & if(\exists i)[a_i \triangle b_i = \emptyset\ or\ a_i \triangle b_i = X for\ more\ than\ one\ i,\ when\ a_i \neq X], \\ c = (c_1, ..., c_n) & otherwise \end{cases}$$

where:
$(\forall i = 1, ..., n)$ $[c_i = a_i \triangle b_i]$, *in accordance with table from Fig. 12.14b.*

Definition 12.5.3 The relation of column absorption, *denoted by* $\triangleleft$, *is defined for binary cubes* a *and* b *(*a_i, $b_i \in \{0,1\}$*), as follows:*

$$a \triangleleft b\ if\ (\forall\ i)[a_i \triangleleft b_i \in \{0, 1, \varepsilon\}] \tag{12.4}$$

$$a \neg \lhd b \ if \ (\exists \ i)[a_i \ \lhd \ b_i \ = \ \emptyset] \tag{12.5}$$

where the operation $a_i \lhd b_i$ *is defined in accordance with the table from Fig. 12.14c.*

Examples.
0100 $\lhd$ 0110,
0100 $\neg\lhd$ 0010,

Definition 12.5.4 The relation of cube (row) absorption, *denoted by* $\sqsubseteq$ *is defined on ternary cubes a and b as follows:*

$$a \sqsubseteq b \ if \ (\forall \ i)[a_i \ \subseteq \ b_i \ = \ \varepsilon] \tag{12.6}$$

$$a \not\sqsubseteq b \ if \ (\exists \ i)[a_i \ \subseteq \ b_i \ = \ \emptyset] \tag{12.7}$$

where the operation $a_i \subseteq b_i$ *is defined in accordance with the table from Fig. 12.14d.*

12.5.3 Generation of Prime Implicants of Multi-Valued Functions

In this subsection we will deal with the generation of prime implicants for multi-valued functions.

We will start with the case in which it is assumed that the number of partitions is minimal, and thus only proper partitions are considered. To find the successor function from the successor table the code for each cell of the table is required. The code of each cell is the concatenation of the one-hot code of column and the cube code of row; however, the encoding is yet unknown, and hence so is the column's code. Our goal is then to select such an auxiliary code for columns, that will allow to check all the possibilities of minimization of the output functions (i.e., covering minterms with prime implicants) before finding the final encoding of the symbols of the table. The code will then make it possible to check all possibilities of joining groups of minterms into prime implicants, i.e., applying logic adjacency to minterms from every two columns. Suppose we have encoded the tables according to some proper partition τ_j. Several possibilities of including columns into prime implicants exist now. This results from the fact, that we assume the possibility of arbitrary permutation of the columns, such that the other partitions are assorted in the best possible manner with τ_j, to minimize the *multi-valued function*, described by a table with columns encoded with τ_j (the encoded tables will be called *transition tables*). To formalize such *generalized merging of columns into groups* we must find the respective code for the columns. For K columns of the successor table we select a code of length K, where 1 is written to the i-th column on the i-th position. For example three symbols are encoded as follows:

1 - 100,
2 - 010,
3 - 001.
We call it a *one-hot code.*

Actually, every cell of the successor table (i.e., a 0-cube) can be described by means of the binary string: $c^i \ = \ c^i_\alpha \ \bullet \ c^i_\beta$ where c^i_α is the successor's block free variables inputs cube in Gray code. and c^i_β is the cube of columns in the code described above, The symbol $\bullet$ stands for concatenation. The operations from subsection 12.5.2 may be utilized to merge the cubes into larger cubes. Both constituents of the string are used. There may be $n_1 \ < \ K$ arbitrary columns in each group after merging. The adopted code is the simpliest one, with which this property can be checked. All the cubes generated by means of joining have the part corresponding to the binary inputs, and the part corresponding to the symbols of columns. The implicants of the multi-valued function will be called *"generalized implicants"* (abbrev. *G-implicants*). The G-implicants which are prime will be called *"generalized prime implicants"* (*GP-implicants*). Let ON be the set of minterms and DC be the set of "don't-cares" of the Boolean function, which corresponds to the successor table, whose columns correspond to symbols and are encoded in accordance with the chosen partition τ_j. We shall find the set of all GP-implicants, and next the covering of minterms with the GP-implicants.

Let $F = (y^1, y^2, ..., y^i, ..., y^m)$ be a multioutput multi-valued Boolean function with n inputs and m outputs, where $y_i = y^i(x_1, ..., x_n)$, $i = 1, ..., m$ are the component single output functions. Any multi-valued minimizer such as Espresso-MV [?] can be used to generate good covering with GP-implicants. Another method to find GP-implicants would be to use the method of graph coloring.

12.5.4 Quality Function for Partitions

Quality function for the set of partitions $\{\tau_1, ..., \tau_r\}$ will be defined as follows:

$$QF(\{\tau_1, ..., \tau_r\}) = \sum_{\tau_i \in \{\tau_1,...,\tau_r\}} q(\tau_i) \tag{12.8}$$

where $q(\tau_i)$ is the total cost of realization of the partition τ_i in predecessor and successor blocks. It is calculated as a sum of costs $q(y_i)$ for all output variables and costs $q(v_i)$ for all intermediate variables. Both these costs are calculated for the case of selecting the best additional partitions assorted with partition τ_i.

The cost q depends on the applied method of realizing the decomposed blocks. Below a few methods of estimating the q are presented, q^s stands for these methods, s = 1,2,3,4.

Notations applied:

- NL_i - number of literals in the i-th implicant,
- NL - total number of literals of all implicants NL = $\sum_{i=1}^{i=NT} NL_i$,
- NT - total number of implicants ,

It is the total number of inputs to the AND level gates and the OR level gates.

Calculation of the quality functions for partition τ involves the following steps:

Algorithm 12.5.1

Calculation of the quality functions for partition τ.

Given are symbolic predecessor and successor functions.

Begin

1. Encode the first block of τ partition with 1 and the second block with 0;
2. Given τ, find the multi-valued minimum cover of the successor function H using GP-implicants. Calculate the binary cost (cost of binary implicants) and the multi-valued cost (cost of GP-implicants). Calculate the weighted total cost of H.
3. Given τ, find the minimum cover of the predecessor function with standard binary implicants.
4. Encode the first block of τ partition with 0 and the second block with 1;
5. Execute steps 2 and 3 with the encoding from step 4.
6. Calculate the minimum of the two costs found in steps 2 and 4 - the cost is computed: $q^{min}(\tau)$.
7. Store in memory the encoding of the partition's blocks to symbols 0 and 1.

End

This algorithm can be also extended to concurrent minimization of functions G and H. We adopt the following notation:

$$\tau_{ij} \; = \; \{ \; \overline{i, j}, \; \overline{other \; states \; from \; A} \; \}$$

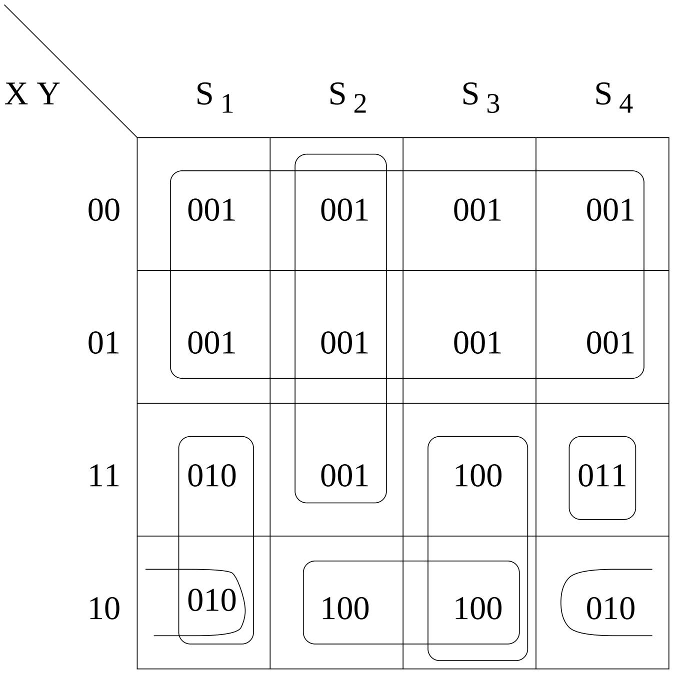


Figure 12.15: Function H for Example 5.1.

Example 12.5.1

Given is the function from Fig. 12.15
$T2 = \{\tau_{12},\ \tau_{13},\ \tau_{14},\ \tau_1,\ \tau_2,\ \tau_3,\ \tau_4\}$.

The method of computing $q(y_i)$ for τ_{13} is presented in Fig. 12.16. For simplification, only assignment of 1's to subscript block of τ is shown. Also, only function H is evaluated. The method can be, however, easily expanded to both block encodings and to the simultaneous evaluation of function G.

Function $y_1 = x\ \bar{y}\ \tau_{23}\ +\ x\ \tau_3$.
Function $y_2 = x\ \bar{y}\ \overline{\tau_{23}}$.

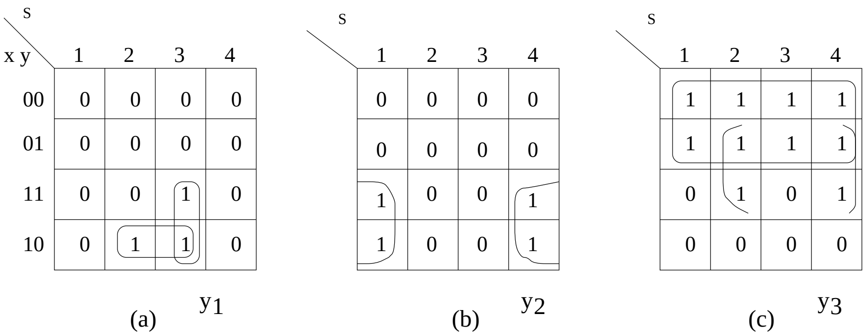


Figure 12.16: Calculations of costs of partition τ_{23} for functions y_1, y_2, y_3

Function $y_3 = \bar{x} + \tau_{24}\ y$.

Depending on the realization method, partition τ_3 can be realized as a separate partition, or as $\tau_{23} \cdot \overline{\tau_{24}}$. The total cost of partition τ_{13} is thus calculated for functions y_1, y_2, y_3.
Similarly, the costs are calculated for all possible partitions from $T2$.

12.5.5 Cost Functions for Final Families

The cost function for final family of partition T is defined as follows:

$$CF_r(TF) = \sum_{v_i \in V} q_r(v_i) + \sum_{y_j \in Y} q_r(y_i). \tag{12.9}$$

Functions q_r are computed in the same fashion as the q functions. The difference lies in the methods of minimization of the predecessor functions and the successor functions. In the quality function every partition was evaluated separately, on the basis of the minimum GP-implicant covers, found for the separate functions. Similarly, every function was computed for the best code found for it.

The cost functions CF_r, however, may be computed either separately or jointly. If the cost functions are computed separately for each of the functions and the functions are realized separately, then Algorithm 5.1 (and Algorithm 12.7.1) guarantee that the minimum solutions are found (see comment 2 to Algorithm 12.5.4). Cost function CF_r may be also computed for the jointly minimized intermediate and output variables for the given family of partitions. Hence there may be common implicants - there exists then a possibility of getting better solutions than those resulting from the assumptions taken previously. However, for this class of solutions there is no assurance that the optimum solution is found.

In what follows we will apply the following notation for the jointly computed cost functions:

${CF_r}^{2j}$ - corresponds with the number of inputs in AND and OR gates,

${CF_r}^{3j}$ - corresponds with the number of literals in the functions, at the same time in repeated implicants the literals are counted one time only.

12.5.6 Algorithm of Encoding for Column-Minimized Tables.

Algorithm of Encoding

Encoding of column-minimized tables is performed using the following algorithm. The input data are symbolic functions G and H.

Algorithm 12.5.2

Begin

E1: Find $T2$ family of all two block partitions using Algorithm 12.5.3. $T2$ family is stored as a list with name $T2$. This algorithm utilizes breadth-first tree search. Hence it arranges the $T2$ list in such an order that it starts with partitions of blocks with the same sizes of blocks, or with blocks whose lengths differ at most by one, and which terminate with partitions having the longer block of length $K-1$;

E2: (Option) Sort the $T2$ list according to some criterion of partitions quality. One of the methods is to sort the $T2$ list in such a way, that among the partitions of the same length of the shorter block, the partition with a smaller value of quality function q is placed first. The values of quality functions are calculated by Algorithm 12.5.1;

E3: Find families (or a single family - in accordance to the value of the user's parameter ALL) of partitions $TO \subseteq T2$ by the means of Algorithm 12.5.4.

The families must satisfy the following conditions:

a) condition of code uniqueness
$\prod_{\tau_j \in TO} \tau_j = \overline{0}$ product is the zero partition.

b) condition of the cost function minimization:

$$min(CF_r) = min(\sum_{v_j \in V} q_r(v_j) + \sum_{y_i \in Y} q_r(y_i)) \tag{12.10}$$

End

As a part of this algorithm the Boolean (multi-output) functions of blocks G and H are found.

The algorithm uses a tree search in solution trees. The states (nodes) of the tree are the vectors of coordinates. Each vector for state with number N is denoted as

$$V(N) = (C_1(N), C_2(N), ..., C_r(N)) \tag{12.11}$$

where C_i, $i = 1, ..., r$ are the coordinates. New states are created from the old ones with use of *operators.*

By NN we will denote the number of the currently created new state of the tree, being the successor of state with number N. By $A \setminus \{b\}$ we denote set (list) A with removed element b. The coordinates have the following meaning:

- $QS(N)$ - part of generated partition,
- $GS(N)$ - set of operators of node N.

The operators correspond to the symbols. Generation of $T2$ family of partitions is done by Algorithm 12.5.3.

Algorithm 12.5.3

Begin

1. *Creation of the initial state of the solution tree:*
   $N := 0;$
   $V(0) = (QS(0), GS(0)) := (\{\}, A);$
   $T2 := \{\}; OPEN := \{V(0)\};$
2. If $OPEN = \{\}$ then return;
   $V(N) = (QS(N), GS(N)) :=$ first element from list $OPEN$;
3. if $GS(N) = \{\}$ then
   begin delete first element from list $OPEN$; go to 2 end
4. $s :=$ first element from $GS(N)$;
   $QS(NN) := QS(N) \cup \{s\};$
   $GS(NN) := GS(N) := GS(N) \setminus \{s\};$

5. add $(QS(NN), A \setminus QS(NN))$ to the beginning of list $T2$.
6. Add new state $NN = (QS(NN), GS(NN))$ to the end of list $OPEN$;
7. Go to 3;

End

Comments.

1. When Algorithm 12.5.3 terminates the $T2$ list is a $T2$ family.
2. The two-block partition $\tau \in T2$ is represented with the single computer word e. Bits 1 of the word correspond to the elements of the first partition block. Then $\overline{e}$ (the word e after bit by bit negation) corresponds to the second block of the partition. Therefore $\tau = \{B_1,\ B_2\}$, where $B_1 = \{a_i \mid e[i] = 1\}$, $B_2 = A \setminus B_1$. The partition may also be represented in the transitional form $\{e,\ \overline{e}\}$. Since we know the value of K then the B_2 block may be easily computed as the bit by bit negation of word e.

The set theory operations are reduced to the operations on the binary words, e.g. product of words, denoted by $\cap$, corresponds to the intersection of sets. Therefore $q(\tau)$ will be identified with $q(e)$. Product of partitions will be denoted by $\cdot p$. In fact Algorithm 12.5.3 and the following algorithms use words and arrays of binary words. In the expression: $\alpha_i \cdot e, \delta_i$ stands for a word with bit 1 in position i, and with the rest of the bits equal to 0. $GS(N)$ is the set of operators for node N, $QS(N)$ is a set of selected partitions for coding in node N, $PR(N)$ is the product of partitions from $QS(N)$. In this problem the operators are arbitrary partitions from $T2$.

Generation of TO family by means of $T2$ family is done as follows:

Algorithm 12.5.4

Begin

1. *Creation of the initial state of the tree*

   $(GS(0), QS(0), PR(0), QF(0)) := (T2, \{\ \}, \overline{1}, 0);$

   $CF_{min} := \infty$; SOLUTION := {}; initialize list $COSTS$;
2. If $OPEN = \{\}$ then

   begin

   if $ALL = 1$ then print "optimum solutions", $SOLUTION$

   else print "optimum solution", $SOLUTION$;

   return

   end ;

   $V(N) = (GS(N), QS(N), PR(N), QF(N)) :=$ element from $OPEN$ list selected according to some search strategy (for instance the first element for the depth-first strategy);

   If $ALL = 1$

   then

   all minimum solutions are searched - delete from $GS(N)$ all such operators e that $CF_{min} < QF(N) + q(e)$

   else

   only some of the minimum solutions are sought - delete from $GS(N)$ all such operators e that $CF_{min} \leq QF(N) + q(e)$;

3. *Extension of state* $V(N)$

(a) If $GS(N) = \{\}$ then
begin
delete first element from list $OPEN$ and go to 2,
end

(b) $e :=$ first element from $GS(N)$; $GS(N) := GS(N) - \{e\}$;

(c) $PR(NN) := PR(N) \cdot p \ \{e, \overline{e}\}$;

(d) $GS(NN) := GS(N) \setminus \{e_i \in GS(N) \mid \{e_i, \overline{e}_i\} \cdot p \ PR(NN) = PR(NN)\}$;

(e) If ($GS(NN) = \{\}$) and ($PR(NN) \neq \overline{0}$) then go to 3; /* backtrack */

(f) $QS(NN) := QS(N) \cup \{e\}$;

(g) Calculate $q(e)$, add new item to $COSTS$
(this step is executed if $q(e)$ is not found in Algorithm 5.1.

(h) $QF(NN) := QF(N) + q(e)$;

(i) If $QF(NN) > CF_{min}$ then go to 3; /* backtrack */

4. /* *Checking of solution* */

```
If  PR(NN) = 0̄ then /* new solution found */
  begin
    Calculate CF_r.
     /* Cost functions q_r(τ_i) are computed for the same
     assignment of the blocks to 0 and 1, as those, already
     found in Algorithm  12.5.1 for the
     best quality function q(τ_i); */
    if CF_r(NN)  =  CF_min then
     begin
       if all solutions are looked for (ALL = 1) then add QS(NN) to list SOLUTION
       else SOLUTION := {QS(NN)};
       go to 3
     end ;
    if CF_r(NN)  <  CF_min
    then
     begin
       CF_min  :=  CF_r(NN);
       SOLUTION := {QS(NN)}
       go to 3
     end ;
  end
  else
  begin
```

add new node to the tree

add $V(NN) = (GS(NN), QS(NN), PR(NN), QF(NN))$ to the beginning of $OPEN$;

go to 2;

end

end.

Discussion and Examples.

Comments to Algorithm 12.5.4.

1. If $QS(NN)$ is the TF family (i.e., if $PR(NN) = \overline{0}$) then the minimum realization of the block functions may be found, and then the real cost of function $CF_r(N)$ may be computed. However, while generating the successive states N_i of the solution tree we don't know the codes of the symbols, so we can't calculate the value of $CF_r(N_i)$. Therefore, to be able to apply the branch-and-bound rule we must have some function for each node that will be used for cutting off. It is useful that QF function can be used with that respect.
2. The value of quality function $QF(N_i)$ is the sum of costs of partitions selected from $QS(N_i)$ and the total cost of output signals realizations. (There is the assumption in the method that the block functions are realized and minimized separately). Therefore, for every $N \in S_k$ (i.e., $QS(N)$ fulfilling conditions of the problem) there is: $QF(N) \leq CF_r(N)$. For every node N_i in a branch leading to N: it holds $QF(N_i) \leq QF(N) \leq CF_r(N)$. If the cost of the last solution was CF_{min}, and in the following process of tree extension we find such node N_i, that QF(N_i) $> CF_{min}$, then the further development of every branch that begins with N_i may lead only to solutions QS(N') with values $CF_r(N') \geq QF(N_i) > CF_{min}$, so we can cut it off at node N_i.
3. If all the nodes for which it holds that $QF(N) \leq CF(N)$ have been already looked through, and we encounter the node $N_j \in S_k$, such that $QF(N_j) = CF(N_j)$, then $QS(N_j)$ is the optimum family of partitions TO. Cost $q(\tau_j)$ of every partition is calculated separately, assuming that all the remaining partitions will be chosen in the best way after selection of the partition assumed. From this assumption there results the possibility of arbitrary permutation of rows of the initial table, which in turn leads to the generation of "generalized prime implicants." The partition cost τ_i obtained in this way will then be at most equal to the realization cost of the i-th block function.
4. If functions are realized separately then Algorithm 5.4 gives a guarantee that the minimum solution is found, because in such case it holds for all nodes N in the solution tree that $CF_r(N) \geq QF(N)$, so that the cut-off nodes cannot include the optimum solution. In the case of common minimization the optimum solutions can be cut-off because there may be $CF_r(N) < QF(N)$.
5. For some variants of the method, the value $q(e)$ is calculated for the first call on the basis of Algorithm 12.5.1, and at the following calls it is found from the incrementally created list of pairs:

$$COSTS = ((e_{i1}\ q(e_{i1})),\ (e_{i2}\ q(e_{i2}))...). \tag{12.12}$$

6. Such a variant of the method results from the trend to find part of the solutions without generating $q(e)$ for all partitions (there may be plenty of them, and much time would be spend before generating any solution - that would preclude the algorithm from the ability to deal with large functions). In such case list $T2$ is not sorted in step $E2$ of Algorithm 5.2 with respect to the values of q functions, but according to some other quicker method (not involving calculations of q functions), or not sorted at all. $COSTS$ list is initialized as $\{\}$.

12.6 The Column Group Selection Problem

In this section we will present two approaches to improve the quality of the Symbol Encoding method from the previous section at the prize of more complex processing.

The essence of the method is this. Normally only one symbolic code is assigned to a column. Let us call each column covered with more than one group of compatibles the *non-unique column.* It can be observed that for each such column there is a choice of a symbol. It is then reasonable to select a symbol that would simplify the encoding. The method presented here is quite similar to the one from the section 12.5, but it improves the quality of the solution at the price of slower processing. Algorithm ??(Alg.6.3.old) can be used as a preprocessor of **every** Symbol Encoding method.

12.6.1 Selection of Symbols based on Map Simplification

Example 12.6.1

For dydactic reasons, we will explain first the method on an example.

[[Example 6.1. is ??. Example 6.2. is ??]]

Fig. 12.17a presents an *initial function table.* The columns, i.e. the cofactors of the bound set {c,d,e}, are denoted by C_i, $i = 0, ..., 7$. Below the columns, their possible symbols are shown. The symbols are created by multi-coloring or any other method from section 4. For instance, column C_0 can be assigned with symbols A, B, or C. Column C_2 can be assigned with symbols A or C.

Figure 12.17b shows the enumeration of columns in the initial function map. The corresponding Symbolic Predecessor Relation Map is shown in Figure 12.17c. For each column of the initial function map (represented by the corresponding cell of the Symbolic Predecessor Map), the symbols that can be used for this column are shown in Figure 12.17c.

Now, the goal is to select in every cell of the Symbolic Predecessor Relation Map such single symbol that the encoding will be minimized. One can see that selecting A in all cells 000, 010 and 100 will allow to create large groups of A's, so it should simplify the encoding. This choice is shown in Figure 12.17d.

This new map from Figure 12.17d is now evaluated using various partitions. In our case we use all two-block partitions, since the table is small. For larger maps only a subset of two-block partitions (or a subset of proper partitions) is tried. (Those partitions are found using a method similar to the one from section 12.5. These two methods can be combined to evaluate each position based on complexities of both functions G and H.)

Suppose that partition $\tau_1 = \{\overline{AB},\ \overline{C}\}$ is tested and it is assumed that block $\overline{AB}$ is encoded with 1 and block $\overline{C}$ is encoded with 0. This is denoted by {Block1, Block0} $= \{\ \overline{AB}, \overline{C}\ \}$. In this example partition $\tau_1 = \{\overline{AB},\ \overline{C}\}$ has cost 3 - see Figure 12.17e, partition $\tau_2 = \{\overline{AC},\ \overline{B}\}$ has cost 13 - see Figure 12.17f, and partition $\tau_3 = \{\overline{BC},\ \overline{A}\}$ has cost 9 - see Figure 12.17g. Therefore, partitions τ_1 and τ_3 are selected as having the smallest total cost among all pairs of partitions whose product is $\overline{0}$. (it is easy to check that $\tau_1 \cdot \tau_2 = \overline{0}$).

The first method of selecting symbols is as determined by the following Algorithm ?? (Alg 6.1.old):

Algorithm 12.6.1

1. For every cell of the Symbolic Predecessor Relation Map (with rows and columns corresponding to set B as in Figure) that contains more than one symbol select such symbol that the encoding is simplified - for instance: a row of the same symbols, a row of the same symbols, or a large implicant of the same symbols.

2. Evaluate this map using different partitions, using the method presented in section 5. Similarly to the method from section 5, for every symbolic map of G either all partitions are tested, or only some subset of proper partitions is calculated for speed improvement (for instance those that were good in previous encodings).

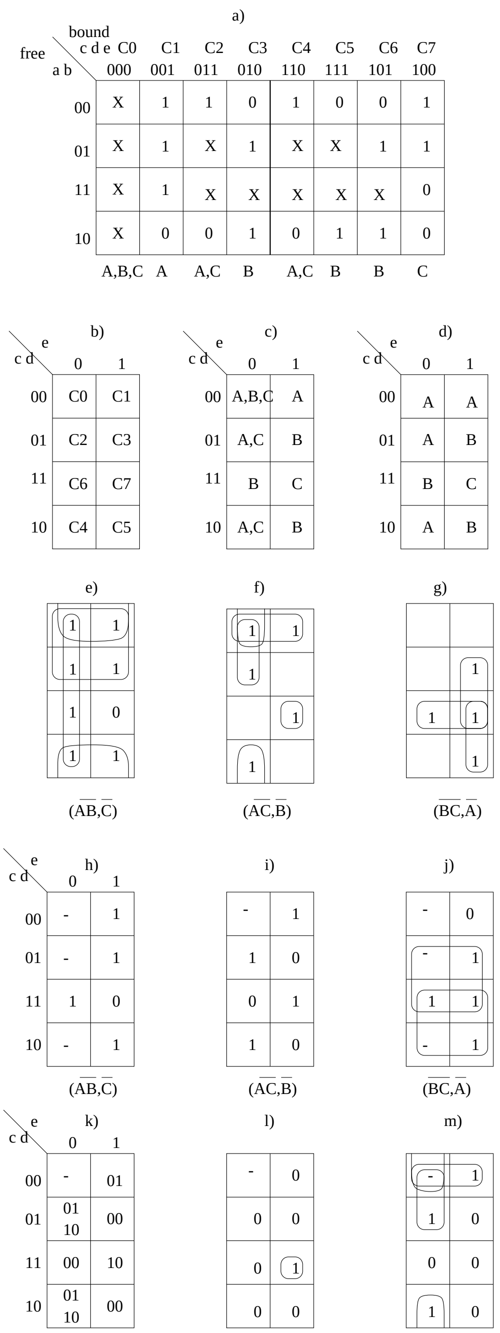


Figure 12.17: The Column Group Selection and Encoding process to Examples 6.1 and 6.2.

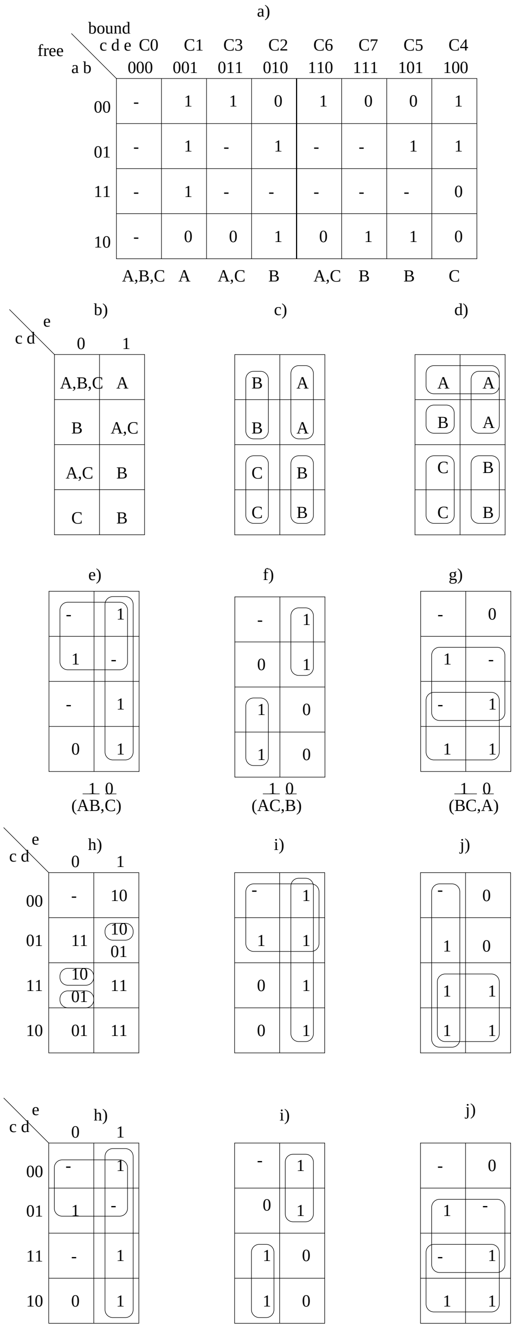


Figure 12.18: The Column Group Selection and Encoding process for Example 6.3.

12.6.2 Column Encoding Based on Boolean Relation Minimization

The second method is to evaluate the partitions using the symbolic relational map of G before making the mapping choice. The algorithm of this method is as follows:

Algorithm 12.6.2

1. Take a proper partition. Encode the symbolic relation map with this partition in such a way that all groups which are included in $Block0$ are encoded with 0, all groups which are included in $Block1$ are encoded with 1, all other (those that intersect both blocks of the partition) are encoded with a don't care. Therefore, when a cell of the map includes symbols from both blocks, it is replaced with a don't care. Otherwise, it is assigned the binary symbol associated to this block.
2. The Boolean function created in step 1 is minimized using standard 2-level minimizer Espresso.
3. The steps 1 and 2 are iterated for various proper partitions. Either all partitions are tested, or only some subset of proper partitions is calculated for speed improvement.
4. The partitions with the smallest cost are selected.
5. The Boolean relation is minimized and changed to a Boolean function, to be further processed in the next stages. This is done using Algorithm ?? (Alg 6.3old) (not presented here in detail; see also section ??(sect 10 old).)

Let us observe, that the step 5 of Algorithm ??(ALg 6.2old) can be applied to any function G realized as an encoded relation map. This map is found using any encoding algorithm.

Example 12.6.2

The encoded map created for symbolic relation map from Figure 12.17c, and partition $\tau_1 = \{\overline{AB}, \overline{C}\}$ = {B1, B0} is shown in Figure 12.17h. Using the Symbolic Relation Map from Figure 12.17c, for partition $\tau_2 = \{\overline{AC}, \overline{B}\}$ = {B1, B0} - the encoded map is shown in Figure 12.17i, and for partition $\tau_3 = \{\overline{BC}, \overline{A}\}$ = {B1, B0} - the encoded map is shown in Figure 12.17j. The same code as previously is selected.

The Boolean Relation Map is created in Figure 12.17h. In cell 010 we have two binary vectors - 01 and 10, which means, any of them can be selected, and the selection should be done in a way to simplify the logic. Selecting the first choice, 01 in both cells, will lead to the solution from Figure 12.17m.

Example 12.6.3

This is a slightly modified Example ?? (Example 6.2old). We just permuted columns in the Initial Map, to demonstrate how such permutation can affect the quality of encoding.

Figure 12.18a presents an initial function map. As previously, below the columns, their possible colorings from multi-coloring are shown. Two selections to create large groups are shown in Figure 12.18c and Figure 12.18d, respectively. They are supposed to simplify the future encoding.

The Symbolic Relation Map from Figure 12.18b is now evaluated using various partitions. For partition $\tau_1 = \{\overline{AB}, \overline{C}\}$ with block $\overline{AB}$ encoded with 1 and block $\overline{C}$ encoded with 0, the Boolean Relation from Figure 12.18e is obtained with cost 2. Similarly, for partition $\tau_2 = \{\overline{AC}, \overline{B}\}$ the cost is 6 - see Figure 12.18f, and for partition $\tau_3 = \{\overline{BC}, \overline{A}\}$ the cost is 2 - see Figure 12.18g, Therefore, partitions τ_1 and τ_3 are selected as having the smallest total cost among all partitions whose product is $\overline{0}$. The resultant code is: A = 10, B = 11 and C = 01. The Boolean relation map corresponding to this code is shown in Figure 12.18h. Making the choices encircled in the map, the solution is the Boolean function shown in Figure 12.18i,j. It can be shown, that the same solution is created by selecting the symbols as in Figure 12.18c and next encoding with $\tau_1 = \{\overline{AB}, \overline{C}\}$ and $\tau_3 = \{\overline{AC}, \overline{B}\}$ - see Figure 12.18k.

Both Algorithm ?? (Alg. 6.1old) and Algorithm ??(Alg.6.2old) can be called for any sets of compatibles. Several heuristics should be added to make the process more efficient. There are several trade-offs among these two approaches. The second method requires an approximate minimizer of Boolean relations. Such program does perhaps exist in U.C. Berkeley tools. Therefore, the second method should be implemented. Yet another variant of this appraoch is present in section 10.

12.7 Column Encoding

This section describes a high-performance algorithm for the Column_Compatibility/Encoding Problem of Decomposition. This method can find applications whenever a very high quality circuit is looked for, in a given realization technology, but when the function is of rather small size.

The method combines two, traditionally independent stages of column minimization and symbol encoding, to the single process of concurrent column minimization and encoding. The basic idea of this method is to select such partitions for the symbols of the not necessarily column minimized table (in particular, the columns), that the circuit realization of this table, composed from blocks G and H, minimizes certain cost function. When these partitions are used for symbol encoding, the number of symbols is reduced, as a result of assigning the same codes to the groups of compatible columns C_i. Let us make a point that the number of symbols is reduced but not necessarily totally minimized. Such an approach corresponds to the search for the maps that are equivalent to the initial map and that minimize the realization.

The branch-and-bound optimum algorithm, as well as an approximative variant, based on multivaluated function minimization are described.

12.7.1 Previous Work

The function decomposition procedure can be either to encode directly the initial map (which leads to having the same codes for many input columns), or first to minimize the number of symbols S_i for columns C_j and follow it with the symbol encoding process. The aim of these methods is, in general, to minimize the semiconductor area in the selected realization technology or some other measure of decomposed function complexity. Various FSM assignment algorithms exist in the literature and can be adopted for these tasks, [?, ?, ?, ?, ?, ?, ?, ?, ?].

Application of the principle of symbol minimization results from the following conviction, that is assumed by most of the systems: "the more the symbols, the more complicated is the realization, hence there are more intermediate variables, and therefore the realization of both block functions is more complicated". Practical examples bear, however, evidence that the map with the minimum number of symbols will not necessarily yield the circuit realization with the minimum total complexity. The symbol minimization is useful, so somehow it must be incorporated to the entire decomposition process, but may be not to the most extent, usually only some reduction of the set of symbols gives the exact optimum solution. Instead of minimizing the number of symbols (compatibles) it is more reasonable to minimize the number of intermediate variables and next, as a secondary criterion, the complexities of functions G and H.

The next design stage is the encoding of the symbols. The graph embedding approach [?, ?], the "column-based" approach [?], and the partition theory [?, ?, ?, ?] are the best known methods applied in the past for state machine assignment and these methods give inspiration to create new algorithms for encoding in decomposition. It is our opinion that the exact minimum solution can be obtained only when the methods based on the partition theory are used. We will, therefore, briefly review these methods. First, it must be mentioned that many of them generate the minimum solution assuming, again, the condition of the minimization of the number of intermediate variables. In such a case for K symbols the partitions of total number k are selected, where $2^k - 1 < K \leq 2^k$. This leads to the concept of *proper partitions,* defined as the two-block partitions in which the number of symbols in the larger block does not exceed 2^{k-1}. The partitions are then selected among the set TP of proper partitions.

Last but not least, the partitions are selected in a way that minimizes the total number of intermediate variables. This condition is expressed by an attempt to select such set TF (called the *final family of partitions*) from set TP of such partitions τ_i, that:

$$\textit{every columns in every block of } \prod_{\tau_i \in TF} \tau_i \textit{ are compatible} \tag{12.13}$$

It is well-known that the minimum solution doesn't have to produce the minimum number of symbols, but also, the minimum number of intermediate variables.

12.7.2 The Principle of Joint Minimization and Encoding of Columns

The method described below replaces two stages:

- minimization of the number of symbols (compatible columns),
- encoding of these symbols.

with one stage of joint column minimization and encoding.

In this joint process certain cost function is optimized, which is related to the realization of the predecessor and successor blocks functions in a selected technology. The goal of this approach is to find a new and efficient method of joint column minimization and encoding, which does not have the previously mentioned weaknesses.

The final family (set) of partitions, TF, is defined as such a family, that the partition product of all partitions from that family satisfies Equation (12.13).

The optimum family, TO, is such a final family that minimizes the value of an assumed cost function.

The family TO in our approach is chosen among all two-block partitions - this gives a possibility of finding better solutions, but is computationally less efficient than using the proper partitions.

The main idea of the method of joint column minimization and encoding proposed here is to select such partitions for the not necessarily column-minimized map, that the two-level AND/OR realizations of the block functions are minimized according to some cost function. Simultaneously, as a byproduct, the number of symbols is reduced (and hence the number of intermediate variables), which results from assigning the same codes to some of the compatible columns (remember that a symbol is a group of columns). This corresponds to the search for such a pair of mappings of the two blocks that their composition is equivalent to the initial mapping of the function and that have the minimum total decomposed function realization.

Separation condition states that each pair of incompatible symbols is separated by at least one partition from TF.

Every set TF that satisfies the separation condition will be referred to as the *final family of partitions.*

A solution to the joint minimization and encoding problem consists of finding the family TO . This family should be such a subset of $T2$, that each of its partitions separates at least one pair of incompatible columns.

12.7.3 Algorithm of Joint Minimization and Encoding of Columns

Relation COMPATIBLE of symbols is defined as the following set of pairs:

$$COMPATIBLE \underset{===}{\overset{def}{}} \{\{a_i,\ a_j\},\ a_i\ \in\ A,\ a_j\ \in\ A \mid \text{ states } a_1 \text{ and } a_j \text{ are compatible}\}. \quad (12.14)$$

A dash (-) denotes the don't care column. $COMB{-}SET(A,2)$ denotes the set of all 2-element subsets of set A. Set $COMPATIBLE$ is found with use of an iterative procedure of comparing columns.

Joint minimization and encoding is realized according to Algorithm 12.7.1.

Algorithm 12.7.1

Begin

E1: Create set $COMPATIBLE$ of pairs of compatible columns;

E2: Find set of pairs of incompatible columns:
$IS = COMB - SET(A, 2) \setminus COMPATIBLE$;

E3: Assign, under the value of variable $T2$, the family $T2$ generated in Algorithm 12.5.3;

E4: Find the set of all such partitions from set $T2$, that separate at least one pair of incompatible columns:
(partition $\tau_j = \{B_1, B_2\}$ separates pair

$$p = \{a_1,\ a_2\} \in S, \quad if \quad (a_1 \in B_1\ and\ a_2 \in B_2)\ or \quad (a_1 \in B_2\ and\ a_2 \in B_1)) \tag{12.15}$$

Delete the remaining partitions from $T2$;

E5: (Option) Sort set $T2$ in such a way that among the partitions with the same length of the shorter block the partitions with smaller value of the cost function stand at first. In another variant, sort set TZ according to another criterion;

E6: Using Algorithm 12.7.2 create the families $TO \in T2$ that satisfy the following conditions:

1. *Condition of separation* of all pairs of incompatible columns

$$(\ \{a_{i1},\ a_{i2}\},\ \in IS)(\exists \tau_j \in TO)[\tau_j \text{ separates the pair } \{a_{i1},\ a_{12}\}] \tag{12.16}$$

   i.e. for each pair of incompatible columns one of its elements is included in one block of partition τ_j, and the other in the second block.

2. *Condition of the minimization of the total cost function CF_r.*

End

Together with step $E6$, generated are the block functions. These are next used at the next stages of the decomposition. According to the selected value of parameter ALL the algorithm finds one or all TO families.

Algorithm 12.7.2

Generation of family (families) TO from family $T2$ by the joint minimization and encoding.

Begin

1. *Creation of initial state of the tree*
   $(GS(0), QS(0), AS(0), PR(0), QF(0)) := (T2,\ \{\},\ IS,\ \overline{1},\ \{\}, 0)$;
   $CF_{min} := \infty$; SOLUTION $:= \{\}$;
   Initialize list $COSTS$;

2. The same as the step 2 in Algorithm 12.5.4, but now $V(N) = (GS(N), QS(N), AS(N), PR(N), QF(N))$;

3. *Extension of state $V(N)$*

   (a) If $GS(N) = \{\}$ then begin delete first element from list $OPEN$; go to 2 end

(b) e := first element from $GS(N)$;

(c) $GS(N) := GS(N) \quad - \quad \{e\}$;

(d) $QS(NN) := QS(N) \quad \cup \quad \{e\}$;

(e) $AS(NN) := AS(N) \setminus \quad \{\{a_i,\ a_j\} \ \in \ AS(N) \mid (e \cdot \ s\ a_i \ = \ a_i \ and\ \overline{e} \ \cdot s\ a_j \ = \ a_j)$ $or \ \ (e \cdot s\ a_j \ = \ a_j \ and\ \overline{e} \ \cdot s\ a_i \ = \ a_i)\}$;

(f) $PR(NN) := PR(N) \ \cdot$ p $\{e, \overline{e}\}$;

(g) $GS(NN) := GS(N) \setminus \{e_1 \in GS(N) | \ (\nexists \{a_i, a_j\} \in AS(NN))$ [e_1 separates $\{a_i, a_j\}$] } ;

(h) If $(GS(NN) = \{\})$ *and* (PR(NN) $\neq \ \overline{0}$) then go to 3; /* backtrack */

(i) Calculate $q(e)$, add news item to $COSTS$ (this step is executed if $q(e)$ is not found in Algorithm 5.1;

(j) $QF(NN) := QF(N) + \ q(e)$;

(k) If $QF(NN) > CF_{min}$ then go to 3; /* backtrack */

4. 4: *Checking if solution*

5. If $AS(NN) = \{\}$ then new solution found.

6. further steps as in Algorithm 12.5.4 with the only difference that

   V(NN) = (GS(NN), QS(NN), AS(NN), PR(NN), QF(NN)).

End

Comment.
Set AS is decreased by those pairs of incompatible columns, that were separated by the partition e (i.e. one column from the pair belonged to one block of partition e and the other column to the second block).

12.8 Fast, Approximate Algorithms for Symbol Encoding and Column Encoding based on Multiple-Valued Logic Minimization

The number of two-block partitions grows very quickly with the increase of the multiplicity index. Therefore, for functions with the multiplicity index of more than ten, the approximate variants of the above two algorithms have been created. One of the possibilities to increase the efficiency is to limit the size of the initial $T2$ set (list) to some of its, heuristically selected, subsets. Again, different variants have been investigated with this respect. Sets of partitions can be selected according to different heuristic (better or less motivated) criteria. The idea is to keep the selected sets of partitions (or sets of blocks) limited in size at each step. In later stages, the sums or products of the selected sets of partitions are taken as an initial set $T2$, and this set is ordered according to some other heuristic rules. The product of partitions from the selected set $T2$ must be a zero partition (in encoding of column-minimised tables), or the selected partitions must satisfy the separation condition (when column-nonminimized table is encoded). If not, arbitrary other partitions are added to the set QS of partitions (we use the notation from previous chapters), to satisfy these conditions. Below, one of the methods of selecting the initial, limited set $T2_{red}$ will be presented.

Each cell of the table of the successor block can be described as an ordered pair:

$(symbol; inputs)$.

where:

symbol is a symbol of the column,

inputs is a 0-cube (minterm) of input variables of H (symbols 0 and 1 only).

The cells can be joined into larger groups (multivaluated implicants or M-implicants), which leads to the minimization of the number of such groups, and hence to the minimization of block functions. The M-implicant is a pair (block_of_symbols; inputs_cube), where *block_of_symbols* is a set of column symbols and *inputs_cube* is a cube of input signals (symbols 0, 1, X).

In the case of column-minimized table, the rules of merging cells are the following:

RULE 1:

If for all cells ({A}; X_i), ({B}; X_r) where $X_r \sqsubseteq X_i \bowtie X_j$
it holds that

$$[(\delta(A, X_r) = \delta(B, X_r)) \; or \; (\delta(A, X_r) = -) \; or \; (\delta(B, X_r) = -)] \tag{12.17}$$

then

$$(\{A\}; X_i) + (\{B\}; X_j) = (\{A, B\}; X_i \bowtie X_j). \tag{12.18}$$

Operation for merging, called *supercube*, and denoted by $\bowtie$, is defined for cubes x and y as follows:

$$x \bowtie y = (x_1, ..., x_n) \bowtie (y_1, ..., y_n) = (z_1, ..., z_n) \tag{12.19}$$

where:

$$\forall \; i = 1, ..., n \; z_i = x_i \; if \; x_i = y_i; z_i = X \text{ otherwise.} \tag{12.20}$$

Let us observe, that this is a general rule for merging, and it is possible in the above definition that $A = B$ or $X_i = X_j$.

Let X_i and X_j be arbitrary cubes and $set1$, $set2$ be sets of symbols. The above rule can be now extended as follows:

$$(set1; X_i) + (set2; X_j) = (set1 \cup set2; X_i \bowtie X_j), \tag{12.21}$$

if for all cells ({s}; X_r), ({r}; X_v)
where

$$s, r \in set1 \cup set2 \ \& \ X_r,\ X_v \sqsubseteq X_i \bowtie X_j \tag{12.22}$$

$$[(\delta(s,\ X_r) = \delta(r,\ X_v)) \ or \ (\ \delta(s,\ X_r) = -\ ) \ or \ (\delta(r,\ X_v) = -)] \tag{12.23}$$

The M-implicants created with RULE 1 will be called *group M-implicants.* Symbol δ denotes a function with vectors of outputs as values.

For a single output variable, the rule is:

RULE 2:

$$(set1; X_i) + (set2; X_j) = (set1 \cup set2; X_i \bowtie X_j) \tag{12.24}$$

if for all cells $(\{s\}; X_r)$, $(\{r\};\ X_v\ )$
where

$$s, r \in set1 \cup set2 \ \& \ X_r,\ X_v \sqsubseteq X_i \bowtie X_j \tag{12.25}$$

$$[\lambda_o(s,\ X_r) = \lambda_o(r,\ X_v) = 1 \ or \ \lambda_o(s,\ X_r) = - \ or \ \lambda_o(r,\ X_v) = -] \tag{12.26}$$

where $\lambda_o(s, X)$, $o = 1, ..., m$ is the value of output signal with number o from the value of $\lambda(s,\ X)$.

RULE 2 is applied for each o, $o = 1, ..., m$ separately. The M-implicants created with RULE 2 will be called *single output M-implicants,* for function y_1, ..., y_m respectively.

Minimization of the total number of M-implicants with RULE 1 and RULE 2 leads to the minimization of the block functions. Minimization of the number of M-implicants is a problem of multioutput, multi-valued function minimization. This problem can be solved using the minimum or approximate methods. In the minimum method all possible M-implicants are generated, and next the covering problem is solved: to find the minimum number of M-implicants that cover all cells and include the minimum number of blocks. An efficient algorithm for this kind of covering problem is proposed in [?]. Below, an approximative variant is presented.

Definition 12.8.1 *M-implicant MI is* row-essential *if there exists row X_i , such that cell ($\{s\}$; X_i) is covered by MI and each cell ($\{r\}$; X_i) is either covered by MI and $\delta(r,\ X_i) = \delta(s,\ X_i)$ or $\delta(r,\ X_i) \neq \delta(s, X_i)$.*

Definition 12.8.2 M-implicant *MI is* column-essential *if there exists column s, such that cell ($\{s\}$; X_i) is covered by MI and each cell ($\{s\}$; X_j) is either covered by MI and $\delta(s,\ X_j) = \delta(s,\ X_i)$ or $\delta(s, X_j) \neq \delta(s, X_i)$.*

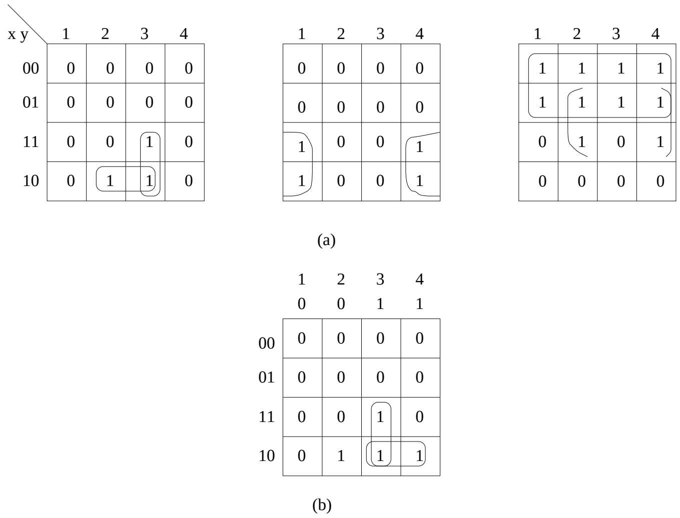


Figure 12.19: Fig. 9.5.1. Function for Example

In the presented approach the row-essential implicants are selected at first, and the covered by them cells are removed from the successor map. Next the procedure is repeated with secondary row-essential M-implicants, until all cells are covered, or no more row-essential implicants exist. In the last case procedure is repeated for column-essential implicants. If no any essential implicants exist, arbitrary implicant is selected (or branching process is executed, as usually).

Example 12.8.1

The row-essential M-implicants for function from Fig. 12.19 are presented in Fig. 12.21a.

They are:

$$(\{A,G\};0X),\quad (\{E,H\};0X),\quad (\{A\};1X),\quad (\{E,G\};1X),$$

$$(\{B,C,D,F\};01),\quad (\{B,C,D,F\};11)\quad (\{C,D\};00)\quad (\{C,D,H\};10).$$

Now M-implicants ({B,F}; 00) , ({B,F}; 10), and ({H}; 11) are selected as secondary column-essential M-implicants. All cells are covered. See Fig. 12.21b.

The finally generated set of blocks for transitions is:

$$TRB\ =\ \{\{A,G\},\{E,H\},\{E,G\},\{A\},\{B,C,D,F\},\{C,D\},\{B,F\},\{H\},\{C,D,H\}\}.$$

Blocks from $TROUB = TRB$ are used to create family $TROUP$ of partitions. This is a step-by-step tree searching process equipped with many heuristics, the goal of which is to create such partitions that:

- the blocks from $TROUB$ are the blocks of those partitions or products of the blocks of these partitions,
- the number of partitions is the minimum,

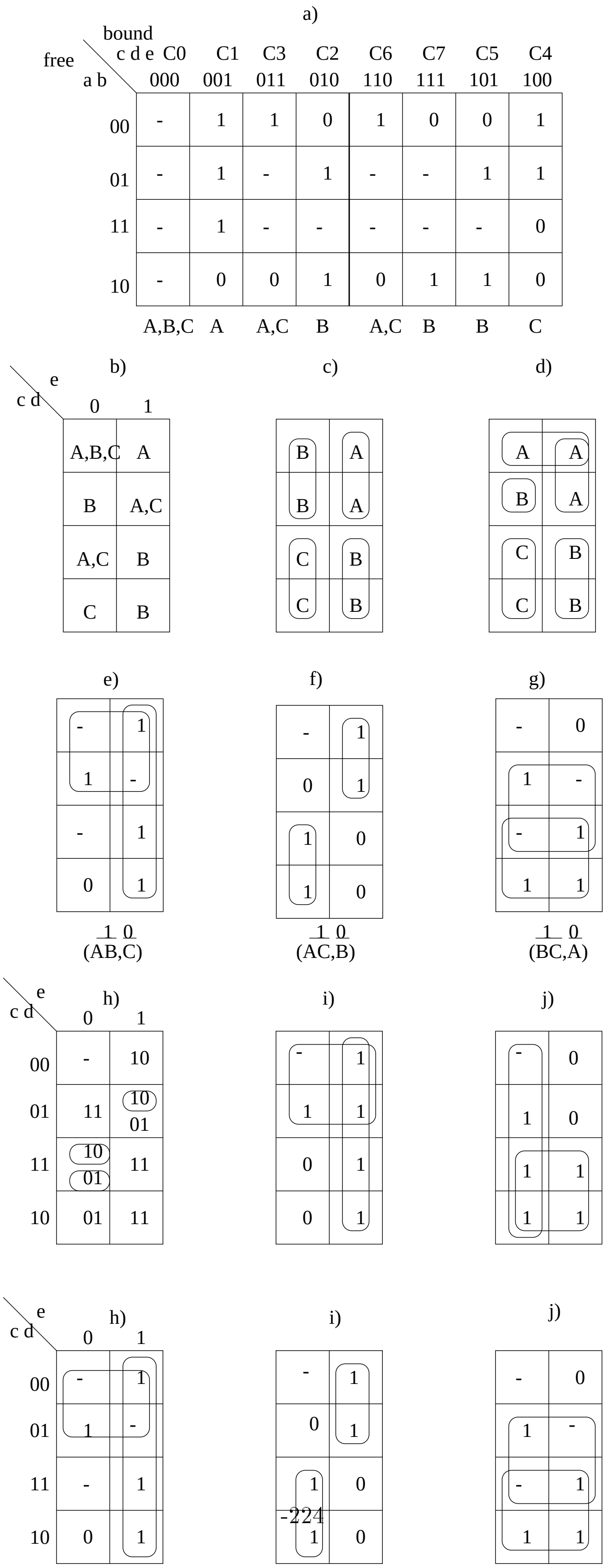


Figure 12.20: Fig. 9.5.1.XXXXXXX Function for Example

Figure 12.21: Fig. 9.5.2. The row-essential M-implicants for function from Fig.

Figure 12.22: Fig. 9.5.3. Graph GCB for set MTROUB for Example 8.2

.

- the separation conditions are satisfied.

The algorithm of creating set $TROUP$ of partitions from sets TRB and OUB is the following.

Algorithm 12.8.1

Begin

1. $TROUB := TRB$;
2. Create set $MTROUB$ of maximal blocks from $TROUB$, by removing blocks included into other blocks:

$$MTROUB \ := \ \{s \in TROUB \mid (\neg \exists \ r \ \in \ TROUB)[\ s \sqsubseteq \ r]\}. \tag{12.27}$$

3. Create graph GCB of consistency for set $MTROUB$. Its nodes correspond to blocks from $MTROUB$. A non-oriented edge exists between two nodes if the corresponding blocks are non-disjoint.
4. The (quasi)minimum coloring of graph GCB is found.
5. Groups of nodes with the same color are used to create set of partitions $TROUP$.
6. Additional partitions are added to $TROUP$, to satisfy the condition:

$$(\forall B \in TROUB)(\exists\{\tau_1, ..., \tau_b\} \in TROUP, b = 1, ..., CARD(TROUP))[B \in \{\tau_1 \cdots \tau_b\}] \tag{12.28}$$

   i.e. each block from $TROUB$ must be either a block of a partition from $TROUP$ or a product of some partitions from $TROUP$.

End

Example 12.8.2

For the map from Fig. 12.19a, the following set can be found:

$TROUB \ = \ \{ \{A\}, \{A, G\}, \{B, C, D, F\}, \{B, F\}, \{C, D\}, \{E, H\}, \{E, G\},\{H\}, \{C, D, H\}, \{A, B, G, F\},\{A, D, E, F, (G$
$\}.$

$MTROUB \ = \ \{\{B, C, D, F\}, \{E, H\}, \{C, D, H\}, \{A, B, G, F\}, \{A, D, E, F, G\}\}.$

The graph GCB for $MTROUB$ is shown in Fig. 12.22a.

After the coloring, shown in Fig. ??b , (FIG 30) the partitions:

$$\{\overline{ABGF}, \overline{CDH}, (E)\}, \quad \{\overline{BCDF}, \overline{EH}, (AG)\}$$

are found. States in parantheses can be added to each of the blocks.

Suppose, that partitions:

$$\{\overline{ABGF}, \overline{CDEH}\}, \{\overline{BCDF}, \overline{AEGH}\}$$

are selected.
Then:

$$\{\overline{ABGF}, \overline{CDEH}\} \cdot \{\overline{BCDF}, \overline{AEGH}\} = \{\overline{BF}, \overline{AG}, \overline{CD}, \overline{EH}\}.$$

Blocks {A,B,G,F}, {B,C,D,F}, {B,F}, {A,G}, {C,D}, {E,H} are then removed from $TROUB$. The remaining blocks are:

$$\{\{A\}, \{H\}, \{E, G\}, \{C, D, H\}, \{A, D, E, F, (G)\}\}.$$

If $\{\overline{ADEF}, \overline{BCGF}\}$ were selected, then product of all selected partitions is zero, and $TROUB$ reduces to { {E,G}, {C,D,H} }. Another good candidate is $\{\ \overline{EG}, \overline{CDH}, (ABF)\}$. For instance, $\{\overline{ABEG}, \overline{CDFH}\}$ can be selected from it.

Finally, the generated set $TROUP$ is:
$\{\{\overline{ABGF}, \overline{CDEH}\}, \{\overline{BCDF}, \overline{AEGH}\}, \{\overline{ADEF}, \overline{BCGH}\}, \{\overline{ABEG}, \overline{CDFH}\}\ \}$.

It must be noted, that depending of the allotted amount of computer time, larger sets $TROUP$ of partitions can be generated from the same data, for instance all partitions greater than the partitions:

$$\{\overline{ABGF}, \overline{CDH}, (E)\}, \{\overline{BCDF}, \overline{EH}, (AG)\}, \ and\ \{\overline{ADEF}, \overline{BCH}, (G)\}.$$

The creation of set $T2_{red}$ of partitions is done according to the following algorithm.

Algorithm 12.8.2

Begin

1. Find the set $TROUP$;
2. $T2_{red} := TROUP$;
3. Sort set $T2_{red}$ according to some criteria: the partitions, that arised in many sets, or that had highest priorities in these sets - obtain highest priorities.

End

Example 12.8.3

For a function from Fig. ?? the blocks cover is:

$$\{(\{1, 2, 3, 4\};\ 0X),\ (\{2\};\ X1),$$

$$(\{1\};\ 1X),\ (\{4\};\ 10),\ (\{4\};\ 11)\},\ (\{2\};\ 10),\ (\{3\};\ 1X)\}.$$

Figure 12.23: Fig. 9.5.4. The cover of the function from Fig......

The output blocks are: {2,3} {2,4}.

$MTROUB$ = { {2,3}, {2,4} }. (Block {1,2,3,4} = A is removed). Partitions $\{\overline{14},\ \overline{23}\}$ and $\{\overline{13}, \overline{24}\}$ are created. Their product includes all blocks from $TROUB$.

Then

$$T2_{red} \ = \ \{\{\overline{13},\ \overline{24}\}, \{\overline{14},\ \overline{23}\}\}.$$

Example 12.8.4

For function from Fig. ??:

$$TRB \ = \ \{\{\overline{0},\ \overline{123}\}, \{\overline{1}, \overline{023}\}, \{\overline{2},\ \overline{013}\}, \{\overline{3},\ \overline{012}\}\}.$$

TRB is taken as a $T2_{red}$ family, but partitions $\{\overline{2},\ \overline{013}\}$, $\{\overline{3},\ \overline{012}\}$, and $\{\overline{1},\ \overline{023}\}$ obtain highest priorities, since they occur most often.

In the case of row non-minimized function, to the set $TROUB$ added is also set TRC of blocks corresponding to the groups of compatible symbols, merged into M-implicants while creating the cover.

Example 12.8.5

For function from Fig. ??a the symbols 0,4 and 1,2 are compatible.

The cover is:

$\{ (\{0\};XX),\ (\{3\};01),\ (\{3\};10),\ (\{1,2\};01),\ (\{2\};10),\ (\{3\};00),\ (\{3\};11),\ (\{1,2\};00),\ (\{1\};11),\ (\{4\};XX)$ }.

(see Fig. 12.23.)

$TRC \ = \ \{\{1,2\},\ \{0,4\}\}.$

$TROUB \ = \ \{\{1,2\},\ \{1,2,3\},\ \{0,1,2\},\ \{0,4\},\ \{1\},\ \{2\},\ \{3\}\}.$

$MTROUB \ = \ \{\{1,2,3\},\ \{0,1,2\},\ \{0,4\}\}.$

Partitions $\{\ \overline{123},\ \overline{04}\}$ and $\{\overline{012},\ \overline{34}\}$ are created.

$$\{\overline{123},\ \overline{04}\} \cdot \{\overline{012},\ \overline{34}\} \ = \ \{\overline{12},\ \overline{0},\ \overline{3},\ \overline{4}\}.$$

Columns 1 and 2 are compatible. The separation condition is satisfied, then these partitions form a TF family.

The cost function takes into account both the complexity of the successor function and the complexity of the predecessor function.

The cost of the final family of partitions may be calculated with respect to the minimum complexity two-level AND/OR realization. Other realization of networks can be also used, if the respective cost and quality functions were defined and the respective programs written.

The approach presented in section 5.6 allows for the exact solution to the single decomposition with a given bound set. Similarly, as in the method from section 5, simultaneously with the minimization and the encoding, the next synthesis stage is here also resolved, i.e., the calculation and the minimization of the block functions. Similarly, as in section 12.3, it is assumed that the block functions are realized either separately or jointly.

Because of the high minimization capabilities of the presented algorithms, the respective programs are computationally not efficient. They are, however, useful for design of functions with small (less than 10) multiplicity index. This is useful when decompositions with small μ are created. It can be also argued, that since algorithms of this kind run only once for each Machine Learning process for which decomposition is applied, it is reasonable to run a computer even for a whole weekend to calculate the corresponding minimum solution. Also, minimum algorithms are required to produce data for comparing different approximative algorithms. Hence we started with an exact minimum algorithm.

The more efficient approximative variants of the methods have been next constructed, and are based on the general notions introduced here. They make use of various search heuristics and limitations of the search space [?]. Not all partitions are generated in them. We start from some partitions and calculate GP-implicants' coverings. They imply sets of other partitions, that can possibly minimize the cost function. The procedure iterates until the set of the selected partitions satisfies all conditions from step $E6$ of Algorithm alg:Algorithm 8.4.1 = 7.1. These algorithms, as well as the comparison of results, will be reported in next sections of this report.

ab \ cdef	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0
01	1	0	1	1	0	0	0	0	0	0	1	1	1	1	0	0
11	1	0	1	1	0	0	0	0	0	0	1	1	0	0	0	0
11	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	1
	A	B	A	A	B	B	B	C	C	C	A	A	D	D	C	C

Figure 12.24: Decomposition table for Example 8.1.

12.9 Symbol Encoding Based on Dychotomies

In this section, we will present one more new method for optimizing both the predecessor logic (output encoding) and the successor logic (input encoding).

This method takes some of the ideas from the input encoding techniques that use dichotomies [?], and some ideas from the output encoding using dominance relationships from [?]. These two methods are combined together for our purpose of solving both these problems concurrently.

12.9.1 The function for Example 8.1

Before explaining the encoding methodology, we will first present an example that will be used in the presentation of our technique.

Consider the following logic function in the initial map, Fig. 12.24.

Notice that the columns correspond to the Bound set and the rows correspond to the Free Set, as previously described. In this example, the columns have already been colored, with symbols A, B, C, D. It is these colors (symbols) that we will try to assign the codes to. The presented method is a symbol encoding technique.

12.9.2 Output Encoding

The single most important idea of our method for output encoding is the concept of dominance [?].

Definition 12.9.1 *A binary representation of a symbol S_1* **dominates** *another symbol S_2 if for each bit '1' in S_2 there is a corresponding bit '1' in symbol S_1.*

The following examples illustrate this concept.

$S_1 > S_2$ for $S_1 = 01011$ and $S_2 = 01001$

$S_1 > S_2$ for $S_1 = 0110$ and $S_2 = 0100$

Why do we want to use the idea of dominance in encoding? It is simple. By developing dominance constraints we constrain how the codes can be assigned to the symbols. For instance, having four variables to encode and assuming that $B > A$, $C > B$, and $D > C$, the following encoding would satisfy these constraints:

$$A = 000 \qquad B = 010 \qquad C = 110 \qquad D = 111$$

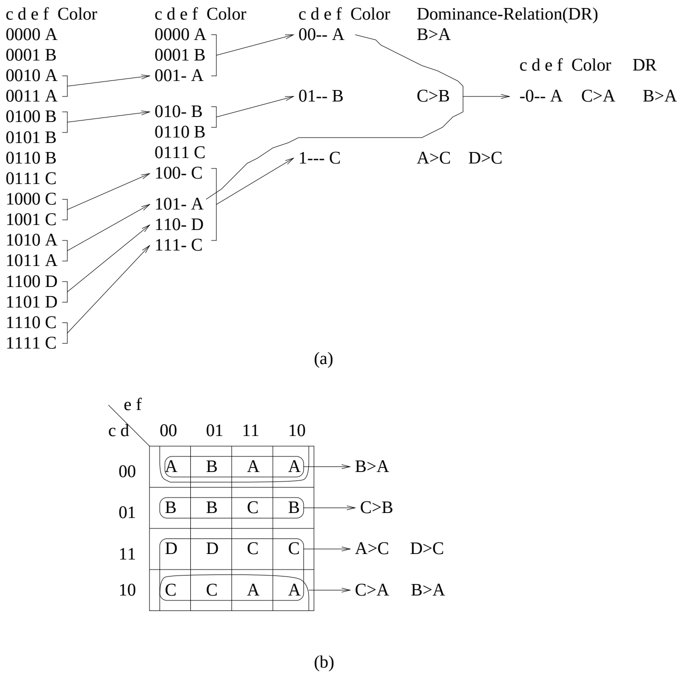


Figure 12.25: Boolean minimization of function using dominance relations for Example 8.1: (a) Tabular form of creating implicants and dominance relations, (b) Kmap for creating implicants and dominance relations.

Let us first consider Figure 12.24 which shows the cubes for input variables c, d, e and f, and also shows the corresponding colors or intermediate symbols. Output encoding of these cubes and associated outputs will be now performed. To do this, the cubes and their outputs are compared, and the cubes will be merged that have the same outputs. For example, cube 0010 associated with symbol A, and cube 0011 associated with symbol A, can be merged into cube 001- associated with symbol A. This operation can be performed again on the cubes for symbols B, C, and D.

When the preliminary minimization of this cover has been completed, these cubes can be tried to be optimized even further. This is done by merging corresponding cubes with don't cares. Take cube 0000, associated with A, and cube 001 associated with A. Merging these two cubes can only occur by using some other cube as a don't care. The only other cube that can be used to merge these two is 0001 associated with B. Because of this operation, it can be now stated that symbol B dominates symbol A. Similarly, for the other cases, symbol C dominates symbol B, C dominates A, A dominates C, and D dominates C. The graphical interpretation of creating groups and domination relation is shown in Figure 12.25.

Two approaches are possible with respect to finding the dominance relations. The first approach finds only as many dominance relations that there are no conflicts between the relations. Another approach generates all the possible dominance relations and then constructs a dominance graph showing all the dominance relations.

- For the first approach we would only find enough dominance relations so that they cover all of the symbols and be not in conflict. For example, $B > A$, $C > B$, and $D > C$ would cover the symbols, and looking for other dominance relations would be unnecessary. This is a fast way of generating the dominance graph for output encoding.

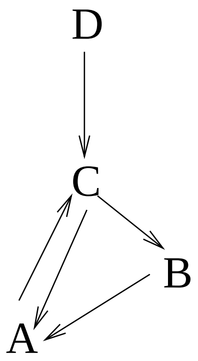


Figure 12.26: Dominance graph for Output encoding with all possible relations for Example 8.1.

- The second approach would be to generate all possible dominance relations, and then to draw the dominance graph for them. Next select from the graph only those relations which are allowed by the following rules. Note that the solution obtained by removing relations from the graph is not necessarily an optimal one. There are many possible solutions for larger graphs. For example, $B > A$, $C > B$, $D > C$ can be one solution, and $B > A$, $B > C$, $A > D$ could be another solution. Obtaining of the final solution using the rules is arbitrary. It may be then a good idea to use the first approach to find the dominance relations instead of finding all possible relations.

We will experiment with these two and also possible approaches to solve this problem.

In case that a decision to generate the complete graph has been done, then the following procedure is observed.

To solve this graph, the conflicting relations have to be removed. This graph can possibly contain conflicting relations, that is, both $A > C$ and $C > A$. Clearly, these cannot both occur simultaneously, because that would imply that the encoding for A and C would be the same. That would be the only way both relations were satisfied, and this is not something that we want.
The dominance graph is shown in Figure 12.26.

The rules applied in removing some of the relations in order to avoid any conflicts are the following:

1. There should be no direct cycles. That is, $A > B$, $B > C$, and $C > A$ is a cycle and is illegal.

2. For an n-bit encoding length, there cannot be a path created from more than n similarly directed relations. That is, $B > A$, $C > B$, $D > C$ for an encoding length of 2 is illegal.

Using these rules, the graph from Fig. 12.27 is drawn for our example. Notice that the conflicting relation from A to C for ($A > C$) has been removed. This would have caused a direct cycle in the graph had it be left there.
Now that the dominance graph has been solved, one can proceed to the input encoding part of the algorithm. One more item needs to be mentioned here. It is possible that the solution that has been obtained from the dominance graph will not fit with the input encoding results, so that it may be necessary to obtain a different solution from the dominance graph in order to obtain a better combination of the two partial encoding methods.

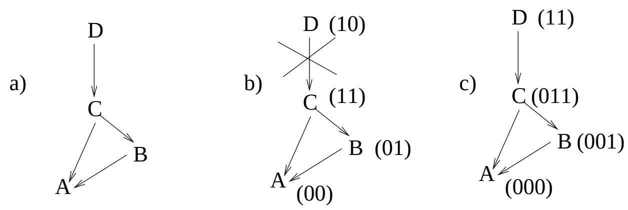


Figure 12.27: The Dominance graph after removal of conflicting relations for Example 8.1.

12.9.3 Input Encoding.

The input encoding algorithm presented here is based upon dichotomies [?].
We first start out with some definitions.

Definition 12.9.2 *A* **dichotomy** *is a disjoint 2-block partition (left:right) of a subset of column indexes of the reduced constraint matrix M. The left side of the partition contains the column indexes of the positions with 0's, and the right side contains the column indexes of the first position with a 1 for that dichotomy. Other dichotomies for the same row will get the other 1's in that row.*

Definition 12.9.3 *A dichotomy D_i=(l_i:r_i) is said to* **cover** *dichotomy D_j=(l_j:r_j) if l_j is a subset of l_i and r_j is a subset of r_i, or if r_j is a subset of l_i and l_j is a subset of r_i.*

Definition 12.9.4 **irredundant dichotomies** *are such dichotomies that are not covered by other dichotomies.*

Definition 12.9.5 *Two dichotomies D_1=(l_1:r_1) and D_2=(l_2:r_2), are said to be* **incompatible dichotomies** *if there exists an unordered pair of indexes (j,k), j,k are elements of l1 or j,k are elements of r_1, such that j is an element of l_2 and k is an element of r_2, or if the same is true with the role of D_1 and D_2 reversed. Otherwise the two dichotomies are considered* **compatible**.

Definition 12.9.6 **A union of two dichotomies**, *D_1=(l_1:r_1) and D_2=(l_2:r_2), denoted $D_1 \cup D_2$, is a dichotomy, D=(l:r), such that l=$l_1 \cup l_2$ and $r = r_1 \cup r_2$, or $l = l_1 \cup r_2$ and $r = r_1 \cup l_2$.*

Definition 12.9.7 *A* **prime dichotomy** *is a dichotomy which is not included in any other dichotomy.*

Definition 12.9.8 *A* **seed dichotomy** *is an elementary dichotomy found in the constraints matrix.*

Definition 12.9.9 *A* **distinct seed dichotomy** *is a seed dichotomy which is covered by one and only one prime dichotomy.*

Definition 12.9.10 *An* **essential prime dichotomy** *is a dichotomy which contains at least one distinct seed dichotomy.*

To perform the input encoding, first binary vectors must be assigned to each of the colors in Positional Cube Notation (PCN). (This notation is also called the one-hot notation and is shown to the right of

a b	Color	PCN	output
00	A	1000	0
01	A	1000	1
10	A	1000	1
11	A	1000	0
00	B	0100	1
01	B	0100	0
10	B	0100	0
11	B	0100	0
00	C	0010	0
01	C	0010	0
10	C	0010	0
11	C	0010	1
00	D	0001	0
01	D	0001	1
10	D	0001	0
11	D	0001	1

a b	MPCN	Output
00	1011	0
01	1001	1
10	1000	1
11	1100	0
01	0110	0
10	0111	0
00	0100	1
11	0011	1

Figure 12.28: Intermediate variables with assigned positional cube notation, and minimization.

the corresponding color). Then the multiple-valued minimization of the cover is performed (the binary inputs). This is shown in Fig. 12.28.

Notice that the table has been reduced by taking rows with similar inputs (a and b) and outputs and merging the cubes for the colors, that is, the positional cube notation. For example, the first rows for colors A, C, and D can be merged to form a row with inputs 00, output 0 and MPCN (merged positional cube notation) 1011. That is the OR of A, C, and D is 1011. This is done repeatedly until all possible mergings have been found. Now, the processing of this cover can be continued.

The cover that we are interested in is for the merged PCN in Fig. 12.29. According to [?] this cover can be reduced further by deleting the rows with single 1s, with all 1s, duplicated rows, and the meet rows, that is, the rows which are the ORs of two or more rows.

One can notice that the 1000 row can be removed, because it has only one 1. The same can be said of the 0100 row. The 1011 row can be deleted because it is the merging of rows 1001 and 0011. The same can be said of 0111 because it is the merging of rows 0110 and 0011. This produces the minimized cover from Fig. 12.30.

Now the cubes from the final cover are used to generate dichotomies based on the definitions presented before. For example, 1001 generates the dichotomies (AD:B) and (AD:C) because it has 1's in the first and fourth columns, and '0s in the second a third columns.

The introduced above definition of irredundant seed dichotomies is now used to extract those dichotomies that are not subsets of other dichotomies. These dichotomies would be used to complete the solution. In this particular example all of the dichotomies are irredundant.

MPCN

ABCD
1011
1001
1000
1100
0110
0111
0100
0011

Figure 12.29: Initial Cover for Input Encoding

	Cover	**Dichotomies Generated per Row**
ABCD	1001	(AD:B) (AD:C)
	1100	(AB:C) (AB:D)
	0110	(BC:A) (BC:D)
	0011	(CD:A) (CD:B)

Figure 12.30: Final Minimized Cover for Input Encoding with Associated Dichotomies.

Prime Dichotomies	(14:2)	(14:3)	(12:3)	(12:4)	(23:1)	(23:4)	(34:1)	(34:2)
(123:4)				X		X		
(124:3)		X	X					
(12:34)			X	X			X	X
(14:23)	X	X			X	X		

Figure 12.31: All Possible Prime Dichotomies and Their Relation to the Seed Dichotomies.

	V_1	**Color**
A	0	0
B	0	1
C	1	1
D	1	0

Figure 12.32: Encoding assigned to intermediate variables based upon input encoding methodology.

Now all possible prime dichotomies are generated. For this example, they are listed in Fig. 12.31.

Finally, the crucial step of combining the output and input encoding steps is executed. In order to take into consideration the dominance relationships from the output encoding stage, the possible colorings for the dichotomies in the input encoding are considered.

To find the colorings, the following facts are taken into consideration. There are some seed dichotomies that have only one intersection with a prime dichotomy. This means that these prime dichotomies are essential dichotomies and must be included in any coloring of the dichotomies. For this example, there are two essential prime dichotomies (AB:CD) and (AD:BC). Since these two dichotomies also cover all of the seed dichotomies, then they are also a complete coloring of all of the seed dichotomies. For other examples, there may be more than the one possible coloring, and all these colorings would need to be determined.

Once the colorings (AB:CD) and (AD:BC) have been determined, the solutions are converted to encodings that satisfy the colors A, B, C, and D. The conversion is performed by making the indexes in each dichotomy correspond to the rows of the following table. There is some flexibility in terms of whether one assigns the indexes in the left side of the dichotomy to 0s or 1s. By applying the rules we try to satisfy the dichotomies by selecting the appropriate conversion. In this example, the left side of the dichotomies has been assigned to 0's and the right side has been assigned to 1's (see Fig. 12.32).

Notice that two of the three dominance relations specified earlier ($B > A$, and $C > A$) have been satisfied. It can be verified that B = 01 dominates A = 00, and C = 11 dominates A = 00. $D > C$ was not satisfied, but this is the price to be paid for concurrently optimizing both the predecessor and the successor logic.

Note that having more colors available (other than the single one that was available), one may have

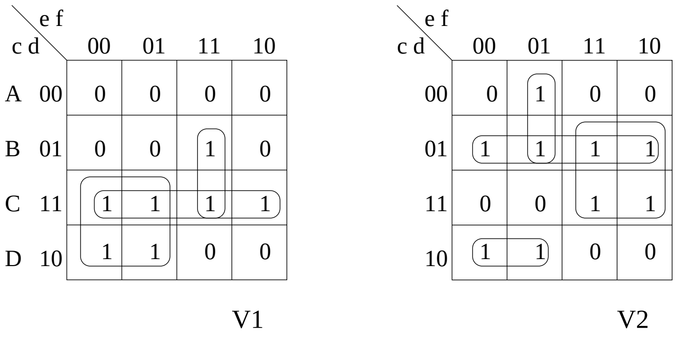


Figure 12.33: *Karnaugh maps of Intermediate Variables for Example 8.1.*

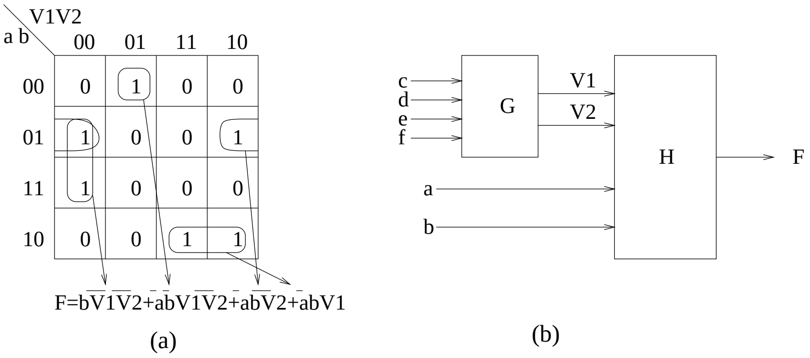


Figure 12.34: Example 8.1: (a) Karnaugh Map of the Output function H, (b) the decomposition diagram.

been able to satisfy also the constraint $D > C$.

Just to reiterate the merging of these two encodings (input and output), the goal is to select the coloring of the dichotomies which satisfies the dominance relations. Remember that to verify that the dominance relations are satisfied, the result of the graph colorings must be converted to actual codes. Only this way will it become obvious that the codes satisfy the dominance relations.

Now the intermediate functions are realized using Karnaugh maps (Fig. 12.33).
From the maps:

$$V1 \; = \; c\overline{e} + cd + d \; e \; f$$

$$V2 \; = \; c\overline{d} \; \overline{e} + \overline{c} \; d + \overline{c} \; \overline{e} f \; + \; d \; e$$

Notice that the dominance relation $B > A$ is used to create the group in the first row for V_1. Also, the dominance relation $C > B$ is used to create a group in the second row for V_2.

Finally, we create the Karnaugh map for the output function.
From the map, we obtain:

$$F \; = \; b \; \overline{v_1} \; \overline{v_2} \; + \; \overline{a} \; \overline{b} \; \overline{v_1} \; v_2 + \overline{a} \; b \; \overline{v_2} \; + \; a \; \overline{b} \; v_1$$

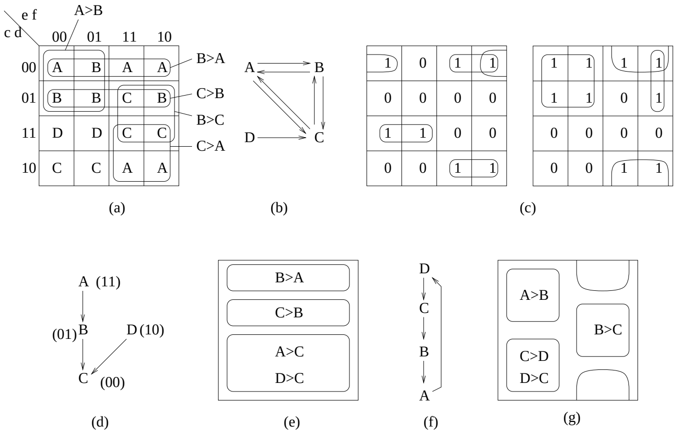


Figure 12.35: More examples of finding implicant covering and Dominance Relations for Example 8.1: (a) the Kmap with groups shown, (b) the complete initial Domination Graph, (c) the functions $V1$ and $V2$, (d) - (g) other variants.

The Kmap for block H and the final decomposition are shown in Figure 12.34.

More examples of finding implicant covering and Dominance Relations for Example 8.1 are shown in Figure 12.35.

A systematic method of evaluating partitions, similar to those from previous chapters is shown in Figure 12.36. Groups AB, AC, AD, etc near arrows correspond to blocks of partitions encoded with one. The costs of these partitions are calculated using Espresso minimization performed on respective Kmaps. The costs of prime implicants are given near the Kmaps. For instance, 223 near the first map means that when A and B are encoded with 1 there are three implicants and their costs are 2,2, and 3. The total cost is 2+2+3 = 7.

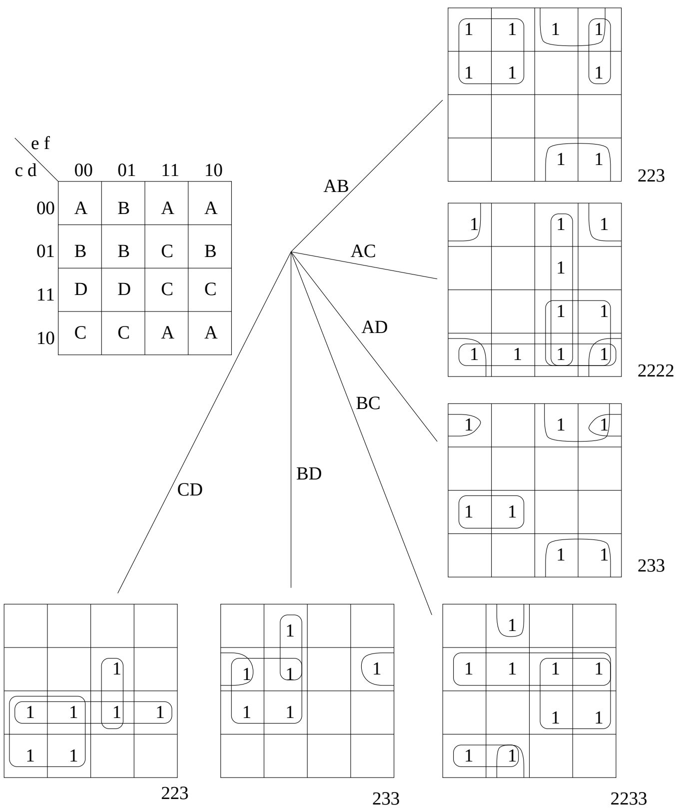


Figure 12.36: A systematic method for evaluating partitions.

12.9.4 Dominance Relation Construction for G Block

The predecessor and sucessor blocks (G and H blocks) can be synthesized with a concurrent input/output encoding method in which dominance relations among the symbolic outputs (columns C_i) of G are obtained and used to simplify the constrained input encoding problem for the H function. In the following a procedure is proposed to obtain the dominance relations and the associated acyclic graphs introduced in the previous section. (the problems of domination relations are presented in [?]).

Example 12.9.1

For the map from Example 12.9.1 one obtains the following specification of function G with symbolic output:

0000 A
0010 A
0011 A
1010 A
1011 A

0001 B
0100 B
0101 B
0110 B

0111 C
1000 C
1001 C
1110 C
1111 C

1100 D
1101 D

The procedure starts with an assignment of colors to the compatible columns of the decomposition chart (cofactors of the bound set variables). In the presented example, the four compatible classes have been colored with A,B,C, and D. The binary enumeration of each column will be treated as a minterm in the 4-valued ON-set of G, i.e. minterms 1100 and 1101 produce value D at the output.

An initial choice for a starting color must be made. Our approach here is based on a heuristic: the groups of minterms (colors) are processed top to bottom, starting with the lower (decimal) valued minterms first and ending with the higher valued ones for the last color D.

Another heurstic may be to select colors that are used in most minterms.

The ON-set of the symbolic output A will be minimized first, with the ON-sets of the following outputs B, C, and D as don't cares. Borrowing of minterms from the ON-sets of B, C, and D is used to achieve a minimal cover for A: eg. borrowing 0001 from the ON-set of B and 1000,1001 from ON-set of C results in a 2-cube cover for value A:

00– A

10– A

Next, the intersection of minterms for D (not 'borrowed' from) and the above cover for A is computed; if it is empty D does not dominate A directly and the corresponding nodes in the dominance graph are not connected. Otherwise, the connection is made as in the Dominance Graph from Fig. 12.37.

The dominance graph now has 3 of the 4 symbols, with only one cover computed (for A). The covers for B and C are computed next to establish the position of D in the graph.

Borrowing from the minterms of C, the B cover is:

```
  C
 / \
B   |
\  /
 A
```

Figure 12.37: The initial Dominance Graph for Example 8.2.

```
     D
    /
   C
  / |
 /  |
B   |
\  /
 A
```

Figure 12.38: The final dominance graph

0001

01–

and the one for C (borrowing both minterms of D):

0111

100-

11–

Since the minterms of D intersect the cover for C , the final dominance graph is as shown in Fig. 12.38:
The pairwise dominance relations for the following input encoding step of the sucessor are:

D > C

C > A

B > A

As we see, the relation C > B had to be removed to have an encoding with the minimum number of bits.

The above, one at a time, introduction of new variables into the graph prevents the formation of cycles. The top-down (from low-valued to high-valued decimal) construction of the cover facilitates the implementation as well.

12.10 Column Encoding Based on Dychotomies

12.10.1 Input-Output Encoding for Simple Disjoint and Complex Disjoint Serial Decompositions

Similarly to the previous section, the encoding methods proposed in this section will be based on combining the input and output encoding methods.

In brief, the Input-Output Encoding Procedure is performed as in the following Algorithm 12.10.1

Algorithm 12.10.1

This algorithm starts from set of columns and corresponding disjoint symbols from graph coloring or clique parititioning. However, symbols of any cliques (also overlapping) can be also used.

1. **Output encoding (G outputs).**
   (a) The output encoding problem for function G is first formulated, and dominance relations are derived for the symbolic variables C_1, C_2,... of G corresponding to the columns of the initial map. (The difference with the method from the previous section is that we are looking here for dominance relations between columns, and not for dominance relations between symbols).
   (b) For each symbolic variable C_i (symbol S_i) find a minimal coer with cubes of the bound set.
   (c) All the input cubes for the column variables following C_i are used as don't cares to produce the above minimal cover. This introduces intersections between the cubes of the cover for C_i and the cubes in covers for other symbolic outputs that follow column C_i.
2. **Binary-multivalued constraint matrix (H inputs).**
   (a) A set of constraints is formulated between the binary variables in the free set and the multivalued variable Column (with values C_1, C_2,...).
   (b) The constraint matrix describes the ON set of the function and since the compatible classes are known at this stage, the don't cares can be assigned only as needed to group columns in a given class, leaving other non-identifying don't cares. Zeroes are assigned for these non-identifying don't cares to increase the number of dichotomies later on. (In other variant we would change all non-identifying don't cares as they are for next states).

   Multivalued minimization of the constraint matrix is not required as in the standard input encoding problem for two-level realization of H.
3. **Seed dichotomy generation.** All pairs of seed dichotomies are formed according to the constraint matrix. Since the multiple-valued minimization is not used, a large number of seed dichotomies is generated, which improves the chance of detecting prime dichotomies without having to add additional 'separating' dichotomies.
4. **Graph of compatible dichotomies.**
   (a) All seed dichotomies which violate the dominance relations for the outputs of G are removed from the list. A dominance relation is not satisfied if any dominating symbols appear in the right part of a dichotomy.
   (b) The graph of all compatible dichotomies is constructed using the following definition of compatibility:

   $(L1; R1)$ is compatible with $(L2; R2)$ iff $L1$ is compatible with $L2$ and $R1$ is compatible with $R2$

de \ abc	C1 000	C2 001	C3 010	C4 011	C5 100	C6 101	C7 110	C8 111
00	X	1	1	0	1	0	0	1
01	X	1	X	1	X	X	1	1
11	X	1	X	X	X	X	X	0
10	X	0	0	1	0	1	1	0
	A	A	A	B	A	B	B	C

Figure 12.39: *Table 1.*

5. **Identification of prime dichotomies.**

   All prime dichotomies are identified and a subset of them is selected such that all seed dichotomies (meeting the dominance conditions) are covered by the members of the set

   dichotomy $(L1, R1)$ covers $(L2, R2)$ iff $L1$ includes $L2$ and $R1$ includes $R2$.

6. **Final code selection.** The final codes for all columns in the decomposition table are formed from the prime dichotomies in the above subset. Identical code words will result for compatible columns. A don't care can be assigned to a code in case a particular column symbol is missing from the corresponding dichotomy.

The following examples 12.10.1 and 12.10.2 will illustrate the above procedure.

Example 12.10.1

The incompletely specified function from Wan Wei (see [?]) will be used. A simple disjoint serial decomposition exists in this case for the free and bound sets, as shown, and the compatible classes have been identified.

STAGE 1. Finding the dominance relations for columns.
Function F is presented in Table 1 from Figure 12.39.
The hand calculation of encoding for this function is shown in Figure 12.40.
Using Clique Partitioning Algorithm from Table 1, we can find all symbols:
Symbol (compatible) A $= \{C_1, C_2, C_3, C_5\}$
Symbol B $= \{C_4, C_6, C_7\}$
Symbol C $= \{C_8\}$

The symbolic relation of G with $C_1, .., C_8$ as outputs is the following:

abc
000 C_1
001 C_1

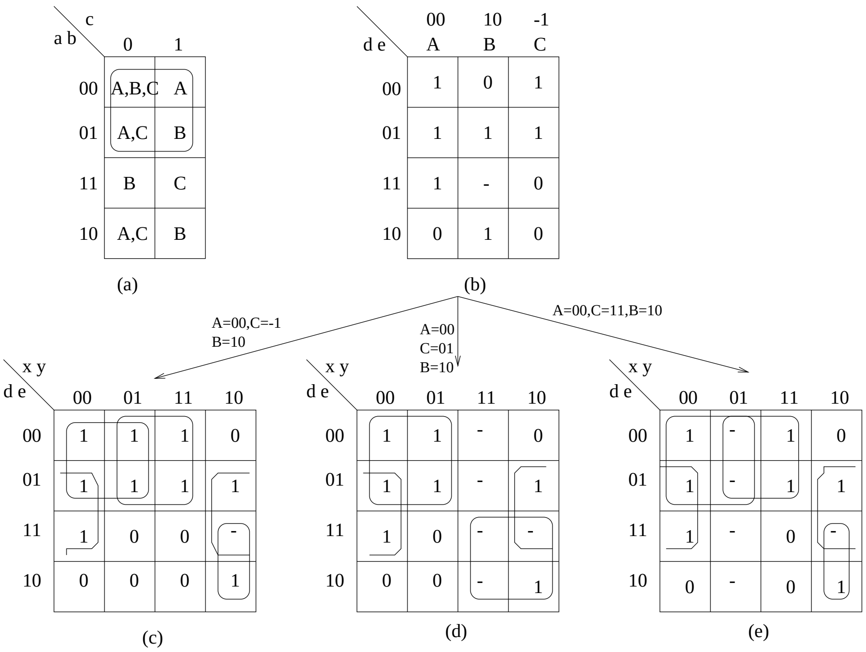


Figure 12.40: *Realization of function from Example 9.1: (a) Symbolic output relation for function G for Kmap from the previous figure, (b) Symbolic Map of function H with encoding A=00,B=10,C=-1, (c),(d),(e) - Kmaps for three different encodings of map from (b)*

010	C_1
100	C_1
000	C_2
001	C_2
010	C_2
100	C_2
000	C_3
001	C_3
010	C_3
100	C_3
000	C_5
001	C_5
010	C_5
100	C_5
011	C_4
101	C_4
110	C_4
011	C_6
101	C_6
110	C_6
011	C_7
101	C_7
110	C_7
111	C_8

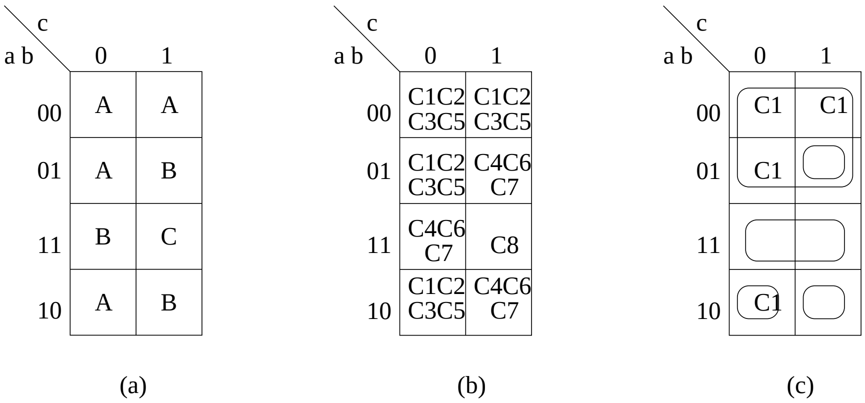


Figure 12.41: *Graphical illustration of stages of encoding.*

Let us observe that the input cubes are the same in each of groups C_1, C_2, C_3, and C_5. For C_1 the cover is:

0XX
100

Let us observe that to create cube 0XX in class C_1, an input 011 is necessary, which is taken from C_4, C_6, or C_7. We can thus say, that the 0XX cube above is formed by considering the input cubes for columns C_4, C_6, C_7, as don't cares.

The same cover exist also for the input cubes in each compatible class for C_2, C_3, and C_5. For each of the groups, C_4, C_6, and C_7, the cover is the same, and it is:

011
101
11X

The 11X cube above is formed by considering the input cube for column C_8 as don't cares. The above cover intersects the cover for C_1, C_2, C_3, C_5. Thus, the following dominance relation can be found:

$$C_4,\ C_6,\ C_7 \ > \ C_1,\ C_2,\ C_3,\ C_5$$

The input cover (cube 111) for C_8 intersects the cover for C_4, C_6, C_7 thus

$$C_8 \ > \ C_4,\ C_6,\ C_7$$

These stages are illustrated graphically in Figure 12.41.

STAGE 2. Finding the dichotomies.

The input constraint matrix for function H (see Fig. 12.42) is created from symbol compatibles and from TABLE 1.

By comparing the matrix for function H with the original table of the initial function one can verify, that all the "non-identifying don't cares" in class B have been assigned '0' . By "non-identifying don't cares" one means those don't cares that are not replaced with cares during combining compatible column according to some compatible.

Now matrix H is used, row-by-row, to generate seed dichotomies. For instance, from the first row of matrix H, the following two dichotomies are created:

d e	C1	C2	C3	C4	C5	C6	C7	C8
00	1	1	1	0	1	0	0	1
01	1	1	1	1	1	1	1	1
11	1	1	1	0	1	0	0	0
10	0	0	0	1	0	1	1	0
	A	A	A	B	A	B	B	C

Figure 12.42: *The input constraint matrix for function H*

$$P1_a \; = \; (C_1C_2C_3C_5C_8;\;\; C_4), \qquad P1_b \; = \; (C_4;\;\; C_1C_2C_3C_5C_8)$$

Similarly, the following other seed dichotomy pairs are created from the first row of H:

$$P2_a \; = \; (C_1C_2C_3C_5C_8;\;\; C_6) \qquad P2_b \; = \; (C_6;\;\; C_1C_2C_3C_5C_8)$$

$$P3_a \; = \; (C_1C_2C_3C_5C_8;\;\; C_7) \qquad P3_b \; = \; (C_7;\;\; C_1C_2C_3C_5C_8)$$

Similarly, the other seed dichotomy pairs are created from other rows of H:

$P4_a = (C_1\ C_2\ C_3\ C_5\ ;\ C_4)$ $\quad P4_b = (C_4\ ;\ C_1\ C_2\ C_3\ C_5)$
$P5_a = (C_1\ C_2\ C_3\ C_5\ ;\ C_6)$ $\quad P5_b = (C_6\ ;\ C_1\ C_2\ C_3\ C_5)$
$P6_a = (C_1\ C_2\ C_3\ C_5\ ;\ C_7)$ $\quad P6_b = (C_7\ ;\ C_1\ C_2\ C_3\ C_5)$
$P7_a = (C_1\ C_2\ C_3\ C_5\ ;\ C_8)$ $\quad P7_b = (C_8\ ;\ C_1\ C_2\ C_3\ C_5)$
$P8_a = (C_4\ C_6\ C_7\ ;\ C_1)$ $\quad P8_b = (C_1\ ;\ C_4\ C_6\ C_7)$
$P9_a = (C_4\ C_6\ C_7\ ;\ C_2)$ $\quad P9_b = (C_2\ ;\ C_4\ C_6\ C_7)$
$P10_a = (C_4\ C_6\ C_7\ ;\ C_3)$ $\quad P10_b = (C_3\ ;\ C_4\ C_6\ C_7)$
$P11_a = (C_4\ C_6\ C_7\ ;\ C_5)$ $\quad P11_b = (C_5\ ;\ C_4\ C_6\ C_7)$
$P12_a = (C_4\ C_6\ C_7\ ;\ C_8)$ $\quad P12_b = (C_8\ ;\ C_4\ C_6\ C_7)$

The dichotomies in a pair are compatible and are included to increase the chance of meeting the dominance relations derived for the outputs of G. After removing the seed dichotomies which do not satisfy the dominance relations from the above list we are left with the following dichotomies:

$P4_b = (C_4\ ;\ C_1\ C_2\ C_3\ C_5)$ $\quad P5_b = (C_6\ ;\ C_1\ C_2\ C_3\ C_5)$
$P6_b = (C_7\ ;\ C_1\ C_2\ C_3\ C_5)$ $\quad P7_b = (C_8\ ;\ C_1\ C_2\ C_3\ C_5)$
$P8_a = (C_4\ C_6\ C_7\ ;\ C_1)$ $\quad P9_a = (C_4\ C_6\ C_7\ ;\ C_2)$
$P10_a = (C_4\ C_6\ C_7\ ;\ C_3)$ $\quad P11_a = (C_4\ C_6\ C_7\ ;\ C_5)$

$P12_b \quad = \quad (C_8 \, ; C_4 \; C_6 \; C_7 \,)$

As we see, the left side is always a dominating column in the dominance relations that were previously found.

The prime dichotomies which cover all seed dichotomies are formed by

$\{ \, P7_b, \quad P12_b \, \} \quad = \quad (\, C_8 \, ; \quad C_1 \; C_2 \; C_3 \; C_4 \; C_5 \; C_6 \; C_7 \,)$

$\{ \, P4_b, \, P5_b, \, P6_b, \, P8_a, \, P9_a, \, P10_a, \, P11_a \, \} \quad = \quad (\, C_4 \; C_6 \; C_7 \, ; \quad C_1 \; C_2 \; C_3 \; C_5 \,)$

The corresponding code words are:

C_1	00	A
C_2	00	A
C_3	00	A
C_4	10	B
C_5	00	A
C_6	10	B
C_7	10	B
C_8	X1	C

The procedure assigned identical code words to all columns in a compatible class;

G will have 2 outputs $x(c,d,e)$ and $y(c,d,e)$ with $H(x,y)$.

Using the above encoding with $C_8 = 11$ results in the following functions:

$x = c\overline{d} + c \; e + \overline{c}d\overline{e}, \qquad y = c\overline{d} \; \overline{e}, \qquad H = \overline{a} \; \overline{x} + \overline{a} \; y + b \; \overline{y} + a \; x \; \overline{y}$

The $C_8 = 01$ code results in the following functions:

$$x \; = \; c \; e + \overline{c} \; d \; \overline{e}$$

$$y \; = \; c \; \overline{d} \; \overline{e}$$

$$H \; = \; b \; \overline{y} + a \; x + \overline{a} \; \overline{x}$$

$$H = \overline{x} \; \overline{d} + e \; \overline{y} + dx$$

Example 12.10.2

The above input-output encoding procedure can be applied to the complex serial disjoint decomposition problem as well. The following function has complex disjoint decomposition as shown:

$$F(x_1, x_2, x_3, x_4) \; = \; \overline{x_1} \; \overline{x_3} \; \overline{x_4} + x_2 \; \overline{x_3} \; \overline{x_4} + \overline{x_1} \; x_3 x_4 + x_2 \; x_3 \; x_4$$

$$F \; = \; H(g_1(x_1, x_2); \; g_2(x_3, x_4)) \; = \; g_1 \; \cdot \; g_2$$

i.e. H is an AND gate.

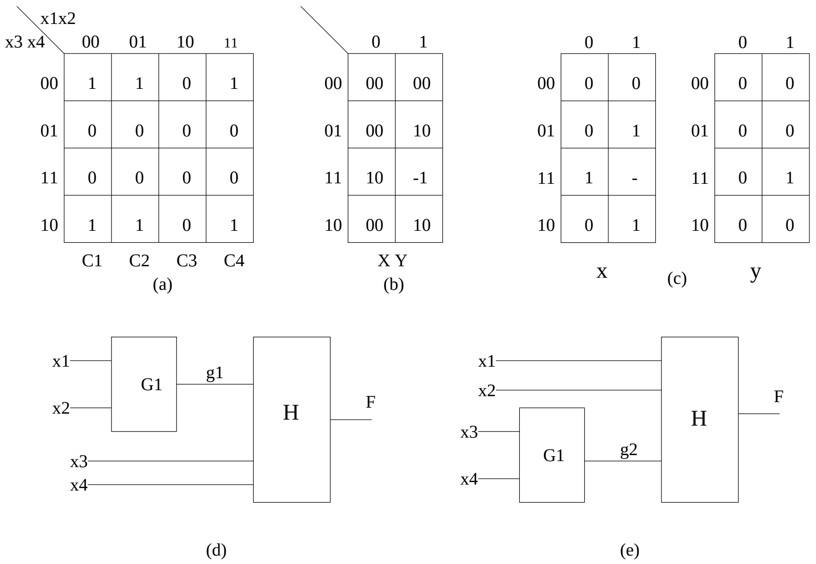


Figure 12.43: (a) The decomposition chart for g_1 in Example 9.2, (b) Functions x and y in Example 9.1, (c) realizations of functions x and y, (d) Decomposition with function g_1, (e) Decomposition with function g_2

x1x2 \ x3 x4	00	01	10	11
00	1	0	0	1
01	1	0	0	1
11	0	0	0	0
10	1	0	0	1
	C5	C6	C7	C8

Figure 12.44: The decomposition chart for g_2

x1 \ x2	0	1
0	C1	C2
1	C3	C4

Figure 12.45: A 2 by 2 Kmap.

X3 X4	C1	C2	C3	C4
00	1	1	0	1
01	0	0	0	0
11	0	0	0	0
10	1	1	0	1

Figure 12.46: The non-minimized binary/MV constraint matrix for g_1

The decomposition chart for g_1 is shown in Figure 12.43a and for g_2 in Figure 12.44. Next the cover of the input cubes for g_1 is formed:

cubes 0X and 1X cover C_1, C_2, and C_4

and

cube 10 covers C_3 ; $C_3 \;\; > \;\; C_1$, C_2, and C_4

x_1x_2	
00	C_1
01	C_1
11	C_1
00	C_2
01	C_2
11	C_2
00	C_4
01	C_4
11	C_4
10	C_3

The above data correspond to the Kmap from Figure 12.45. The non-minimized binary/MV constraint matrix for g_1 is shown in Figure 12.46:

The only seed dichotomy meeting the dominance constraint is (C_3 ; C_1 C_2 C_4).

Function g_1 can then be encoded with a single bit, according to:

x1 x2	C5	C6	C7	C8
00	1	0	0	1
01	1	0	0	1
10	0	0	0	0
11	1	0	0	1

Figure 12.47: The seed dichotomies

$$C_3 = 1; \qquad C_1 = C_2 = C_4 = 0$$

Repeating these steps for function g_2 produces:

$0X$ and $1X$ cover C_5, C_8

01 and 10 cover C_6 , C_7 ; $\qquad C_6, C_7 > C_5, C_8$

The seed dichotomies from Figure 12.47 are:

$s1_a = (\ C_5\ C_8\ ;\ C_6\ ),\qquad s1_b = (\ C_6\ ;\ C_5\ C_8\ )$
$s2_a = (\ C_5\ C_8\ ;\ C_7\ ),\qquad s2_b = (\ C_7\ ;\ C_5\ C_8\ )$

Dichotomies $s1_b$ and $s2_b$ meet the dominance constraint and assign 0 to $C_5 = C_8 = 0$ and 1 to $C_6 = C_7 = 1$. Using the above encoding for C_1,C_2,C_3, and C_4 results in $g_1 = x_1\ \overline{x_2}$ and the encoding for C_5, C_6, C_7, C_8 results in:

$$g_2 = x_3\ \overline{x_4} + \overline{x_3}\ x_4, \qquad H_1\ (\text{ for } g_1\ ) = \overline{x_3}\ \overline{x_4} + x_3\ x_4 = \overline{g_2}$$
$$H_2\ (\text{ for } g_2\ ) = \overline{x_1} + x_2 = \overline{g_1}$$

Therefore the final $H = \overline{g_1}\ \cdot\ \overline{g_2}$ (see Fig. 12.48. This is the resulting form for H iff simple disjunctive decomposition exists for both free and bound variable sets (interchanged).

12.10.2 Encoding for G and H functions in Non-Disjoint decompositions

The above procedure can be used for the synthesis of G and H in a non-disjoint decomposition of the form $F = H(G(B), A)$ where B and A are non-disjoint subsets of the variables of F. The following function can be decomposed in the above non-disjoint form L:

$$F(x_1, x_2, x_3, x_4)\ =\ \overline{x_1}\ \overline{x_2}\ \overline{x_3}\ \overline{x_4} + \overline{x_1}\ \overline{x_2}\ x_3\ \overline{x_4} + \overline{x_1}\ x_2\ x_3\ x_4 +$$

$$x_1\ \overline{x_2}\ \overline{x_3}\ \overline{x_4} + x_1\ \overline{x_2}\ x_3\ x_4 + x_1\ x_2\ x_3\ \overline{x_4}\ =\ \sum(0, 2, 7, 8, 11, 14)$$

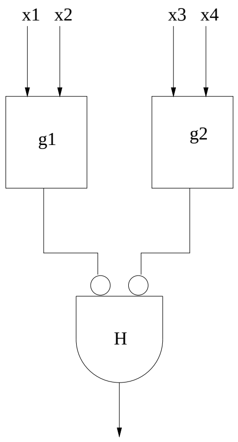


Figure 12.48: The final realization of function H

x1 \ x2 x3 x4	C1	C2	C3	C4	C5	C6	C7	C8
0	1	0	1	0	0	0	0	1
1	1	0	0	1	0	0	1	0

Figure 12.49: Function with bound set $x_2x_3x_4$

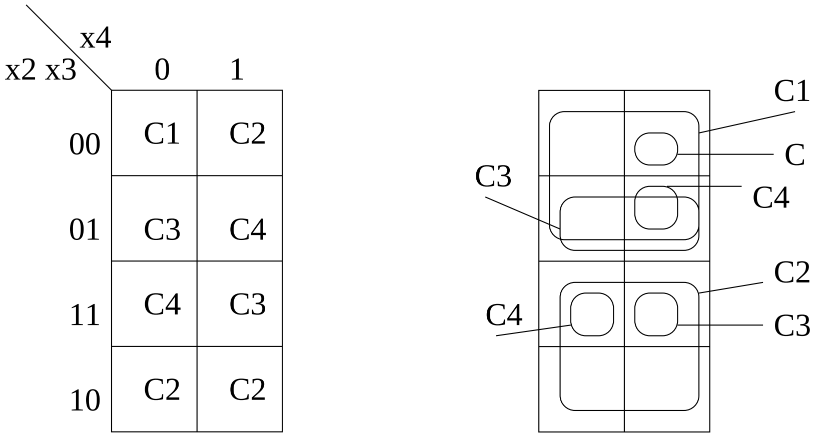


Figure 12.50: Notation for columns and groups

This function with bound set $x_2x_3x_4$ is drawn in Fig. 12.49:
Functions $G(x_2, x_3, x_4)$ and $H(G(x_2, x_3, x_4), x_1, x_3)$ are chosen for the simple non-disjunctive form above ; as the decomposition chart of F shows, the column multiplicity is 4, but because of the selected non-disjoint bound/free sets, $G(x_2, x_3, x_4)$ will be a single output function:
The input cubes for each column (G output) are:

000 C_1
001 C_2
100 C_2
101 C_2
010 C_3
111 C_3
011 C_4
110 C_4

See, Figure 12.50.
The covers for input-cube sets are (0XX for C_1 generated by treating 001 cube of C_2, 010 cube of C_3, and 011 cube of C_4 as don't cares, etc.)

0XX C_1
001 C_2
1XX C_2
01X C_3
111 C_3
011 C_4
110 C_4

since covers for C_2, C_3 , and C_4 intersect the cover of C_1, $C_2\ C_3\ C_4 \ > \ C_1$.

The covers for C_4, C_3 intersect the cover of C_2 thus $C_4\ C_3 \ > \ C_2$. Similarly we have $C_4 \ > \ C_3$.
The input constraints matrix for H is given by table from Fig. 12.51:
The corresponding seed pairs are:

x1 x2 \	C1	C2	C3	C4
00	1	0	-	-
01	-	-	1	0
11	1	0	-	-
10	-	-	0	1

(a)

x1 \	C1	C2	C3	C4
0	1	0	1	0
1	1	0	0	1

(b)

Figure 12.51: The input constraints matrix for H

x2 \ x3 x4	00	01	11	10
0	1	0	0	1
1	0	0	1	0

(a)

x1 x3 \ G	0	1
00	1	0
01	1	0
11	0	1
10	1	0

(b)

Figure 12.52: (a) The K-map for the predecessor block, (b) The successor $H(G(x_2, x_3, x_4), x_1, x_3)$

$$s1_a = (C_1 \ ; C_2), \qquad s1_b = (C_2 \ ; C_1),$$

$$s2_a = (C_3 \ ; C_4), \qquad s2_b = (C_4 \ ; C_3).$$

After removing the dichotomies which contradict the dominance relations, we are left with:

$$s1_b = (C_2; \ C_1) \text{ and } s2_b = (C_4; \ C_3) \ (s3_b \text{ and } s4_a \text{ repeat})$$

Prime dichotomy $p_1 = (\ C_2\ C_4\ ;\ C_1\ C_3\ )$ covers all of the above seed dichotomies. One bit is therefore required to encode the output of G with $C_2 = C_4 = 1$, and $C_1 = C_3 = 0$. However, since in the seeds for $s1_b$ and $s2_b$ the left and right parts have been interchanged (to satisfy the dominance conditions), one needs to complement the codes. In result: $C_1 = C_3 = 1$, and $C_2 = C_4 = 0$. Thus, the K-map for the predecessor block can be created (Fig. 12.52a).

$$G = \overline{x_2}\ \overline{x_4} + x_2\ x_3\ x_4$$

The successor $H(G(x_2, x_3, x_4), x_1, x_3)$ is derived from the original decomposition chart for F, as shown in Figure 12.52b.

Note that the $G = 1$ column corresponds to C_1 and C_3 entries in the original chart (for the corresponding x_1 and x_3 values). Similarly the $G = 0$ column maps the C_2 and C_4 values from the original chart. From the above K-map:

$$H(G(x_2, x_3, x_4), x_1, x_3) \; = \; (\overline{x_1} + \overline{x_3})G + x_1 x_3 \overline{G}$$

The non-disjoint bound/free set selection and encoding used above produced a single output predecessor function and correct image H of F.

12.11 Input-Output Encoding Combined with Column Minimization

This section addresses some of the well known problems that are dealt with in input and output encoding, and, based on that we develop one more step-by-step method to encode symbols in order to achieve a more simplified total of the predecessor and the successor functions. Techniques used involve the selection of a suitable cover set, heuristics to optimize the quality of encoding at low computational cost, multiple constraint satisfaction using an edge-weighted connection graph, and use of Hamming distances to aid in assigning codes which will result in simpler functions.

The format of this section is as follows: subsection ?? introduces some of the basic definitions, in subsection ?? the primary encoding phase is covered, in subsection ?? the final phase of the encoding approach is presented, in subsection ?? an example encoding problem is solved using the encoding approach presented in subsections ?? and ??, and finally in subsection ?? conclusions are presented.

12.11.1 Basic Definitions

$H(A, G(B, C))$ represents the decomposed function such that $F = H(A, G(B, C))$.

The sets A and B are the free and bound sets respectively where $A \cup B = X$, and $A \cap B = \emptyset$.

The set C is some subset of A when a nondisjoint decomposition is necessary and is an empty set otherwise.

A *Compatible Class* (CC) is a set of columns that are mutually compatible.

Maximum CC (MCC) is a CC that cannot include any additional columns and still maintain compatibility.

Decomposition Function Cardinality (DFC) is a measure used to evaluate the quality or effectiveness of a decomposition. More specifically it is an integer value which is equal to $DFC = \sum_{i=1}^{n} 2^{I_i} \times O_i$ where I_i and Oi are the number of inputs and outputs to block i and n is the total number of blocks.

12.11.2 PHASE I: Primary Encoding Phase

Evaluation of the cost function ratio G_{DFC}/H_{DFC} and selection of a suitable cover set

The purpose of this subsection is to outline a heuristic procedure that is used to give higher priority to the encoding of a sub-function which is much larger than it's counterpart sub-function. It is believed that giving priority will increase the overall number of don't cares in G and H (combined) and that this will ultimately result in a lower DFC, as well as in a reduction of the computation time. The cost function ratio is an approximate measure of the relative sizes of the predecessor and successor functions. It is defined as follows:

Cost Function Ratio $CFR = Cost1 = G_{DFC}/H_{DFC}$

Where G_{DFC} is the DFC of sub-function G and H_{DFC} is the DFC of sub-function H. CFR is used to give a very rough estimate of the ratio of complexity and/or computation time required by each. This in turn is used as criteria for selecting percentages of strategies. These percentages are used to determine how a *cover set* should be chosen in order to provide more efficient encoding of the larger sub-function.

The following are the ranges that have been set for the CFR ratio:

Range 1: $CFR \leq .25$
Range 2: $.25 < CFR \leq .5$
Range 3: $.5 < CFR \leq 2$
Range 4: $2 < CFR \leq 4$
Range 5: $CFR > 4$

For Range 1, X = 75 U = 0.0
For Range 2, X = 50 U = .25
For Range 3, X = 33 U = .5
For Range 4, X = 25 U = .75
For Range 5, X = 15 U = 1.0

where X is the percent of don't cares, and U is the parameter to be used in function $FLOOR$ below. The cover set is selected using the following heuristics:

1. Find the top X % of the columns which have the greatest number of specified output values.

2. From the minimal cover of columns with symbols (groups of columns), remove from symbols all redundant occurences of the columns found in step 1). [1] All redundant occurences of other columns should remain in symbols. Keep track of which symbols each column is compatible with.

3. Add the number of $FLOOR(U \times number\_of\_unused\_codes)$ classes to the minimal cover from step 2) to form the new cover. Selection of the additional classes (symbols) is performed as follows: From the minimal cover, select the symbol(s) which have the greatest number of columns that are compatible with $2^k - 1$ symbols, k=1,2,.... Before adding these new symbols to the cover found in step 2), remove ALL occurences of the top X % of columns which were found in step 1).

Edge-Weighted Connection Graph

The *Edge-Weighted Connection Graph (EWG)* is a data structure that will be used by us.

What exactly is the edge-weighted connection graph? The EWCG as defined here is a graph with nodes corresponding to symbols and edges between any two nodes signifying that there exists at least one column which is compatible with both symbols connected by the edge. With few exceptions, each edge has an associated weight (integer value) which is equal to the number of columns which are compatible with the same two symbols. This is somewhat similar to prime implicants covering minterms.

What is the purpose of using an EWCG?

There are two main reasons why an EWCG was used in the encoding approach presented in this section. One of the reasons is to enable multiple types of constraints to be satisfied together in a systematic and controlled manner which might not otherwise be feasible to attempt. The other major reason is that once the graph has been constructed following the three rules outlined below, then the graph may be mapped directly to a hypercube of dimension n (where n is the number of encoding bits

[1] The number of symbols in the minimal cover is equal to the minimum column multiplicity, and the symbols or classes do not necessarily correspond to MCC's.

in the symbolic codes). Also note that this will ensure that all symbols connected by an edge in the graph may be given code words which differ by only one bit.

How is the EWCG built and what are the rules that must be satisfied?

The graph is built by incrementally adding edges between nodes (symbols) in the graph, and systematically checking whether any of the Rules 1 - 3 given below are violated. If no rules are violated, an edge and an associated edge weight is assigned between the two nodes in the graph. The rules that must be satisfied are given below:

Rule 1: No node may have more edges than the number of bits in the code word.

Rule 2: No cycles of odd length.

Rule 3: No two intersecting faces (face A and face B) of size four may have more than two symbols in common.

Procedure for resolving rule violations:

1. If there is a violation in $Rule$ 1, then a determination is made as to whether the value of the edge in question is worth the cost of removing an edge(s).
2. If it is not worth the cost ($Cost2 \leq 0$), then no changes are made and the edge in question is removed from the list of edges yet to be considered.
3. If so, then $Rule$ 2 is checked.
4. If $Rule$ 2 is violated then a similar determination is made as to whether the removal of another edge(s) is worth the cost of assigning the new edge being considered.
5. If it is not worth the cost ($Cost2 \leq 0$), then no changes are made and the edge in question is removed from the list of edges yet to be considered.
6. If so, then $Rule$ 3 is checked.
7. If $Rule$ 3 is violated, then no changes are made and the edge in question is removed from the list of edges yet to be considered.
8. If there is not a violation in $Rule$ 1, then $Rule$ 2 is checked.
9. If $Rule$ 2 is violated then a determination is made as to whether the removal of an edge(s) is worth the cost of assigning the new edge being considered.
10. If it is not worth the cost ($Cost2 \leq 0$), then no changes are made and the edge in question is removed from the list of edges yet to be considered.
11. If so, then $Rule$ 3 is checked.
12. If $Rule$ 3 is violated, then no changes are made and the edge in question is removed from the list of edges yet to be considered.
13. If there is not a violation in $Rule$ 1 or $Rule$ 2, but there is a violation in $Rule$ 3, then a determination is made as to whether the removal of a different edge(s) is worth the cost of assigning the new edge being considered.
14. If it is not worth the cost($Cost2 \leq 0$), then no changes are made and the edge in question is removed from the list of edges yet to be considered.

The cost associated with a particular assignment is determined via a cost function which takes into account several factors. The cost function is not called, unless certain rule is violated. The cost function is formulated as follows:

$$Cost2 = N - P \tag{12.29}$$

$$N = EW_{jk} + .5W^{u}_{jk} \tag{12.30}$$

$$P = EW_{yz} + .5W^{u}_{yz} \tag{12.31}$$

$$EW_{xy} = f_{xy} + W^{p}_{xy} + 1 \tag{12.32}$$

$$W_{xy} = \sum_{i=4}^{m} f_{xy_i} \times w_i \tag{12.33}$$

Variables defined:

N is a weighted value used for the new edge being considered.

P is a weighted value used for one of the edges being considered for removal from the graph.

m is the size of the largest group of symbols compatible with any column.

EW_{xy} is the edgeweight for the edge between S_x and S_y.

W^{u}_{xy} is a temporary weight given to the pair of symbols x and y corresponding to unplaced groups.

W^{p}_{xy} is a weight given to the pair of symbols x and y corresponding to placed groups.

f_{xy} (frequency) is the number of columns that are compatible with groups of size 3 or less and which include symbols S_x and S_y.

f_{xy_i} is the number of columns compatible with groups of symbols of size i which contain symbols S_x and S_y.

w_i is a weight associated groups of size i and is equal to the number of don't cares that the resulting column code will have if the maximum number of symbols in a group of size i are satisfied in a single face imbedding constraint ($w_i = FLOOR(Log_2 i)$).

The first set of edges to be considered for assignment to the graph are the 1/3 of the symbol pairs S_{ij} that are compatible with the greatest number of columns. As stated before, each of these candidate edges must be checked for rule violations before being assigned. The order of consideration of these edges is in descending order based on the frequencies $f_{xy_{total}}$ for each pair of symbols which are not assigned yet. Where $f_{xy_{total}}$ is the total number of columns which are compatible with symbols S_x and S_y.

The second set of edges to be considered for assignment to the graph are the edges corresponding to symbol pairs that are contained in groups of symbols of size four or larger. The order of consideration of these edges is in descending order based on the weights W^{u}_{ij} for each pair of symbols which are not assigned yet. Hence, the first edge in this set to be considered is the edge with the largest weight W_{ij}^u.

The last set of edges to be considered for assignment are all remaining pairs of symbols not considered yet. The order of consideration is in descending order of $f_{xy_{total}}$.

Procedure for Incrementally Building the Graph

i) From the first set of edges to be considered, check the edge having the greatest $f_{xy_{total}}$ for rule violations. If there are rule violations, follow the procedure outlined for resolving rule violations above. If there are no rule violations or the violations were resolved, assign an edge between S_x and S_y and an associated edge weight EW_{xy}.

ii) Remove S_{xy} from the first set of edges to be considered for assignment to the graph. Also, if the edge S_{xy} was assigned, then remove all groups of three symbols which contain both S_x and S_y from the list LGA of groups to be assigned. If the edge corresponding to S_{xy} cannot be assigned, then remove all groups from the list LUG of unassigned groups that can not be satisfied as a result. Then take the edge with the next largest $f_{xy_{total}}$ and repeat step i).

iii) Repeat step i) until there are no more pairs of symbols S_{xy} in the first set to be considered.

iv) Repeat step i) for the second set of edges to be considered for assignment to the graph.

v) Repeat step i) for the third set of edges to be considered for assignment to the graph.

vi) Now that the graph has been constructed, check each of the unsatisfied constraints to see if removing the necessary edge(s) from the graph and adding the unsatisfied constraint will be worth the cost (i.e. $Cost2 < 0$).

    If so, then update the graph accordingly.

vii) Finally, assign any of the *Optional Constraints* that can be satisfied without removing any edges from the graph.

    By "Optional Constraints" we understand constraints that if satisfied will not yield more don't cares in a particular code word, but will provide an optional encoding for certain columns. This can be used to reduce Hamming distances between the output codes in sub-function G.

Mapping the Graph to the Hypercube

This step simply involves assigning nodes in the graph to vertices in the hypercube in such a way that nodes that are connected by an edge in the graph will be assigned to adjacent vertices in the hypercube. Each vertex of the hypercube has a binary codeword which differs in exactly one bit from each of the other vertices that it is connected to by an edge.

The process of mapping to the hypercube is as follows:

1) Start with the largest group(constraint) of 2^k symbols and assign the group to 2^k codes such that the supercube of these codes does not contain any of the codes not in the constraint.

    Example:

    constraint $= (S_1, S_4, S_3, S_7)$

    000 S_1

001 S_4

011 S_3

010 S_7

---

0 - - supercube contains only codes contained in the constraint.

2) Find the largest group (constraint) of 2^k symbols which has the largest intersection with the group assigned in 1). Assign this group.
3) Find the largest group (constraint) of 2^k symbols which has the largest intersection with more than one of the previously placed groups(if one exists).
4) Repeat step 3) until no more groups intersect more than one of the previously placed groups.
5) Repeat step 2) until all groups which intersect with the group assigned in step 1) have been placed.
6) If any groups have not been placed yet, then repeat step 1) for the largest of the remaining groups.
7) If any nodes (symbols) in the graph do not have any edges, they may be assigned to any of the remaining codes.

For symbols in the graph which do not have any edges, use of Hamming distances will greatly improve the selection from among the remaining codes. Symbols without edges correspond to classes of columns that do not appear in any other classes (i.e. they are disjoint output classes).

12.11.3 Assignment of Supercube of Symbol Codes to Corresponding Columns

For many of the columns, the task of assigning them codes is trivial once the symbol codes have been established. However, certain columns may not be assigned to 2^k symbol codes (where k is an integer). In this case, an objective cost function is used to determine which combination of symbol codes to assign to a column being considered. For example, if a column is compatible with five different symbols, then there are five different combinations of four(2^2) symbol codes which can be used to encode the column. Of these five, certain combinations may not be able to be combined because they may include a code of a symbol which is not one of the symbols compatible with the column(i.e. one or more symbols may not satisfy a particular face embedding constraint).

Once the symbol codes have been assigned, a function $ENCODE(C_i)$ is called repeatedly to encode all columns. The trivial and non-trivial cases handled by function $ENCODE(C_i)$ are explained below. There are two conditions that must be checked for each column. If both conditions are satisfied the column is considered a trivial case and is encoded accordingly. If either condition is not satified then the column is treated as a non-trivial case.

Conditions required for trivial encodings:

1) The number of symbols that a column is compatible with is equal to 2^k(where k is an integer).
2) The supercube of 2^k symbol codes which column C_i is compatible with do not include any of the codes not in the constraint.

For the trivial cases, the supercube of 2^k symbol codes may be assigned as the column's code word.
Examples:
000 S1
010 S2
—-
0-0 supercube = column code

110 S4
101 S6
111 S9
100 S3
—-
1- - supercube = column code

For non-trivial cases, the assignment of codes is as follows:

Find all combinations of 2^k symbols which column C_i is compatible with(where $2^k < m < 2^{k+1}$ and m is the total number of symbols that C_i is compatible with). Store the supercube of each combination in a temporary assignment list TAL. Remove any codes from this list which violate the face embedding constraints. Then from the remaining candidate codes determine which code will result in the smallest hamming distances between adjacent codes in sub-function G.

This is done using a cost function which determines the sum of the hamming distances between all codes in G (already assigned) which are adjacent to the code of column C_i. The cost function is as follows:

$$Cost3 = \sum_{j=1}^{n} HAM\_DIST(CODE(C_i), CODE(C_j))$$

where C_i is the column which is being considered for an encoding and C_j is a column which has its corresponding code in G adjacent to the code of C_i. The value "n" is the total number of codes in G which are adjacent to the code of C_i, or more simply, n is equal to the number of variables in the bound set. $CODE(C_i)$ and $CODE(C_j)$ are the corresponding codes of the two columns.

The HAM_DIST function returns the number of bits in two code words which are not compatible (i.e. a "1" in one code word where there is a "0" in the other code word—don't cares are considered compatible with any bit value).

The candidate code with the lowest cost is assigned as the column code. This procedure is repeated for all columns remaining to be encoded from the list $LNTC$ of non-trivial cases.

12.11.4 PHASE II: Finalizing the encoding

Determining if the elimination of certain Don't Cares in sub-function H is justified

1) For each symbol S_i do the following:

   i) Find the position of all don't cares in S_i that would result if all columns which appear in S_i and at least one other symbol(class) were removed.

   ii) For each don't care found in step 1), find the number of columns (up to 4) which have a care bit in the corresponding position (only consider columns which are found in S_i and at least one other symbol). Add them to a list corresponding to the don't care being considerd and increment a count for each column (this will provide the number of don't cares covered in S_i by each column–this count may be used in the cost function evaluation). If four columns are found which cover the corresponding don't care then take no action.

iii) Repeat step ii) until a don't care is found which is covered by less than four coulmns or until no more don't cares are left to be checked.

iv) After all the lists of columns have been found which cover don't cares in S_i, then for each list which contains only one column evaluate the cost function below:

$$Cost4 \;=\; 1- \mid DC(S_i, C_j) \mid \;+\; ADJ\_IN\_G(C_j) - ADJ\_IN\_H(S_i, C_j) \qquad (12.34)$$

where:
$DC(S_i, C_j)$ is the position(s) of a don't care in S_i covered only by column C_j,
$\mid DC(S_i, C_j) \mid$ is the total number of don't cares in S_i covered only by C_j,
$ADJ\_IN\_G(C_j)$ or "Adjacency factor in G" is a measure of how many adjacent output codes in sub-function G will no longer be compatible if the code for symbol S_i is removed from the column code,
$ADJ\_IN\_H(S_i, C_j)$ or "Adjacency factor in H" is equal to the number of outputs in sub-function H (adjacent to the output position(s) in S_i under consideration) which will remain incompatible with the care outputs of C_j if C_j is not removed from symbol S_i.

If Cost4 ≤ 0 then re-encode column C_j and remove it from all the lists generated in ii). Re-encoding of columns is accomplished by function RE_ENCODE(S_i, C_j) which is described in the next section.
Repeat step iv) until all lists have been checked.

2) Repeat step 1) until all symbols have been checked.

3) Make additional passes as needed (it means, repeat step 2) until no more changes are made).

Re-Encoding of Columns which have a low Benefit/Cost ratio

If the value of Cost4 determined in the previous section is less than or equal to zero, then the function RE_ENCODE(S_i, C_j) is called to re-encode column C_j. The process of re-encoding C_j is as follows:

1) Remove S_i from the set of symbols which column C_j can be encoded with.

2) Then call function ENCODE(C_j) to give a new encoding to column C_j.

Note: $ENCODE(C_j)$ is the same function used in the primary encoding phase.

12.11.5 Encoding Example Solved Step-by-Step

Shown in Figure 12.53 is the function F which is being decomposed in this example. Refer back to previous sections describing the encoding process as needed for further clarification. Since this example was solved using the minimum number of MCC's prior to completion of the set cover selection procedures, the procedures outlined in section 12.11.3.1 are not included in this example.

F

ab \ cdef	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	0	1	0	0	-	0	0	1	1	0	-	-	0	0	1	-
01	-	-	-	0	-	1	0	-	-	0	0	-	1	-	-	0
11	0	1	-	0	1	0	-	-	1	1	-	-	1	-	-	1
10	-	-	-	0	-	-	0	-	1	-	0	0	-	-	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	S2 S3	S5 S6	S1 S2 S3 S4	S3	S1 S4 S5 S6	S2	S3 S4	S5 S6	S6	S4	S3 S4 S5	S1 S2 S3 S4 S5	S1	S1 S2 S3 S4	S5	S4 S5

H

ab \ xyz								
	0	0	0	0	1	1	-	-
	1	1	0	0	-	-	-	-
	1	0	0	1	-	1	-	-
	-	-	0	0	0	1	-	-
	S1	S2	S3	S4	S5	S6		

Function F shown with corresponding symbols each column is compatible with.
Also shown is sub-function H with a minimal cover of disjoint classes of columns.

Figure 12.53: Function F for Encoding Example

Building the Edge-Weighted Connection Graph

STEP 1)
Identify the first set of edges to be considered for assignment to the Edge-Weighted Connection Graph (1/3 of symbol pairs which are compatible with most columns). Each of the edges(symbol pairs) in this set is marked with a star in Figure 12.54(b).

STEP 2)
Begin assigning edges to the graph starting with the edge corresponding to the pair of symbols denoted $S_{ij}(S_i$ and $S_j)$ that has the greatest frequency $f_{ij_{total}}$. $f_{ij_{total}}$ is the total number of columns which are compatible with symbols S_i and S_j. Initially the graph appears as in Figure 3(a).

The first edge to be considered is S_{34}. Since there are no other edges in the graph, no rules are violated and the edge and it's corresponding edge weight are assigned to the graph. The edge weight is calculated as follows:

$$EW_{34} = f_{34} + W_{34}^p + 1 = 2 + 0 + 1 = 3$$

$W_{34}^p = 0$ since no groups of size 4 or greater that have been placed that also contain symbols S_3 and S_4.

The graph now appears as in Figure 12.55b.

The next edge to be assigned is the one with the next largest $f_{ij_{total}}$. It turns out that there is a three way tie for the next largest frequency. Use the following criteria for breaking ties:

For each S_{ij} that are tied, find the symbol $S_k \in S_{ij}$ which has the most edges. Then compare these values and select the pair which has the lowest number of edges from these(i.e. select the least from the greatest). If there is still a tie select randomly.

In this case there is still a tie so S_{14} is chosen randomly and checked for rule violations. Since there are no violations, it is assigned to the graph.

$$EW_{14} = f_{14} + W_{14}^p + 1 = 0 + 0 + 1 = 1$$

Similarly for the next two edges which had the next largest $f_{ij_{total}}$. No rules were violated so the edges S_{23} and S_{45} are assigned to the graph.

$$EW_{23} = f_{23} + W_{23}^p + 1 = 1 + 0 + 1 = 2$$
$$EW_{45} = f_{45} + W_{45}^p + 1 = 2 + 0 + 1 = 3$$

This concludes the assignment of the first set of edges. The graph now appears as in Figure 12.55c.

STEP 3)
Determine the second set of edges to be considered. The second set of edges to be considered can be found in the column W_{ij} in Figure 12.54b which are not equal to zero. An example of how one of these weights is calculated is as follows:

a)

Group Size	C_i-(set of symbols that column C_i is compatible with)
5	C_{11}-(1,2,3,4,5)
4	C_2-(1,2,3,4) C_{13}-(1,2,3,4) C_4-(1,4,5,6)
3	C_{10}-(3,4,5)
2	C_{15}-(4,5) C_6-(3,4) C_7-(5,6) C_1-(5,6) C_0-(2,3)
1	C_3-3 C_5-2 C_8-6 C_9-4 C_{12}-1 C_{14}-5

b)

Pairs of Symbols	$f_{ij\,total}$	W_{ij}
S_{12}	3	6
S_{13}	3	6
S_{14}	* 4	8
S_{15}	2	4
S_{16}	1	2
S_{23}	* 4	8
S_{24}	3	6
S_{25}	1	2
S_{26}	0	0
S_{34}	* 5	6
S_{35}	2	2
S_{36}	0	0
S_{45}	* 4	4
S_{46}	1	2
S_{56}	2	2

a) Table showing all columns and the corresponding sets of symbols they are compatible with. b) Table showing the total number of columns compatible to each symbol pair(frequency) and the corresponding weight(number of don't cares that can be introduced from groups greater than size 3 if all constraints are satisfied).

Figure 12.54: Tables (a) and (b) for the Example

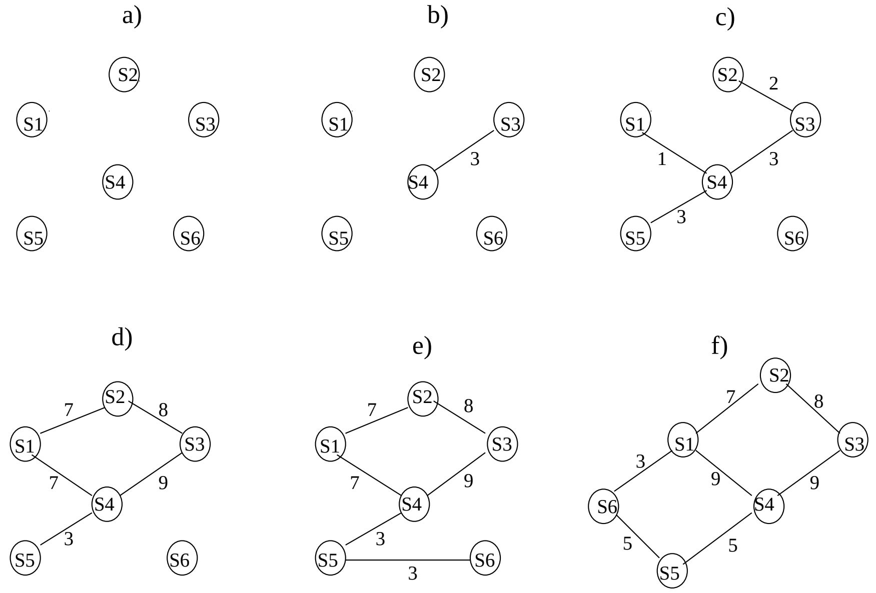


Figure 12.55: Construction of the Edge Weighted Connection Graph.

$$W_{xy} = \sum_{i=4}^{m} f_{xy_i} \times w_i \tag{12.35}$$

$$W_{14}^{u} = \sum_{i=4}^{5} f_{14_i} \times w_i = (f_{14_4} \times w_4) + (f_{14_5} \times w_5) = (3x2) + (1x2) = 8 \tag{12.36}$$

Recall that f_{xy_i} is the number of columns compatible with groups of symbols of size i which contain symbols S_x and S_y

STEP 4)

Assign edges in the second set in descending order according to the values of W_{ij}^{u}. The first edge to be considered is either S_{23} or S_{14} since they tie for the greatest W_{ij}^{u}. However, since both have already been placed, simply find the next largest W_{ij}^{u} and remove S_{23} and S_{14} from the second set of edges to be considered. There is a three way tie for the next edge to be considered(S_{12},S_{13} and S_{24}). Use the same criteria for breaking ties as done for the first set of edges. The tie is broken and S_{12} is selected next. The following was the tie breaking criteria used:

Max edges from S_{12} is $S_1 = S_2 = 1$.
Max edges from S_{13} is $S_3 = 2$.
Max edges from S_{24} is $S_4 = 2$

MIN(MAX(1,2),MAX(1,3),MAX(2,4)) = MIN(1,2,2) = 1.

There are no rule violations so edge S_{12} and it's corresponding edge weight are assigned. This satisfies the first group constraint(or face embedding constraint) of size 4 where the set of symbols in the group constraint is (1,2,3,4).

Once edge weights have been updated, the graph will appear as in Figure 12.55d. Note that 3 columns are compatible with the group of symbols(1,2,3,4). For each edge in the group constraint, an additional weight of 2(for 2 don't cares) is added to each edge weight for each column that is compatible with the group constraint(i.e. W^p_{ij} = 3 x 2 = 6).

Update edges as follows:

$$EW_{12} \;=\; f_{12} + W^p_{12} + 1 \;=\; 0 + 6 + 1 \;=\; 7 \tag{12.37}$$

W^p_{12} is calculated as follows:

$$W^p_{12} \;=\; \sum_{i=4}^{5} f_{12_i} \times w_i \;=\; (f_{12_4} \times w_4) + (f_{12_5} \times w_5) \;=\; (2x2) + (1x2) = 6 \tag{12.38}$$

Similarly, the other edges in the group constraint are updated.

$$EW_{14} \;=\; f_{14} + W^p_{14} + 1 \;=\; 0 + 6 + 1 \;=\; 7 \tag{12.39}$$

$$EW_{23} \;=\; f_{23} + W^p_{23} + 1 \;=\; 1 + 6 + 1 \;=\; 8 \tag{12.40}$$

$$EW_{34} \;=\; f_{34} + W^p_{34} + 1 \;=\; 2 + 6 + 1 \;=\; 9 \tag{12.41}$$

The next edge selected is S_{13}. *Rule* 1 is not violated, but *Rule* 2 is violated, because a cycle of odd length is identified (S_{12}, S_{23}, S_{13}). Therefore the cost function $Cost2$ must be evaluated to determine if adding the new edge to the graph and removing one of the other edges in the cycle will be worth the cost. The following are two evaluations of $Cost2$ if S_{12} or S_{23} are to be removed:

For S_{12}, $Cost2 = N$ - P = $(EW_{13} + .5W^u_{13}) - (EW_{12} + .5W^u_{12})$ = (0 + .5(6)) - (7 + .5(0)) = 3 - 7 = -4

Since Cost2 $\leq$ 0, this change is not worth the cost and edge S_{12} remains in the graph.

For S_{23}, $Cost2 = N - P = (EW_{13} + .5W^u_{13}) - (EW_{23} + .5W^u_{23})$
= (0 + .5(6)) - (8 + .5(0)) = 3 - 8 = -5

Since Cost2 $\leq$ 0, this change is not worth the cost. Therefore no changes are made and edge S_{13} is removed from the list to be considered.

The next edge selected is S_{24}. Again *Rule* 1 is not violated, but *Rule* 2 is violated, because a cycle of odd length is identified (S_{12}, S_{24}, S_{14}). The cost function $Cost2$ is evaluated again for each edge that can break the cycle.

For S_{12}, $Cost2$ = -4

Since $Cost2 \leq 0$, this change is not worth the cost.

For S_{14}, $Cost2$ = -5

Since Cost2 $\leq$ 0, this change is not worth the cost. Therefore no changes are made and edge S_{24} is removed from the list to be considered.

The next edge considered is S_{15}. Again, a cycle of odd length is identified (S_{15}, S_{45}, S_{14}).

For S_{14}, Cost2 = -6
For S_{45}, Cost2 = -3

Since Cost2 $\leq$ 0, no changes are made and edge S_{15} is removed from the list to be considered.

For the next largest W_{ij}^{u} there is a five way tie. Using the procedure outlined for breaking ties results in the following:

Max edges from S_1 or S_6 = 2
Max edges from S_2 or S_5 = 2
Max edges from S_3 or S_5 = 2
Max edges from S_4 or S_6 = 3
Max edges from S_5 or S_6 = 1

MIN(MAX(1,6),MAX(2,5),MAX(3,5),MAX(4,6),MAX(5,6))
= MIN(2,2,2,3,1) = 1.

Therefore S_{56} is the next edge to be considered. No rules are violated so edge S_{56} is added to the graph. The corresponding edge weight is EW_{56} = 3. The graph now appears as in Figure 12.55e.

Next, there is a four way tie. Randomly, S_{25} was selected from those above which had 2 as the maximum number of edges connected to one of its symbols. *Rule* 1 and *Rule* 2 are satisfied, but *Rule* 3 is violated because this edge created a new face of size 4 which intersects a different face with more than 2 nodes (symbols) in common. Select edges to be evaluated with S_{25} which are not in the same face(i.e. S_{12} and S_{14} are edges in face$(S_{12}, S_{23}, S_{34}, S_{14})$ but not in face $(S_{25}, S_{45}, S_{34}, S_{23})$. If $Cost2 \geq 0$, make changes to the graph accordingly(add new edge and remove the other).

For S_{12}, $Cost2$ = 3 - 7 = -4
For S_{14}, $Cost2$ = 3 - 8 = -5

Since Cost2 $\ngeq$ 0, no changes are made and edge S_{25} is removed from the list to be considered.

The next edge considered is S_{35}. Rule 2 is violated because a cycle of odd length is identified(S_{35}, S_{45}, S_{34}).

For S_{34}, Cost2 = 2 - 9 = -7
For S_{45}, Cost2 = 2 - 4 = -2

Since Cost2 $\leq$ 0, no changes are made and edge S_{35} is removed from the list to be considered.

The next edge considered is S_{35}. There are no rule violations therefore S_{16} is assigned to the graph with its corresponding edge weight. Update the edge weights of those pairs which are elements of the group. Edges updated are:

$EW_{16} = f_{16} + W_{16}^{p} + 1 = 0 + 2 + 1 = 3$

$EW_{56} = f_{56} + W_{56}^{p} + 1 = 2 + 2 + 1 = 5$

$EW_{45} = f_{45} + W_{45}^{p} + 1 = 2 + 2 + 1 = 5$

$EW_{14} = f_{14} + W_{14}^{p} + 1 = 0 + 8 + 1 = 9$

The graph now appears as in Figure 12.55f.

This completes the second set of edges to be considered for assignment to the graph. As it turns out, this is the last set of edges for consideration because there are no remaining edges which have not been considered yet.

Therefore this completes the construction of the graph!

Mapping the Graph to the Hypercube

The next step is to map the graph directly to a hypercube. The following mapping of symbols to codes was found using the procedure outlined in section 12.11.3.3.

First, we select the largest group(constraint) of 2^k symbols and assign it to 2^k codes such that the supercube does not contain any of the codes not in the constraint. This can be ensured by maintaining the connectivity relationship in the hypercube that is found in the graph. It turns out that there is a tie for the largest constraint. The two groups in the graph which are largest are (S_1, S_2, S_3, S_4) and (S_1, S_4, S_5, S_6). Select the group which is compatible with the greatest number of columns. Therefore group (S_1, S_2, S_3, S_4) is selected. Arbitrarily four codes which differ only in one bit are chosen for the constraint

000
001
011
010
—-
0- - supercube does not contain any codes not in the constraint

These codes are assigned so that the connectivity relationship in the graph is maintained.
000 S_2
001 S_3
011 S_4
010 S_1

Next find the largest group (constraint) of 2^k symbols which has the largest intersection with the group already assigned. Assign this group. The group found is (S_1, S_4, S_5, S_6). Since S_1 and S_4 are already assigned, only S_5 and S_6 need to receive codes. The remaining codes available are:

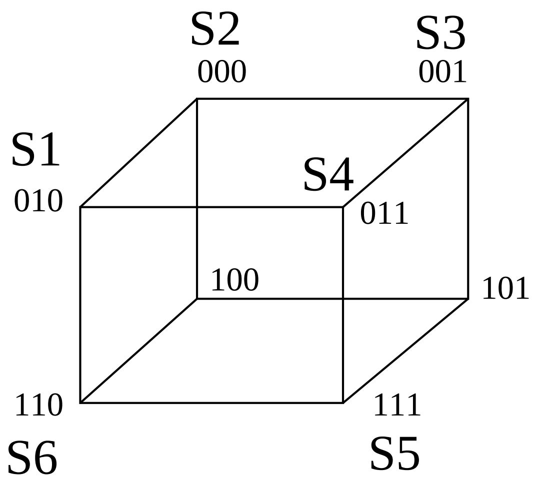


Figure 12.56: Assignment of codes resulting from the mapping of the Edge-Weighted Connection Graph to the Hypercube.

100
101
111
110

S_5 is adjacent to S_4 in the graph and therefore must receive a code which differs by only one bit from the code of S_4. From the availabe codes, the only one which differs from code 011 by one bit is code 111. Therefore S_5 is assigned to code 111. Next, S_6 must be assigned a code which is adjacent to S_1 and S_5. Therefore, it must be given a code from the available codes which differs by only one bit from 010 and 111. Therefore S_6 is coded with 110.

111 S_5
110 S_6

These code assignments and the relationships between the groups in the graph can be seen in the hypercube of Figure 12.56.

Example Problem—Assignment of Supercube of codes to corresponding columns

Finally, the last part of the primary encoding phase is to assign the supercube of symbol codes to the corresponding columns. Trivial cases correspond to columns which may be assigned to the supercube of codes of all symbols which it is compatible with. There are two types of non-trivial cases. One type of non-trivial case is where a column is compatible with some number of symbols (m) where $2^k < m < 2^{k+1}$ and k is some integer(This implies more than one optional supercube of 2^k codes). The other type of non-trivial case is where the supercube of the codes includes a code which is not in the constraint.

Using the procedure outlined in section 12.11.3.4, we begin by assigning codes to columns for cases considered trivial. At first all columns which are compatible with 2^k symbols are considered trivial to encode. The supercube is checked to see if any codes are included which are not in the constraint. If so, that column is placed in the list of non-trivial cases $LNTC$ to be coded after all trivial cases. If not, then the column is assigned to the supercube of all codes in the constraint.

For trivial cases the order of assignment of codes does not matter. So, we begin with the columns in order as they appear in Figure 12.53. The first column to be encoded is column 0 which is compatible with symbols S_2 and S_3. Since there are 2^k symbols compatible with column C_0 (where $k = 1$ in this case), we proceed to check the second condition required for a trivial encoding.

000 S_2
001 S_3
—-
00- supercube

Since the supercube does not contain any codes that are not in the constraint of column C_0, then C_0 is encoded with "00-". Similarly, the remaining columns are coded which do not fall into the category of non-trivial. It turns out that all columns except for C_{10} and C_{11} are encoded as trivial cases (they receive the supercube of all symbols they are compatible with). These can be seen in the Table from Figure 12.57.

For non-trivial cases we encode columns starting with the largest number of symbols that any one column is compatible with. Therefore column C_{11} is encoded first. The largest group of 2^k symbols that C_{11} is compatible with is 4. There are 5 possible groups of 4 symbols ((1234), (1345),(1245),(1235),(2345)). Of these, only one is possible because the supercube of the others include codes which are not in the constraint. For example, the supercube of symbols (1235) is " $-\ \ -\ \ -$ ". This includes the code of symbol S_6 which is not in the constraint. Therefore C_{11} is encoded with the supercube of symbols (1234) resulting in the code "$0\ \ -\ \ -$".

For the only other non-trivial case(C_{10}), there are 3 possible groups of 2^k symbols codes that C_{10} might be able to be encoded with(k = 1 in this case). The 3 groups are ((34),(35),(45)). Group (35) is not allowed since it's supercube ($-\ -\ 1$) includes the code of an unused code (1 0 1). For the remaining groups (34) and (45) we apply the cost function $Cost3$ to determine which codes(($0\ -\ 1$) or ($-\ 1\ 1$)) to select. The following are the columns that are adjacent to column C_{10} in G:

0010 (column C_2)
1000 (column C_8)
1011 (column C_{11})
1110 (column C_{14})

$Cost3\ =\ \sum_{j=1}^{n}$ HAM_DIST(CODE(C_i),CODE(C_j))

$$Cost3 = HAM\_DIST(CODE(C_{10}), CODE(C_2)) + HAM\_DIST(CODE(C_{10}),\ CODE(C_8))\ +$$

$$HAM\_DIST(CODE(C_{10}),\ CODE(C_{11}))\ + HAM\_DIST(CODE(C_{10}),\ CODE(C_{14}))$$

For C_{10} encoded with "$0\ -\ 1$" the cost is:

$Cost3(C_{10}) = 0 + 2 + 0 + 1 = 3$

Columns	Symbols	Symbol Codes	Supercube	Acceptable Encodings
0	S2,S3	000 001	00-	000 001 00-
1	S5,S6	111 110	11-	111 110 11-
2	S1,S2,S3,S4	010 000 001 011	0--	010 000 001 011 00- 01- 0-0 0-1 0--
3	S3	000	000	000
4	S1,S4,S5,S6	010 011 111 110	-1-	010 011 111 110 11- 01- -10 -11 -1-
5	S2	000	000	000
6	S3,S4	001 011	0-1	001 011 0-1
7	S5,S6	111 110	11-	111 110 11-
8	S6	110	110	110
9	S4	011	011	011
* 10	S3,S4,S5	001 011 111	--1	001 011 111 0-1 -11
* 11	S1,S2,S3,S4,S5	010 000 001 011 111	---	010 000 001 011 111 00- 01- 0-0 0-1 0-- -11
12	S1	010	010	010
13	S1,S2,S3,S4	010 000 001 011	0--	010 000 001 011 00- 01- 0-0 0-1 0--
14	S5	111	111	111
15	S4,S5	011 111	-11	011 111 -11

Circled codes indicate the column codes determined by the algorithm presented in this paper.

Figure 12.57: Application of Supercube Operation in Encoding.

For C_{10} encoded with "$-$ 1 1" the cost is:

$Cost3(C_{10}) = 0 + 1 + 0 + 0 = 1$

The code resulting in the lowest cost is chosen. Therefore column C_{10} is assigned the code "$-$ 1 1".

Note that the use of the unused codes can improve the quality of the encoding in some cases. The approach in this section does use the unused codes when heuristics in the cover set selection determines that they should be used. However, there is a trade-off and complicated heuristics may be required to ensure that there will be a benefit from particular uses of the unused codes. In the example above one would not benefit "overall" if he would encode C_{10} with the supercube of the three symbols it is compatible with plus an unused code. If one would use the unused code(1 0 1) then the code "$-$ $-$ 1" can be given to column S_{10}. But this forces to assign one of the don't care columns in H to either symbol S_3, S_4 or S_5. If S_5 is chosen, since it has the fewest care bits, then column "1 0 1" in the successor function H will be changed from all don't cares to a partially specified column, as illustrated below:

From To

$-$ 1

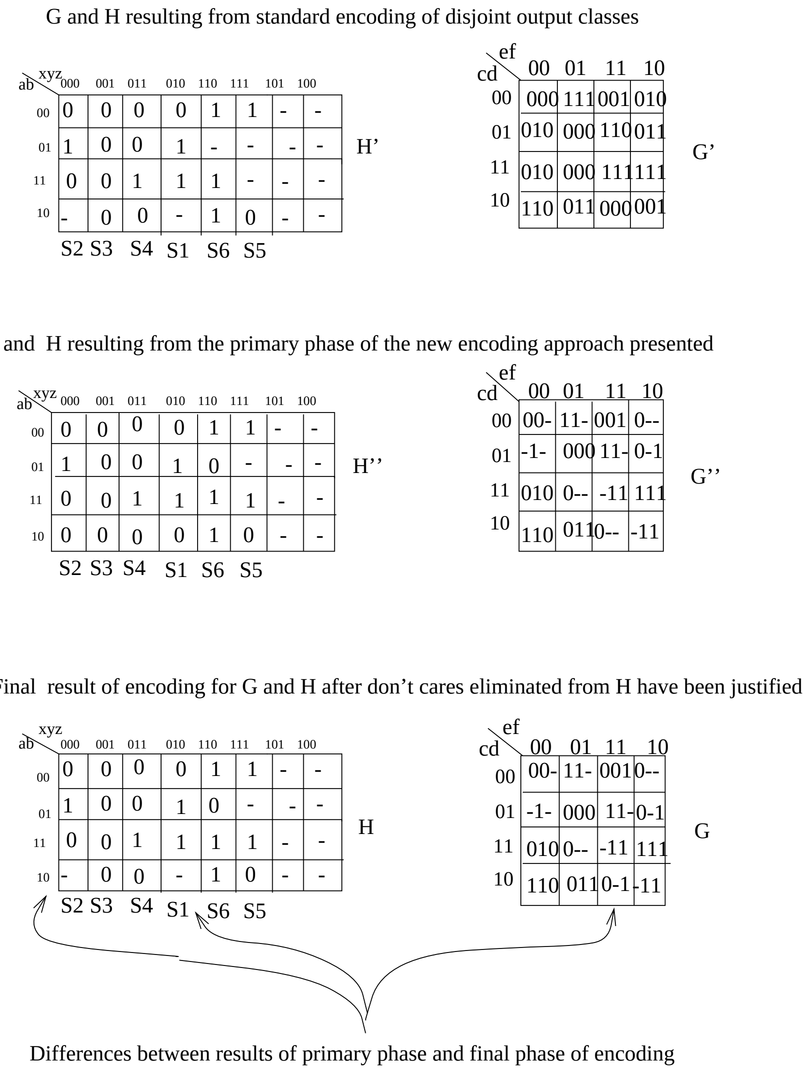


Figure 12.58: Subfunctions G and H for the Example.

– –
– –
– 0

This results in adding one don't care to the predecessor function G and removing two don't cares from the successor function H. In some cases it may be a good trade-off, but in certain cases it may be that far more don't cares would be removed from H (changed to specified values) than would be introduced into G.

12.11.6 Concluding Remarks

Unlike some approaches which optimize only the input encoding or only the output encoding, the approach presented in this section allows to satisfy multiple constraints and objectives for input and output encodings concurrently in a relatively systematic manner. Though not shown in the example, the method described in the subsection for cover set selection can be used to satisfy the objective to

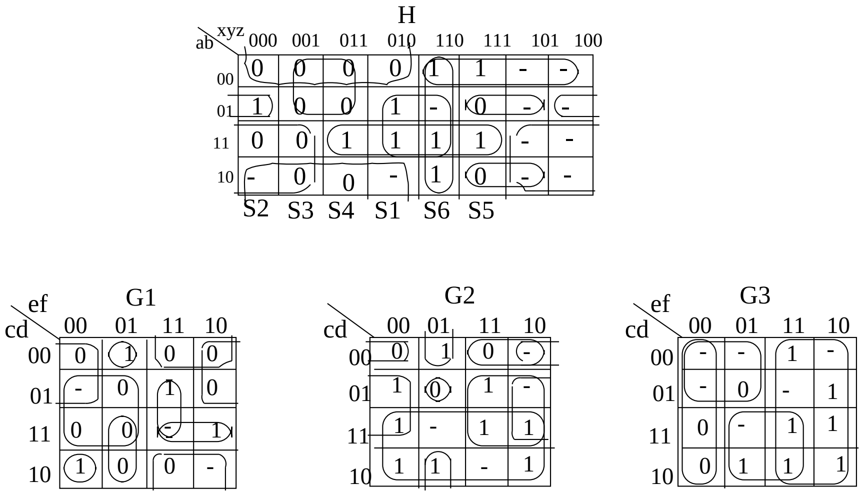


Figure 12.59: Minimization of subfunctions G and H for Example

increase the number of don't cares in the sub-function G while minimizing the loss of don't cares in the sub-function H.

The example of encoding illustrated how a substantial number o don't cares can be introduced into the sub-function G as a result of encoding columns with codes of multiple symbols. Also demonstrated was how multiple constraint satisfaction can be accomplished through a relatively simple set of heuristics applied to an edge-weighted connection graph. Use of minimum Hamming distances between the optional codes and the adjacent codes already assigned illustrated how the complexity of the resulting function could be simplified.

Though the results from small examples calculated by hand are very promising, numerous tests on the implemented algorithm need to be done in order to get an accurate assessment of the potential of this method.

12.12 Future Work on Encoding for Decomposition

We have presented several new algoritms:

1. Symbol Encoding by Partitions and Boolean Function Minimization, (Perkowski)
2. Column Encoding by Column Selection and Boolean Relation Minimization, (Perkowski)
3. Column Encoding by Column Selection and Boolean Function Minimization, (Perkowski)
4. Symbol Encoding by Partitions and Multiple-Valued Minimization, (Perkowski)
5. Symbol Encoding by Partitions and Multiple-Valued Minimization, (Perkowski)
6. Symbol Encoding by Dychotomies and Dominance Relations, (Almeria, Iliev)
7. Column Encoding by Dychotomies. (Iliev, Perkowski)

The Column Encoding algorithms combine column minimization and symbol encoding methods and take into account the optimization of both the predecessor and successor logic. Most of these methods take some ideas from the existing methods that would perform only the input encoding, the output encoding, or the FSM state assignment. Moreover, some of the proposed methods work on the non-minimized or partially minimized columns, thus essentially enlarging the search space, and potentially improving the results. It is now the task of our group to practically compare these algorithms in computer programs, and test them on one of our decomposers: TRADE, DEMAIN and GUD. The future user of GUD will be able to select either of these approaches.

Since the concept of a Boolean relation is a generalization of the concept of a don't care, the techniques described here can be generalized for column minimization problem with symbolic relations.

Chapter 13

Strategy

Chapter 14

Special Decompositions

14.1 Applications of Spectral Methods

Various spectral methods have been tried to solve the Boolean Decomposition problem. In 1971 Lechner formulated a decomposition method based on orthogonal transforms. Most authors concentrate on various kinds of Walsh transforms (Hadamard, Paley, Karczmarz - these transforms differ only in ordering of their coefficients which is irrelevant from the point of view of logic synthesis), although there are few papers that use Fast Fourier, Haar, or Reed-Muller transform for binary functions and Christenson Transform for multiple-valued logic.

Hurst, Miller and Muzio [?] tried to apply orthogonal transform techniques to decomposition, but this approach could not be practically successful at the time of their publication (1970's), because no fast and memory efficient methods for orthogonal transforms were available at that time. With the recent introduction of fast methods for spectral transforms based on BDDs (Clarke and Zhao 1993 [?]) the method of Hurst, Miller and Muzio may obtain some practical value again.

Tokmen [?], discussed disjoint decomposability of multiple-valued functions by spectral means. Muzio and Hurst [?], found complete and reduced sets of orthogonal spectral coefficients for logic design. Thus, this method could be useful for spectral-based decomposition.

Varma and Trachtenberg [?] created a fast algorithm for the optimal state assignment of large Finite State Machines using spectral methods. Because the algorithm is very fast comparing to other published state assignment algorithms, it can be used for encoding purposes in decomposition.

Recently, the work by Dressig et al. [?, ?, ?] reformulated the spectral approach in the form of "groupability" analysis method, especially tuned for FPGA mapping. It should be further investigated whether the "spectral groupability" of Dressig et al has any advantages over "functional groupability" of Bochmann and Steinbach, that we have already introduced above as one of the leading recent ideas worthy our further investigations.

In this section we will present in more detail some spectral approaches that may be used in a general-purpose decomposer.

Decomposition of Boolean functions into linear blocks (linear blocks are multi-output combinational functions implemented with EXOR gates only) and nonlinear blocks (which requires a complete base AND, OR, NOT for instance) was discussed in Karpovsky [?], and R. Lechner and A. Moezzi [?].

It was shown that by examining the autocorrelation function of Boolean function F one can easily design a linear preprocessor so that a cascade implementation of this function would have a reduced complexity. No equivalent procedures for such a decomposition in the Boolean domain have been yet proposed. This fact demonstrates that at least in some problem formulations the spectral methods can solve problems that the functional methods can not.

Since the general AC decomposition is time consuming, research has been concentrated on finding ways of decomposing a function into sub-functions of restricted types (a decomposition into sub-functions of certain type will be called a "biased" decomposition. In essence, all practical decompositions are biased (even assuming disjointness of the bound and free sets of variables in a disjoint decomposition is also a "bias"). Special decompositions with EXOR (linear) preprocessor or EXOR postprocessor (EXOR decomposition from TRADE, [?]) can be considered, that assume the usage of EXOR operators of sub-functions or input variables. For instance, these ideas occur also in recent Machine Learning related papers of Wnek [?].

Varma and Trachtenberg found that EXOR decomposition is especially good for functions that belong to the class of arithmetic, error correcting and translating logic, such as code converters, adders, and functions with embedded additions. Symmetric functions, which are usually difficult, also benefit from the linearization process (linearization process consists in creating new variables that are EXORs of some subsets of input variables. It is also called linear decomposition). Randomly defined functions are not considerably affected by linear decompositions.

Functions with more than about 20 variables, which so far have been very difficult to handle with spectral methods, can be now treated efficiently. This is in striking contrast with other spectral methods which ALWAYS required exponential number of operations and storage.

Spectral methods are used in POLO system [?] to detect EXOR part of a Boolean function. Since POLO system uses spectral methods together with programs based on the "unate paradigm" such as those in the SIS package from U.C. Berkeley, POLO can easily handle not only functions that are close to the strongly unate ones, but the strongly non-unate functions as well. The decomposition of Boolean functions with pre- and post-linear parts by spectral means leads to greatly simplified circuits, which may have application in Machine Learning. Moreover, these circuits are highly testable. They can also be efficiently implemented in several existing FPGA technologies. Such decompositions are possible using the spectral approach only. They create a layer of EXOR gates at the inputs, at the outputs, or at the inputs and outputs of the multi-output function. The "EXOR Decompositions" can be applied in tandem with any other logic synthesis method, which in some cases (for instance, for arithmetic circuits) leads to a powerful synergy of the methods.

An interesting possibility investigated in POLO was to relate the two areas of logic synthesis - that have had previously very little in common - the standard "functional" approach (that uses such functional representations as cube calculus and decision diagrams), and the spectral methods. This approach gave new insights to spectral methods and allowed to "demystify" these methods as basically the "pattern matching" methods. It allowed also to create several new families of generalizations of spectral transforms. This approach can be extended to realize the algorithms using the Binary Decision Diagrams (BDDs) and the Shared, Reduced, Ordered Kronecker Decision Diagrams (KDDs) (which are EXOR-based generalizations of BDDs [?, ?]). This representation would make all the algorithms more memory efficient.

In addition, several other families of new orthogonal transforms have been found [?, ?, ?]. These transforms include the Arithmetic Transform, the Adding Transform and the Generalized Walsh Transform [?, ?]. Because of the analogous properties to the Walsh Transform, all these new transforms could find the same applications in logic synthesis, and particularly in the decomposition and characterization of Boolean functions as the Walsh Transform (the last transform is better known). However, in contrast to the classical Walsh Transform, but similarly to the Fixed Polarity Reed-Muller Transform, these new transforms have various "polarities" of input variables - so practically we can consider entire "families" of such transforms. In particular, they allow to create pre- and post- processors with some standard gates, whose types correspond to the selected type and polarity of the transform. These gates include EXOR, AND/EXOR universal gate of Davio Expansion, and other "standard cells" with high EXOR component. Similarly to Walsh, the "remainder" function, created after the removal of the pre- and post- processors, can be synthesized using any other logic synthesis method.

It should be noted here, that all the previously mentioned spectral methods that were historically based on Walsh Transform can be now investigated in light of the new transforms. Moreover, all the transforms can be now represented with efficient Decision Diagrams Methods to create such diagrams for transforms have been recently introduced. Although some of these approaches have advantages for limited fan-in libraries of cells with high EXOR component, it is not yet sure, how these decompositions compare to AC decompositions for DFC minimization. All these are open research problems.

a
b g1
a F1
c
d
e

Figure 14.1: Serial decomposition with single output predecessor, one stage

a
c g2
a1 F2
d
e

Figure 14.2: Stage two decomposition F_2

14.2 Roth-Karp Decomposition

14.2.1 Introduction

Roth's serial decomposition technique, [?], can be illustrated as follows. Given $F(a, b, c, d, e)$ a selection is made of free and bound variable sets (these sets are not necessarily disjoint) and tests are made for the existance of single-output or multiple-output predecessor functions. Let us assume a single-output predecessor $g_1(a, b)$, as illustrated in Figure 14.6.

Function $F_1(g_1(a, b), a, c, d, e)$ is the image of F and its ON-OFF sets can be computed from the ON-OFF sets of F in terms of the outputs $g_1(a, b), a, c, d$, and e (as shown in a later example). Let us assume the simple disjoint case, i.e. $g_1(a, b)$ is a single-output function. The image of F will exist if the following conditions hold :

Given functions $F(B, A)$ and $g_1(B)$, there exists a function f such that:

$$F(B, A) \leq f(g_1(B), A) \tag{14.1}$$

(i.e. for each set of inputs B, A where the value of F is defined, f has the same value) if and only if:

1. input cubes b_i and b_j, are incompatible with respect to F ($F(b_i, a_k) \neq F(b_j, a_k)$), where same a_k implies $g_1(b_i) \neq g_1(b_j)$

   or equivalently,

2. $g_1(b_i) = g_1(b_j)$ implies compatible input cubes b_i and b_j, with respect to F, ($F(b_i, a_k) = F(b_j, a_k)$)

Function f is then called the image of $F(B, A)$, such that for all cubes b_i, a_j where the value of F is defined, $f(g_1(b_i), a_j)$ has the same value. Compatible cubes intersect and $\neq$ denotes different values of a scalar Boolean function.

If F_1 is still not realizable with a library component, its image $F_2(g_2(a, c), g_1, e, d)$ is formed if a single-output predecessor $g_2(a, c)$ exists.

The ON-OFF sets of F_2 can be also derived from the sets for F_1 in terms of g_2, g_1, e, and d; if F_2 is a realizable component then this is the last step of the decomposition process, otherwise the image of F_3 has to be obtained, etc.

	ON		OFF
	a b c d		a b c d
$u_1 =$	1 0 1 x	$v_1 =$	1 1 x 1
$u_2 =$	1 x 1 0	$v_2 =$	1 x 0 x
$u_3 =$	0 1 x x	$v_3 =$	x 0 0 x

Table 14.1: ON OFF cubes for bound set $B = \{ab\}$ and free set $A = \{cd\}$

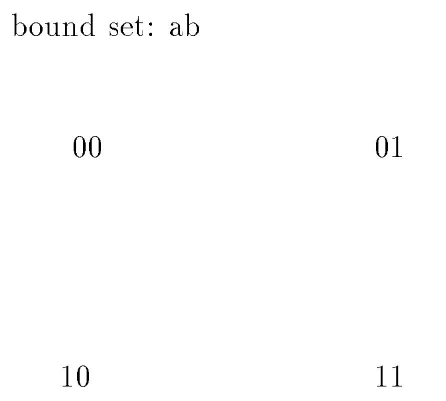


Figure 14.3: Compatibility Graph of $g_1(a, b)$ values

The test for existance of a single-output predecessor is as follows: We want to test if $F(a, b, c, d)$ has an image $F_1(g_1(B), A)$ where the bound set $B = \{a, b\}$ and the free set $A = \{c, d\}$. Let us arrange the ON and OFF array of F as in Table 14.6.

Let's denote the set of ON cubes by $U = \{u_1, u_2, u_3\}$ and the set of OFF cubes by $V = \{v_1, v_2, v_3\}$. Any two cubes u and v, one in the ON set and the other in the OFF set, must be mapped to distinct cubes in the image of F; if u and v have common free-set parts $u - B, v - B$, then the bound-set parts $u - A, v - A$, are the only distinguishing features of u and v and $g_1(u - A) \neq g_1(v - A)$ must hold. Otherwise some cubes of the image F_1 would appear in both the ON and OFF sets of F_1.

In the above example:
$u_1 - A =$ 10, $u_2 - A =$ 1x, $u_3 - A =$ 01;
$u_1 - B =$ 1x, $u_2 - B =$ 10, $u_3 - B =$ xx;
$v_1 - A =$ 11, $v_2 - A =$ 1x, $v_3 - A =$ x0;
$v_1 - B =$ x1, $v_2 - B =$ 0x, $v_3 - B =$ 0x;
The above requirement can be also stated as follows: the bound-parts of u and v are incompatible if their free-parts intersect.

For example $u_1 - B$ and $v_1 - B$ intersect at 11; $F(1011) = 1$, $F(1111) = 0$. Thus we must have:
$F_1(g_1(10), 11) \neq F_1(g_1(11), 11)$ and this requires $g_1(10) \neq g_1(11)$.

All pairs of cubes u, v must be considered to determine if a decomposition with g_1 exists. A systematic way to do this is to list all pairs which intersect and the corresponding incompatible bound-part cubes. Figure 14.9 shows a graph of compatible values of predecessor $g_1(a, b)$.
This graph can be interpreted as the following set of implications :

- $u_1 - B$ intersects $v_1 - B$ implies $g_1(10) \neq g_1(11)$

a

b g1

a

b g2 H

c

d

Figure 14.4: Two-output predecessor block in serial decomposition

- $u_3 - B$ intersects $v_1 - B$ implies $g_1(01) \neq g_1(11)$
- $u_3 - B$ intersects $v_2 - B$ implies $g_1(01) \neq g_1(1\text{x})$
- $u_3 - B$ intersects $v_3 - B$ implies $g_1(01) \neq g_1(\text{x}0)$

The above Compatibility Graph from Fig. 14.9 cannot be partitioned into two cliques of compatible elements, therefore $g_1(a, b)$ has 3 distinct values:

$g_1(10) \neq g_1(11) \neq g_1(01)$.

This implies that $g_1(a, b)$ can not be a single-output binary function. The Roth-Karp algorithm [?] considers a multiple-output predecessor at this point, such that the number of mutually compatible bound-part sets $\leq 2^t$ where t is the number of outputs of the predecessor: $F_1(g_1(B), g_2(B), \ldots, g_t(B), A)$.

In the above example there were three compatible bound-part sets 11, 10, and 01. Hence a decomposition can be found if a two-output predecessor block is used $3 < 2^2, t = 2$. Two-output predecessor block, as required by Roth's test for single-output/Ashenhurst predecessor above is shown in Figure 14.9.

Another way to state the above requirement is that the ON set of F must be identical to the ON set of H (image of F). The ON terms of F above can be mapped to the distinct values of $g_1(ab)g_2(ab)cd$ (minterms in the ON set of the image H). The truth table for F is as in Table 14.7.

a	b	c	d	F
1	0	1	0	1
1	0	1	1	1
1	1	1	0	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1

Table 14.2: Truth table for F

With a, b as the bound set variables, the decomposition chart shows that one possible mapping is ab = 10 to g_1g_2 = 00, ab = 11 to g_1g_2 = 01, and ab = 01 to g_1g_2 = 10. The truth table for H can then be filled in as in Table 14.8.

Each row corresponds to a row in the table for F, with variables ab mapping to g_1g_2 as specified above. This represents an arbitrary encoding of the columns in the decomposition chart of F given by Kmap from Table 14.9.

g_1	g_2	c	d	H
0	0	1	0	1
0	0	1	1	1
0	1	1	0	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1

Table 14.3: Truth table of H, for $g_1(a,b)$, $g_2(a,b)$

cd-ab	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	-	1	0	1
10	-	1	1	1
	C_1	C_2	C_3	C_4

Table 14.4: Kmap

The relation between the input cubes of the bound set variables ab and g_1g_2 is shown in Table 14.10.
Columns C_1, C_3 form compatible $class_1$, encoded with $g_1g_2 = 01$, C_2 is $class_2$ with $g_1g_2 = 10$, C_4 is $class_3$ with $g_1g_2 = 00$. H depends on the predecessor functions g_1 and g_2 according to:

$$H = g_1\overline{g_2} + \overline{g_1}c\overline{d} + \overline{g_2}c$$

Also $g_1(a,b) = \overline{a}b$ and $g_2(a,b) = \overline{ab} + ab$.

The final image:
$H = \overline{a}b + (a+\overline{b})c\overline{d} + (a\overline{b} + b\overline{a})c$

a	b	g_1	g_2
0	0	0	1
0	1	1	0
1	0	0	0
1	1	0	1

Table 14.5: Bound set encoding table

The above observations have been stated formally in two theorems by Roth and Karp [?]:

Theorem 14.2.1 *Given $F(B,A)$ and $g_1(B)$, there exists $F_1(g_1(B),A) = F$ iff for all cubes b_i and b_j included in B, b_i incompatible (non-intersecting) with b_j implies $g_1(b_i) \neq g_1(b_j)$.*

Theorem 14.2.2 *If the cubes of bound set B can be partitioned into k mutually compatible sets then a decomposition with a t-output predecessor exists, where $k \leq 2^t$ and*

$$F = F_1(g_1(B), \ldots, g_t(B), A)$$

A circuit synthesis procedure was presented in Karp [?], based on a-priori knowledge of all the predecessor (library) functions $g_1(), g_2(), \ldots, g_k()$. This requirement can not be maintained in a general decomposition framework. Instead, any form of the predecessors $g_i()$ can be assumed and the DFC minimization strategy should rely on bound and free set evaluations for decomposition forms such as simple (column multiplicity = 2) disjoint, iterative disjoint, and complex disjoint and non-disjoint. The image of F should be constructed at each level as in the example above.

The iterative disjoint form has been defined in Karp [?], pp.300-303 as shown in Figure 14.10, with single output predecessor at each stage.

S1 g1

S2 g2

S3 g3

Figure 14.5: Iterative serial disjoint decomposition

The $S1, S2$, and $S3$ are disjoint sets and cover all variables of F, and all $g_i()$ functions are single output.

14.3 Bibilo-Yenin Decomposition.

Ideas of Ashenhurst and Curtis, and particularly Roth and Karp have been re-implemented by Bibilo and Yenin before 1987 and published in a book [?]. However, the comparison of the quality of their method with the quality of similar approaches in the West is difficult, since they did not use benchmark functions. In the book there is no critical evaluation of the proposed improvements, no references to recent Western literature, and no comparisons to Western programs. They use more efficient algorithms for partitioning and column compaction which is performed together with the encoding. This is perhaps a very good idea worth further study. It is somehow similar to the concurrent state minimization and state assignment, as proposed in [?, ?].

Bibilo and Yenin report to solve functions of 40 input variables, and they tested their programs on randomly generated functions. Since randomly generated functions are very hard, the performance of their approach seems to be good, but it is hard to assess. Some of their algorithms should be implemented and compared.

14.4 He and Tolkersen Decomposition.

In 1993 He and Tolkersen proposed a new Boolean extraction algorithm based on K-maps. Because of K-map size restrictions, their algorithm is not very practical as presented. However, there are certain new ideas in their paper which can be perhaps adapted to one of the modern representations of Boolean functions, such as BDDs or partitioned representations. This will be discussed in the sequel.

14.5 Spectral Approach of Shen and McKellar.

The main drawback of using the basic method of Disjoint Decomposition is to check $2^m - m - 2$ "maps" for an m-input function. However, it is possible to devise algorithms that use necessary conditions for the existence of a decomposition in order to prune out certain combinations of input variables which do not belong to bound sets of any decompositions. This effectively speeds up the procedure in all but pathological cases.

The search for a fast algorithm for the disjunctive decompositions of switching binary functions was studied by Shen at el [?, ?, ?]. They presented a fast algorithm based on testing a necessary condition for decomposability of switching functions. During this process, candidate bound (CB) sets were generated. The CB sets are a smaller subset of all possible bound sets. By performing an additional minor test on CB sets, Shen et al were able to determine the decomposability of a switching function in disjunctive form. Shen and McKellar [?, ?, ?] devised an algorithm that detected candidate partitions for disjoint decompositions of logic functions but required further testing of candidates, even though the set of candidates contained far fewer partitions that the original set of all possible partitions. They also showed that Reed-Muller (RM) canonical form was easier to test for decompositions than the disjunctive normal form (DNF). Thus, the idea of a speedup resulting from applying a new function representation has been further reinforced.

14.6 Roth-Karp Decomposition

14.6.1 Introduction

Roth's serial decomposition technique, [?], can be illustrated as follows. Given $F(a,b,c,d,e)$ a selection is made of free and bound variable sets (these sets are not necessarily disjoint) and tests are made for the existance of single-output or multiple-output predecessor functions. Let us assume a single-output predecessor $g_1(a,b)$, as illustrated in Figure 14.6.

Figure 14.6: Serial decomposition with single output predecessor, one stage

Function $F_1(g_1(a,b),a,c,d,e)$ is the image of F and its ON-OFF sets can be computed from the ON-OFF sets of F in terms of the outputs $g_1(a,b),a,c,d,$ and e (as shown in a later example). Let us assume the simple disjoint case, i.e. $g_1(a,b)$ is a single-output function. The image of F will exist if the following conditions hold :

Given functions $F(B,A)$ and $g_1(B)$, there exists a function f such that:

$$F(B,A) \leq f(g_1(B),A) \tag{14.2}$$

(i.e. for each set of inputs B,A where the value of F is defined, f has the same value) if and only if:

1. input cubes b_i and b_j, are incompatible with respect to F $(F(b_i,a_k) \neq F(b_j,a_k))$, where same a_k implies $g_1(b_i) \neq g_1(b_j)$

   or equivalently,

2. $g_1(b_i) = g_1(b_j)$ implies compatible input cubes b_i and b_j, with respect to F, $(F(b_i,a_k) = F(b_j,a_k))$

Function f is then called the image of $F(B,A)$, such that for all cubes b_i, a_j where the value of F is defined, $f(g_1(b_i),a_j)$ has the same value. Compatible cubes intersect and $\neq$ denotes different values of a scalar Boolean function.

If F_1 is still not realizable with a library component, its image $F_2(g_2(a,c),g_1,e,d)$ is formed if a single-output predecessor $g_2(a,c)$ exists.

Figure 14.7: Stage two decomposition F_2

The ON-OFF sets of F_2 can be also derived from the sets for F_1 in terms of $g_2,g_1,e,$ and d; if F_2 is a realizable component then this is the last step of the decomposition process, otherwise the image of F_3 has to be obtained, etc.
The test for existance of a single-output predecessor is as follows: We want to test if $F(a,b,c,d)$ has an image $F_1(g_1(B),A)$ where the bound set $B=\{a,b\}$ and the free set $A=\{c,d\}$. Let us arrange the ON and OFF array of F as in Table 14.6.

Let's denote the set of ON cubes by $U=\{u_1,u_2,u_3\}$ and the set of OFF cubes by $V=\{v_1,v_2,v_3\}$. Any two cubes u and v, one in the ON set and the other in the OFF set, must be mapped to distinct cubes in the image of F; if u and v have common free-set parts $u-B, v-B$, then the bound-set parts $u-A, v-A$, are the only distinguishing features of u and v and $g_1(u-A) \neq g_1(v-A)$ must hold. Otherwise some cubes of the image F_1 would appear in both the ON and OFF sets of F_1.

In the above example:
$u_1 - A = 10$, $u_2 - A = 1\text{x}$, $u_3 - A = 01$;

	ON		OFF
	a b c d		a b c d
$u_1 =$	1 0 1 x	$v_1 =$	1 1 x 1
$u_2 =$	1 x 1 0	$v_2 =$	1 x 0 x
$u_3 =$	0 1 x x	$v_3 =$	x 0 0 x

Table 14.6: ON OFF cubes for bound set $B = \{ab\}$ and free set $A = \{cd\}$

$u_1 - B =$ 1x, $u_2 - B =$ 10, $u_3 - B =$ xx;
$v_1 - A =$ 11, $v_2 - A =$ 1x, $v_3 - A =$ x0;
$v_1 - B =$ x1, $v_2 - B =$ 0x, $v_3 - B =$ 0x;
The above requirement can be also stated as follows: the bound-parts of u and v are incompatible if their free-parts intersect.

For example $u_1 - B$ and $v_1 - B$ intersect at 11; $F(1011) = 1$, $F(1111) = 0$. Thus we must have:
$F_1(g_1(10), 11) \neq F_1(g_1(11), 11)$ and this requires $g_1(10) \neq g_1(11)$.

All pairs of cubes u, v must be considered to determine if a decomposition with g_1 exists. A systematic way to do this is to list all pairs which intersect and the corresponding incompatible bound-part cubes. Figure 14.9 shows a graph of compatible values of predecessor $g_1(a, b)$.

Figure 14.8: Compatibility Graph of $g_1(a, b)$ values

This graph can be interpreted as the following set of implications :

- $u_1 - B$ intersects $v_1 - B$ implies $g_1(10) \neq g_1(11)$
- $u_3 - B$ intersects $v_1 - B$ implies $g_1(01) \neq g_1(11)$
- $u_3 - B$ intersects $v_2 - B$ implies $g_1(01) \neq g_1(1\text{x})$
- $u_3 - B$ intersects $v_3 - B$ implies $g_1(01) \neq g_1(\text{x}0)$

The above Compatibility Graph from Fig. 14.9 cannot be partitioned into two cliques of compatible elements, therefore $g_1(a, b)$ has 3 distinct values:

$g_1(10) \neq g_1(11) \neq g_1(01)$.

This implies that $g_1(a, b)$ can not be a single-output binary function. The Roth-Karp algorithm [?] considers a multiple-output predecessor at this point, such that the number of mutually compatible bound-part sets $\leq 2^t$ where t is the number of outputs of the predecessor: $F_1(g_1(B), g_2(B), \ldots, g_t(B), A)$.

In the above example there were three compatible bound-part sets 11, 10, and 01. Hence a decomposition can be found if a two-output predecessor block is used $3 < 2^2, t = 2$. Two-output predecessor block, as required by Roth's test for single-output/Ashenhurst predecessor above is shown in Figure 14.9.

Figure 14.9: Two-output predecessor block in serial decomposition

Another way to state the above requirement is that the ON set of F must be identical to the ON set of H (image of F). The ON terms of F above can be mapped to the distinct values of $g_1(ab)g_2(ab)cd$ (minterms in the ON set of the image H). The truth table for F is as in Table 14.7.

a	b	c	d	F
1	0	1	0	1
1	0	1	1	1
1	1	1	0	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1

Table 14.7: Truth table for F

With a, b as the bound set variables, the decomposition chart shows that one possible mapping is $ab = 10$ to $g_1g_2 = 00$, $ab = 11$ to $g_1g_2 = 01$, and $ab = 01$ to $g_1g_2 = 10$. The truth table for H can then be filled in as in Table 14.8.

g_1	g_2	c	d	H
0	0	1	0	1
0	0	1	1	1
0	1	1	0	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1

Table 14.8: Truth table of H, for $g_1(a,b)$, $g_2(a,b)$

Each row corresponds to a row in the table for F, with variables ab mapping to g_1g_2 as specified above. This represents an arbitrary encoding of the columns in the decomposition chart of F given by Kmap from Table 14.9.

cd-ab	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	-	1	0	1
10	-	1	1	1
	C_1	C_2	C_3	C_4

Table 14.9: Kmap

The relation between the input cubes of the bound set variables ab and g_1g_2 is shown in Table 14.10.

Columns C_1, C_3 form compatible $class_1$, encoded with $g_1g_2 = 01$, C_2 is $class_2$ with $g_1g_2 = 10$, C_4 is $class_3$ with $g_1g_2 = 00$. H depends on the predecessor functions g_1 and g_2 according to:

$$H = g_1\overline{g_2} + \overline{g_1}c\overline{d} + \overline{g_2}c$$

Also $g_1(a,b) = \overline{a}b$ and $g_2(a,b) = \overline{a\overline{b}} + ab$.

The final image:
$H = \overline{a}b + (a + \overline{b})c\overline{d} + (a\overline{b} + b\overline{a})c$
The above observations have been stated formally in two theorems by Roth and Karp [?]:

a	b	g_1	g_2
0	0	0	1
0	1	1	0
1	0	0	0
1	1	0	1

Table 14.10: Bound set encoding table

Theorem 14.6.1 *Given $F(B,A)$ and $g_1(B)$, there exists $F_1(g_1(B),A) = F$ iff for all cubes b_i and b_j included in B, b_i incompatible (non-intersecting) with b_j implies $g_1(b_i) \neq g_1(b_j)$.*

Theorem 14.6.2 *If the cubes of bound set B can be partitioned into k mutually compatible sets then a decomposition with a t-output predecessor exists, where $k \leq 2^t$ and*

$$F = F_1(g_1(B), \ldots, g_t(B), A)$$

A circuit synthesis procedure was presented in Karp [?], based on a-priori knowledge of all the predecessor (library) functions $g_1(), g_2(), ..., g_k()$. This requirement can not be maintained in a general decomposition framework. Instead, any form of the predecessors $g_i()$ can be assumed and the DFC minimization strategy should rely on bound and free set evaluations for decomposition forms such as simple (column multiplicity = 2) disjoint, iterative disjoint, and complex disjoint and non-disjoint. The image of F should be constructed at each level as in the example above.

The iterative disjoint form has been defined in Karp [?], pp.300-303 as shown in Figure 14.10, with single output predecessor at each stage.

Figure 14.10: Iterative serial disjoint decomposition

The $S1, S2$, and $S3$ are disjoint sets and cover all variables of F, and all $g_i()$ functions are single output.

14.7 PLA Decomposition.

Several authors, including Ciesielski, [?, ?], Devadas [?, ?], and Sasao, [?] proposed to apply Boolean decomposition to decompose a PLA into two or more cascaded PLAs. However, from the point of view of the mathematical apparatus used, these methods have little in common with the classical functional decomposition problem formulation, and can be rather treated as a multiple-valued minimization and input-encoding problems (for multiple-valued minimization see Brayton [?], Muzio et al [?]. and Sasao [?]). Nevertheless, the techniques introduced by them are very efficient, and, because of certain similarities of multiple-valued logic and decomposition, they should be considered by the proponents of the classical FD model as a possible source of enhancing ideas. Although their methods have been presented entirely in the PLA technology framework, they can be perhaps also modified to other types of "macro-blocks" or "universal cells".

14.7.1 Approach of Tsutomu Sasao to Multiple-Valued PLA Decomposition.

Contrary to Ciesielski and other previous authors, Sasao used classical AC decomposition concepts to PLA decomposition.

In 1989 Tsutomu Sasao, [?], presented an application of multiple-valued logic to serial decomposition of PLAs. His approach resembles the classical Curtis decomposition with all predecessor blocks G combined to a single multi-output block. Sasao divides the problem into two subproblems: *a partition problem* and *an encoding problem*. His first stage is very similar to all multiple-valued Curtis-like decomposition algorithms, with the only difference that the entries in the table are still binary (he discusses a multiple-valued input, binary output functions). Since this is a Curtis-like decomposition and not an Ashenhurst-like decomposition, he allows the column multiplicity index value to be higher than 2.

In order to find a simple circuit he looks for a partition with the smallest multiplicity value. For the n variable function there are 2^n different partitions. When n is small ($\leq$16) a brute force method is used. For larger n the following approach is advocated. The number of variables in the bound set is denoted by n_1. Each column of the decomposition chart is represented by a logical expression. The number of expressions in the chart is 2^{n_1}. Therefore when n_1 is small the decomposition chart can be represented with small memory storage. An equivalence of two columns is checked as an equivalence of two logical expressions.

Sasao was one of the first authors to discuss the encoding problem in the framework of functional decomposition. He experimented with the "one-hot" encoding and the "minimum-length" encoding, and he calculated the total sizes of the predecessor and successor's PLAs. A very simple encoding algorithm was used by him - "the more frequent the pattern occurs in the decomposition chart, the more the number of 0's in the code". Sasao found that the one-hot encoding worked better than the minimum-length encoding. The paper was concluded with the desire of finding a good heuristic method to find a good partition more efficiently and to find a better heuristic for the minimum-length encoding.

Chapter 15

Bi-Decomposition of Functions.

15.1 The Approach of Steinbach et al.

The approach of Steinbach et al is the result of many years of research of Bochmann, Posthoff and Dresig. This is a 'gate-type' variant of a decomposition. Two-input gates: AND, OR, EXOR, and a single-input NOT gate are assumed. The decompositions for these gates can be either disjoint or non-disjoint.

Although this is is not a Functional Decomposition and the concept of multiplicity index is not used, this decomposition is more similar to AC than any other 'gate type' decomposition from the literature. It can be also shown that their conditions for decomposability can be derived from the basic theorems of Ashenhurst.

The sources of these original ideas seem to start from the independent works of Davio and Zakrevskij. Bochmann himself was perhaps influenced by the work of Zakrevskij's group, one of the three top logic synthesis groups in the former Soviet Union. Zakrevskij was the creator of the original algebraic ideas of this line of research; such as the definitions of differential, maximum and minimum. Some of them may go back perhaps to the early papers of Davio, Thayse and Deschamps. Many of the respective early papers are still unavailable to us.

All recent ideas of this German group are implemented in programming system XBOOLE, therefore we will be referring to the *XBOOLE approach.*

There are five properties that are interesting about the approach of XBOOLE:

- The totally algebraic formulation of the decomposition problem. The definition of Boolean operators that are powerful and universal, and do not depend on the actually used representation of Boolean functions. Being purely Boolean concepts, these operators can be implemented in conjunction with any other representations of binary functions. This can be taken advantage of by reformulating these operators for other types of decompositions: multiple-valued, fuzzy, and other logic. The decomposition method in the most general sense expressed by the operators should remain the same. The symbolic operators used in XBOOLE are: complementation, cofactor, AND, OR, MIN, MAX, and EXOR. The main difficulty in generalizing these ideas to multiple valued logic is in having several complementation operators in multiple-valued logics.

- The efficient use of Ternary Vector Lists, a kind of 'Cube Calculus', in which all the cubes are disjoint. They are being made disjoint ('orthogonalized'), immediately after execution of every operation that may produce non-disjoint cubes.

- The decomposition used by XBOOLE is actually not an AC decomposition, since it decomposes each function to two-input gates: AND, OR and EXOR, and possibly also Inverters. This approach allows to decompose large functions relatively quickly. Contrary to other 'Gate Decomposition' methods, we discuss XBOOLE system here in more detail, because:

  - several of its ideas are still very close to the AC decomposition (for instance the general decomposition models or variable partitioning),
  - we believe that its methods are general and can be adapted for Machine Learning purposes.

  We will call it 'XBOOLE Decomposition'. Such decomposition can be one of 'Special Bias' Decomposition in the future decomposer proposed in this document. It is much closer to AC decomposition than other 'gate-oriented' Boolean decompositions, like those of Zaky [?], Stoffers, Dietmeyer and Duley, and Darringer [?].

- Every step in the decomposition can have a large amount of different results because of different algorithms, different basis elements, and the ambiguity of the decomposition. The program allows then to be controlled by some parameters in such a way that the result optimally meets some kind of user requirements. Thus, it can be tuned to various technological parameters, such as the area, the number of gates, the depth of the circuit, the power consumption, or the circuit's testability. An overview of different kinds of XBOOLE decompositions is given in [?, ?].

- The method can efficiently handle the don't cares.
- The method can be easily converted to either the BDDs with don't cares (ternary BDDs, an extension of a BDD that has three terminal nodes, corresponding to '0', '1', and '-') or to the pairs of standard BDDs that represents ON and OFF sets. XBOOLE uses a pair of cube represented completely specified functions q (we denote it by ON) and r (we denote it by OFF)to represent an incompletely specified function.

XBOOLE decomposition is based on the concept of 'groupability'.

Definition 15.1.1 *(groupability, Steinbach).*
A Boolean function $f(X_a, X_b, X_c)$ is said to be (disjunctively, conjunctively of exclusive-or) groupable with the operation $\bigcirc \in$ { AND, OR, EXOR } and variable sets X_a, X_b if

$$f(X_a, X_b, X_c) \;=\; g(X_a, X_c) \;\bigcirc\; h(X_b, X_c) \tag{15.1}$$

Although not all Boolean functions are groupable, it has been proven in [?] that all Boolean functions, which are not groupable by the EXOR gate decomposition, are always disjunctively or conjunctively weakly groupable.

Definition 15.1.2 *(weak groupability) (Steinbach).*
A Boolean function $f(X_a, X_b)$ is said to be (disjunctively, conjunctively, or exclusive-or) weakly groupable with operation $\bigcirc \in$ { OR, AND, EXOR } and the variables set X_a if

$$f(X_a, X_c) \;=\; g(X_a, X_c) \;\bigcirc\; h(X_c) \tag{15.2}$$

A weakly groupable function can be implemented by several weak groupings. For a disjunctively or conjunctively weakly groupable function $f(X_a, X_b)$ the function $h(X_c)$ can always be selected in such a way that $g(X_a, X_c)$ can be implemented more easily than $f(X_a, X_c)$, although the number of variables is not altered.

The question 'is an incompletely specified function with the functions $ON(X)$ and $OFF(X)$ for X_a and X_b groupable?' can be answered using the following existence theorems.

The theoretical basis is explained in works of Steinbach, Le, Dresig, Bochmann, and Posthoff.

Theorem 15.1.1 *(disjunctively groupable) (Steinbach).*
An incompletely specified function with the functions $ON(X)$ and $OFF(X)$ for X_a and X_b is disjunctively groupable, *if and only if*

$$ON(X) \wedge max_{X_a}{}^k \; OFF(X) \wedge max_{X_b}{}^k \; OFF(X) = 0 \tag{15.3}$$

The differential equation 15.3 implies that no pattern from the ON-set (represented by ON(X)) may be in the projection of the OFF-set (OFF(X)) in the X_a as well as in the X_b direction.

Dual situations exist in conjunctively groupable functions.

Theorem 15.1.2 *(conjunctively groupable) (Steinbach).*
An incompletely specified function with the functions $ON(X)$ and $OFF(X)$ for X_a and X_b is conjunctively groupable, *if and only if*

$$OFF(X) \wedge max_{X_a}{}^k \; ON(X) \wedge max_{X_b}{}^k \; ON(X) = 0 \tag{15.4}$$

The exclusive-or groupability for X_a and X_b is determined step by step. First, the derivation of the incompletely specified function for one variable x_a is calculated using Lemma 15.1.1.

Lemma 15.1.1 *(derivation of an incompletely specified function) (Steinbach).*
The derivation of an incompletely specified function with respect to variable x_a forms also an incompletely specified function with the functions

$$ON_a(X) \;=\; max_{x_a}\, ON(X) \;\wedge\; max_{x_a}\, OFF(X) \tag{15.5}$$

$$OFF_a(X) \;=\; min_{x_a}\, ON(X) \;\vee\; min_{x_a}\, OFF(X) \tag{15.6}$$

Theorem 15.1.3 *(exclusively-or groupable) (Steinbach).*
An incompletely specified function with the functions $ON(X)$ and $OFF(X)$ is exclusively-or groupable for one variable x_a and set of variables X_b, if and only if

$$ON_a(X) \;\wedge\; max_{X_b}{}^{k}\; OFF_a(X) \;=\; 0 \tag{15.7}$$

The test (15.7) for (x_{a1}, X_b) and (x_{a2}, X_b) only confirms the exclusive-or groupability for $((x_{a1}, x_{a2}), X_b)$, if there is at least one specified function within the incompletely specified function which meets the requirements for x_{a1} as well as for x_{a2}. The probability of finding such specified functions is increased by using Theorem 15.1.4.

Theorem 15.1.4 *(restrictions of an incompletely specified function) (Steinbach).*
The reduced incompletely specified function with the functions

$$ON(X) := ON(X) \vee OFF(X \setminus x_a, \overline{x_a}) max_{X_b}{}^{k} ON_a(X) \vee ON(X \setminus x_a, \overline{x_a}) max_{X_b}{}^{k} OFF_a(X) \tag{15.8}$$

$$OFF(X) := OFF(X) \vee ON(X \setminus x_a, \overline{x_a}) max_{X_b}{}^{k} ON_a(X) \vee OFF(X \setminus x_a, \overline{x_a}) max_{X_b}{}^{k} OFF_a(X) \tag{15.9}$$

contains all Boolean functions of the primary incompletely specified functions that are exclusive-or groupable for (x_a, X_b).

The exclusive-or weak groupability is a property of all Boolean functions. But only functions that satisfy theorems 15.1.5 or 15.1.6 enable the disjunctively or conjunctively weak groupings.

Theorem 15.1.5 *(disjunctively weakly groupable) (Steinbach).*
An incompletely specified function with the functions ON(x) and OFF(x) is disjunctively weakly groupable for X_a if and only if

$$ON(X) \;\wedge\; \overline{max_{X_a}{}^{k}\; OFF(X)} \;\neq\; 0 \tag{15.10}$$

The inequality (15.10) implies that not all patterns of the ON-set $(ON(X))$ may be in the projection of the OFF-set $(OFF(X))$ in the X_a-direction.

The patterns of ON(X) which are not covered by the 'zero-projection' belong to the function ON(X_c). Here, dual situations with regard to conjunctively weakly groupable functions can be also observed.

Theorem 15.1.6 *(conjunctively weakly groupable) (Steinbach).*
An incompletely specified function with the functions $ON(X)$ and $OFF(X)$ is **conjunctively weakly groupable** *for X_a, if and only if,*

$$OFF(X) \;\wedge\; \overline{max_{X_a}{}^{k}\; ON(X)} \;\neq\; 0 \tag{15.11}$$

If only those properties of incompletely specified functions are used which have been mentioned above, a convergent decomposition method can be developed. This is due to the following theorem.

Theorem 15.1.7 *(groupable) (Steinbach).*
Every Boolean function that is not disjunctively weakly groupable and not conjunctively weakly groupable has the property of exclusive-or groupability.

The algorithm is based on the idea that the more symmetrical and the larger the vectors X_a and X_b are, the better is the timing behavior of the design. The demand for symmetry is based on the fact that Boolean functions with equal number of variables produce circuits with similar numbers of gates.

A maximum size of X_a and X_b results in a smaller vector X_c, to make functions $g(X_a, X_c)$ and $h(X_b, X_c)$ being dependent on less variables.

As a criterion for the best grouping the smaller one of X_a and X_b should be as large as possible. If more than one grouping has the same number of variables in the smaller vector, the grouping with the largest vector is the best. In contrast to the area-optimizing algorithm where one vector is filled first, the numbers of variables in vectors X_a and X_b is increased alternately.

The algorithm uses a heuristic approach with certain rules when to add or subtract a variable, and which one, from sets X_a and X_b. In this sense it is very different from all other variable partitioning methods from the literature.

The largest examples presented in the paper are of a medium size: misex2, has 25 inputs and 18 outputs, duke2 has 22 inputs and 29 outputs. It is then not known, what are the practical limits of the method. The functions are smaller than in Lai/Pedram's approach, but comparable to other recent top decomposers by Bibilo, Luba, and TRADE.

15.2 Strong AND/OR/EXOR Decompositions.

In this section we will present our implementation in GUD-2 of the "gate-type" decomposition approach of Steinbach, Bochmann, Posthoff and Dressing. This is "gate-type" decomposition because two-input gates: AND, OR, and EXOR are assumed. The decompositions for these gates can be either disjoint or non-disjoint. Although this is not a Functional Decomposition and the concept of multiplicity index is not used, this decomposition is more similar to AC decomposition than any other "gate type" decomposition from the literature. This approach allows to decompose large functions relatively quickly, and the method can handle efficiently don't cares.

Symbolic Boolean operators are defined that are powerful and universal, do not depend on the actually used representation of Boolean functions, and, being purely Boolean concepts, can be implemented in conjunction with any other representations of binary functions. This can be taken advantage of by reformulating these symbolic operators for other types of decompositions: multiple-valued, fuzzy, and other types of logics. The general decomposition method expressed by applying these symbolic operators should remain the same.

The symbolic operators are: complementation, derivative, cofactor, AND, OR, MIN, MAX, and EXOR. In contrast to XBOOLE, which uses Ternary Vector List (TVL) data structure, and Luba who uses lists of minterm numbers, we use here a new kind of data structure, a *CDB*, introduced in section 3.3.

15.2.1 Boolean Operations

The following Boolean operations are necessary; *derivative, minimum* and *maximum.* They are defined as follows:

1. Derivative:
$$\frac{\partial f(a,b,c)}{\partial a} = f(a=0,b,c) \oplus f(a=1,b,c) \tag{15.12}$$

2. Minimum
$$\min_a f(a,b,c) = f(a=0,b,c) \cdot f(a=1,b,c) \tag{15.13}$$

3. Maximum
$$\max_a f(a,b,c) = f(a=0,b,c) + f(a=1,b,c) \tag{15.14}$$

For the Boolean operations defined above, we can see that all operations use cofactors of the given function. The derivative operation is the EXOR of positive and negative cofactors of the given function with respect to certain variable. The minimum operation is the AND of positive and negative cofactors of the given function with respect to certain variable, and the maximum operation is the OR of the positive and negative cofactors of the given function with respect to certain variable. If we want to find operations of derivative, minimum or maximum for more than one variable, the above operation definition should be applied iteratively.

Example 15.2.1

For a given function:

$$f(a,b,c,d) \;=\; a\overline{b} \;+\; \overline{a}b \;+\; \overline{a}cd \tag{15.15}$$

Derivative:

$$\frac{\partial^2 f(a,b,c,d)}{\partial a \partial b} = f(a=0,b=0,c,d) \oplus f(a=0,b=0,c,d)$$
$$\oplus f(a=0,b=1,c,d) \oplus f(a=1,b=0,c,d) \oplus$$
$$f(a=1,b=1,c,d) = cd \oplus 1 \oplus 1 \oplus 0 \;=\; cd \tag{15.16}$$

Minimum:

$$min^2_{[ab]} f(a,b,c,d) \;=\; f(a=0,b=0,c,d) \cdot f(a=0,b=1,c,d) \cdot f(a=1,b=0,c,d) \cdot$$
$$f(a=1,b=1,c,d) = cd \cdot 1 \cdot 1 \cdot 0 \;=\; 0 \tag{15.17}$$

$$min^2_{[cd]} f(a,b,c,d) \;=\; f(a,b,c=0,d=0) \cdot f(a,b,c=0,d=1) \cdot f(a,b,c=1,d=0) \cdot$$
$$f(a,b,c=1,d=1) = (\overline{a}b + a\overline{b}) \cdot (\overline{a}b + a\overline{b}) \cdot (\overline{a}b + a\overline{b}) \cdot$$
$$(\overline{a}b + a\overline{b} + \overline{a}) = \overline{a}b + a\overline{b} \tag{15.18}$$

Maximum:

$$max^2_{[ab]} f(a,b,c,d) \;=\; f(a=0,b=0,c,d) \;+\; f(a=0,b=1,c,d) \;+$$
$$f(a=1,b=0,c,d) \;+\; f(a=1,b=1,c,d) = cd \;+\; 1 \;+\; 1 \;+\; 0 \;=\; 1 \tag{15.19}$$

Graphical illustration of operations derivative, minimum and maximum for minterms in a completely specified function using K-maps are shown in Fig. 15.1.

Let us observe that the concepts of these three operators are absent from decompositions introduced by Luba et al, and are used in a different way in Ternary Vector List representation of XBOOLE.

15.2.2 Two-input Decompositions

By a *two-input decompositions* we understand the decomposition of a binary function $f(A,B,C)$ into two functions $g(A,C)$ and $h(B,C)$ with non-empty, disjoint sets of variables A and B.

Depending upon the used operation $\circ$, which is OR, AND, or EXOR two-input gate, the decomposition is called OR, AND or EXOR, respectively. (They are also called disjunctive, conjunctive and linear decompositions, respectively). The best decomposition occurs, if all variables can be divided into disjoint sets A and B. If possible, these sets should also have equal numbers of elements. If $g(X)$ is equal to a single variable x_i and for $h(X) \;=\; h(X \setminus x_i)$, the decomposition degenerates into a simple splitting-off of a variable. The weakest form of the decomposition (the splitting-off mentioned above) exists if A and B contain only one variable each.

For each binary function and each possible partitioning of set of variables X to three disjoint variable sets A, B, and C, $X = (A,B,C)$, the definitions of OR, AND and EXOR decompositions are formulated.

Definition 15.2.1 *A Boolean function $f(A,B,C)$ is said to be strong (AND, OR, EXOR) decomposable with the operation $\bigcirc \in \{$ AND, OR, EXOR $\}$ and variable sets A, B if*

$$f(A,B,C) \;=\; g(A,C) \;\bigcirc\; h(B,C) \tag{15.20}$$

Although not all Boolean functions are decomposable, it has been proven that all Boolean functions, which are not decomposable by the EXOR gate decomposition, are always disjunctively or conjunctively weakly decomposable.

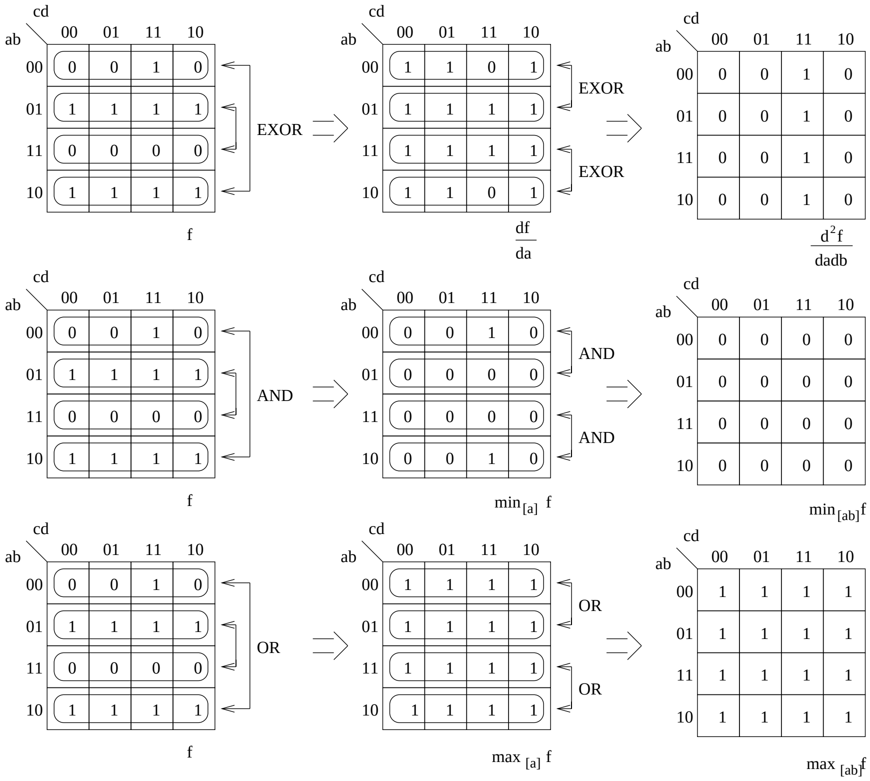


Figure 15.1: Graphical illustration of operations derivative, minimum and maximum for minterms

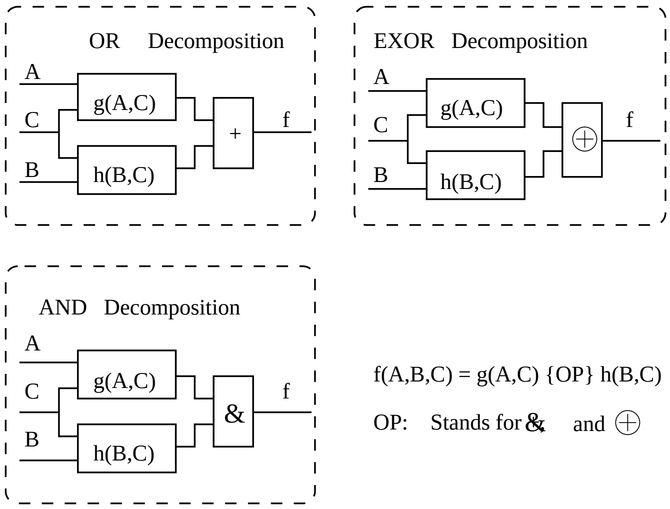


Figure 15.2: Strong OR (Disjunctive), AND (Conjunctive) and EXOR (Linear) Decompositions.

ab \ cd	00	01	11	10
0 0	0	0	1	X
0 1	1	X	1	1
1 1	0	0	X	0
1 0	1	1	1	1

f(a,b,c,d)

ab \ cd	00	01	11	10
0 0	0	0	1	0
0 1	1	0	1	1
1 1	0	0	0	0
1 0	1	1	1	1

$f(a,b,c,d)_{ON}$

ab \ cd	00	01	11	10
0 0	1	1	0	0
0 1	0	0	0	0
1 1	1	1	0	1
1 0	0	0	0	0

$f\,(a,b,c,d)_{OFF}$

Figure 15.3: The Care Functions f_{ON} and f_{OFF} of function $f(a,b,c,d)$ to Example 20.2

Definition 15.2.2 *A Boolean function $f(A,B)$ is said to be* (disjunctively, conjunctively, or linearly) weakly decomposable *with operation* $\bigcirc \in \{$ *OR, AND, EXOR* $\}$ *and the variables set A if*

$$f(A,B) \;=\; h(A,B) \;\bigcirc\; g(A) \tag{15.21}$$

Strong decompositions are illustrated in Figure 53.

The algorithm is based on the idea that the more symmetrical and the larger the vectors A and B are, the better is the design. The demand for symmetry is based on the fact that Boolean functions with equal number of variables in sets A and B produce circuits that are better balanced.

A maximum size of A and B results in a smaller vector C, making functions $g(A,C)$ and $h(B,C)$ to depend on less variables.

As a criterion for the best partitioning the smaller set of A and B should be as large as possible. If more than one partitioning has the same number of variables in the smaller vector, the partitioning with the largest vector is the best. In contrast to the area-optimizing algorithm where one vector is filled first, here the numbers of variables in vectors A and B are increased alternately.

Example 15.2.2

Let us look at the example in Fig. 15.3, where $f(a,b,c,d)$ stands for the original Boolean function, $f(a,b,c,d)_{ON}$ stands for the $ON - set$ function with don't care in $f(a,b,c,d)$ set to zero, and $f(a,b,c,d)_{OFF}$ stands for $OFF - set$ function with 0s in $f(a,b,c,d)$ set to 1, and 1s and don't cares set to 0.

Example 15.2.3

Graphical ilustrations of *derivative, min*, and *max* operations for function from Figure 15.4a are shown in Fig. 15.4b, c, d, respectively.

For two disjoint partitions of the set $X = (\ A, B, C\ )$ and X = $(\ A', B', C'\ )$ we define the half-order relation $\preceq$ as in Equation 15.22:

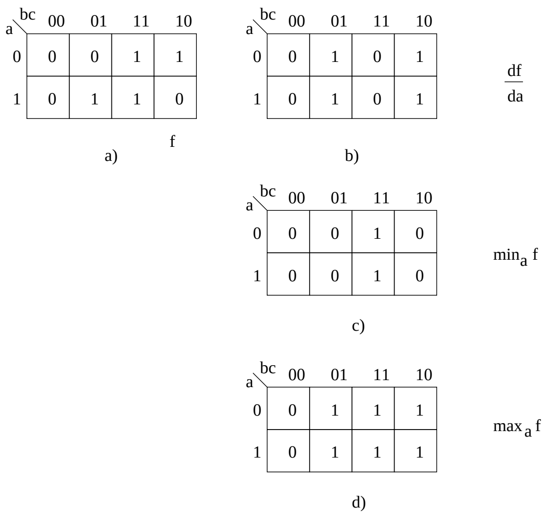


Figure 15.4: Example 20.3: (a) function $f(a,b,c)$, (b) its derivative with respect to variable a, (c) its minimum with respect to variable a, (d) its maximum with respect to variable a

$$(A,B,C) \;\preceq\; (A',B',C') \Longleftrightarrow (A \subseteq A') \;\wedge\; (B \subseteq B) \tag{15.22}$$

From all the compact variable partitionings that exist for a binary function $f(X)$ we select the partitioning that is the largest with respect to the half-order relation 15.22.

A weakly decomposable function can be implemented by several weak partitionings. For a weakly AND or OR decomposable function $f(A,B)$ the function $g(A)$ can always be selected in such a way that $h(A,B)$ can be implemented more easily than $f(A,B)$, although the number of variables is not decreased.

The question "is an incompletely specified function with the functions $f(X)_{ON}$ and $f(X)_{OFF}$ for A and B decomposable?", can be answered using the following existence theorems.

Theorem 15.2.1 *An incompletely specified function with functions $f(X)_{ON}$ and $f(X)_{OFF}$ for A and B is* OR decomposable, *if and only if*

$$f(X)_{ON} \;\cdot\; \max_A{}^k f(X)_{OFF} \;\cdot\; \max_B{}^k f(X)_{OFF} \;=\; 0 \tag{15.23}$$

The above equation implies that no pattern from the ON-set may be in the projection of the OFF-set in the A direction and the projection of the OFF-set in the B direction.

Dual situations exist in AND decomposable functions.

Theorem 15.2.2 *An incompletely specified function with functions $f(X)_{ON}$ and $f(X)_{OFF}$ for A and B is* AND decomposable, *if and only if*

$$f(X)_{OFF} \;\cdot\; \max_A{}^k f(X)_{ON} \;\cdot\; \max_B{}^k f(X)_{ON} \;=\; 0 \tag{15.24}$$

Example 15.2.4

Using the above theorem is illustrated in Figure 15.5.

The function $find\_A\_B$ determines strong nonlinear (AND or OR) decomposability, if such a decomposability exists. The conditions for the decomposability can be described using care functions $f(X)_{ON}$ = M1 and $f(X)_{OFF}$ = M0, from bundle $FB(X,f)$.

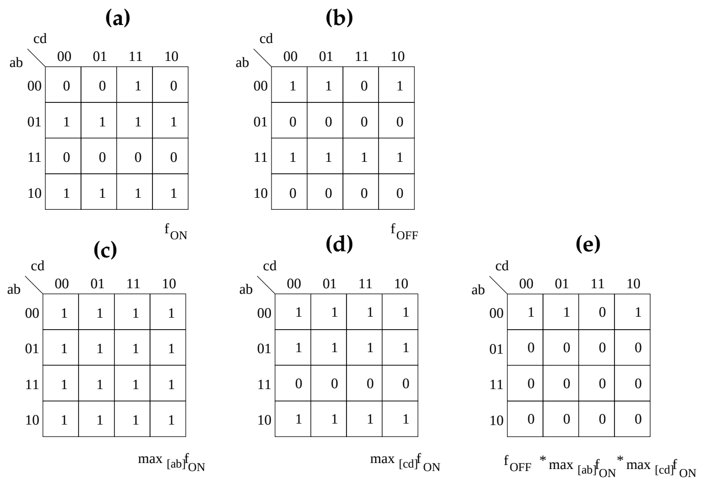


Figure 15.5: Function f to Example 20.4 - graphical calculation of max and intersection

Algorithm 15.2.1

```
Subroutine find_A_B

GIVEN:

Bundle of function $f$ FB(x, f).

Type:  ' d ' for OR
       ' c ' for AND decomposability.

Function find_A_B(FB, type, A, B), returns the
value "true", if FB is decomposable according to the
required strong "type" of decomposability ( 'd' or 'c').
In this case the paritioning sets A and B are returned.

 function find_A_B(FB:bundle;type:char;
                    var A,B:cube_list):boolean;
 var u,v1,v2,v1n,v2n,h1,h,M1,M0 : bdd;
 var a,b,pa,CLI : cube_list;
 var gef : boolean;

 begin
 M1 := get_M1(FB);
 M0 := get_M0(FB);
 CLI := get_CLI(FB);
 BI := get_BI(FB);

 if type = 'd' then begin u := M1; v1 := M0; v2 := M0 end;
             else begin u := M0; v1 := M1; v2 := M1 end;
 gef := false;
```

```
a := sv_first(u); /* loop through variable a */
while (SV_SIZE(a) > 0) and not gef do
  begin              /* Initial solution a = variable, B = set */
  v1n := MAXK(v1,a,CLI,BI);
  h1 := ISC(u,v1n);
  b := SV_NEXT(CLI,a);
  while (SV_SIZE(b) > 0) do (* loop through variable b *)
    begin
    v2n := MAXK(v2,b,CLI,BI);
    h := ISC(h1,v2n);
    if TE(h) then begin gef := true; v2 := v2n end;
    b := SV_NEXT(CLI,b);
    end;
  if gef then B := SV_DIF(CLI,v2) else a := SV_NEXT(CLI,a);
      (* set B created here *)
  end;
if gef then
  begin              /* strong solution  A-set/B- set */
  v1 := v1n;
  pa := SV_DIF(v1,B);
  h1 := ISC(u,v2);
  a := SV_NEXT(pa,a);
  while (SV_SIZE(a) > 0) do
    begin
    v1n := MAXK(v1,a,CLI,BI);
    h := ISC(h1,v1n);
    if TE(h) then v1 := v1n;
    a := SV_NEXT(pa,a);
    end;
  A := SV_DIF(u,v1);    /* set A is created here */
  end;
end;
```

If the decomposability was determined, then the bundles of functions $g(A,C)$ and $h(B,C)$ are calculated. Different possibilities exist for partitioning. For each of the three types of partitioning we would like to find large functions.

The bundles $FB(A,C,\{g\})$ with $g(A,C)_{ON}$ = M1g and $g(A,C)_{OFF}$ = M0g as well as $FB(B,C,\{h\})$ with $h(B,C)_{ON}$ = M1h and $h(B,C)_{OFF}$ = M0h can be calculated for the OR decomposition with the formulas 15.25 to 15.28, if $FB(x,\{f\})$ is AND decomposable with respect to (A, B, C).

$$g(A,C)_{ON} \;=\; \max_B{}^k f(X)_{ON} \cdot \overline{\max_B{}^k f(X)_{OFF}} \tag{15.25}$$

$$g(A,C)_{OFF} \;=\; \max_B{}^k f(X)_{OFF} \tag{15.26}$$

$$h(B,C)_{ON} \;=\; \max_A{}^k [\, f(X)_{ON} \cdot \overline{g(A,C)_{ON}}\,] \tag{15.27}$$

$$h(B,C)_{OFF} \;=\; \max_A{}^k f(X)_{OFF} \tag{15.28}$$

For the AND decomposition one obtains the decomposed functions g, h using formulas 15.29 to 15.32.

$$g(A,C)_{ON} \;=\; \max_{B}{}^{k} f(X)_{ON} \qquad (15.29)$$

$$g(A,C)_{OFF} \;=\; \max_{B}{}^{k} f(X)_{OFF} \;\cdot\; \overline{\max_{B}{}^{k} f(X)_{ON}} \qquad (15.30)$$

$$h(B,C)_{ON} \;=\; \max_{A}{}^{k} f(X)_{ON} \qquad (15.31)$$

$$h(B,C)_{OFF} \;=\; \max_{A}{}^{k}[\; f(X)_{OFF} \;\cdot\; \overline{g(A,C)_{OFF}}\;] \qquad (15.32)$$

The subroutine *nonlinear_decomposition* analyzes a given function bundle, first for the OR decomposition, next for the AND decomposition. It looks for compact partitioning, and if the answer is "yes", it forms bundles using formulas 15.25 to 15.28 or 15.29 to 15.32.

Algorithm 15.2.2

```
Subroutine nonlinear_decomposition

FB(x, f): bundle of the function f.

Function nonlinear_decomposition(FB, M1g, M0g, M1h, M0h, type),
returns value "true", in case a
nonlinear decomposition of the bundle with bdds M1 and
M0 is possible.  In this case the subfunction bundles for
g(A, C) and h(B, C)
for the selected strong decomposability are calculated and
returned together with their type ' d ' or ' c '.
Function g is specified by M1g and M0g.
Function h is specified by M1h and M0h.

 function nonlinear_decomposition
        (FB:bundle;var M1g,M0g,M1h,M0h:bdd;
                       var type:char):boolean;
 var u,v,ug,vg,uh,vh : bdd;
 var A,B : cube_list;
 var gef : boolean;

 begin

 M1 := get_M1(FB);
 M0 := get_M0(FB);
 CLI := get_CLI(FB);
 BI := get_BI(FB);

 type := 'd';
 gef := find_A_B(FB,type,A,B);
       /* strong AND, strong OR  */
 if not gef then
   begin
   type := 'c';
```

```
  gef := find_A_B(FB,type,A,B);
   /* calculates functions g and h
           for 'd' or 'c' decomposition */
  end;
if gef then
  begin
  if type = 'd' then begin u := M1; v := M0 end;
               else begin u := M0; v := M1 end;
  ug := DIF(MAXK(u,B,CLI,BI),MAXK(v,B,CLI,BI),CLI,BI);
  vg := MAXK(v,xb,CLI,BI); (* uses A and B to find g and h *)
  uh := MAXK(DIF(u,ug),A,CLI,BI);
  vh := MAXK(v,A,CLI,BI);
  if type = 'd' then begin M1g := ug; M0g := vg;
                           M1h := uh; M0h := vh end
               else begin M1g := vg; M0g := ug;
                           M1h := vh; M0h := uh end;
  end;
nonlinear_decomposition := gef;
     /* informs by returning a flag */
end;
```

15.2.3 EXOR Decompositions

The EXOR decomposability for sets A and B is determined step by step. First, the derivative of the incompletely specifed function for a single variable x_a is calculated using Lemma 15.2.1.

Lemma 15.2.1 The derivative of an incompletely specified function $f(X)$ with respect to variable x_a *is an incompletely specified function defined as follows:*

$$f_{x_a}(X \setminus x_a)_{ON} := \max_{x_a} f(X)_{ON} \cdot \max_{x_a} f(X)_{OFF} \tag{15.33}$$

$$f_{x_a}(X \setminus x_a)_{OFF} := \min_{x_a} f(X)_{ON} + \min_{x_a} f(X)_{OFF} \tag{15.34}$$

Example 15.2.5

Figure 15.6 illustrates calculation of the formulas from Lemma 15.2.1 using Kmaps.

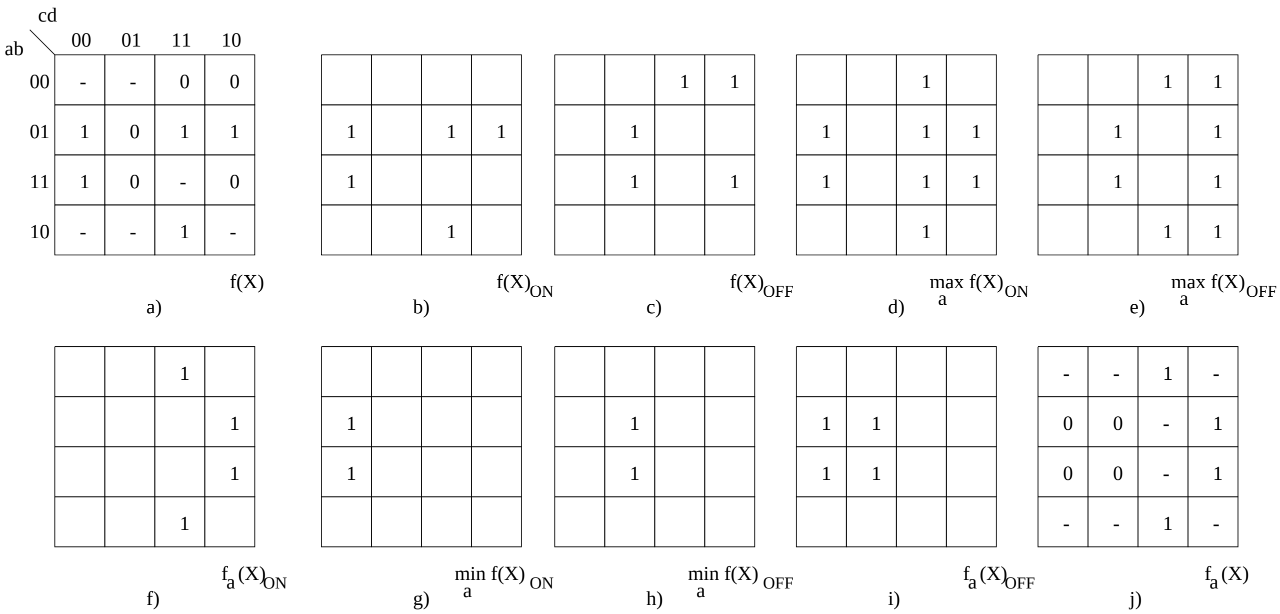


Figure 15.6: Calculation of a derivative $f_a(X)$ of an incompletely specified function $f(X)$ from Example 20.5 using min and max formulas on Kmaps

Figure 15.6f shows $f_a(X)_{ON} = \max_a \ f(X)_{ON} \cdot \max_a \ f(X)_{OFF}$, and Fig. 15.6i illustrates $f_a(X)_{OFF} = \min_a \ f(X)_{ON} + \min_a \ f(X)_{OFF}$. Finally, Fig. 15.6j shows the composite incompletely specified derivative composed from Fig. 15.6f and Fig. 15.6i.

The same map of incomplete derivative from Fig. 15.6j can also be obtained directly, by "folding" the K-map with respect to variable a and calculating new values symetrically to a as follows: $0 \oplus 1 = 1 \oplus 0 = 1$, $0 \oplus 0 = 1 \oplus 1 = 0$, $0 \oplus - = 1 \oplus - = -$.

The ON-set of the derivative with respect to variable x_a in equation 15.33 assumes value 1 for every argument change combination relative to variable x_a. Values 0 are created in the derivative with respect to variable x_a, equation 15.34, when with respect to variable x_a either two 1 minterms or two 0 minterms face each other. With the subroutine *derivative* the care functions $f(X)_{ON} = M1$ and $f(X)_{OFF} = M0$ of the derivative are found. This is calculated by the subroutine *derivative* presented below.

Algorithm 15.2.3

```
Subroutine derivative

GIVEN:

M1: bdd of the ON-set of bundle FB(x, f).

M0: bdd of the complement of the OFF-set of bundle FB(x, f).

A:  { xa }, A is a set containing xa,
     a single variable for which the derivation is executed.

Procedure derivative(M1, M0, xa, M1a, M0a),
which calculates the bdds of the derivative bundle FB(X \ xa, fa).

 procedure derivative(M1,M0:bdd,A:cube_list;var M1a,M0a:bdd);
 begin
 M1a := ISC(MAXK(M1,xa),MAXK(M0,xa));
 M0a := UNI(MINK(M1,xa),MINK(M0,xa));
 end;
```

Definition 15.2.3 Function $f(X)$ is EXOR decomposable *with respect to variable x_a and set of variables B if:*

$$\frac{\partial^2 f(X)}{\partial x_a \partial x_j} = 0 \tag{15.35}$$

for all variables x_j in set B.

Definition 15.2.4 *Function $f(X)$ is linearly groupable with respect to variable sets A and B if:*

$$\frac{\partial^2 f(X)}{\partial x_i \partial x_j} = 0 \tag{15.36}$$

for all variables x_j in set B, and for all variables x_i in set A.

Example 15.2.6

Given is function $f(X) = f(a,b,c,d)$ from Figure 15.7a. Its graphically obtained derivatives with respect to variables a and b are shown in Figure 15.7b and Figure 15.7c, respectively.
It is immediately seen in Figure 15.7b that:

$$\frac{\partial^2 f(X)}{\partial a \partial c} = \frac{\partial^2 f(X)}{\partial a \partial d} = 0$$

It is immediately seen in Figure 15.7c that:

$$\frac{\partial^2 f(X)}{\partial b \partial c} = \frac{\partial^2 f(X)}{\partial b \partial d} = 0$$

Function $f(X)$ is then EXOR decomposable with respect to variable a and set of variables $\{c,d\} = B$. Similarly, function $f(X)$ is EXOR decomposable with respect to variable b and set of variables

ab \ cd	00	01	11	10
00	1	0	1	0
01	0	1	0	1
11	0	1	0	1
10	0	1	0	1

$f(X)$

a)

1	1	1	1
0	0	0	0
0	0	0	0
1	1	1	1

$\frac{d\,f(X)}{d\,a}$

c)

1	1	1	1
1	1	1	1
0	0	0	0
0	0	0	0

$\frac{d\,f(X)}{d\,b}$

c)

Figure 15.7: Checking EXOR decomposability to Example 20.6: (a) function $f(X)$, (b) a derivative with respect to variable a, (c) a derivative with respect to variable b

$\{c, d\} = B$. Therefore $f(X)$ is EXOR decomposable with respect to sets $A = \{a, b\}$ and $B = \{c, d\}$.

When such sets A and B are found, the EXOR decomposability is calculated from the formulas:

$$g(A, C) \;=\; f(A, B = c0, C) \tag{15.37}$$

$$h(B, C) \;=\; \max_A{}^k \; [\; f(X) \;\oplus\; g(A, C) \;] \tag{15.38}$$

Example 15.2.7

Given is function from Fig. 15.8. $B = 00$ was selected (Fig. 15.8a) and respective $g(A, C) = f(a = 0, b = 0, c, d) = c \oplus \overline{d}$ is found, Fig. 15.8b. Next $h(B, C)$ is found (Fig. 15.8e) by EXOR-ing $f(X)$ (Fig. 15.8c) and $g(A, C)$ (Fig. 15.8d). Finally *max* operation with respect to set A is applied to remove vacuous variables (Fig. 15.8f). Finally, $f = (c \oplus \bar{d}) \;\oplus\; \bar{a}b$.

However, below more efficient methods to test EXOR decomposability and to find functions g and h will be given.

Theorem 15.2.3 *(Dresig 1988). An incompletely specified function with functions $f(X)_{ON}$ and $f(X)_{OFF}$ is EXOR decomposable for variable x_a and set of variables B, if and only if:*

$$\max_B{}^k f_a(X)_{ON} \;\cdot\; \max_B{}^k f_a(X)_{OFF} \;=\; 0 \tag{15.39}$$

The above test for (x_{a_1}, B) and (x_{a_2}, B) only confirms the EXOR decomposability for $(\{x_{a_1}, x_{a_2}\}, B)$, if there is at least one specified function within the incompletely specified function which meets the requirements for x_{a_1} as well as for x_{a_2}.

Example 15.2.8

Figure 15.9 illustrates the calculation of EXOR decomposability according to Theorem 15.2.3. Fig. 15.9 shows that $f(X)$ is groupable with respect to a and $\{c, d\}$ and with respect to b and $\{c, d\}$. In this case, function is also groupable with respect to $\{a, b\}$ and $\{c, d\}$.

The likelihood of finding such functions is increased by using Theorem 15.2.4.

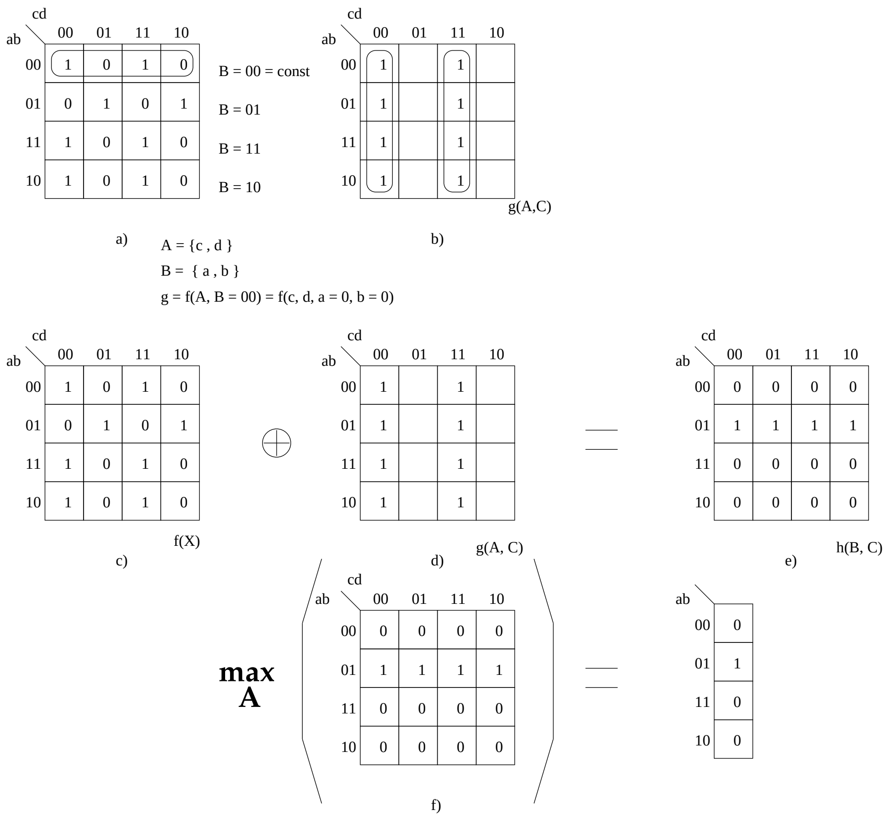


Figure 15.8: Function to Example 20.7

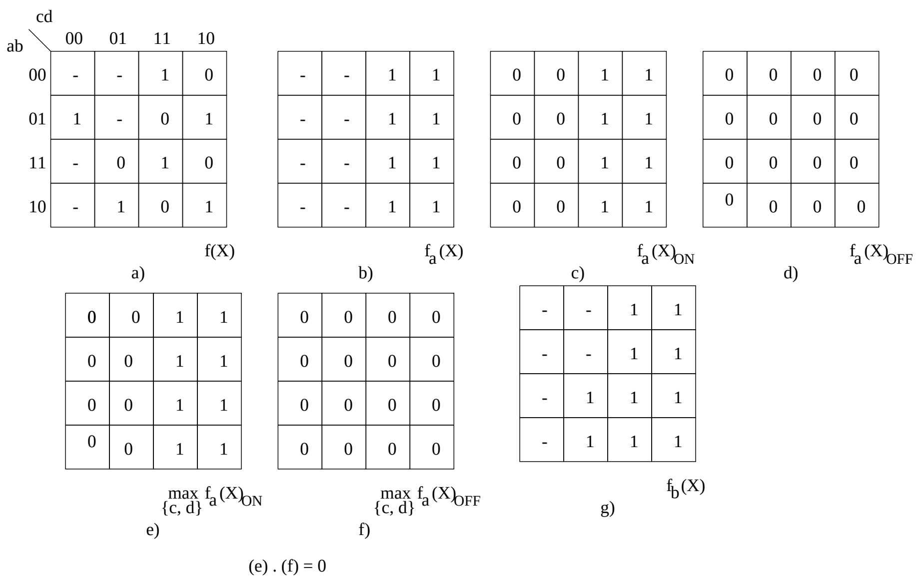


Figure 15.9: Calculation of EXOR decomposability to Example 20.8

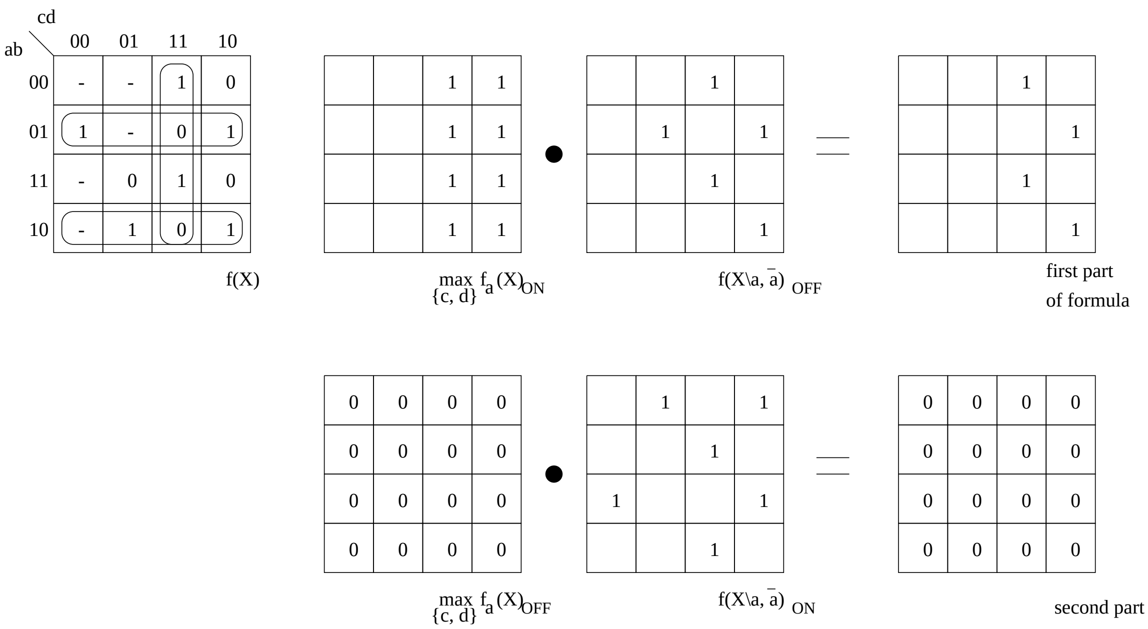


Figure 15.10: Illustration of using Theorem 20.4 in Example 20.9

Theorem 15.2.4 *The reduced incompletely specified function with care functions:*

$$f(X)_{ON} \; := \; f(X)_{ON} \; + \; f(X \setminus x_a, \overline{x_a})_{OFF} \; \cdot \; \max_B{}^k f_a(X)_{ON} \; + \; f(X \setminus x_a, \overline{x_a})_{ON} \; \cdot \; \max_B{}^k f_a(X)_{OFF} \tag{15.40}$$

$$f(X)_{OFF} \; := \; f(X)_{OFF} \; + \; f(X \setminus x_a, \overline{x_a})_{ON} \; \cdot \; \max_B{}^k f_a(X)_{ON} \; + \; f(X \setminus x_a, \overline{x_a})_{OFF} \; \cdot \; \max_B{}^k f_a(X)_{OFF} \tag{15.41}$$

includes all Boolean functions that are EXOR decomposable for (x_a, B). Operation $f(X \setminus x_a, \overline{x_a})_{OFF}$ denotes mirror operation *with respect to variable x_a. It means, in every minterm, literal x_a is changed to $\bar{x_a}$, and literal $\bar{x_a}$ is changed to x_a.*

Example 15.2.9

Figure 15.10 illustrates the usage of Theorem 15.2.4.

The question, whether a binary function is EXOR decomposable with respect to variable sets A and B, cannot be simply answered by using a single equation. It becomes especially complex for strongly unspecified functions and weak decompositions. An iterative method described below must be used in such cases.

One can observe, that if set A is first limited to single variable x_a, then the EXOR decomposability $f(A, B, C)$ as in Figure 15.8 is obtained when the derivative of $f(X)$ with respect to variable x_a is constant for each cofactor of set B. We first analyze if such functions, EXOR decomposable with respect to variable x_a, are contained in the incompletely specified function f. These functions may be again incompletely specified, and our idea is to keep as many of their don't cares as possible.

If in the derivative with respect to variable x_a there exists a constant function in every cofactor of set B, then after the derivative for operation B applied to this function we obtain:

$$f(X)_{ON}(C) = 0$$

Formula 15.39 is used as a condition for the linear groupability (x_a, B, C).

While formula 15.39 helps only to find a reduced groupability, we will use it now to create the solution method for the EXOR decomposability for arbitrary variable sets A and B. Using formula 15.39 the EXOR decomposability of any variable $x_i \in A$ can be checked with respect to set B. Unfortunately, executing this test successively for different $x_i \in A$, will not provide EXOR decomposability for sets B and A. That is because the individual EXOR decomposabilities with respect to $x_i \in A$ and B fix the values for certain originally freely available don't cares which can next lead to conditions mutually contradicting themselves.

If we limit the function groupable according to equation 15.39 to additionally satisfy formulas 15.40 and 15.41 in cofactors, then EXOR decomposable functions are preserved relative to these A and B. This will happen for most, but not for all such functions, so some EXOR decomposable functions with respect to sets A and B may be not found.

The subroutine *check_EXOR_variable_set* shown below reduces a function relative to $x_i \in A$ and B using EXOR decomposability formulas 15.40 and 15.41, to a new function. In this function, if the decomposability exists for $x_j \in A$ and B, then the probability is substantially higher than for the original function that at least one EXOR decomposable function with respect to $(x_i, x_j \in A)$ and B.

Algorithm 15.2.4

```
Subroutine check_EXOR_variable_set.

GIVEN:

M1: bdd of the ON-set of bundle FB(x, f).

M0: bdd of the complement of the OFF-set of bundle FB(x,f)

M1a:  bdd of the ON-set of bundle FB(x, fa).

M0a:  bdd of the complement of the OFF-set of bundle FB(x, fa).

A:  { xa } a set containing a single variable xa.

B: a set of variables.

Procedure check_EXOR_variable_set(M1, M0, M1a, M0a, A, B),
limits bundle FB(x, f) to a cofactor,
for which variable xa is linearly grouped to set B.


 procedure check_EXOR_variable_set
      (var M1,M0:bdd;M1a,M0a: bdd,A,B:cube_list);
 var h0,h1,h2,h3 : bdd;
 begin
 h0 := CEL(M0,A,'10-');
   /* calculate the mirror function with respect to variable xa, */
```

```
  /* it means, in every minterm of M0, literal xa is changed    */
  /* to \xa and literal \xa is changed to xa                    */
 h1 := CEL(M1,A,'10-'); /* do the same transformation for set M1 */
 h2 := MAXK(M0a,B);
 h3 := MAXK(M1a,B);
 M1 := UNI(M1,UNI(ISC(h0,h3),ISC(h1,h2)));
         /* use equation (4.51) */
 M0 := UNI(M0,UNI(ISC(h1,h3),ISC(h0,h2)));
         /* use equation (4.52) */
 end;
```

In the above procedure, M1 was calculated according to equation 15.40 and M0 according to equation 15.41.

Similarly as for the nonlinear decomposing procedures subroutine *linear_grouping* finds a strong EXOR decomposition, if function f is EXOR decomposable. This subroutine relies on the formulas 15.22, 15.33, 15.34, 15.39, 15.40, and 15.41 and uses the subroutines *derivative* and *check_EXOR_variable_set*. The function is constructed to find strong EXOR decomposability as efficiently as possible.

Algorithm 15.2.5

```
Subroutine linear_grouping

GIVEN:

M1: bdd of the ON-set of bundle FB(x, f).

M0: bdd of the complement of the OFF-set of bundle FB(x, f).

Function linear_grouping(M1, M0, A, B),
returns value "true", if there is an
EXOR decomposable function in the bundle with M1 and M0.
In this case, sets A and B for strong partitioning are determined.

 function linear_grouping(var M1,M0:bdd;A,B:cube_list):boolean;
 var u,v,un,vn,M1a,M0a,h : bdd;
 var a,b,pa : cube_list;
 var gef : boolean;

 begin
 gef := false;
 a := sv_first(M1);
 while (SV_SIZE(a) > 0) and not gef do
   begin             /* Initial solution for sets A, B. */
   derivative(M1,M0,a,M1a,M0a);
   u := M1a;
   v := M0a;
   b := SV_NEXT(M1,a);
   while (SV_SIZE(b) > 0) do
     begin
```

```
    un := MAXK(u,b);
    vn := MAXK(v,b);
    h := ISC(un,vn);
    if TE(h) then
                begin
                  gef := true;
                  u := un;
                  v := vn
                end;
    b := SV_NEXT(u,b);
    end;
  if gef then
          begin
            A := a;
            B := SV_DIF(M1a,u);
           end
          else SV_NEXT(M1,a);
  end;
if gef then
  begin   /* compact solution for A and B */
  pa := SV_DIF(M1a,B);
  check_EXOR_variable_set(M1,M0,M1a,M0a,A,B);
          /*                             */
  a := sv_first(pa);
  while (SV_SIZE(a) > 0) do
    begin
    derivative(M1,M0,a,M1a,M0a);
    h := ISC(MAXK(M1a,B),MAXK(M0a,B));
    if TE(h) then
      begin
      check_EXOR_variable_set(M1,M0,M1a,M0a,a,B);
           /*         */
      xa := SV_UNI(A,a);
      end;
    a := SV_NEXT(pa,a);
    end;
  end;
linear_grouping := gef;
end;
```

For EXOR decomposability there are very many different subfunctions h and g. Finding good functions h and g is difficult, since EXOR operation is involved here. During the analysis of the EXOR decomposability we limited already the function in order to obtain the EXOR decomposability for the individual A[i] in connection with B.

However, the reduced function can be, with a very low probability, not EXOR decomposable.

The iterative construction of function $g(A, C)$ is done as follows. First, using equation 15.42 the initial value of function $u(A, C)$ is calculated:

$$u(A, C) \;:=\; \max_B{}^k [f(X)_{ON} \;+\; f(X)_{OFF}] \tag{15.42}$$

Next functions $g(A,C)_{ON}$ and $g(A,C)_{OFF}$ are initialized:

$$g(A,C)_{ON} = 0, \qquad g(A,C)_{OFF} = 0$$

Next, equations 15.43 and 15.44 are applied:

$$g(A,C)_{ON} := g(A,C)_{ON} + u(A,C) \cdot f(B = xb0)_{ON} \tag{15.43}$$

$$g(A,C)_{OFF} := g(A,C)_{OFF} + u(A,C) \cdot f(B = xb0)_{OFF} \tag{15.44}$$

Now, in order to exclude contradictions, the function $u(A,C)$ is shrinked by sharping the above selected cofactor care functions.

$$u(A,C) := u(A,C) \cdot \overline{g(B = xb0)}_{ON} \cdot \overline{g(B = xb0)}_{OFF} \tag{15.45}$$

This whole process of using equations 15.43 to 15.45 is iterated, and in each step 15.45 function $u(A,C)$ is reduced to a smaller ON part.

For all don't cares of $u(A,C)$, which are not completely covered, we must select a care value for function $g(A,C)$. Finally, $u(A,C)$ becomes 0, which is detected by the program.

Thus, function $g(A,C)$ has been calculated.

$$f(X)_{ON} := g(A,C)_{ON} \cdot f(X)_{OFF} + g(A,C)_{OFF} \cdot f(X)_{ON} \tag{15.46}$$

$$f(X)_{OFF} := g(A,C)_{ON} \cdot f(X)_{ON} + g(A,C)_{OFF} \cdot f(X)_{OFF} \tag{15.47}$$

The subfunction $g(A,C)$ is finally found, when in 15.45 $u(A,C) = 0$ results. Otherwise the iterative process is continued again with $xb0$ selected according to equations 15.43 and 15.44.

The EXOR operation of function $g(A,C)$ and $f(X)$ creates incompletely specified function $h(B,C)$. This is done in formulas 15.46 and 15.47. Now, we calculate the (more specified) function $h(B,C)$ on the basis of the incompletely specified function obtained in equations 15.46 and 15.47.

The same iterative method is used to create $h(B,C)$ as was used to find function $g(A,C)$. With the starting functions $h(B,C)_{ON} = 0$ and $h(B,C)_{OFF} = 0$, after unique calculation of equation 15.48 the functions $h(B,C)_{ON}$, $h(B,C)_{OFF}$ and $u(B,C)$ are iteratively calculated, and different values $xa0$ are tried, until $u(B,C)$, calculated in step 15.51), results in 0.

$$u(B,C) := \max_{A}^{k}[f(X)_{ON} + f(X)_{OFF}] \tag{15.48}$$

$$h(B,C)_{ON} := h(B,C)_{ON} + u(B,C) \cdot f(A = xa0)_{ON} \tag{15.49}$$

$$h(B,C)_{OFF} := h(B,C)_{OFF} + u(B,C) \cdot f(A = xa0)_{OFF} \tag{15.50}$$

$$u(B,C) := u(B,C) \cdot \overline{h(A = xa0)}_{ON} \cdot \overline{h(A = xa0)}_{OFF} \tag{15.51}$$

The subroutine *strong_linear* analyzes a given function $f(X)$ with respect to strong EXOR decomposability. It forms a pair of subfunctions (g,h) using equations 15.42 to 15.51, if $f(X)$ is an EXOR decomposable function. The two functions $g(A,C)$ and $h(B,C)$ are generated the same way. A verification is recommended of functions g and h calculated by *strong_linear*, since it is possible that two paritionings (x_i, B) and (x_j, B) are not compatible with one another. This can only happen with an extraordinarily small probability, but cannot be excluded.

Algorithm 15.2.6

```
subroutine strong_linear

M1: bdd of the ON-set of bundle FB(x, f).

M0: bdd of the complement of the OFF-set of bundle FB(x, f).

Function EXOR_compact(M1, M0, M1g, M0g, M1h, M0h) returns value "true",
if a EXOR paritioning of the bundle with M1 and M0 is
possible.  In this case the subfunctions for g(A, C) and h(B, C)
of a strong EXOR decomposition are returned.

 function strong_linear(M1,M0:bdd;
      var M1g,M0g,M1h,M0h:bdd):boolean;
 var M0n,M1n : bdd;
 var A,B : cube_list;

 procedure iteration
     (M1w,M0w,grv:bdd;var M1r,M0r:bdd);
   /* the comments inside this procedure are for function g.  */
   /*  Analogically, procedure interation is next used        */
   /*  for function h                                         */
 var grv0,u,u1,h1,h0 : bdd;
 begin
 u1 := UNI(M1w,M0w);  /* Use (4.61) to calculate      */
 u  := MAXK(u1,grv);  /* the value of  u(A,C)         */
 M1r := EMPTY(u);     /* Initialize g_ON to zero.     */
 M0r := EMPTY(u);     /* Intitalize g_OFF to zero.    */
 while not TE(u) do   /* until u(A,C) becomes zero... */
   begin
   grv0 := MAXK(CEL(STV(u1,1),grv,'010'),u);
   h1 := MAXK(ISC(M1w,grv0),grv); /* calculate h1 in (4.62) */
   h0 := MAXK(ISC(M0w,grv0),grv); /* calculate h0 in (4.63) */
   M1r := UNI(M1r,ISC(u,h1));
    /* use equ. (4.62) to calculate new value of g(A,C)_ON */
   M0r := UNI(M0r,ISC(u,h0));
    /* use equ. (4.63) to calculate new value of g(A,C)_OFF     */
   u := DIF(DIF(u,h1),h0); /* New value of u(A,C) is calculated */
   u1 := ISC(u,u1);        /* from equation (4.64)              */
   end;
 end;

 begin
 if linear_grouping(M1,M0,A,B)
  then
    begin
    iteration(M1,M0,B,M1g,M0g);
      /* iteration uses rules 4.62 to 4.64 to calculate function g  */
      /*   for set B                                                */
    M1n := UNI(ISC(M1g,M0),ISC(M0g,M1));
```

```
         /* equation  (4.65)  */
   M0n := UNI(ISC(M1g,M1),ISC(M0g,M0));
         /* equation  (4.66)  */
         /* these were EXORing of f(X) with the function h found */
   iteration(M1n,M0n,A,M1h,M0h);
     /* iteration uses rules 4.67 to 4.70 to calculate function h  */
     /*   for set A                                                */
   strong_linear := true;
     /* informs that strong linear decomposition was found         */
  end
 else
   strong_linear := false;
end;
```

15.3 Weak AND/OR/EXOR Decompositions.

A strongly decomposable function is decomposed to two simpler subfunctions. Otherwise (e.g. in case of the symmetrical functions) we must reiterate to other synthesis procedures. One of such procedures, called *weak decomposition*, is based on *weak variable paritioning*, and is presented in Fig.15.11. Figure 15.11a presents the *Weak AND Decomposition*, and Figure 15.11b presents the *Weak OR Decomposition.*

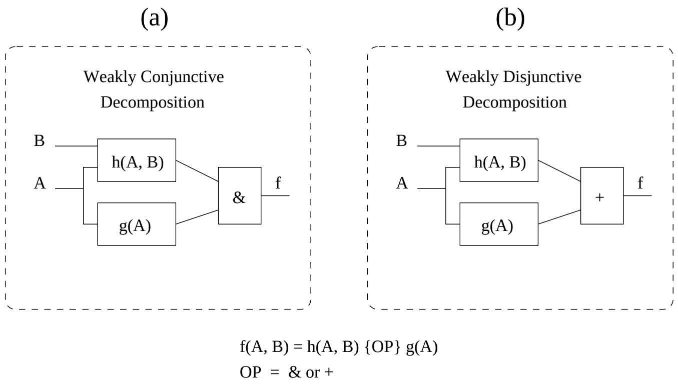


Figure 15.11: Weak AND and OR Decompositions

The weak variable partitioning can be made for AND, OR or EXOR decompositions. A binary function is nonlinearly weakly groupable, if the respective criteria have been satisfied.

Every function has EXOR (linear) decomposability, and can be decomposed respectively. Using the weak EXOR partionings we transform function $f(A,B)$ to an EXOR of function $g(A)$, selected by us, and function $h(A,B)$ that is found by EXORing $f(A,B)$ and $g(A)$.

Although every function is weak EXOR decomposable, this should be our last resort in the decomposition strategies. The weak AND or OR decompositions always simplify the function. Compared with $f(A,B)$, function $g(A)$ has fewer variables. Function $h(A,B)$ has more don't cares, so it is easier to realize than $f(A,B)$ and it can become perhaps strongly decomposable. If not, a weak decomposability is repeated.

In a function bundle $FB(X,\{f\})$ with the above care functions, at least one weak paritioning relative to set B exists, AND or OR, if the following Theorem 15.3.1 or Theorem 15.3.2 are satisfied, respectively.

Theorem 15.3.1 *An incompletely specified function with care functions $f(X)_{ON}$ and $f(X)_{OFF}$ is weak OR decomposable with set B if and only if:*

$$f(X)_{ON} \cdot \overline{\max_B{}^k f(X)_{OFF}} \neq 0 \tag{15.52}$$

Example 15.3.1

A completely specified function $f(A,B)$ is presented in Fig. 15.12a. Let us assume that set A corresponds to rows and set B to columns. It is easy to find function $g(A)$ for OR decomposition from the patterns of rows. If there are rows that have no 0's, and only 1's and -'s, then such rows can be used to determine function $g(A)$, see Fig. 15.12b. The condition in Theorem 15.3.1 means that there are rows of only 1's and -'s. Let us observe, that by replacing true minterms from function $g(A)$ in function $f(A,B)$ with "don't care" minterms, one obtains function $h(A,B)$, Fig. 15.12c, that is, in general, a function of at least variables from B and at most variables from $A \cup B$. If it is a function of only variables from B the OR decomposition is strong, otherwise it is weak.

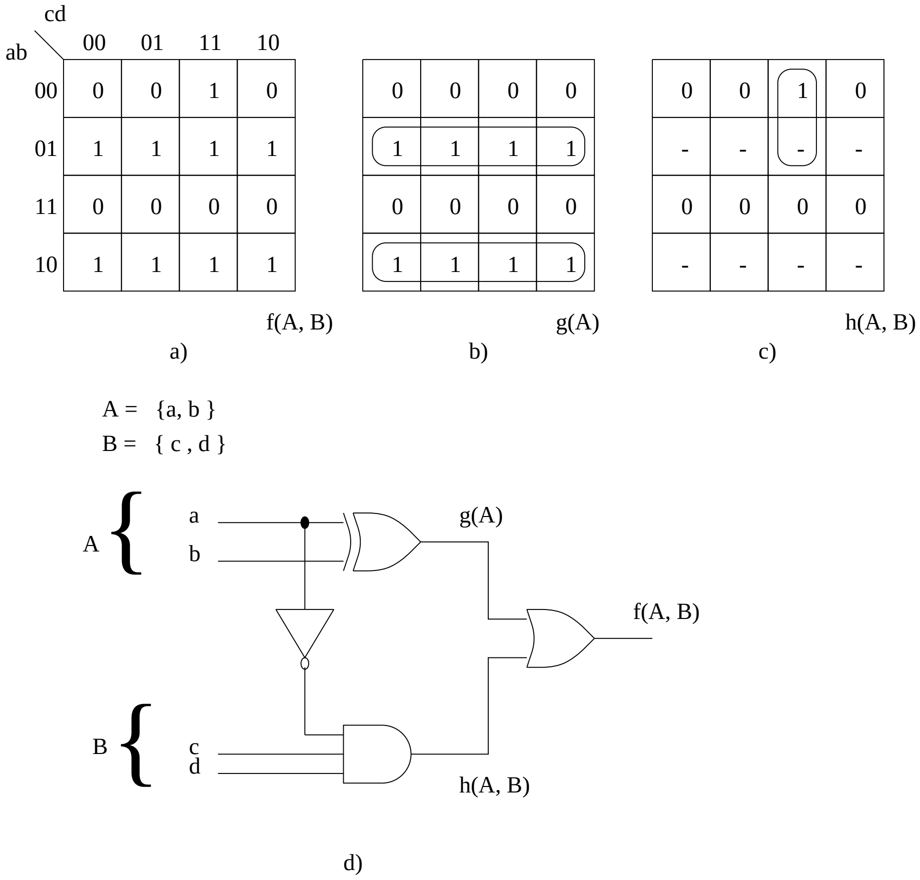


Figure 15.12: Steps to OR decomposition to Example 21.1

The final OR decomposition of function from Fig. 15.12a is shown in Fig. 15.12d.

Concluding, finding OR decompositions in Karnaugh maps is completely algorithmic and easy to find by hand, and patterns of only 1's and -'s in rows are used.

Example 15.3.2

Given is function $f(A, B)$ with $A = \{a, b\}$ and $B = \{c, d\}$ from Fig. 15.13. The enumeration of minterms for partitions is shown in Fig. 15.13.

Using partitions, the above presented steps from Example 15.3.1 are executed as follows.

$$\Pi_f = \{\{0, 1, 3, 8, 9, 10, 11\};\ \{2, 4, 5, 6, 7, 12, 13, 14, 15\}\}$$

$$\Pi(A) = \{\{0, 1, 2, 3\};\ \{4, 5, 6, 7\};\ \{12, 13, 14, 15\};\ \{8, 9, 10, 11\}\};$$

The order of blocks in the above partition is the same as the natural order of cofactors, $\bar{a}\bar{b}, \bar{a}b, a\bar{b}, ab$.

$$\Pi(A) \cdot \Pi_f = \{\{(0, 1, 3)(2)\};\ \{(\emptyset)(4, 5, 6, 7)\};\ \{(\emptyset)(12, 13, 14, 15)\};\ \{(8, 9, 10, 11)(\emptyset)\}\}$$

We assume in the above formula that in each cofactor the OFF-part is first and the ON-part is next. For instance, in the second block, the OFF-part is empty.

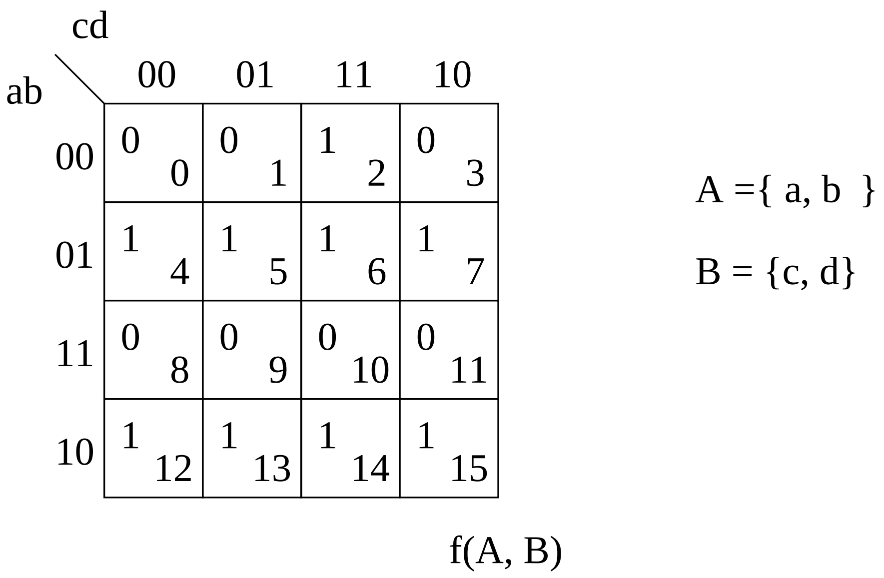


Figure 15.13: Function to Example 21.2 with enumeration of minterms

$$g(A) = \overline{a}\ b + a\ \overline{b}$$

$h(A, B) = \{\{0, 1, 3, 8, 9, 10, 11\};\ \{2, 4, 5, 6, 7, 12, 13, 14, 15\} - \{4, 5, 6, 7\} - \{12, 13, 14, 15\}\} = \{\{0, 1, 3, 8, 9, 10, 11\};\ \{2\}\}$
Above, $\{2\}$ is the ON-part of $h(A, B)$.
A weak AND decomposition is a dual to the weak OR decomposition.

Theorem 15.3.2 *An incompletely specified function with functions $f(X)_{ON}$ and $f(X)_{OFF}$ is* weakly AND decomposable *with set B, if and only if:*

$$f(X)_{OFF} \cdot \overline{\max_B{}^k\ f(X)_{ON}} \neq 0 \qquad (15.53)$$

Example 15.3.3

Let us assume that set A corresponds to rows and set B to columns. It is easy to find function $g(A)$ for AND decomposition from the patterns of rows. If there are rows that have no 1's, and only 0's and -'s, then such rows can be used to determine function $g(A)$, see Fig. 15.14. Condition in Theorem 16.2 means that there are rows of only 0's and -'s. Let us observe, that by replacing false minterms (zeros) of function $g(A)$ in function $f(A, B)$ by "don't cares", one obtains function $h(A, B)$ that is a function of at least variables from B and at most variables from $A \cup B$, Fig. 15.14. If it is a function of only variables from B the AND decomposition is strong, otherwise it is weak.

The final AND decomposition of function from Fig. 15.14a is shown in Fig. 15.14d.

Concluding, finding AND decompositions in Karnaugh maps is completely algorithmic and easy to find by hand, and is based on patterns of 0's and -'s in rows.

If only those properties of incompletely specified functions are used which have been mentioned above, a convergent decomposition method can be developed. This is due to the following theorem.

Theorem 15.3.3 *Every Boolean function that is not weak OR decomposable and not weak AND groupable has the property of EXOR decomposability.*

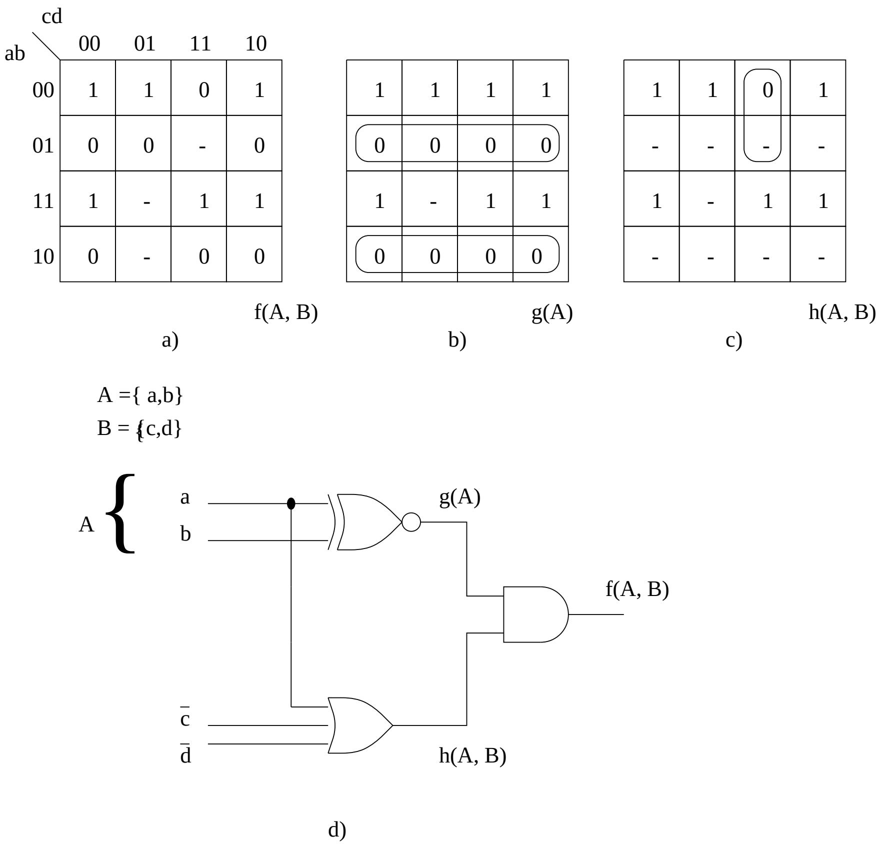


Figure 15.14: Steps of AND decomposition: (a) function $f(A,B)$, $A = \{a,b\}$, $B = \{c,d\}$, (b) calculation of function $g(A)$, (c) calculation of function $h(A,B)$, (d) final circuit for AND decomposition

If the variable set B cannot be further increased in Equation 15.52 or 15.53, without these inequalities changing to equalities, then weak decomposability exists. For each weak decomposability there are several weak partitionings. In the formulas 15.54 to 15.61 the indicated weak partitionings were selected in such a way that as simple as possible function $h(A, B)$ is created.

Under the prerequisite that equation 15.52 is satisfied, the *Weak AND Decomposition*, see Fig. 15.11b, can be made into bundles $BF(A, \{g\})$ with $g(A)_{ON} =$ M1g and $g(A)_{OFF} =$ M0g as well as $BF(A, B, \{h\})$ with $h(A, B)_{ON} =$ M1h and $h(A, B)_{OFF} =$ M0h after the formulas 15.54 to 15.57.

$$g(A)_{ON} = \max_B{}^k f(A,B)_{ON} \cdot \overline{\max_B{}^k f(A,B)_{OFF}} \tag{15.54}$$

$$g(A)_{OFF} = \max_B{}^k f(A,B)_{OFF} \tag{15.55}$$

$$h(A,B)_{ON} = f(A,B)_{ON} \cdot \overline{g(A)_{ON}} \tag{15.56}$$

$$h(A,B)_{OFF} = f(A,B)_{OFF} \tag{15.57}$$

If the inequality 15.53 is satisfied, then a *Weak AND Decomposition*, Fig. 15.11b, is obtained from formulas 15.58 to 15.61.

$$g(A)_{ON} = \max_B{}^k f(A,B)_{ON} \tag{15.58}$$

$$g(A)_{OFF} = \max_B{}^k f(A,B)_{OFF} \cdot \overline{\max_B{}^k f(A,B)_{ON}} \tag{15.59}$$

$$h(A,B)_{ON} = f(A,B)_{ON} \tag{15.60}$$

$$h(A,B)_{OFF} = f(A,B)_{OFF} \cdot \overline{g(A)_{OFF}} \tag{15.61}$$

The subroutine *nonlinear_weak* analyzes BDDs of a bundle for weak OR and AND decomposability, and forms (if possible) the weak subfunctions using 15.54 to 15.57, or 15.58 to 15.61, respectively.

Application of these formulas to calculate OR and AND decompositions is shown in the next examples.

Example 15.3.4

Figure 15.15 illustrates graphically using of the above formulas in a weak OR decomposition.

Example 15.3.5

Figure 15.16 illustrates graphically using of the above formulas in a weak AND decomposition.

Example 15.3.6

Figure 15.17 illustrates graphically the weak EXOR decomposition. Formulas 15.52 and 15.53 are not satisfied. Let us observe that any row or set of rows of the Kmap can be selected as function $g(A)$. It is good to select indentical rows. Row $a\ b$ has been selected in Fig. 15.17a, and EXOR-ed from $F(A, B)$ as $g(A) = a\ b$, thus creating $h(A, B)$ from Fig. 15.17b. The final circuit is shown in Fig. 15.17c.

Algorithm 15.3.1

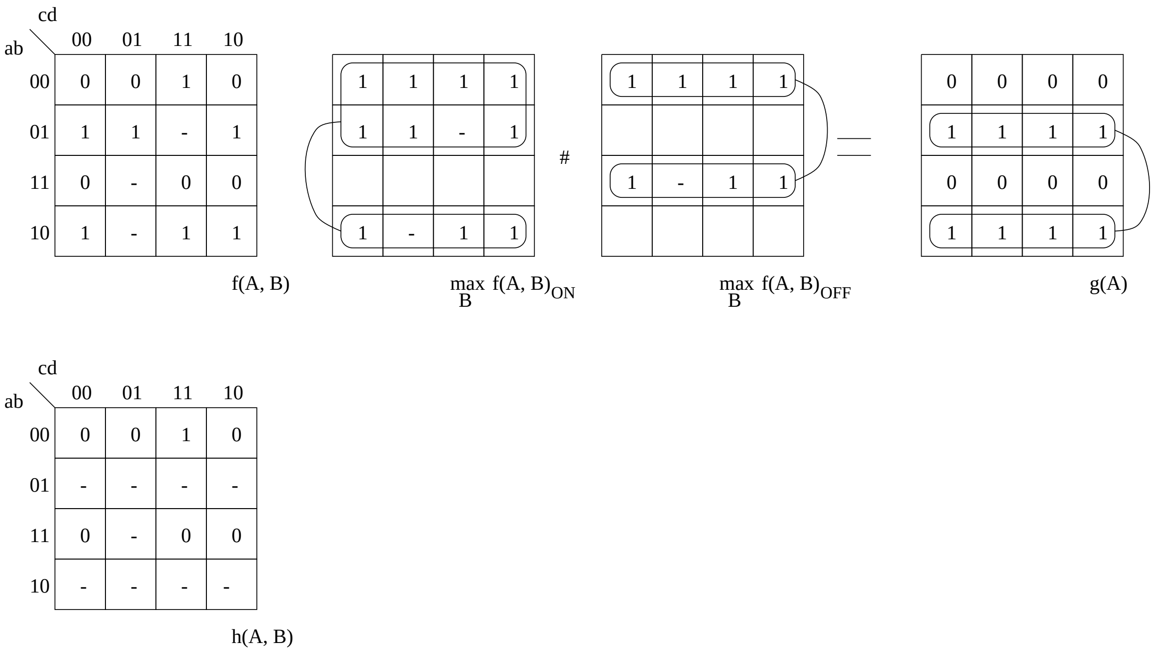


Figure 15.15: Graphical illustration of using the formulas in weak OR decompositions to Example 21.5

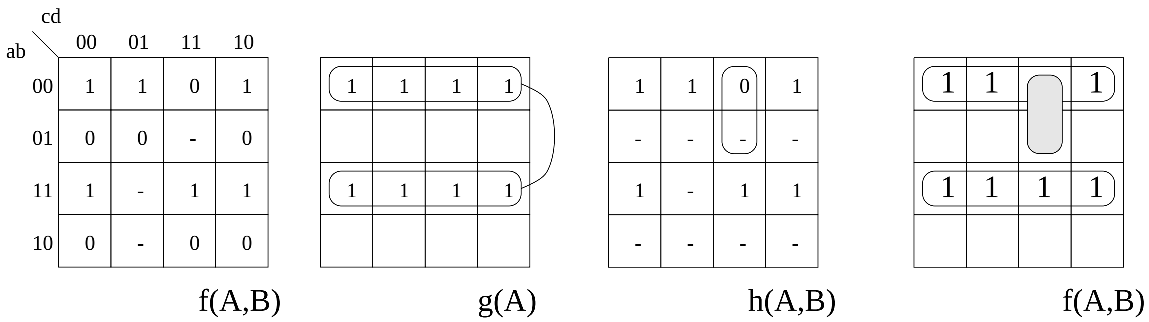


Figure 15.16: Graphical illustration of using the formulas in weak AND decompositions to Example 21.6

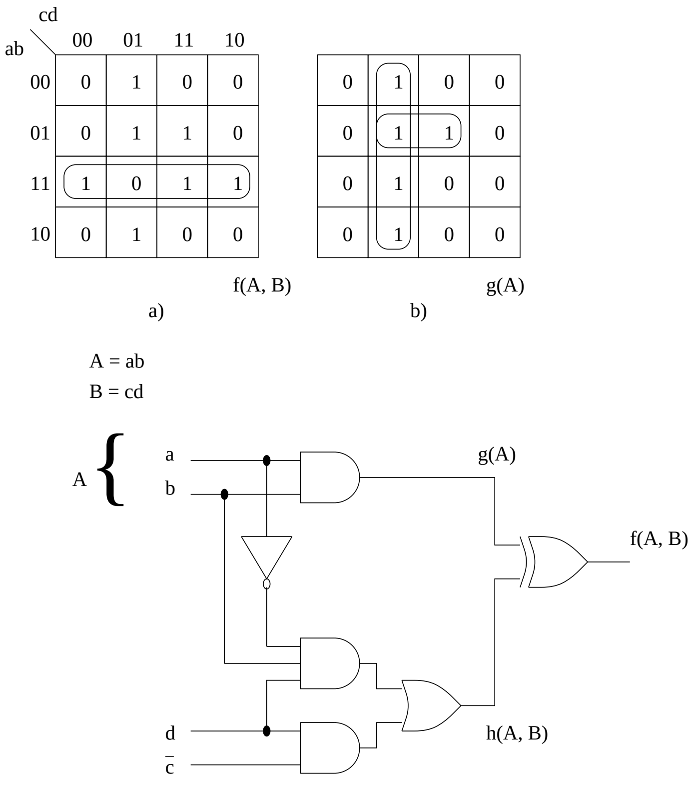


Figure 15.17: Graphical illustration of using the formulas in weak EXOR decomposition

Subroutine nonlinear_weak

GIVEN:

M1: bdd of the ON-set of function f from bundle BF(x, f).

M0: bdd of the complement of the OFF-set of function f from bundle BF(x, f).

Function nonlinear_weak(M1, M0, M1g, M0g, M1h, M0h, type)
returns value "true", if a weak AND or OR partitioning
of a bundle with M1 and M0
is possible. In this case the weak subfunction
bundles for g(A) and h(A, B) are calculated
for the selected compact
partitioning and returned together with their
types, ' d ' or ' c ', respectively.
One decomposition of type ' d ' or ' c ' is returned,
or type ' n ' is returned.

```
 function nonlinear_weak
   (M1,M0:bdd;var M1g,M0g,M1h,M0h:bdd;
                     var type:char):boolean;
 var u,v,ug,vg,uh,vh,wt,wtn,b,B : bdd;
                            b,B : cube_list;
    gef : boolean;
 begin
 gef := false;
 type := 'd';
 B := SV_DIF(M1,M1);
 repeat  /* repeat until 'c' or 'd' decomposition found,   */
         /*  or no more sets B exist                        */
   if type = 'd' then begin u := M1; v := M0 end
                else begin u := M0; v := M1 end;
   wt := v;
   b := sv_first(u);
   while SV_SIZE(b) > 0 do
     begin
     wtn := MAXK(wt,b);
     if not TE(DIF(u,wtn)) then begin
                              wt := wtn;
                              B := SV_UNI(B,b);
                              end;
     b := SV_NEXT(u,b);
     end;              /* end of creating set B    */
   if SV_SIZE(B) > 0 then
      begin
      ug := DIF(MAXK(u,B),MAXK(v,B)); /* use eq. (3) */
      vg := MAXK(v,B);                /* use eq. (4) */
      uh := DIF(u,ug);                /* use eq. (5) */
```

```
      vh := v;                        /* use eq. (6) */
      gef := true;
      if type = 'd' then begin
                          M1g := ug; M0g := vg;
                          M1h := uh; M0h := vh
                       end
                 else begin
                          M1g := vg; M0g := ug;
                          M1h := vh; M0h := uh
                       end;
      end
    else if type = 'd' then type := 'c' else type := 'n';
 until gef or (type = 'n');
 nonlinear_weak := gef;
 end;
```

The algorithm of the decomposition is the following.

Algorithm 15.3.2

```
FUNCTIONS:

grp_test: tests the decomposability

calc_g_ON, calc_g_OFF: calculate care functions of function g.
calc_h_ON, calc_h_OFF: calculate care functions of function h.
calc_f: calculate function f from function g(A,C) and h(B,C).

end_test: test the possibility of using a function that depends on
only one variable.

choice_fct: select a single function from an incompletely
specified function.

write_gate: write the designed gate into the netlist.

decompose(ON,OFF)

{

< grp_op, A,B > = grp_test(ON,OFF);
 /* grouping operator and sets A, B are returned   */
 /* as the result of the test for groupability.    */


g_ON = calc_g_ON(f_ON,f_OFF,A,B,grp_op);
g_OFF = calc_g_OFF(f_ON,f_OFF,A,B,grp_op);
  /* respective incomplete function g(A,B) is      */
  /* calculated for A, B, and operator.            */
```

```
if ( end_test(g_ON,g_OFF) )
 fct_g = choice_fct(g_ON,g_OFF);
   /* if this is a one-variable function, realize it. */
else
 fct_g = decompose(g_ON,g_OFF);
   /* else keep recursicely decomposing.              */

h_ON = calc_h_ON(f_ON,f_OFF,A,B,grp_op,fct_g);
h_OFF = calc_h_OFF(f_ON,f_OFF,A,B,grp_op,fct_g);
 /* respective incomplete function h(A,B) is           */
 /* calculated for A, B, operator and function g.      */
 /* By, now, recursion to create g is completed, and   */
 /* g is completely specified.                         */

if ( end_test(h_ON,h_OFF) )
 fct_h = choice_fct(h_ON,h_OFF);
else
 fct_h = decompose(h_ON,h_OFF);
  /*    now the same method was used to create         */
  /*    a completely specified function h              */
fct = calc_f(fct_g,grp_op,fct_h);
  /* calculate completely specified function f         */
  /* from completely specified functions g and h.      */

write_gate(grp_op);
  /*   print operator of this level.                   */
  /*   This is done recursively.                       */

return(fct);

}
```

Interesting method is also used in XBOOLE to find sets A and B. It is based on the following Monotony Theorem:

Theorem 15.3.4 *If an incompletely specified function has no OR decomposition with respect to sets A and B then there is no OR decomposition of this function for $\{A, x_i\}$ and B. There is also no OR decomposition for A and $\{B, x_i\}$.*

This is used in the above algorithm.

Algorithm 15.3.3

```
 Calculate initial partitioning A = {xi}, B = {xj};

 For all variables xi in X \ {xi,xj} do
  begin
  if
    card(A) < card(B)
```

```
then
  if GR((A,xi),B) then A := A + xi;
  else
    if GR(A,(B,xi)) then B := B + xi;
else
  if GR(A,(B,xi)) then B := B + xi;
  else
    if GR(A,xi),B) then A := A + xi;
end;
```

Chapter 16

PUB Decompositions

16.1 PUB Decompositions

16.1.1 Disjoint PUB Decomposition

The method shown here was first introduced by Perkowski, Uong and Brown in 1987, [?, ?]. It can use arbitrary representation of switching functions. Although this method is similar to Curtis Decomposition, it differs from it. Curtis Decomposition encodes outputs of the bound set. PUB Decomposition uses free set variables for the address inputs of a conceptual multiplexer, and, in contrast to Curtis, it does not require the encoding process. (See comparison of PUB and Curtis belsow). Thus, PUB creates the intermediate subfunction H with a higher value of DFC, but possibly, this subfunction can be built from simpler gates. It must be then further investigated, which of the Curtis and PUB decompositions creates a less expensive realizations from gates of a small fan-in in multi-level decomposition. Below, PUB type of decomposition will be explained using cube calculus, but, like any other decomposition, it can be executed in arbitrary function representation.

A Boolean function of n *input variables,* $X_1, X_2, \ldots, X_n$ will be given in a form of *arrays of cubes.* Each *cube* will have n positions. Each position of a cube can be a value of the respective variable:

- 0 - in negative form,
- 1 - in positive form,
- X - don't care form.

The function will be specified as the *ON-array* and *OFF-array.* ON-array is an array of *ON-cubes.* OFF-array is an array of *OFF-cubes.* ON-cube is a cube with value 1. OFF-cube is a cube with value 0. By ON(00XXX) we will denote the set of ON-cubes of $f(0, 0, c, d, e)$. Similarly by OFF(XX010) we will denote the set of OFF-cubes of $f(a, b, 0, 1, 0)$.

Theorem 16.1.1 *Two functions f_1 and f_2 are* compatible *if and only if* $[ON(f_1) \cap OFF(f_2) = \emptyset]$ *and* $[ON(f_2) \cap OFF(f_1) = \emptyset]$

Definition 16.1.1 *The loop pair incompatibility graph is an undirected graph*

$$G = (N, RS)$$

defined as follows:

- N *- is the set of loops,*
- RS *- is the set of edges,*
- $(c_1,\ c_2) \in RS \Longleftrightarrow$ *loop functions* $f(c_1)$ *and* $f(c_2)$ *are not compatible (such functions are called incompatible).*

The nodes of the graph are loops (cofactors). An edge is created in the incompatibility graph between two nodes when the corresponding loop functions of these nodes are not compatible.

Let us observe that the Loop-Pair-Incompatibility-Graph is a complement of the Loop-Pair-Compatibility-Graph, i.e. has an edge betwen two nodes if there is no edge in the other graph, and vice versa.

The optimal algorithm for the clique partitioning of the Loop-Pair-Compatibility-Graph is based on minimal coloring of the Loop-Pair-Incompatibility-Graph. The coloring is an assignment of colors to graph nodes in which any nodes linked by an edge receive different colors. Minimal coloring is a coloring with the minimum number of colors. If the number of colors is two then the classical decomposition is possible: Function F_1 being the merging of loop functions from all nodes of color 1 is found. Next function F_2 being the merging of loop functions from nodes of color 2. If F_1 and $\overline{F_2}$ are compatible then the Ashenhurst decomposition exists. Otherwise, even when the number of colors exceeds two - the minimal coloring result can still be used for the efficient realization with multiplexers, or general PUB Decompositions.

Example 16.1.1

Let us assume a function $f(a, b, c, d, e)$ specified as follows:

ON(f) = {10000, 110X1, 00X11, 01101, 10110, 11100}
OFF(f) = {110X0, X1010, 0XX10, 101X1, 1010X, X0101, 00000, 01001}

ab \ cde	000	001	011	010	110	111	101	100
00	0	-	1	0	0	1	0	-
01	-	0	-	0	0	-	1	-
11	0	1	1	0	-	-	-	1
10	1	-	-	-	1	0	0	0

Figure 16.1: A Karnaugh map for Example 9.1

The function is presented in a Karnaugh map from Fig. 16.1.

We illustrate an application of the optimal graph-coloring algorithm for free set $A = \{c, d, e\}$ corresponding to columns.

1. Find cofactors in the form of arrays of cubes.

   ON(XX000) = {10XXX} OFF(XX000) = {00XXX, 11XXX}
   ON(XX001) = {11XXX} OFF(XX001) = {01XXX}
   ON(XX010) = {} OFF(XX010) = {0XXX, X1XXX, 11XXX}
   ON(XX011) = {00XXX, 11XXX} OFF(XX011) = {}
   ON(XX100) = {11XXX} OFF(XX100) = {10XXX}
   ON(XX101) = {01XXX} OFF(XX101) = {x0XXX}
   ON(XX110) = {10XXX} OFF(XX110) = {0XXXX}
   ON(XX111) = {00XXX} OFF(XX111) = {10XXX}

2. Discard cofactors of only zeros and cofactors of only ones.

   After removing the loop XX011 of ones and the loop XX010 of zeros (such loops can be easily found, since they have empty sets OFF(XX011) and ON(XX010), respectively) we obtain the following set of loops:

   {XX000, XX001, XX100, XX101, XX110, XX111}

   These loops become the nodes of the incompatibility graph.

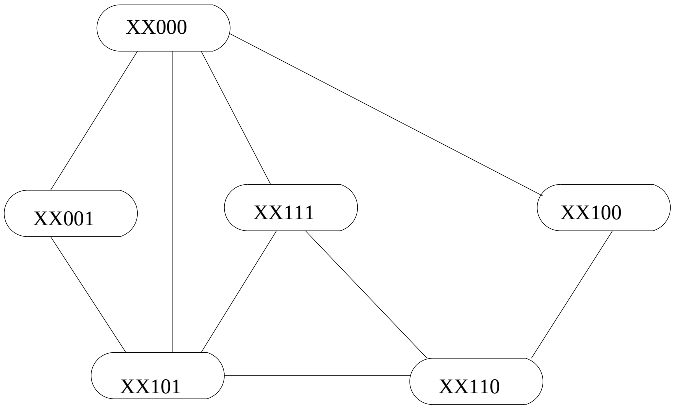


Figure 16.2: The loop pair incompatability graph for Example 9.1

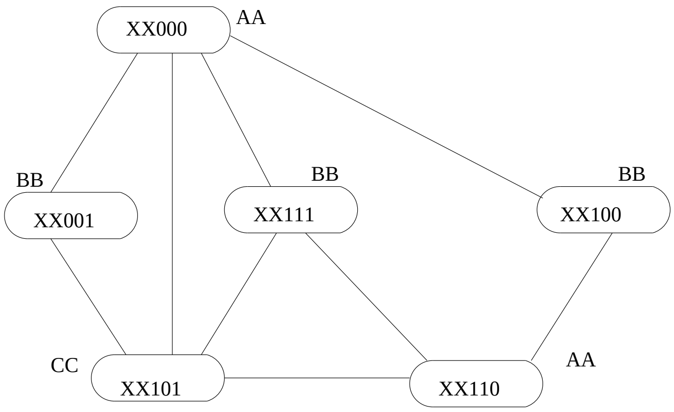


Figure 16.3: The graph from Fig. 26 after coloring

3. Create the graph of loop pair incompatibility.

   For the pair of loops XX000, XX001 we have:

   ON(XX000) $\cap$ OFF(XX001) = {10XXX} $\cap$ {01XXX} = $\emptyset$
   OFF(XX000) $\cap$ ON(XX001) = {00XXX, 11XXX} $\cap$ {11XXX} = {11XXX} $\neq \emptyset$

   then these loops are incompatible and an edge is added to the graph. Similarly, using this method the entire loop pair incompatibility graph from Fig. 16.2 is created.

4. Find minimal coloring of the loop pair incompatibility graph.

   After coloring with a minimum number of colors the graph from Fig. 16.2 becomes as in Fig. 16.3.

   The minimal coloring has 3 colors: AA, BB, and CC. The Ashenhurst decomposition thus does not exist but this result is useful to minimize the design with the multiplexer. We have then to realize only three different functions: AA, BB, and CC to feed the output successor (multiplexer) for B = $\{c,\ d,\ e\}$. The set of columns colored with color AA is = {XX000, XX110}. The set of columns with color BB is {XX001, XX111, XX100} and the set for color CC is {XX101}.

   The next stage is to merge the compatible loop pairs.

   For color AA:

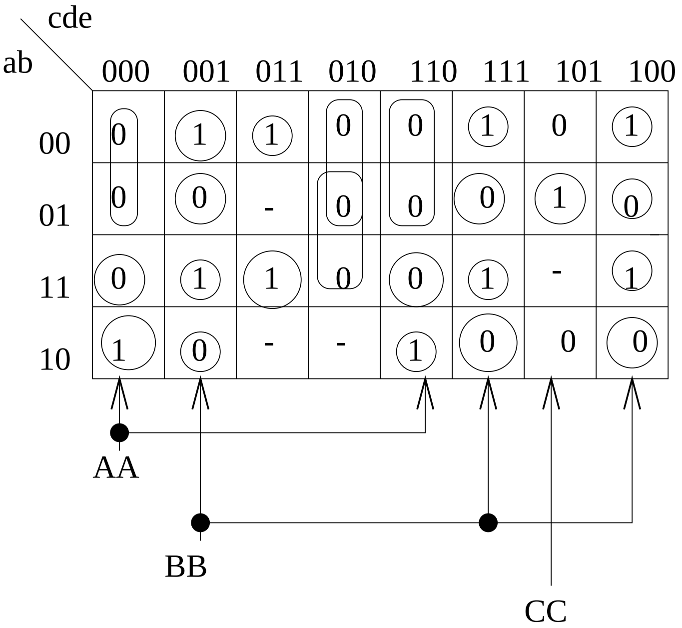


Figure 16.4: The function after the completion of loop functions, the columns colored with the same colors are shown (the column multiplicity of 3)

ON(XX000) ∪ ON(XX110) = {10XXX} ∪ {10XXX} = {10XXX}
OFF(XX000) ∪ OFF(XX110) = {00XXX, 11XXX} ∪ {0XXXX} =
{0XXXX, 11XXX} (since 00XXX $\sqsubset$ 0XXXX)

Let us observe that this is not a set of prime implicants for this column.

For color BB:

ON(XX001) ∪ ON(XX111) ∪ ON(XX100) =
{11XXX} ∪ {00XXX} ∪ {11XXX} = {11XXX, 00XXX}
OFF(XX001) ∪ OFF(XX111) ∪ OFF(XX100) =
{01XXX} ∪ {10XXX} ∪ {10XXX} = {01XXX, 10XXX}

For color CC:

ON(XX101) = {01XXX}
OFF(XX101) = {X0XXX}

5. Complete the function f according to the optimal coloring found in the previous step: the functions being the mergings of all loop functions of the same color replace the original loop functions. Now our K-map looks as in Fig. 16.4.
6. Depending on the number of colors and compatibility of F_1 and $\overline{F_2}$: either find the Ashenhurst decomposition, or create a realization that uses a multiplexer, and that minimizes the number of

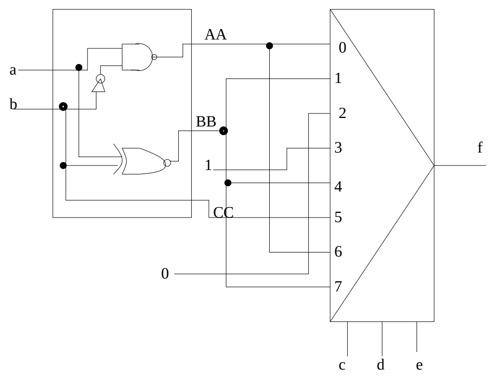


Figure 16.5: The PUB decomposition with column multiplicity of 3 realized with a multiplexer and gates

different functions given to the data inputs of the multiplex, and the total complexity of these functions.

The respective realization with multiplexers is shown in Fig. 16.5. The names of the wires: AA, BB, and CC correspond to the colors of the combined functions. Let us observe that the don't cares have been utilized to simplify function CC.

The above method based on minimal graph coloring of the loop pair incompatibility graph is a starting point to the generalized decompositions introduced in [?].

In PUB decomposition, a function is called $\nabla(k)$-*decomposable* when the row multiplicity on non-trivial functions is $\nabla(k)$. It means that $\nabla(k)$, or less, of the data inputs to the multiplexer are determined to be the non-trivial functions, where k is the number of address variables to this multiplexer.

The *Strong PUB Decomposition,* the most similar to Curtis from the DFC-minimization point of view, assumes that $\nabla(k) = 2^{k-1}$. We created also the concept of the *Weak PUB Decomposition,* or $\nabla(k)$-decomposition, where $\nabla(k)$ is an integer $\in [\ 2^{k-1},\ 2^k - 1\ ]$, defined by the user for each value of k.

The *trivial functions* are defined here for PUB decompositions as any functions that are trivial or easy to realize. For instance, the following sets of trivial functions have been considered [?]:

1. constants 0 and 1,
2. constants 0 and 1, and literals (variables and their negations),
3. constants 0 and 1, products of arbitrary number of literals, sums of arbitrary number of literals, EXORs of arbitrary number of literals.

The following can be observed regarding the checking of trivial functions in PUB decompositions.

1. In the first two cases it is very easy to find trivial functions. Constant 0 is assigned, when the ON-set of the cofactor function f_{COF_i} is empty. Constant 1 is assigned, when the OFF-set of the cofactor function f_{COF_i} is empty. Literal ${x_r}^i$, $i = 0, 1$, is assigned, when the intersection of ${x_i}^i$ with the OFF-set of cofactor function f_{COF_i} is empty, and the intersection of x_i with the ON-set of cofactor function f_{COF_i} is non-empty.
2. To find if f_{COF_i} is *a product of literals*, the supercube SUP(f_{COF_i}) of all true minterms in f_{COF_i} is found. If SUP(f_{COF_i}) $\cap$ OFF(f_{COF_i}) $\neq \emptyset$ then SUP(f_{COF_i}) or a product of any subset of literals

from SUP(f_{COF_i}) that does not intersect OFF(f_{COF_i}) is a product_of_literals data function for f_{COF_i}.

3. To find if f_{COF_i} is *a sum of literals,* the supercube SUP(f_{COF_i}) of all false minterms in f_{COF_i} is found. If SUP(f_{COF_i}) $\cap$ ON(f_{COF_i}) $\neq \emptyset$ then $\overline{SUP(f_{COF_i})}$ or a complement of any subset of literals from SUP(f_{COF_i}) that does not intersect ON(f_{COF_i}) is a sum_of_literals data function for f_{COF_i}.

4. To find if f_{COF_i} is *an EXOR of literals* $\{x_j, x_k, x_l\}$, one has to calculate the expression:

$$EXOR\_EXPR = f_{COF_i} \oplus x_j \oplus x_k \oplus x_l \tag{16.1}$$

If the ON-set of EXOR_EXPR = $\emptyset$, then $x_j \oplus x_k \oplus x_l$ is an exor_of_literals data function for f_{COF_i}. Let us observe, that it is sufficient to check exors of literals with only one literal negated.

It is obvious, that all the above conditions can be tested on K-maps, arrays of cubes, partitions, or any other representation of Boolean functions.

In addition, these rules can be used to not only the original input variables, but also to any intermediate variables created in the decomposition process as well as the output variables (i.e. the output functions).

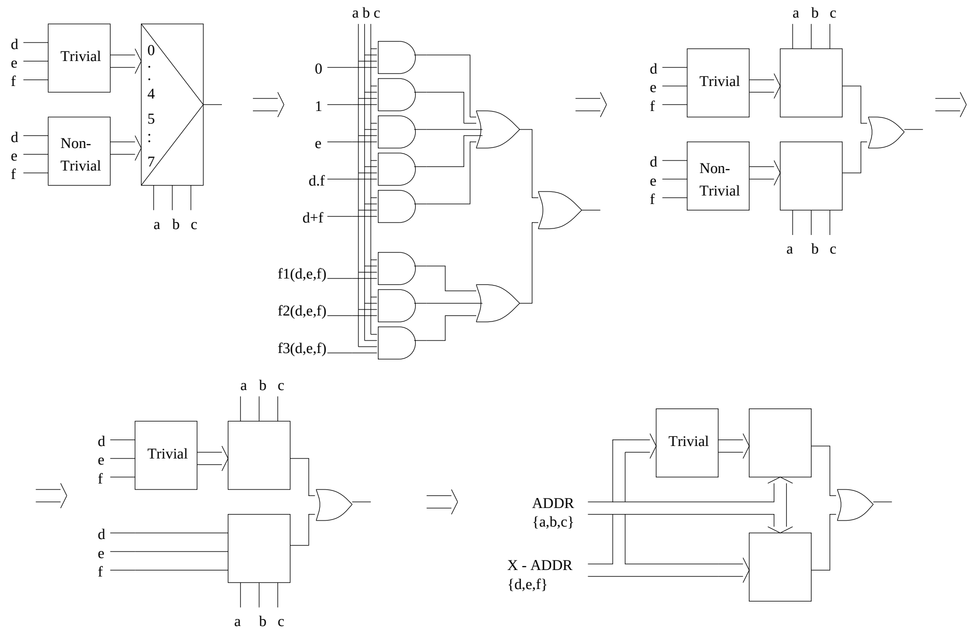


Figure 16.6: Different views at the Disjoint PUB Decomposition

Figure 16.6 presents a step-by-step transformation to explain different views at the Disjoint PUB Decomposition. The final structure is a general diagram of a disjoint PUB decomposition.

16.1.2 Non-disjoint PUB Decomposition

When bound and free sets are non disjunctive, PUB function decomposition is refered to as non-disjoint PUB decomposition. To ilustrate the decomposition process let us use an example:

abe \ cde	000	001	011	010	110	111	101	100
000	0			0	0			-
001		-	1			1	0	
011		0	-			-	1	
010	-			0	0			-
110	0			0	-			1
111		1	1			-	-	
101		-	-			0	0	
100	1			-	1			0

Figure 16.7: K-map of a function to a non-disjoint PUB decomposition from Example 9.2

Cofactor	ON	OFF
XXX000	100XXX	000XXX, 110XXX
XXX001	111XXX	011XXX
XXX011	111XXX, 001XXX	-
XXX010	-	0X0XXX, 110XXX
XXX110	100XXX	0X0XXX
XXX111	001XXX	101XXX
XXX101	011XXX	001XXX, 101XXX
XXX100	110XXX	100XXX

Table 16.1: ON and OFF cofactor cubes of a function from Example 9.2

Example 16.1.2

Let us analyse the function from Example 16.1.1 and define the bound (B) and free (A) sets as being: $B = \{a, b, e\}, A = \{c, d, e\}$. Function's K-map is shown in Fig. 16.7 and the ON and OFF sets are:

ON(f) = {001X11, 010101, 110001, 110100, 111011, 101110, 10000X}
OFF(f) = {000000, X00101, 0X1X10, X11010, 010001, 110000, 101111, 10010X}

Following the procedure described in section 16.1.1 we then find cofactors of columns (Table 16.1). Cofactors XXX011 and XXX010 may be discarded as they contain only 1's and 0's.

The incompatibility graph for the function cofactors is shown in Fig. 16.8. We need only two colors to color the graph which means that the function may be Ashenhurst-decomposable. It would be decomposable if the functions F_1 and F_2 coresponding to color 1 and color 2 would meet the following condition: F_1 is compatible to $\overline{F}_2$. Let us now determine F_1 and F_2. Merging all the loop functions of the same color we obtain new functions (specified by ON and OFF sets):

AA:
ON(XXX000) ∪ ON(XXX110) ∪ ON(XXX101) = {100XXX, 011XXX}
OFF(XXX000) ∪ OFF(XXX110) ∪ OFF(XXX101) = {0X0XXX, 110XXX, 001XXX, 101XXX}

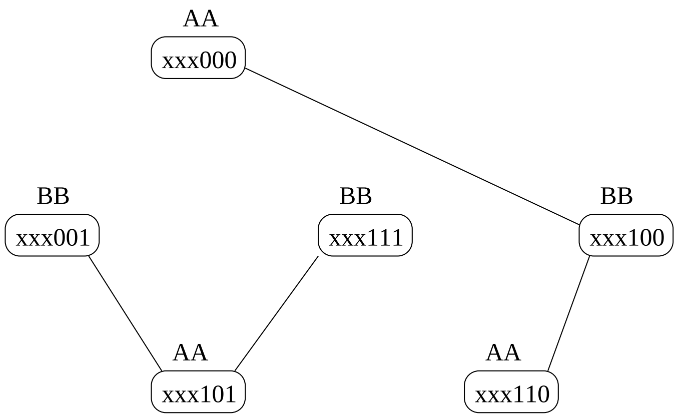


Figure 16.8: Incompatibility graph for non-disjoint PUB decomposition

BB:
ON(XXX001) ∪ ON(XXX111) ∪ ON(XXX100) = {111XXX, 001XXX, 110XXX}
OFF(XXX001) ∪ OFF(XXX111) ∪ OFF(XXX100) = {011XXX, 101XXX, 100XXX}

The K-map coresponding to the new functions is shown in Fig. 16.9. Functions F_1 and F_2 coresponding to colors AA and BB are: $F_1 = a\overline{b}\overline{e} + \overline{a}be = (a \oplus b)(a \oplus c)(\overline{b \oplus e})$, $F_2 = ab + \overline{a}\,\overline{b} = \overline{a \oplus b}$. The condition F_1 compatible to $\overline{F_2}$ is not met in our case so the function is not Ashenhurst-decomposable. The PUB realization of our function using a multiplexer is shown in Fig. 16.10.

Fig. 16.11 compares the disjoint and non-disjoint Curtis and PUB decompositions.

abe \ cde	000	001	011	010	110	111	101	100
000	0				0		0	
001	0	1			0	1	0	1
011	1	0			1	0	1	0
010	0				0		0	
110	0	1			0	1	0	1
111		1				1		1
101	0	0			0	0	0	0
100	1	0			1	0	1	0

AA

BB

Figure 16.9: The K-map coresponding to the completed function with coloring of compatible columns shown and the multiplicity index of 2

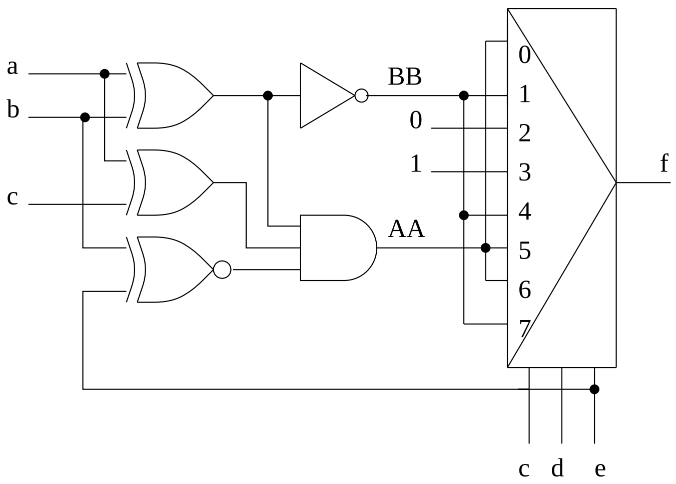


Figure 16.10: Realization of the PUB decomposition of function from Example 9.2 using a multiplexer

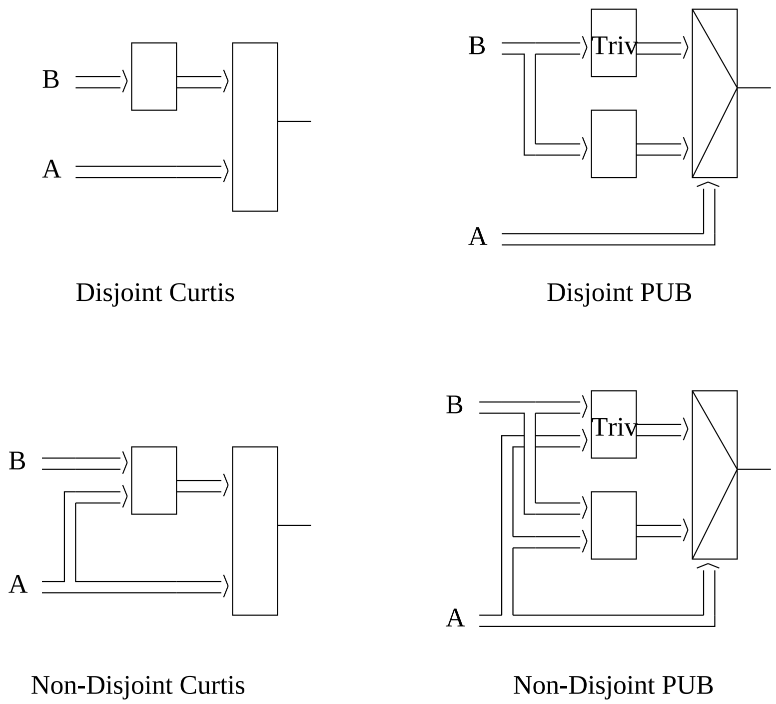


Figure 16.11: Comparison of the disjoint and non-disjoint Curtis and PUB decompositions

16.2 Decomposition of Fuzzy Logic Functions.

Decomposition of fuzzy logic functions was first discussed by Abraham Kandel in [?]. Later he found errors in his paper which, after removing, made his method completely impractical. Therefore, in the next paper by Francioni and Kandel [?], they propose a much improved method for disjoint decomposition.

The disadvantages of their new method include however the following:

1. Lack of intuitive explanation how their decomposition rules have been created makes it difficult to improve the algorithm.
2. The algorithm is strictly tabular, and no computer implementation exists.
3. The algorithm is quite complex and requires another program to minimize fuzzy functions.

The non-disjoint decomposition of fuzzy logic functions remains an open problem. Francioni and Kandel mention interesting examples of applications of fuzzy logic decomposition, including medical diagnosis, hardware realization of fuzzy logic circuits, pattern recognition, and document retrieval systems.

16.2.1 Introduction

Fuzzy Logic

Definition 16.2.1 *Fuzzy set, defined as A, is a subset of universe of discourse U, where A is characterized by a membership function $u_A(x)$. The membership function $u_A(x)$ is associated with each point in U and is the "grade of membership" in A. The membership function $u_A(x)$ is assumed to range in the interval [0,1], 0 corresponding to non-membership and 1 corresponding to full membership. The ordered pairs form $\{(x, u_A(x))\}$ to represent the fuzzy set member and the grade of membership, [?].*

Operations:

The intersection operation of two fuzzy sets uses the symbols: $\cap, *, \wedge$, AND, or min. The union operation of two fuzzy sets uses the symbols: $\cup, \vee, +$, OR, or max.

- Equality of two sets is defined as $A = B \leftrightarrow u_A(x) = u_B(x)$ for all $x \in X$.
- Containment of two sets is defined as A subset $B, A \subseteq B \leftrightarrow u_A(x) \leq u_B(x)$ for all $x \in X$.
- Complement of a set A is defined as $\overline{A}$, where $u_{\overline{A}}(x) = 1 - u_A(x)$ for all $x \in X$.
- Intersection of two sets is defined as $A \cap B$ where $u_{A \cap B}(x) = \min\{(u_A(x), u_B(x))\}$ for all $x \in X$ where $C \subseteq A, C \subseteq B$ then $C \subseteq A \cap B$.
- Union of two sets is defined as $A \cup B$ where $u_{A \cup B}(x) = \max\{(u_A(x), u_B(x))\}$ for all $x \in X$ where $D \supseteq A, D \supseteq B$ then $D \supseteq A \cup B$, [?].

Example 16.2.1 Let $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and consider the following two fuzzy sets A and B:
$A = \{(3, 0.8), (5, 1), (6, 0.6)\}$ and $B = \{(3, 0.7), (4, 1), (6, 0.5)\}$

Then:
$A \cap B = \{(3, 0.7), (6, 0.5)\}$
$A \cup B = \{(3, 0.8), (4, 1), (5, 1), (6, 0.6)\}$
$\overline{A} = \{(1, 1), (2, 1), (3, 0.2), (4, 1), (6, 0.4), (7, 1), (8, 1), (9, 1), (10, 1)\}$

Identities:

Idempotency:

$X + X = X$, $X * X = X$

Commutativity:

$X + Y = Y + X$, $X * Y = Y * X$

Associativity:

$(X + Y) + Z = X + (Y + Z)$, $(X * Y) * Z = X * (Y * Z)$

Absorption:

$X + (X * Y) = X$, $X * (X + Y) = X$

Distributivity:

$X + (Y * Z) = (X + y) * (X + Z)$, $X * (Y + Z) = (X * Y) + (X * Z)$

Complement:

$\overline{\overline{X}} = X$

DeMorgan's Laws:

$\overline{(X + Y)} = \overline{X} * \overline{Y}$, $\overline{(X * Y)} = \overline{X} + \overline{Y}$

Transformations:

Some transformations of fuzzy sets with examples follow:

$\overline{x}b + xb = (x + \overline{x})b \neq b$

$xb + x\overline{x}b = xb(1 + \overline{x}) = xb$

$\overline{x}b + x\overline{x}b = \overline{x}b(1 + x) = \overline{x}b$

$a + xa = a(1 + x) = a$

$a + \overline{x}a = a(1 + \overline{x}) = a$

$a + x\overline{x}a = a$

$a + 0 = a$

$x + 0 = x$

1 1

X = 1 - X

0 X 0 X

Figure 16.12: Membership Function and Inverse

$x * 0 = 0$

$x + 1 = 1$

$x * 1 = x$

Example 16.2.2

$$a + xa + \overline{x}b + x\overline{x}b = a(1 + x) + \overline{x}b(1 + x)$$
$$= a + \overline{x}b$$

Example 16.2.3

$$a + xa + \overline{x}a + x\overline{x}a = a(1 + x + \overline{x} + x\overline{x})$$
$$= a$$

Differences between Boolean logic and Fuzzy logic

In Boolean logic the values of a variable and its negation are always disjoint $(X * \overline{X} = 0)$ and $(X + \overline{X} = 1)$ because the values are either zero or one. In fuzzy logic the membership functions can be either disjoint or non-disjoint. The membership function is determined by the grade of membership and can be any value in the interval $[0, 1]$ from zero to one (it means, both arguments and values are in this interval). Fuzzy membership functions can be any function that can be realized in the interval from zero to one. For simplicity, the term 'grade of membership' of a variable in a set will be replaced by the term 'fuzzy variable'. An example of a fuzzy membership function X with its inverse membership function is shown in Figure 16.12.

Another fuzzy logic membership function, which is linear, can be defined by the function shown in Figure 16.13.

With the membership function from Figure 16.13 the inverse of the membership function is shown in Figure 16.14.

The fuzzy intersection of variables X and its complement $\overline{X}$ is not empty, or is not equal to zero because the membership functions are non-disjoint. In Boolean logic the intersection or AND of a variable and its complement is zero. The intersection of membership functions from X Figure 16.13 and its complement $\overline{X}$ Figure 16.14 is shown in Figure 16.15.

X

1

0

Figure 16.13: Membership function X

1

X = 1 - X

0

X

Figure 16.14: A complement of the membership function X

X

1

0

X

Figure 16.15: Intersection $X * \overline{X} \neq \emptyset$

X

1

0 X

Figure 16.16: Union $X + \overline{X} \neq 1$

X

1

0 X

Figure 16.17: $X \oplus X = X * \overline{X} + \overline{X} * X = X * \overline{X}$

The fuzzy union of the variables X and its complement $\overline{X}$ is not equal to one because the membership functions are non-disjoint. In Boolean logic the union or OR of a variable and its complement is one. The union of membership functions from X Figure 16.13 and its complement $\overline{X}$ Figure 16.14 is shown in Figure 16.16.

The xor of the fuzzy variables X and X is shown in Figure 16.17 which is the same as the intersection of X and $\overline{X}$.

The xor of the fuzzy variables X and $\overline{X}$ is shown in Figure 16.18 which is the same as the union of X and $\overline{X}$.

16.2.2 The Problem of Decomposition of Fuzzy Functions

Theorem 16.2.1 *Let y, z be a partition on X. A completely specified Boolean function $f(X)$ possesses a simple disjunctive decomposition with bound set Y and free set Z if and only if its Y/Z decomposition map has a column multiplicity $v \leq 2$.*

The proof of this theorem is based on the Shannon's expansion theorem which proves that for a Boolean function f,

$$f(x_1, x_2, ..., x_n) = x_1 f(1, x_2, ..., x_n) + \overline{x_1} f(0, x_2, ..., x_n)[?].$$

1 X

0 X

Figure 16.18: $X \oplus \overline{X} = X * X + \overline{X} * \overline{X} = X + \overline{X}$

y z \ w x	00	01	10	11
00				I
01				
10				
11	I	I	I	

Figure 16.19: Decomposition map of Example 4.

Unfortunately, the Shannon's expansion theorem (and therefore Theorem 16.2.1) does not hold true for fuzzy functions which are not disjoint binary functions but non-disjoint infinity-valued functions. The next example will shows that a function can be decomposed correctly for binary functions but does not decompose correctly for fuzzy functions.

Example 16.2.4 Let the function f be

$$f(w, x, y, z) = w\overline{x}yz + \overline{w}\;\overline{x}yz + \overline{w}xyz + wx\overline{y}\;\overline{z}.$$

The Y/Z decomposition map of f corresponding to the Boolean representation of the variables and to the partition $Y = \{w, x\}$, $Z = \{y, z\}$ is shown in Figure 16.19.

Therefore the Boolean function represented by f has a simple disjunctive decomposition of the form

$$F[G(w, x), y, z] = G\overline{y}\;\overline{z} + \overline{G}yz$$

where $G(w, x) = wx$.

However, if f is a fuzzy function, then the variables represent grades of membership in a set, $u(x_i)$, and take on values between 0 and 1 inclusive and not just 0 and 1.

Let $u(w) = 0.4, u(x) = 0.2, u(y) = 0.9$ and $u(z) = 0.9$.

Then $f(w, x, y, z) = w\overline{x}yz + \overline{w}\ \overline{x}yz + \overline{w}xyz + wx\overline{y}\ \overline{z}$

$= 0.4 * 0.8 * 0.9 * 0.9 + 0.6 * 0.8 * 0.9 * 0.9 + 0.6 * 0.2 * 0.9 * 0.9 + 0.4 * 0.2 * 0.1 * 0.1$

$= 0.4 + 0.6 + 0.2 + 0.1$

$= 0.6$

But

$F[G(w, x), y, z] = (wx)\overline{y}\ \overline{z} + \overline{(wx)}yz$

Using DeMorgan's Law and distributive law we obtain:

$= wx\overline{y}\ \overline{z} + \overline{w}yz + \overline{x}yz$

$= 0.4 * 0.2 * 0.1 * 0.1 + 0.6 * 0.9 * 0.9 + 0.8 * 0.9 * 0.9$

$= 0.1 + 0.6 + 0.8$

$= 0.8$

As the previous example shows Theorem 16.2.1 does not hold. Next graphical methods are used to correct the decomposition of fuzzy functions.

16.2.3 Graphical Representation of Fuzzy Functions for Decomposition.

In Boolean logic maps the symbols "1", "0", and "-" are used to describe a binary function. These symbols denote the values of cells (minterms). Since there is only a finite number of unique terms in a fuzzy function, a symbol I can be used to show if a term is present [?].

Fuzzy Maps

As presented by Schwede and Kandel [?], the fuzzy map may be regarded as an extension of the Veitch diagram [?], which forms the basis for the Karnaugh map [?]. Fuzzy maps pictorially describe the set of all fuzzy implicants which represent a fuzzy function. In a K-map of n variable minterms can be represented by 2^n areas in the map. A fuzzy map of n variables can be represented by 4^n areas in the map. The symbol I is used to represent a term in the fuzzy function, $F(x_1, x_2, ..., x_n)$. For two variables fuzzy map, the columns are labeled $x_1\overline{x}_1, x_1, \overline{x}_1, 1$ and the rows are labeled $x_2\overline{x}_2, x_2, \overline{x}_2, 1$ as shown in Figure. ?? [?].

The construction of fuzzy maps of min or + as intersection, and max or * as union, the area where I is placed is easy to determine. The intersection and union of two variables are shown in Figure ??a and ??b, respectively.

The column and row headings are conventionally replaced with quarternary numbers representing the binary headings. There are four combinations for each variable x_i, i in $1, 2, ..., n$ variables, to be represented in the headings of the rows and columns.

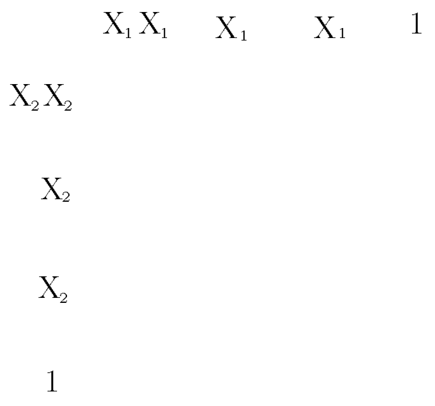


Figure 16.20: Fuzzy Map

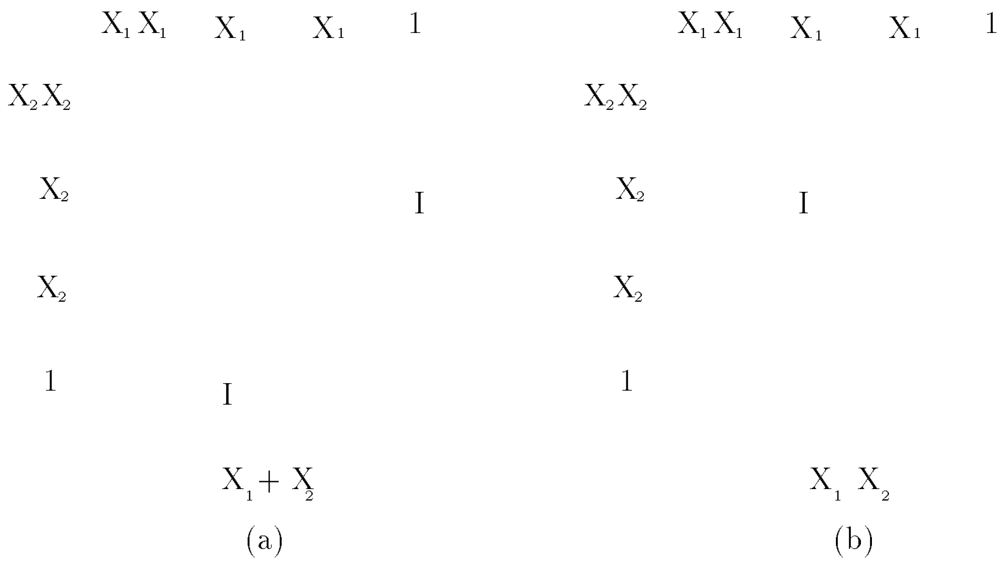
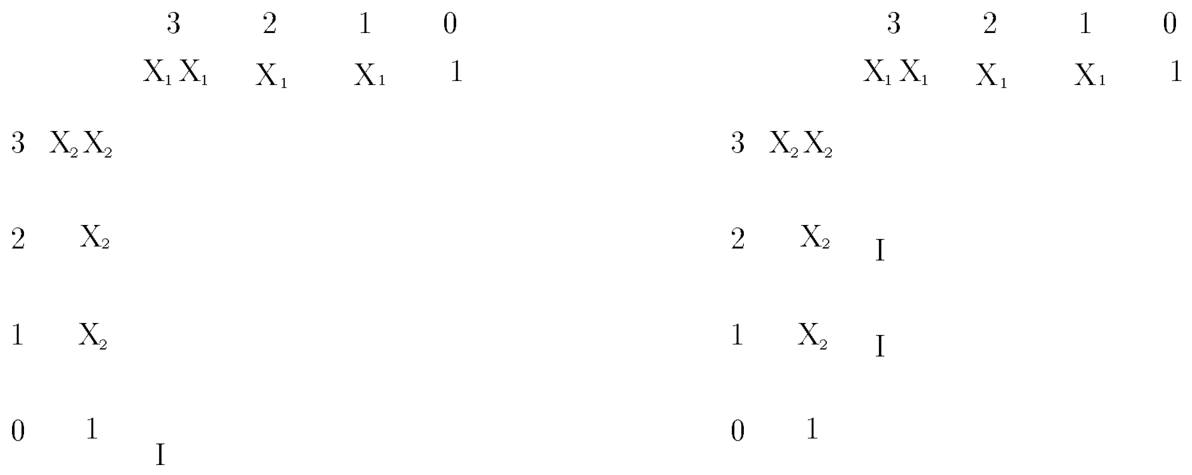


Figure 16.21: Max and Min representations using fuzzy maps with (n = 2)



(a) (b)

Figure 16.22: $F(x_1, x_2) = x_1\overline{x}_1$, $F(x_1, x_2) = x_1\overline{x}_1x_2 + x_1\overline{x}_1\overline{x}_2$

1. This heading is vacuous in x_i. The pair $x_i\overline{x}_i$ is denoted by 00 and is represented by 0.
2. This heading includes $\overline{x}_i$ but not x_i. The pair $x_i\overline{x}_i$ is denoted by 01 and represented by 1.
3. This heading includes x_i but not $\overline{x}_i$. The pair $x_i\overline{x}_i$ is denoted by 10 and represented by 2.
4. This heading includes x_i and $\overline{x}_i$. The pair $x_i\overline{x}_i$ is denoted by 11 and represented by 3.

Fuzzy map representation has important properties which distinguish it from Boolean maps. One of the most obvious of these is that the squares (each representing a different implicant) are not disjoint in the sense that the absence of an I in a square does not necessarily exclude the associated implicant from an equivalent representation of the function. Conversely, the presence of an I does not imply essentially the associated implicant. The two different representations of the same function Figures ??a and ??b are examples of these properties. The two fuzzy maps are of size $n = 2$ and the function in Figure ??a is the same as the function in Figure ??b because x_2 is vacuous so $F(x_1, x_2) = x_1\overline{x}_1$ can be expanded to $F(x_1, x_2) = x_1\overline{x}_1x_2 + x_1\overline{x}_1\overline{x}_2$.

S-Maps

S-maps are another way to arrange two-variable fuzzy maps for n variables. To construct an n-variable S-map, whole one- or two- variable fuzzy maps are treated as though they were squares of an S-map on $n-1$ or $n-2$ variables. This method is just iterated for n variable S-maps. These subsets of the logical space are called sub-maps and are a very important feature of S-maps. As in fuzzy maps, the binary headings for the columns and rows are converted to a quarternary representation as shown in Figure ??. The sub-map boundaries are indicated by the vertical solid lines.

S-maps have some important properties as follows:

1. The dominant square of a fuzzy map is the square representing the minterm 1.
   The dominant square of the entire S-map is designated 0 while the dominant square of any sub-map is the lowest numbered square in that sub-map. The dominate square of the entire S-map in the S-map shown in Figure ?? is square $X_1 = 0$, $X_2 = 0$, and $X_3 = 0$. The sub-map dominate squares are $X_1 = 3$ $X_2 = 0$ $X_3 = 0$, $X_1 = 2$ $X_2 = 0$ $X_3 = 0$, and $X_1 = 1$ $X_2 = 0$ $X_3 = 0$.

	3				2				1				0				X_1
	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	X_2
3																	
2																	
1																	
0																	
X_3																	

Figure 16.23: S-map for n = 3.

Theorem 16.2.2 *(Kandel [?]).*
A sub-map subsumes (is included by) a second sub-map if the minterm represented by the dominant square of the first subsumes the minterm represented by the dominant square of the second sub-map.

2. **Theorem 16.2.3** *(Kandel [?]).*
   Every subset (pattern of implicant squares) of a sub-map is included in the dominant square of a subsumed sub-map.

3. **Theorem 16.2.4** *(Kandel [?]).*
   Any subset of a sub-map is included by the same pattern on a subsumed sub-map.

The same manipulations used on a two-variable map can be used on an n-variable S-map. On S-maps entire sub-map sized patterns behave as single implicants as on the two-variable map.

Both fuzzy maps and S-maps help to decompose fuzzy switching functions.

- Fuzzy map is used to find if a decomposition exists.
- S-map is used to determine the decomposition, it means, to calculate the G and H functions.

16.2.4 Fuzzy Logic Multiplexers

Fuzzy logic multiplexers are like binary logic multiplexers in that the values of one or more control variables influence what input data goes to a single output. They both have many inputs and one output. In binary logic multiplexers the control variable or variables select which input from the multiplexer is transmitted to the output. In binary logic multiplexers the only input that is transmitted to the output is selected by the control variable or variables. In fuzzy logic multiplexers the inputs can be transmitted simultaneously to the output selected by the inputs values and the control variable value(s). In the fuzzy logic multiplexer, the different input variables are compared with a constant 1, a control variable or variables, their complements, and the control variable or variables and their complements, in parallel, using minimum operations. The maximum of these minimum operations makes the output. A fuzzy logic multiplexer and its internal realization with fuzzy gates are shown in Figure ??.
This multiplexer works as follows:

1. The first input is compared with the constant 1, which is the highest value, because the control variable is vacuous so the input is the minimum value.
2. The second input value is compared to the inverted control variable $1 - X$.
3. The third input is compared to the control variable X.

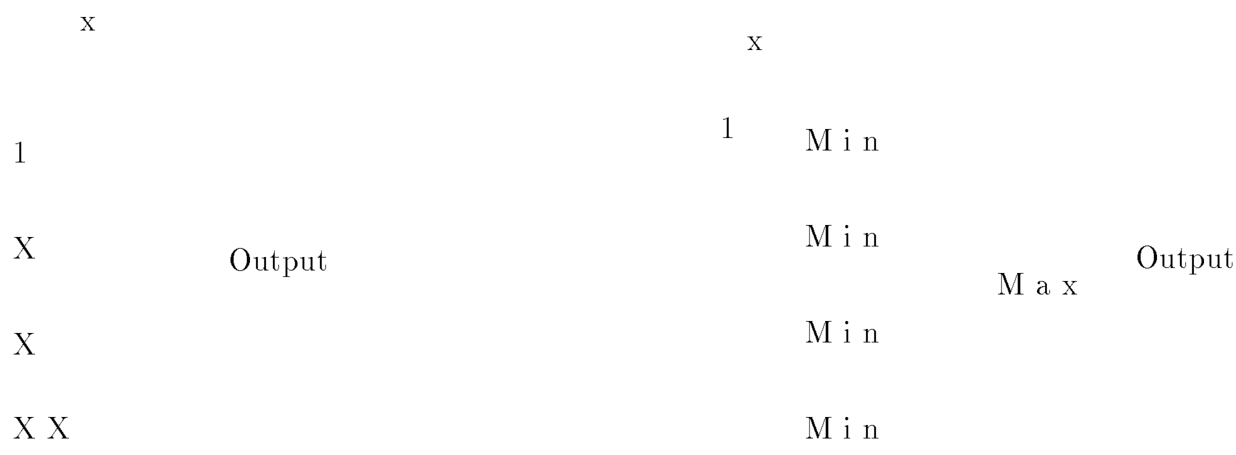


Figure 16.24: Fuzzy Multiplexer, (a) symbol, (b) internal structure using a Fuzzy Inverter, Min,and Max gates

4. The fourth input value is compared with the control variable X and the inverted control variable value $1 - X$.

After these four minimum comparisons are done the maximum of these four is the output.

Example 16.2.5
$f(w,x,y,z) = \overline{w}\ \overline{x}\ \overline{z} + w\overline{x}z + \overline{w}yz + wy\overline{z} = (\overline{w}\ \overline{z} + wz)\overline{x} + (\overline{w}z + w\overline{z})y$

If we let $G(w,z) = \overline{wz} + wz$

then $f(w,x,y,z)$ can be decomposed as follows:

$$f(w,x,y,z) = G\overline{x} + \overline{G}y = F[G(w,z),x,y].$$

The decomposed fuzzy circuit uses a fuzzy multiplexer as shown in Figure ??. The data inputs to the multiplexer with 2 address variables are labeled like in fuzzy maps with a quarternary representation. The function that the multiplexer needs to perform is $G\overline{x} + \overline{G}y$. The inputs to the multiplexer are G and $\overline{G}$ which needs to go to the multiplexer inputs $\overline{x}$ and y. The $\overline{x}$ input is labeled 10 because x is the first column and y is the second column. The 10 input label is translated to $\overline{x}$ and 1 in the quarternary representation the 1 and $\overline{x}$ is intersected to equal $\overline{x}$. The y input is labeled 02 because in translating from the quarternary representation 0 is 1 and 2 is y, 1 intersected with y results in y.

16.2.5 Decomposition of fuzzy functions

Definition 16.2.2 *A fuzzy switching function, $f(X)$, is said to be* **normal** *in x_i if it is in disjunctive normal form where no minterm includes an $x_i\overline{x}_i$, and is written $f_i(X)$. If $f(X)$ is normal in y_i for every y_i in $Y = y_1, y_2, ..., y_n$, then $f(X)$ is said to be normal in Y and is written $f_y(X)$.*

Definition 16.2.3 *Let $\{Y/Z\}$ be a* **partition** *on the set of fuzzy variables X. A fuzzy decomposition chart for a function $f(X)$ with a $\{Y/Z\}$ partition of X is a reduced fuzzy map where the variables of Y define the columns, the variables of Z define the rows and the following columns are excluded: the 00 column or column heading 0 and the 11 column or column heading 3. A fuzzy switching function, normal in Y, is represented on a fuzzy decomposition chart by placing an I in the column and row representing each minterm of the function and leaving the other positions blank.*

x y

w
G
z

for x for y
0 = 1 0 = 1
1 = x 1 = y
2 = x 2 = y
3 = xx 3 = yy

G

0 0
1 0
2 0
0 1
0 2
1 1
1 2
2 1
2 2
3 0
0 3
1 3
2 3
3 1
3 2
3 3

Output

Figure 16.25: A Decomposition of a Fuzzy Function from Example 19.5 using a Fuzzy Multiplexer.

X_3 \ $X_1 X_2$	2 2	2 1	1 2	1 1	0 2	0 1	2 0	1 0
3			I		I			
2							I	
1			I		I		I	
0							I	

Figure 16.26: Fuzzy decomposition chart for 3 variables

Definition 16.2.4 *CAP is a* **unary operation** *denoted by the symbol* $\hat{}$ *such that:* $\hat{0} = 0$; $\hat{1} = 2$; *and* $\hat{2} = 1$. *This corresponds to permutation of input values.*

Definition 16.2.5 *Columns in a fuzzy decomposition map are said to be* **I-equivalent** *if and only if all I's or blanks in every row of one column is the same as another column. Every column is I-equivalent to itself.*

Example 16.2.6 In the fuzzy decomposition chart for 3 variables shown in Figure ??, where x_1 and x_2 are bound and x_3 is free, columns 12 and 02 are I-equivalent. Column 20 is not I-equivalent to any other column. And columns 22, 21, 11, 01 and 10 are all I-equivalent.

Definition 16.2.6 *The number of distinct columns in a fuzzy decomposition chart which are non-vacuous in I is called the* column distinction count *and is designated by c. (This count does not include blank columns.)*

Definition 16.2.7 *A fuzzy decomposition chart represents a* trivial case *of fuzzy switching function normal in Y to be decomposed if and only if all the columns containing an I are I-equivalent.*

Definition 16.2.8 *These functions are used for column compatibility later on in Definition ??.*

Let $T = \{0, 1, 2\}$.

COMPL 1 is a function $f : T^4 \longrightarrow T^2$

such that

$$f(a, b, c, d) = \begin{cases} \hat{b}\hat{c} & \textit{if } b = c \\ bc & \textit{otherwise} \end{cases}$$

COMPL 2 is a function $f : T^4 \longrightarrow T^2, T^2$

	3	2	1	0	x_i
3					
2		a			
1				a	
0			a	g	
x_j					

Figure 16.27: DIP 1 $a(x_i, x_j) = x_i x_j$, $\overline{a} = \overline{x}_i + \overline{x}_j$

	3	2	1	0	x_i
3					
2				b	
1			b		
0		b		g	
x_j					

Figure 16.28: DIP 2 $b(x_i, x_j) = \overline{x}_i \overline{x}_j$, $\overline{b} = x_i + x_j$

such that

$f(a, b, c, d) = bc, da.$

Note. When referencing these functions, ab and cd will represent column headings and should be arranged so that $ab < cd$ with respect to their decimal values.

Definition 16.2.9 *A* decomposition implicant pattern *(DIP) is one of the 5 implicant patterns shown in the S-maps of Figure* **??**, **??**, **?? ??**, *and* **??** *where a, b, c, d and e represent sub-map patterns; g is any sub-map pattern or can have no sub-map pattern; and no implicants appear in either the 3 row or the 3 column or any other place except g.*

The next definition is based on the representation of the DIP function on fuzzy decomposition charts as opposed to S-maps. This definition provides a procedure to determine whether or not a fuzzy function corresponds to some DIP.

Definition 16.2.10 *The columns of a fuzzy decomposition chart are* compatible *if and only if:*

x_j \ x_i	3	2	1	0
3				
2			c	
1				c
0		c		g

Figure 16.29: DIP 3 $c(x_i, x_j) = \overline{x}_i x_j$, $\overline{c} = x_i + \overline{x}_j$

x_j \ x_i	3	2	1	0
3				
2				d
1		d		
0			d	g

Figure 16.30: DIP 4 $d(x_i, x_j) = x_i \overline{x}_j$, $\overline{d} = \overline{x}_i + x_j$

x_j \ x_i	3	2	1	0
3				
2		e	e	
1		e	e	
0				g

Figure 16.31: DIP 5 $d(x_i, x_j) = x_i \oplus x_j$, $\overline{d} = x_i \odot x_j$

X_3 \ X_1X_2	2 2	2 1	1 2	1 1	0 2	0 1	2 0	1 0
3	I					I		I
2								
1	I					I		I
0	I							

(a)

X_3 \ X_1X_2	2 2	2 1	1 2	1 1	0 2	0 1	2 0	1 0
3		I	I	I				
2	I	I	I	I				
1	I	I	I	I				
0								

(b)

Figure 16.32: Illustration of compatible columns for a Fuzzy function

1. *for any ab column where $a = 0$ or $b = 0$, if column ab is I-equivalent to column ba then the only other column non-vacuous in I is the column represented by COMPL 1 (a, b, b, a)*

   or

   if column ab is I-equivalent to column $\hat{b}\hat{a}$ then
   the only other column non-vacuous in I is the column represented by COMPL 1 $(a, b, \hat{b}, \hat{a})$ or ab is I-equivalent to $\hat{a}\hat{b}$ and all other columns are vacuous of I's;

   or

2. *(if all ab columns where $a = 0$ or $b = 0$ are vacuous in I then) for any ab column, where $a \neq 0$ and $b \neq 0$, if column ab is I-equivalent to column $\hat{a}\hat{b}$ then*
   the only other columns non-vacuous in I are the columns represented by COMPL 2 $(a, b, \hat{a}, \hat{b})$ which must also be I-equivalent.

Example 16.2.7 In Figure ??a the columns are compatible. In Figure ??a from Definition ??.1, the ab column where $a = 0$ or $b = 0$, is the column 01. It is I-equivalent to ba column 10. Then the only other column non-vacuous in I is the column represented by COMPL 1 $(a, b, b, a) =$ COMPL 1 $(0, 1, 1, 0)$ $= f(a, b, c, d) = \hat{b}\hat{c} = 22$ which is the only other non-vacuous column in I. The columns in Figure ??a are compatible from Definition ??.1.

Example 16.2.8 In Figure ??b the columns are not compatible. In Figure ??b column 21 is I-equivalent to 12, column 21 is I-equivalent to 11, and column 12 is I-equivalent to 11. From Definition ??.2 (if all ab columns where $a = 0$ or $b = 0$ are vacuous in I then) for any ab column, where $a \neq 0$ and $b \neq 0$, if column ab or 21 is I-equivalent to column $\hat{a}\hat{b}$ or 12 then the only other columns non-vacuous in I are the columns represented by COMPL 2 $(a, b, \hat{a}, \hat{b})$ = COMPL 2 $(1, 2, 2, 1) = f(a, b, c, d) = bc$, $da = 22$, 11 which must also be I-equivalent. But columns 22 and 11 are not I-equivalent so these columns in Figure ??b are not compatible.

Theorem 16.2.5 *If a fuzzy switching function in Y, $f_y(X)$, is decomposable on a $\{Y/Z\}$ partition of X, $Y = \{y_1, y_2\}$, then it represents a trivial case or it can be represented by a decomposition implicant pattern (DIP) with the bound variables as the outside row and column headings.*

As an example the labels in Figure ??, X_3 is the outside row and X_1 is the outside column. This next theorem is used to determine if a fuzzy switching function can be decomposed or not.

xz \ wy	22	21	12	11	02	01	20	10	00
33									
32									
31									
23									
13						I		I	
03									
30									
22									I
21									
12									
11									
02	I								
01									
20									
10									
00									

0 = 1
1 =x
2 =x
3 = x x

y x z z

w x z z

x z

w y z

Figure 16.33: Fuzzy decomposition chart for $f(w, x, y, z)$

Theorem 16.2.6 *Let $\{Y/Z\}$ be a partition on the fuzzy set X where $Y = \{y_1, y_2\}$. A completely specified fuzzy switching function normal in Y possesses a fuzzy simple disjunctive decomposition with fuzzy bound set Y and fuzzy free set Z if and only if:*

1. *its Y/Z fuzzy decomposition map represents a trivial case, or*
2. *its Y/Z fuzzy decomposition map has columns that are all compatible from Definition ?? and the column distinction count $c \leq 2$.*

The function in Figure ??a is a decomposable function because the column distinction count is $c \leq 2$ and the columns are compatible while in Figure ??b the columns are not compatible so the function is not decomposable.

Example 16.2.9 Let

$$f(w, x, y, z) = \overline{x}\ \overline{y} z \overline{z} + xz + \overline{w}\ \overline{x} z \overline{z} + wyz$$

Let $Y = \{w, y\}$ and $Z = \{x, z\}$. Since $f(w, x, y, z)$ is normal in Y, its fuzzy decomposition map is as shown in Figure ??.

The column distinction count for this chart is 2. The columns 01 and 10 are I-equivalent and the only other column non-vacuous in I is column 22. Since COMPL 1 $(0, 1, 1, 0) = 22$, the columns 22, 01,

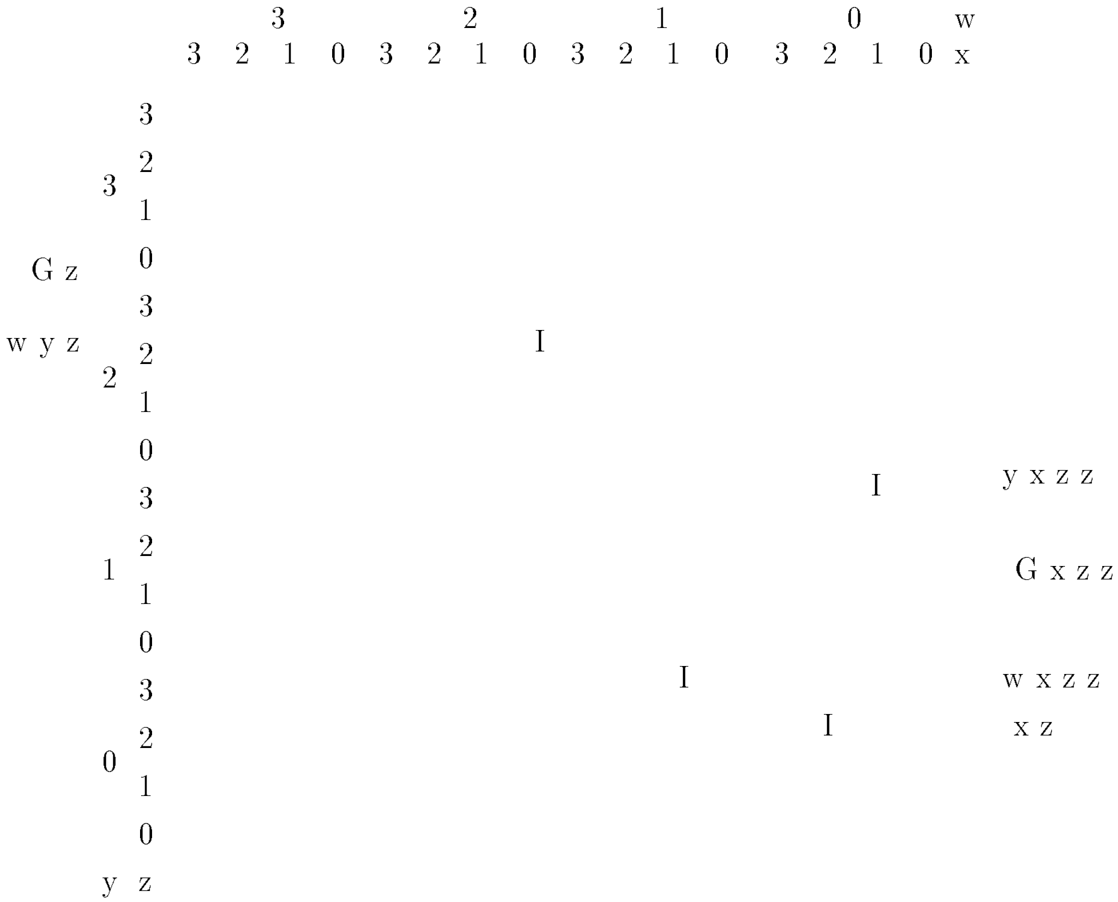


Figure 16.34: S-map of $f(w, x, y, z)$ representing DIP 1.

and 10 are compatible.
Therefore, by Theorem ??, $f(w, x, y, z)$ is decomposable with w and y as bound variables and x and z as free variables. Figure ?? shows the S-map of this function. According to Theorem ??, $f(w, x, y, z)$ can be represented by DIP 1 Figure ?? where the outside row y corresponds to x_i and the outside column w corresponds to x_j.
Using Table ?? this DIP implies:

$G(w, y) = wy$, $\overline{G}(w, y) = \overline{w} + \overline{y}$.

$$f(w, x, y, z) = (wy)z + (\overline{w} + \overline{y})\overline{x}z\overline{z} + xz$$

Therefore substituting $G(w, y) = G(Y)$ and $\overline{G}(w, y) = \overline{G}(Y)$

$$f(w, x, y, z) = F[(G(w, y), x, z)] = G(Y)z + \overline{G}(Y)\overline{x}z\overline{z} + xz$$

The decomposed fuzzy circuit is shown is Figure ??.
Now let $g = G(w, y) = wy$.

Then

$h(X)$	$\overline{h}(X)$	DIP
x_i	$\overline{x_i}$	-
$x_i x_j$	$\overline{x}_i + \overline{x}_j$	1
$\overline{x}_i \overline{x}_j$	$x_i + x_j$	2
$\overline{x}_i x_j$	$x_i + \overline{x}_j$	3
$x_i \overline{x}_j$	$\overline{x}_i + x_j$	4
$x_i x_j + \overline{x}_i \overline{x}_j$	$x_i \overline{x}_j + \overline{x}_i x_j$	5

Table 16.2: Two variable matching DIPs.

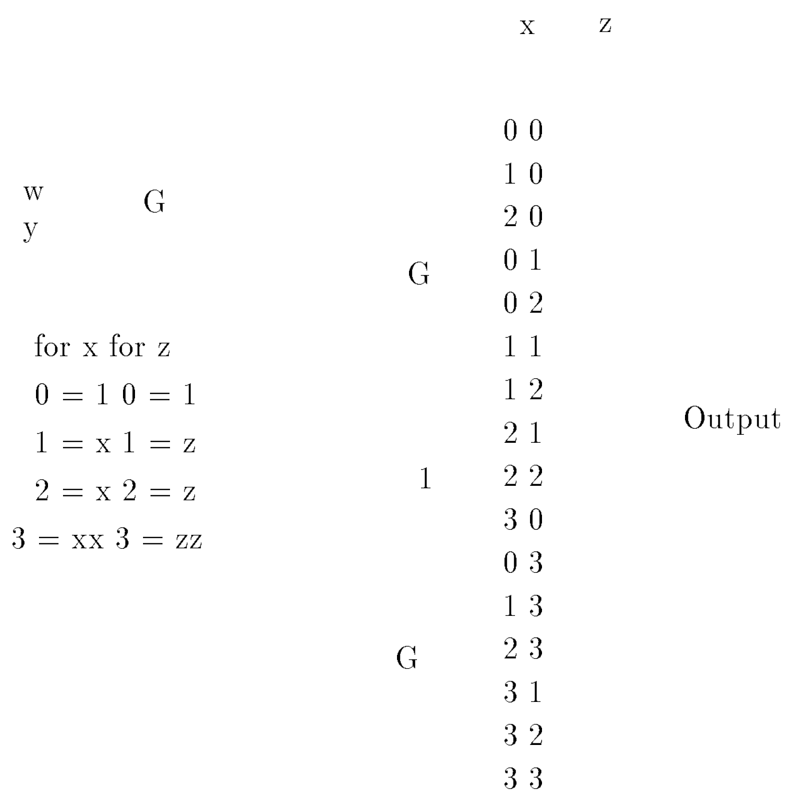


Figure 16.35: The decomposed fuzzy circuit

g x

z

g x z z

	22	21	12	11	02	01	20	10	00
3				I					
2					I		I		
1									
0									

g z

x z

Figure 16.36: $F(g, x, z)$ map

g	3				2				1				0			
z	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0
3																
2														I		
1									I							
0						I										

x

g z

H z

x z

g x z z

H z z

Figure 16.37: S-map of $F(g, x, z)$ representing DIP 2

$F(g, x, z) = gz + \overline{g}\ \overline{x} z\overline{z} + xz.$

The fuzzy decomposition map for $F(g, x, z)$ with g and x as the bound variables is shown in Figure ??.
Again, the column distinction count is 2. The columns 02 and 20 are
I-equivalent and the only other column non-vacuous in I is column 11. Since COMPL 1 (0, 2, 2, 0) = 11, the columns 11, 02, and 20 are compatible. Therefore, by Theorem ??, $F(g, x, z)$ is decomposable with g and x as bound variables and z as a free variable.

The S-map is represented by $F(g, x, z)$ is shown in Figure ??. This gives according to Theorem ??, $F(g, x, z)$ can be represented by DIP 2 Figure ?? where the outside row is x corresponds to x_i and the outside column g corresponds to x_j.
From the Table ??

$H(g, x) = \overline{g}\ \overline{x}$, $\overline{H}(g, x) = g + x.$

$g = G(w, y) = wy$, $\overline{g} = G(w, y) = \overline{w} + \overline{y}$

so by De Morgan's Law and distributive law we obtain:

$H(w, y, x) = \overline{(wy)}\overline{x} = (\overline{w} + \overline{y})\overline{x} = \overline{w}\ \overline{x} + \overline{y}\ \overline{x}.$

$f(w, x, y, z) = H(w, y, x) = (\overline{g}\ \overline{x})z\overline{z} + (g + x)z$

And therefore

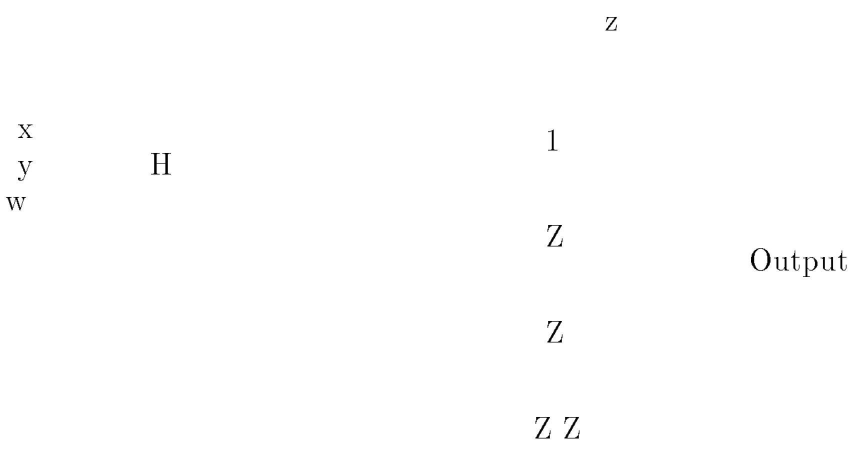


Figure 16.38: The decomposed fuzzy circuit

$$f(w,x,y,z) = F[H(w,x,y),z] = H(w,y,x)z\overline{z} + \overline{H}(w,y,x)z \ [?]$$

The decomposed circuit is shown in Figure ??.
Substituting in H and G we get:

$$f(w,x,y,z) = F[H(w,x,y),z] = H(w,y,x)z\overline{z} + \overline{H}(w,y,x)z = (\overline{g}\ \overline{x})z\overline{z} + (g+x)z$$

$$f(w,x,y,z) = (\overline{g}\ \overline{x})z\overline{z} + (g+x)z$$

$$= (\overline{w}\ \overline{x} + \overline{y}\ \overline{x})z\overline{z} + (wy+x)z$$

This decomposed function takes 5 not, 6 min, 3 max operations while the original function, $f(w,x,y,z) = \overline{x}\ \overline{y}z\overline{z} + xz + \overline{w}\ \overline{x}z\overline{z} + wyz$, takes 6 not, 9 min, and 3 max operations. Decomposing this function makes the circuit simpler by one fuzzy not, and three fuzzy min operations.

16.2.6 Another Decomposition Example. Application of decomposition:

An example of using decomposition will start from a decision table. Such tables are used in medical diagnostics. A fuzzy decision table can be used to represent symptoms and test results which define a particular medical condition. A decision table can represent a fuzzy valued switching function. The decomposition could produce a function made up of variables representing symptoms and tests, or just of the tests, which could be evaluated by personnel other than a doctor. The doctor evaluates the patient's conditions without having to evaluate all the variables. Decomposition can be used to isolate functions which when evaluated could determine whether further tests are even necessary. Less tests are better for the patient by being less expensive and by avoiding tests which involve a high risk for the patient. For an example, Table ?? is a very simple version of a fuzzy decision table which represents conditions for diabetes. The variables represent the results of certain tests which are made and evaluated in order to determine if a person has diabetes. Let:

- Variable x be represented by the amount of sugar present in the urine based on a urine analysis.
- Variable y be represented by the amount of sugar present in the blood derived by a blood test.

Tests	Rule No. 1	Rule No. 2
Urine analysis		x
Blood test	y	
Glucose tolerance test	$\overline{z}$	$\overline{z}$
Positive Diabetes	[1 - 0.6]	

Table 16.3: Fuzzy decision table

- variable z be represented by the tolerance level of sugar by the body based on the results of a glucose tolerance test.

The rules that determine if a person has a diabetes are:

1. Rule 1. If a person has a high level of sugar in the blood and a low level of tolerance, this person may have diabetes.
2. Rule 2. If a person has a large amount of sugar in the urine and a low level of tolerance, this person may have diabetes.

The value in the table represents the grades of membership of the function which determines when someone has diabetes.
The fuzzy valued switching function which represents this decision table is

$$f(x, y, z) = y\overline{z} + x\overline{z}$$

If we let $G(x, y) = x + y$ then this function can be decomposed as follows:

$$f(x, y, z) = (x + y)\overline{z} = G(x, y)\overline{z} = F[G(x, y), \overline{z}]$$

Since the limit of f for the diabetes condition is a grade membership of 0.6, if $G(x, y)$ has a grade membership less than 0.6 then the glucose tolerance test need not be administered to deduce that the person does not have diabetes.[?]

Concluding, Example **??** shows that decomposition of a fuzzy function works correctly, but there is more considerations to be made in order to have a function decomposable. This makes decomposition even more rare to occur than a binary decomposition. When decomposition does work the amount of operations is reduced. We are investigating improvements and generalizations to the presented methods of fuzzy function decomposition.

16.3 The Decomposition of Continuous Functions.

The only paper about generalizing the AC decomposition to continuous functions is one by Ross et al. [?]. However, the method of orthogonal decision diagrams for continuous variables, developed by Pierzchala, Perkowski and Grygiel, should be also easily adaptable to AC decomposition, because of the close link of orthogonal and AC decompositions [?].

Perkowski, Pierzchala and Grygiel developed methods to realize Galois Field operations for small fields in analog/mixed *Field Programmable Analog Array - FPAA*. Paper [?] gives an example of realization of GF(4) operations. In this way, Galois Field type of expansions can be realized in hardware in an analogous way to Galois(2) expansions, i.e. AND/EXOR canonical forms. The regular array structure of the FPAA allows also to realize structures based on Orthogonal Expansions which are not Galois Expansions. The structure of the corresponding circuit resembles a PLA as well, and has columns corresponding to orthogonal functions over GF(2^n), where the functions are realized by cascades of Galois Addition and Galois Multiplication gates, and pass gates. Each column is Galois-multiplied by a constant, and next these values are Galois-added in rows. Thus, the column is a generalization of an AND-term of a PLA, and a row is a generalization of an OR-term (interestingly, a very similar array structure allows also to realize fuzzy logic circuits). Further, the trellis expansions of functions can be also generalized to continuous logic, where obviously variable repetition is required in a general case. Several other generalizations to other algebras, including fuzzy logic, as well as their realizations in regular array structures have been also described in another paper (not published yet). This is a fruitful area of our current research.

In another, yet unpublished paper, they observe that general decision diagrams, universal cells, regular structures and decompositions are so general, that they apply not only to binary logic, but also to multiple-valued logic, Galois Fields and other algebras (fields and some rings). They are also applicable to continuous, i.e. analog logic, for instance fuzzy logic. This helped to create the concept of universal analog-digital "tissue" which would allow to realize a very wide category of logics and other formal systems in hardware. A practical realization of these efforts is the concept of Field Programmable Analog (Mixed) Array which is for analog (mixed) circuits what FPGA is for binary circuits [?].

Perkowski and Hong [?] and Luba and Lasocki [?] presented a case of decomposing a function with various numbers of values in each variable. The recent research of the author is on the expansion of these methods to include also continuous and fuzzy variables. It is done using generalized decision diagrams that have nodes corresponding to binary, multiple-valued, continuous and fuzzy variables.

Chapter 17

Types of Decompositions. Comparison.

17.1 Towards a Unified Approach to Decomposition.

Among several types of decomposition of single-output Boolean functions the most familiar is a simple disjoint decomposition. Let F be a multiple-valued or binary function representing functional dependence $Y = F(X)$, where X is a set of multiple-valued or binary variables and Y is a set of multiple-valued or binary input, binary output variables. Let $X = A \cup B, A \cap B = \emptyset$ and $C \subseteq A$.

Then there is a functional decomposition of F iff

$$F = H(A, G(B, C)) = H(A, Z) \tag{17.1}$$

where G and H denote functional dependencies: $G(B, C) = Z$ and $H(A, Z) = Y$, Z is a set of two-valued variables. If $C = \emptyset$ then H is called a simple *disjoint decomposition* of F. B is called a *bound set*, and A is called a *free set*.

Example of such a decomposition is shown in the block diagram from Fig. ??.

The function F of inputs a, b, c, and d is decomposed to two subfunctions here:

- Subfunction 2, denoted by $G(c, d)$ of inputs c, d. This subfunction will be called a *predecessor function (a predecessor block, a recoder)*. Set of variables $\{c, d\}$ is a *bound set.*

- Subfunction 1 of inputs a, b and G.
  This subfunction, denoted by H, will be called a *successor function (a successor block, an image function)*. Set of variables $\{a, b\}$ is a *free set.*

As we see, instead of realizing a four-variable function we have now to realize one two-input function and one three-input function. Both functions can be next either directly realized, minimized using other circuit optimization techniques, or further decomposed.

Because each of the subfunctions can be further decomposed, a multi-level, tree-like implementation structure is created. (If the same function is used twice, the two instances can be merged (factorized), leading to blocks with fan-out higher than the value of 1 - the circuit is thus no longer a tree, and it becomes a Directed Acyclic Graph (DAG).

Because in the area of circuit synthesis many computer programs for Boolean minimization, especially those that attempt at finding optimal solutions, are practicaly able to solve problems of limited size or even have constraints on the number of inputs, the decomposition into smaller circuits not only produces results of a smaller cost, but can also lead to optimal solutions sooner.

Decomposition can also assume specific kind of realization of a block: PLA, CLB of FPGA or PLD, standard cell libraries, etc.

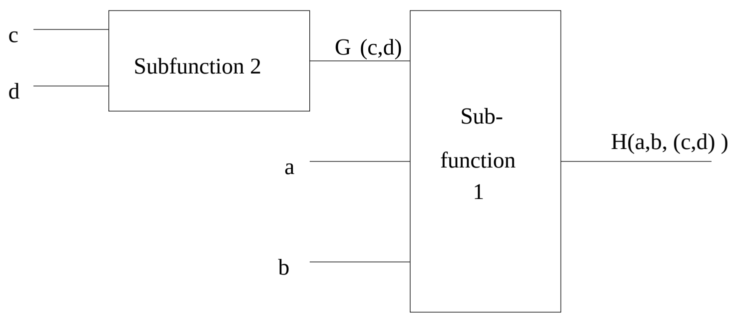


Figure 17.1: Fig.2.1., former fig1. An example of a simple disjunctive decomposition

Similarly, many problems related to decision making in the area of AI or data base design, are of the following nature: given a decision table - find the minimal decision algorithm associated with the table. This process includes reduction of attributes as well as reduction of decision rules and removing all superfluous decision rules from the algorithm. The above methodology, however, when compared with circuit design approaches, suffers from the absence of functional decomposition strategies, as those applied for binary functions, especially for FPGA-based implementations [?, ?, ?]. The promising results of these approaches in AI seem to indicate that the concept of functional decomposition should be investigated more generally and in more detail.

Understanding the principles of Boolean decomposition is therefore important for everybody who is interested in circuit synthesis, partitioned PLAs, FPGAs, automatic acquisition of rules for expert systems, Machine Learning, data-base compression, Knowledge Discovery from Data Bases (KDD), etc.

In some of our explanations below we will use multiplexers. A topic of design using multiplexers is closely related to decomposition and finds applications both in VLSI design and FPGA design. They are also closely linked to Binary Decision Diagrams (BDDs), since it is straightforward to implement a BDD with multiplexers. Use of multiplexers and generalized multiplexers in this report will be applied by us as a didactic method to simplify our explanations, and make them more intuitive. Multiplexers will make also an easy link to multiple-valued, fuzzy and continuous logic, and the *if-then-else* rules of Machine Learning and KDD.

One important topic related to decomposition is the *representation* of the Switching Functions, both binary, multiple-valued, fuzzy and continuous. We will illustrate the functions with Maps (both binary and multiple-valued), Cube Calculus Arrays, Decision Diagrams, and Partitions. It is very desirable that the reader be able to make mental transitions from any of these representation models to any other. To help him to do this, we use the graphical methods. We will pay much attention in this report to graphical methods, not to expect the reader to use them in hand design, but rather to help him in creating mental analogies that may be next instrumental in designing new methods and algorithms for decomposition.

17.2 Introduction

Pattern Theory group at Avionics Laboratory of Wright Laboratories applies the so-called Ashenhurst-Curtis Decomposition of Boolean functions as a new Machine Learning paradigm [?]. Many decomposition ideas have been implemented in the programming system Flash developed by this group [?]. Flash is the system of programs created to implement and experiment with various variants of Functional Decomposition and associated Boolean transformations.

Although much research has been done recently world-wide on the decomposition of Boolean functions for logic circuit design applications, it seems to be a lack of recent monograph books or even survey papers in the area of Boolean and Multiple-Valued Logic Decomposition, despite their important applications in circuit design and Pattern Theory. This survey is an attempt to fill this gap.

The functional decomposition is an extremely wide and multifaceted topic, it is located in the very center of the area of logic synthesis. Therefore, in writing this survey, we had to be restricted to some specific point of view; some kind of special interest in the decomposition. We decided to look at the issues from the point of view of their usefulness to the goals of the Pattern Theory group at Wright Labs, [?].

17.2.1 Goals of Pattern Theory Group

The goals of the Pattern Theory work, as we understand them, are the following:

- To create practical Machine Learning algorithms that will give useful results on military data. Which means, at the minimum, functions of 30 binary-input, binary-output variables. Better, about 100 variables.

  The program must be robust across various classes of data. It must have better generalization capabilities than the well-known "Decision Tree" constructing program C4.5, as well as other recent decision-tree-based learning or decision-diagram-based learning programs. (The decision-diagram-based approaches were presented in 1994 by Ron Kohavi [?] and Arlindo Oliveira [?].) The new program is also expected to be faster than the current version of Flash.

- To make contribution to the methods of "discovery". Upgrade Flash in such a way that it would be able to "invent" new algorithms and digital structures. For instance: it took about 100 years from the invention of discrete Fourier Transform to the discovery of Fast Fourier Transform by Cooley and Tucker. We would like to be able to find similarity patterns in data in a few seconds, and create a structure of an FFT automatically. Flash can already "invent" simple circuits such as an adder, but is unable to find more complex circuits, such as a combinational sorter, comparator, maximum of k numbers, lattice realizations of symmetric functions, and butterfly architectures. One of the reasons of this inability is that at present, Flash cannot deal with multi-output functions. Second, it cannot detect symmetries or some other global properties of the underlying Boolean functions.

- To advance the state of the art of the **decomposition theory** with respect to many new aspects of decomposition that are specific to Machine Learning applications. It must result not only in a better program, but also, in original and new research results that will be appreciated by the research community, as measured by high quality conference and journal papers. The papers should make *substantial* and *fundamental* contributions to the theory of decomposition especially when applied to Machine Learning.

There are other points to be made about a special nature of the decomposition for Pattern Theory. The first very important observation is the following:

In the past, the work in the area of Ashenhurst-Curtis Decomposition (AC decomposition) was conducted for application in circuit design, [?], [?]. For problems taken from engineering practice in circuit design, when the number of input variables increases, the number of don't cares increases slightly with the number of minterms (the minterms are also called *true minterms* or *positive samples*), and the number of zeros (called also *false minterms* or *negative samples*) increases dramatically. This kind of examples can be found in the well-known benchmark sets for logic synthesis: MCNC Benchmarks, ISCAS Benchmarks, and other. These benchmarks have been created by designers from industrial companies and universities, and deal with Boolean functions from: computer, DSP and telecommunication, such as large controllers, state machines, decoding logic, arithmetic and interfaces.

This is in sharp contrast to practical functions from Machine Learning applications, where, with the increase in the number of input variables, there is only a small increase in the number of both positive and negative samples, but a dramatic increase in the number of don't cares. For instance, it is reasonable to expect that for a function of 100 input variables there will be not more than 10,000 cares.

This distinction makes completely different requirements on the decomposition programs in the both areas; "circuit design", and Machine Learning, a point that has been not yet sufficiently observed and appreciated. *This fact calls for the development of totally new approaches to synthesis, and is a very positive opportunity for the Pattern Theory group,* since if properly utilized, it would allow the group to become the leaders in the field. Instead of adapting the algorithms created for circuit design, the group should create new algorithms *from scratch*, taking into account the specifics of a problem as the main characteristics.

17.2.2 Variants of the System

We believe, that with respect to what was mentioned in previous subsections, the following separate variants of Flash system should be created.

1. Decomposition to minimize DFC. DFC stands for *Decomposed Function Cardinality.* DFC of a circuit decomposed to blocks is the sum of DFC measures of the component blocks. The DFC of a block is $n2^k$ where k is the number of block inputs and n is the number of block outputs.

2. Decomposition to find good extrapolation of don't cares. This is assumed to be the same as the "DFC minimization", if a really minimum value of DFC is found. However, any decomposition method has some kind of bias, even an obvious bias of having disjoint sets of variables in the bound and free sets. Moreover, unavoidably an additional bias must be introduced for functions with many variables, where all solutions to partial combinational problems will be approximate because of the high-order NP-hardness of the entire problem. It is then not sure that the DFC is the absolutely best measure to be used as a heuristic in the decomposition process.

3. Decomposition to recognize hardware structure. Such decomposition would require, at the very minimum, the introduction of multi-output functions and blocks with many outputs, which currently do not exist in Flash.

Of course, these variants can share most data structures and routines, but we feel that these goals would be easier to satisfy in separate "limited" versions, and not in a single "super-version", that would try to "do everything right."

17.2.3 Terminology and Sources of Information

In the most general formulation, the problem of decomposition of a single-output Boolean function F is to find a representation of F which consists of a composition of simpler Boolean functions. There are several types of decomposition, so to be precise about the subject of our presentation, we will first give respective definitions. Decomposition is the process of expressing a function of n variables as a function of functions of fewer variables. For instance, function $F(X)$ is decomposed to functions H and G, such that $F(X) = H(A, G(B))$, where A and B are proper subsets of the set of inputs $X = A \cup B$, $A \cap B = \emptyset$. When $A \cap B = \emptyset$, the decomposition is called a "disjoint decomposition", otherwise it is called a "non-disjoint decomposition". G is a single-output function. H and G are called *component functions* of F, G is called a *predecessor function* and H is called a *successor function.* H has fewer variables than F. A is called the *free set*, and B is called the *bound set.* In Ashenhurst formulation function G has one output. In Curtis formulation function G can have more than one output, but no more than the number of variables in B.

The main assumption of Flash is that it uses the Curtis type of Boolean Decomposition, in order to minimize some cost measure, the DFC. However, an important point is that there is some confusion in the technical literature, graduate textbooks, and understanding of experts in the field, of what actually constitutes the Ashenhurst/Curtis Decomposition and its many special cases and extensions.

The same type of decomposition is discussed under different names in the literature, when applied to different data representations. This in our opinion, is not correct. On the other hand, the name Ashenhurst/Curtis Decomposition has been used to practically quite different types of decomposition, linked only by their superficial references to the original papers by Ashenhurst and Curtis. An attempt has been made while preparing this survey to find the original papers, give precise and uniform definitions, and attribute the name of the decomposition to its original inventor.

Until now, the main source of information on decomposition in USA is the book by Curtis [?] that presents early American research on what is now called the Ashenhurst-Curtis decomposition. Other monograph books include: Bochmann and Steinbach [?] (in German) that presents mostly German and European research, Bibilo and Yenin [?] (in Russian) that presents only the research from the former Soviet Union, and two monographs by Luba [?, ?] (in Polish) that concentrate mainly on the approach developed at the Warsaw University of Technology.

We will define precisely the Ashenhurst decomposition [?], the Curtis decomposition [?], the Flash decomposition [?], the XBOOLE decomposition [?], the Perkowski et al (PUB) decomposition [?, ?, ?], the Luba decomposition [?], the Roth/Karp decomposition [?, ?], and others.

By defining a decomposition, our main distinguishing criterion will be the number of decompositions possible for a given type of decomposition, and not the data representation or some secondary implementation details of the algorithm.

We found that the following types of decomposition are the most interesting for the above-mentioned requirements of the Pattern Theory:

- Ashenhurst decompositions, defined in [?];
- Curtis decompositions, defined in [?];
- Roth/Karp decomposition [?, ?, ?, ?, ?];
- Sasao multiple-valued PLA decomposition [?, ?, ?];
- the PUB decomposition of Perkowski, Hoang, and Brown, [?, ?];
- Yang/Ciesielski PLA decomposition [?, ?, ?];
- Luba's Partition-Based binary [?, ?], and multiple-valued decomposition [?, ?];
- Bochmann/Posthoff/Dresig/Steinbach algebraic AND/OR/EXOR decomposition [?], so-called XBOOLE Decomposition;
- Zakrevskij/Bibilo decomposition [?, ?];
- Orthogonal decompositions of Perkowski et al. [?, ?, ?]; and
- Lai/Pedram/Vrudhula approach based on BDDs [?].

17.3 Classes of Functional Decomposition

The questions to be answered in this section are:

1. Does a decomposition which creates blocks having more outputs than inputs belong to the category of Ashenhurst/Curtis Decomposition ?

2. Is Roth/Karp decomposition [?, ?, ?] an example of AC decomposition ?

The answer to the first question is no. According to the criteria mentioned in section ?? and applied in this paper, we will not treat such a decomposition as an Ashenhurst/Curtis (AC) decomposition. Following the original papers, only a decomposition which creates blocks with the number of outputs smaller then the number of inputs will be denoted as being the AC decomposition, otherwise there is no DFC gain. A decomposition that creates blocks which have the number of outputs greater than or equal to the number of inputs is certain generalization of the AC decomposition but is not the AC decomposition. All these types of decompositions will be called the *Functional Decompositions*. Ross et al [?] defined such a general decomposition, but it is not actually realized in Flash.

The answer to the second question is yes, but the Roth/Karp decomposition uses a different data representation. Analogously, the Shen/Kellar [?, ?, ?] and Varma/Trachtenberg decompositions [?] are particular cases of the AC decomposition.

When the blocks have overlapping inputs, then every circuit structure, for instance a SOP, or a multi-level NOR network [?], can be obtained by some sort of decomposition. We will call them switching circuit decompositions, or Boolean Decompositions in case of binary functions. Some of these methods have been called "decompositions" by their authors. Again, only if the number of inputs is larger than the number of outputs of a block, and the concept of the multiplicity index is used, the decomposition will be called the AC Decomposition.

We will base our definitions on books on decomposition that have been published until now: Curtis [?], Bibilo [?], Steinbach [?], Luba [?, ?], as well as on the graduate textbooks in logic synthesis: Zvi Kohavi [?], Prather [?], Bolton [?].

We will then define the following types of decomposition:

1. Ashenhurst decompositions, defined in [?].

2. Curtis decompositions, defined in [?].

3. Roth/Karp decomposition [?, ?].

4. Sasao multiple-valued PLA decomposition [?].

5. Ross decomposition, defined in [?] (called Ashenhurst-Curtis there).

6. Perkowski/Hoang/Brown decomposition (PUB), [?, ?].

7. Yang/Ciesielski PLA decomposition [?, ?, ?].

8. Luba's Partition-Based binary and multiple-valued decomposition [?, ?].

9. Bochmann/Posthoff/Dresig/Steinbach algebraic AN/OR/EXOR decomposition, so-called XBOOLE Decomposition [?].

10. Zakrevskij/Bibilo decomposition [?, ?].

11. Orthogonal decompositions of Perkowski [?, ?, ?].

12. Pedram's decomposition using BDDs [?].

13. Murgai/Brayton's decomposition using BDDs [?, ?].

14. Jozwiak generalized decomposition [?, ?].

What was called Ashenhurst-Curtis Decomposition (ACD) by Ross et al [?] in the past, now will be called a Functional Decomposition (FD). This will be a kind of "mother of all methods", but only for the family of decompositions, which are based on what Tim Ross calls "basic decomposition condition" [?] (counting the column multiplicity). According to this criterion, the decompositions of XBOOLE [?], Yang/Ciesielski [?] or orthogonal [?], will be not included to the class of FD. However, Ashenhurst [?], Curtis [?], Ross [?], Sasao [?], Perkowski et al. [?, ?], and Zakrevskij/Bibilo [?, ?] will be examples of Functional Decompositions. Next, particular decomposition cases attributed to various authors will be derived from the FD model, by imposing various constraints on the general FD model.

Other decompositions will be discussed in much less detail, and only if we feel that discussing them may be useful for improving the decomposition model used in Flash. The question arises:

- *'If the FD is applied without any restrictions, will it generate every possible circuit? Can then every circuit structure be treated as a special case generated by certain restrictions imposed on the FD model?'*

The answer is no. For instance, SOP and ESOP circuits, or Dietmeyer-like decompositions [?] cannot be created as particular cases of this model. It is, however, perhaps possible to further expand the FD model to include these classes, but it will be not attempted in this survey report.

Concluding, the decompositions based on column multiplicity criterion will be called the Functional Decompositions (FD). If they assume in addition less outputs than inputs to every block, they will be called the Ashenhurst/Curtis Decompositions (ACD). In addition, if there is only one binary wire from the predecessor block to the successor block, the decomposition will be called the Ashehurst Decomposition. For more than one wires connecting the predecessor and successor blocks, the AC decomposition will be called the Curtis decomposition.

17.4 The Goals of this Survey.

The goals of writing this survey were the following:

1. Find what work has been already done by other researchers - not to repeat the same effort again.
2. Find what approaches to decomposition did practically work well when implemented in computer programs.
3. Find some partial ideas, heuristics, or representations that can be borrowed, integrated, and enhanced into our future improved programs.
4. Find what are the "open research problems" related to decomposition, or just what are the problems that have been not yet formulated nor solved.
5. Find how people are approaching the problems that we found most important in our research in the past:
   - representation.
   - variable partitioning.
   - coloring/covering for column compatibility.
   - column encoding.
   - subfunction reusing.
   - vacuous variable reduction.

Here, our goal will be not to present systematically all ideas that can be found in the decomposition literature. We feel that there is a lot of repetitions and re-formulations in the published papers and the authors seem to be not familiar with fundamental papers published previously. Some papers are strictly and solely related to K-map representations, that we consider obsolete, and besides, do not introduce any new ideas. Some other are only reformulations of known ideas to the cube calculus representation.

Our survey concentrates on papers and ideas that seem to be the most fruitful for the implementation of decomposing programs. Especially the implementation of the approximate programs for a large number of incompletely specified variables, but also for the exact programs that may be useful to evaluate the quality of the approximate programs.

Therefore, we devote a considerable amount of attention also to some general logic synthesis problems that will be useful to design such decomposers. These general problems include:

- representation issues, such as the use of various types of decision diagrams to represent functions,
- provable search constraints, such as finding first partial or total symmetries in input variables of the functions, and then use these symmetries to limit the search space of the bound sets of variables, without sacrificing the quality of the decomposition.
- applications of heuristic methods to reduce the search, such as variable ordering techniques for BDDs that are strictly related to the problem of variables partitioning into bound and free sets.

We have tried to give the list of literature as complete as possible. It includes some really rare internal reports (some of them, but not all, are available to us). We provide also the literature positions that we have found only indirectly - in other referenced papers (or through personal conversations). We are continuously collecting literature on decomposition, so future editions of this survey will include more references on both new publications, and existing publications that are not yet available to us.

17.5 Functional Decomposition Versus Algebraic Methods

17.5.1 Decomposition is not 'Algebraic'

Binary Decomposition is not a Boolean concept, it means, that it can be explained without using the concept of Boolean algebra and gates. What is necessary is only to describe a function (as a discrete mapping) in terms of composition of other functions. An analogous property is true for multiple-valued signals. The concept of decomposition can be explained without resorting to Post logic, Galois logic, or any other specific multiple-valued algebra to realize the functions, [?, ?]. Therefore, one can offer a very abstract and comprehensive treatment of decompositions of the binary, multiple-valued and fuzzy logic functions.

However, the above observation does not mean that some kind of 'algebraic' concepts cannot be used in the decomposition process. For instance, in binary decomposition one can use Binary Decision Diagrams [?] for data representation, and the Binary Decision Diagrams are a concept of Boolean algebra. A similar property is true for the multiple-valued data, where multiple-valued logic diagrams can be used instead of binary decision diagrams to represent the functions.

The main advantage of the functional decomposition approach of the current Flash program is that it is not using any algebraic or even Boolean methods. (Although the Flash program is now only for binary logic, the approach applied in it can be very easily expanded to the logic with multiple-valued inputs and outputs, which is exactly one of generalizations that will be presented below.)

In other synthesis methods, one assumes certain sets of blocks to realize the 'gates', for instance the AND, OR, and EXOR set [?], the NOR set [?], the AND/EXOR set [?, ?], or a multiplexer set [?]. Such approach is currently dominating in the industry (the technologies of standard cells, PLDs, sea of gates, etc.). Although the 'universal logic module' approaches, such as the 'lookup-table' approach to design with Xilinx FPGAs, assume, similarly to Flash, arbitrary lookup tables as 'gates', but, contrary to Flash, these approaches assume a limited fan-in to these tables (for instance; 4, 5 or 6 inputs are used in the older version of TRADE, [?]). This is in a clear distinction to the approach of Flash, where no restriction on the number of inputs to a logic block is assumed.

There are also some other kinds of decomposition, for instance the approach in the XBOOLE system, that assume a decomposition to certain types of Boolean gates. In the case of XBOOLE two-input AND, OR, EXOR, and an inverter. Even though we find such decompositions interesting from the Pattern Theory point of view, and these decompositions have relation to the functional decompositions, in this survey we will concentrate only on presenting the approaches that have a similar 'philosophy' to Flash.

The approach used in Flash program has the following consequences:

- It cannot be used directly to hardware realization of circuits, and some another logic minimization programs, such as Espresso, must be used to minimize and realize the blocks which are non-decomposable by the method of Flash. In the future, another 'gate-oriented' decomposers or synthesizers can be used inside Flash on equal terms with Espresso. The realization of non-decomposable blocks, seems however to be of only a secondary importance from the Machine Learning point of view.
- There exists a possibility that the cost evaluation (DFC) of large blocks is not an accurate prediction of complexity. The argument for the current approach in Flash is, however, that such blocks are rarely created by the program.
- The FD approach can be easily expanded in a natural and general way to multiple-valued, fuzzy and continuous logic. Such an extension would be practical when one assumes a software realization of arbitrary complexity blocks. However, one of the very important goals of the Pattern

Theory research is to combine the advantages of Flash for discrete data with the advantages of the ABTECH's 'abductive network' approach for continuous data [?]. ABTECH's approach is to compose a network from certain types of blocks rather than decompose. It uses particular kind of continuous blocks: third order polynomials of real-valued input variables. Such blocks are analog circuit (continuous logic) counterparts of the limited-fan-in logic gates in the logic design area. If the current approach of Flash were used for continuous data, then some additional mechanisms should be developed to map the 'non-decomposable blocks' to the polynomial blocks. There may be a concern that such a mapping will not be an optimal one. One then may ask a question: 'why not to try to compose the function from existing types of blocks?' Since, in any case, for continuous functions, ABTECH has to assume certain blocks, such as polynomials, why not to try to uniformly apply the composition methods for both multiple-valued and binary variables? Such an approach, though totally different from the one applied in Flash, definitely appears to be reasonable, and has been used in both digital and analog design [?]. Therefore, some papers on this type of decomposition, with blocks taken from certain library of blocks, will also be briefly reviewed in this survey.

- The current philosophy of Flash is that it is better to assume no knowledge on the types of gates, because assuming any kind of element would be considered a dirty patch-up, that would only add a bias. An argument against this philosophy is that, in any case, assuming some kind of bias is necessary to effectively construct some objects, or to learn some patterns. One possible approach would be to have one bias compete with another bias in order to produce the best learning algorithm. If there were no bias at all, there would be a danger that never-ever anything interesting would be found for larger data sets because the search space would be prohibitively large. The current approach of Flash is that first we want to use FD to investigate the whole space of solutions without a bias. Then we want to study how imposing various kinds of constraints would affect the quality and the structure of solutions. As already noted, other approaches are also possible, including the ones that would assume, or 'guess and check', several specific biases.

17.5.2 The Controlling of Biases in the Decomposition

There are many possible ways of dealing with the 'bias' problem.

1. One possible bias is to restrict the search space based on some heuristic principles. For instance, the algorithm created by Craig Files [?] restricts heuristically the search space of all variable partitions to a subset of the original space of all partitions. By using this particular algorithm, and not some other search-restricting algorithm, we make a certain decision. Such decision is a bias in itself.

2. Certain structural properties of the decomposed function can be assumed, which would sometimes allow to prove that searching some subspace of the entire space would be sufficient to find the exact solution in this narrower sense. For instance, this can be achieved by investigating partial and total symmetries of the input variables [?, ?].

3. A certain bias is introduced by assuming blocks with less than 5 inputs, or some other small number of inputs. This bias may be useful when a circuit realization of the decomposition is considered.

4. Any other heuristic way of constraining the search space is a bias.

5. Some other methods of restricting and controlling the decomposition search have been shown by Steinbach [?, ?] (for improving testability, and for improving power consumption - Schneider [?]), Luba [?, ?, ?], and other authors. They select one of few types of available decompositions, depending on the satisfaction of some conditions by the data.

An ideal solution would be to have many methods built into a general decomposition method. This would allow a parameterized user control of the decomposition process. This would be like a user-controlled, dynamical bias, possibly tunable or learnable with each data set.

6. An universal method which would generate specialized methods for particular types of restricted structures. In the past the following structures have been investigated:

   - various kinds of trees [?, ?],
   - Maitra cascades [?],
   - tandem networks of Butler [?],
   - partially symmetric sub-functions [?, ?].

   All these networks are particular cases of the FD decomposition.

   Although we are not aware of using such an approach within the FD context, the OR / AND / EXOR strong-weak decomposition of Steinbach [?, ?] or some special AND, OR and EXOR input decompositions described by Luba [?, ?] and Wnek [?] are good examples of similar decomposition approaches outside the AC framework.

One of the goals of this survey will be to develop an understanding of how the different constraints affect the final DFC value, the generalization abilities of the algorithm, and its learning time. While surveying the literature, we will be paying a special attention to all the issues of the bias problem in multi-level circuit synthesizing programs.

In the following sections we will list other important issues, related to the decomposition, which have guided us in selecting the papers to be covered in this survey.

17.6 Research Areas Related to Functional Decomposition Applicable to Pattern Theory

In this section we will list the research topics and options which exist in functional decomposition.

1. **Type of variables.** On what kind of data the algorithm operates. We assume that in all cases the don't cares exist in data representation.
   (a) **Binary logic** (each variable is from a set 0,1), [?, ?, ?, ?].
   (b) **Multiple-valued data** (each variable is from a finite set of integers), [?, ?, ?, ?].
   (c) **Fuzzy** (each variable is from a continuous interval [0,1]), [?, ?].
   (d) **Analog, or continuous** (each variable is an arbitrary real number), [?].
   (e) **Integer** (each variable is an arbitrary integer), introduced in [?], but not for AC decomposition.
   (f) **Mixed**. New research area.
2. **Representation of Functions.**
   (a) **Lists or Arrays of Samples**, [?, ?, ?, ?].
   (b) **Binary Cubes and Multiple-Valued Cubes** [?, ?, ?, ?, ?].
   (c) **Binary Decision Diagram (BDD)** [?]. **Ternary Binary Decision Diagrams** with extension for don't cares - three terminal nodes: $0, 1, X$ [?], the method most successful computationally until 1994.
   (d) **A pair of BDDs, one for ON-set and one for OFF-set** to specify an incompletely-specified function. A new approach, proposed below.
   (e) **Kronecker Decision Diagram (KDD) [?] with extension for don't cares,** or **a pair of KDDs, one ON-set and one for OFF-set.** These representations have not been proposed yet for decomposition.
   (f) **Other binary decision diagrams:** zero-suppressed [?], pseudo-KFDD [?], EVBDDs [?], etc. These are new approaches which have not been applied to decomposition yet (with an exception of EVBDDs in Pedram).
   (g) **New classes of Decision Diagrams that expand data beyond binary**: Orthogonal [?, ?] multiple-valued KFDD, multiple-valued pseudo KFDD, symmetric diagrams, Bryant's Binary Moment Diagrams [?], etc. These are all new approaches, none used for decomposition yet.
   (h) **Fuzzy Functions** and **Fuzzy Decision Diagrams**. These are both new approaches.
3. **Input Variable Partitioning to Free and Bound Sets.** How to find all, some or few good bound sets.
   (a) **State-space Search** (simple genetic search [?], genetic classifiers, expert systems, heuristic search [?, ?], search in planning spaces.)
   (b) **State-space Search-restricting theorems.** [?].
   (c) **State-space search-restricting heuristics.** [?].
   (d) **Applying learning to improve search performance.** [?].
   (e) **Use strong heuristic to generate just a few candidates.** [?].

(f) **BDD-based.** [?, ?].

(g) **Information theory based** - correlation of variables, order of variables [?, ?].

(h) **Symmetry based.** [?, ?].

4. **Column Minimization Problem,** [?, ?, ?, ?]. How to find the minimum number of compatible columns?

   (a) **Graph-coloring approach** for the Incompatibility Graph [?, ?].

   (b) **Set covering approach** for the Compatibility Graph [?, ?, ?].

   (c) **"Set covering with weights of rows"** for the Compatibility Graph. New approach.

   (d) **Covering/assignment approach of Bibilo et al** for the Compatibility Graph, [?]. (This method solves the encoding problem concurrently with the column minimization problem.)

   (e) **Reduced Covering approach of Luba et al** for the Compatibility Graph, [?].

5. **Encoding of groups of columns.** Columns are also called *symbolic sub-functions,* or *cofactors* with respect to variables from the bound set. Sets of columns are encoded to simplify the logic.

   (a) **Encoding based on the Fast Linear Ordering.** Wei Wan et al, [?].

   (b) **Perkowski/LocNguyen fast rule-based heuristic encoder** [?].

   (c) **Quadratic Assignment algorithms.**

   (d) **Sasao's method** (one hot code and binary code with maximum zeros [?, ?, ?]).

   (e) **Yang and Ciesielski's approach.** [?, ?].

   (f) **Devadas' approach.** [?, ?].

   (g) **Spectral Approach** (Varma and Trachtenberg [?], Karpovsky [?]).

   (h) **The encoding method of Brayton et al encoding method,** [?].

6. **Choice of Bound Partitions and Order of Creating Partitions.** Bound Partitions are selected in such a way that some kind of a special restricted structure is created, such as:

   (a) trees, tandem structures and other limited decomposition structures (Butler [?, ?], Butler and Breeding [?], and Bender [?]).

   (b) totally and partially symmetrical variables are used in predecessor blocks [?, ?].

   (c) trees of non-repeated variables [?, ?, ?].

   (d) cascade structures [?, ?, ?].

   (e) iterative networks (such as a comparator of two numbers of an arbitrary length each).

   (f) combinational systolic structures, pipelines, FTT-like butterflies, or other computer structures.

7. **Realization of Non-decomposable blocks.** Each block evaluated as non-decomposable by ACD is then realized:

   (a) With **SOP Minimizer** such as Espresso, [?].

   (b) With **ESOP Minimizer** such as Exorcism, [?].

   (c) With a **decision diagram,** such as a BDD.

   (d) With a **combination of programs,** which would select the best solution, such as the current variant of Flash which combines Exorcism and Espresso.

(e) A **transformations of a non-decomposable block** is executed to make this block decomposable, such as Wei Wan's EXOR transformation [?], or Steinbach's EXOR transformation.

8. **Sub-function Reusing.**

   (a) **Reuse During Encoding**, this is the approach of TRADE [?] (*TRAnsformation and DEcomposition*).

   (b) **Reuse by Re-entering Sub-functions as Input Variables**, this general logic synthesis approach has not been used yet in the framework of AC decomposition.

9. **Technological constraints** - Block packing (TRADE), XBOOLE [?].

10. **Removal of largest sets of vacuous variables.** Luba [?].

11. **Dealing with multiple-output functions: Luba, TRADE, Pedram et al;** [?, ?, ?].

Below we will briefly present the history of decomposition research and the modern look at the issues which are the most important for Pattern Theory.

17.7 Special Decompositions.

Because the general decomposition is difficult, one has to look to different ways of dealing with the problem. One of them may be to consider several particular cases of decomposition in more detail, and have a program that would evaluate and combine several efficient "mini-decomposers".

Applications of functional decomposition to test generation are described in [?].

Decomposition for minimization of Power Consumption is in the paper by Schneider, 1994, [?].

Paper [?] introduces new kind of decomposition that is useful for NAND and NOR gates. The paper is very similar to Perkowski's [?] paper which introduced the concept of dissected pairs for this task.

The following special structures of circuits for decomposers have been analyzed:

1. Maitra cascades,
2. Iterative circuits, pipelined logic,
3. Trees with repeated leaf variables,
4. Trees without repetitions of leaf variables,
5. Tandem networks of Butler.

17.8 Decomposition Types

We will distinguish the *decomposition type* from the *representation of data in the decomposition.*

By *a decomposition method*, we will understand the ordered quadruple of:

- *the decomposition type,* which is the basic rule of decomposing a function block to more than one function block.
- *the representation of functions* in the decomposition, which is, what kind of knowledge type and data structure is the function represented with (arrays, arrays of cubes, partitions, lists, trees, graphs, spectral coefficients, etc).
- the type of functions - the functions may be: binary, multiple-valued, fuzzy, continuous; binary functions may be completely specified or incompletely specified; multiple-valued functions may be completely specified, have standard don't cares, or have generalized don't cares. All kinds of multiple-valued (MV) and binary functions are known, but until now, only completely specified fuzzy and contiunous functions have been presented in the literature.
- the method to solve the column multiplicity problem (if the decomposition requires this step).

Therefore,

DECOMPOSITION_METHOD =
<*decomposition type, function representation, function type, column_multiplicity_solver*>

Most, if not all, approaches to decomposition have been described in a recent extensive survey on this subject by Perkowski et al, [?]. The special emphasis in this survey has been on the decomposition types, and the decomposition representations. In the present report, however, we will concentrate only on the methods and ideas that are **directly** relevant to our unified decomposition method presented below.

In 1988 Perkowski, Uong and Brown, created a decomposition method for multi-output, multi-valued functions based on graph coloring [?, ?]. The method had two variants of the decomposition type. One was based on a bound set (columns of the map) and was the same as Curtis, but was using the graph coloring instead of the set covering for the Column Multiplicity Index minimization problem in case of incompletely specified functions. Part of this variant, for binary functions, has been next programmed by Wei Wan [?]. Another variant, which we will call here the Perkowski-Uong-Brown Decomposition type (*PUB Decomposition,* for short), was based on minimizing the Row Multiplicity Index, and detecting the so-called *trivial functions.* If a cofactor can be realized in an easy way (be one of the trivial functions) then it is not counted to the Row Multiplicity Index. Although similar to Curtis, the PUB Decomposition is different, since it is based on patterns in a free set and not on patterns in the bound set. In addition, PUB Decomposition does not require to encode the bound set, as the Curtis decomposition does, and has some strong relation to multi-level design with multiplexers. Some functions decomposable in the sense of Curtis are not decomposable in PUB sense, and vice versa. Therefore we can talk about a PUB decomposition type. While for an arbitrary function F the number of all Curtis and Roth-Karp decompositions is the same, the number of PUB decompositions is different. This is an additional argument for us to distinguish the PUB decomposition type.

Concluding, the decomposition methods presented by Perkowski, Uong, Brown and Wan can be characterized as follows:

<{*disjoint Curtis, non-disjoint Curtis, disjoint PUB, non-disjoint PUB, column multiplicity based EXOR*}, {*minterms, cube arrays*},{*binary, multiple valued; complete, incomplete*}, {*clique partitioning, graph*

coloring} >

(Because the Ashenhurst decomposition is a special case of both Curtis and PUB decompositions, it is not separately mentioned here.)

A similar variant of a method has also been created by these authors for the design of PLAs with pair decoders (such PLAs were presented by T. Sasao in [?]). A generalized decomposition, as discussed in [?], but for incompletely specified functions, was also developed.

Independently, Luba et al created a complete decomposition theory based on partition calculus [?]. Luba's method is basically a classical Curtis decomposition, but it uses a new representation. A distinguishing feature of this method is an original calculus based on the representation of a function by a family of partitions over the set of cubes (or minterms). Their decomposition procedure can be also applied to incompletely specified, multiple-valued functions. In addition, it is suitable for various implementation styles, including standard PLAs, PLDs, PLAs with two-bit decoders, FPGAs, and information systems. Some stimulus for the work of Luba's group also came from the data compressing problems in Machine Learning, pattern recognition and in other areas of AI. In Machine Learning the idea of reduction of an instance space is well known: an approach to compress sets of examples, attributes and attribute-value tuples was presented in a technique called a partition triple [?]; other approaches to reduction of an instance space can also be found in [?]. Concluding, the decomposition methods presented by Luba, Rybnik, Lasocki et al can be characterized as follows:

<{*disjoint Curtis, non-disjoint Curtis*}, {*partitions of minterms, partitions of cubes*}, {*binary, multiple valued; complete, incomplete*}, {*clique partitioning,set covering,concurrent minimization and encoding*} >

In another independent effort, Wright Laboratories created a program FLASH, based on classical Ashenhurst/Curtis decomposition, but oriented towards applications in Machine Learning. The decomposition methods presented by Ross et al in WL can be characterized as follows:

<{*disjoint Curtis, non-disjoint Curtis*}, {*minterms*},{*binary, multiple valued output, continuous; complete, incomplete*}, {*clique partitioning, graph coloring*}>

Bochmann and Steinbach developed a system XBOOLE that includes a non-standard approach to decomposition, based on a new class of AND/OR/EXOR decompositions for two-input gates. The decomposition methods presented by Bochmann and Steinbach can be characterized as follows:

<{*strong AND,OR,EXOR, weak AND,OR,EXOR*}, {*cube arrays*}, {*binary; complete, incomplete*}, {*Boolean function manipulation*}>

Therefore, XBOOLE decomposition is an essentially new decomposition type.

As the result of the analysis of the above approaches, as well as other most successful decomposition methods and programs for Ashenhurst-Curtis Decomposition, we arrived at several conclusions that we next used to create a new unified theory and program. This new program, called *Generalized Unified Decomposer* **(GUD)**, has been designed with the goal to outperform all the existing approaches to those decompositions that are not technology-related but minimize certain general cost functions related to function complexity. GUD-1 is now a part of our MULTIS decomposition environment that ultimately will include FRED, TRADE (not yet connected), GUD, DEMAIN, SOP minimizers, and several variable partitioning and format changing programs.

Such a superior program was requested for the high-demand applications that are of interest to Pattern Theory group at Avionics Labs of Wright Laboratories, U.S. Air Force. Our generalization of the decomposition is for multiple-valued logic, fuzzy and continuous logic, and information systems as well. It can be also used for designs implemented with multiplexers and generalized multiplexers in these signal representations.

In a similar ways we generalized the decompositions for multiple-valued logic, fuzzy and continuous logic, and information system as well. We created variants that can be used also for designs implemented with multiplexers and generalized multiplexers. However, at present these variants are not as general as our binary decompositions.

In this report only binary and some multiple-valued decompositions will be discussed. Other multiple-valued decompositions, fuzzy decompositions and continuous decompositions will be presented in separate reports.

In this report we totally unify the methods developed in the past by the four aforementioned groups: Portland State University, Warsaw Technical University, XBOOLE group at Freiberg University in Germany, and Wright Laboratories, and we add several new methods not known from the literature. Therefore, the GUD program will also be able to execute decompositions that have never been formulated before. It will be a test-bed to develop and compare several known and new partial ideas related to decomposition. Concluding, the decomposition methods introduced by this research can be characterized as follows:

<{*disjoint Curtis, non-disjoint Curtis, disjoint PUB, non-disjont PUB, strong AND,OR,EXOR, weak AND,OR,EXOR, column multiplicity based EXOR,column multiplicity based AND, column multiplicity based OR*}, {*minterms, cube arrays, partitions of minterms, partitions of cubes*}, {*binary, multiple valued output; complete, incomplete, generalized don't cares*}, {*Boolean function manipulation, weighted clique partitioning, weighted graph coloring*}>

17.9 Search Strategies.

Several search strategies have been used in the literature to find the best partitions for both the disjoint and non-disjoint cases. For the disjoint case, the following methods are known:

1. Depth First search (we do not recall any particular papers, but this is the most natural method).
2. Breadth First search (one variant in Flash, designed by Mark Axtell).
3. Best Bound (Haomin Wu/Perkowski), and Limited Best Bound search (Haomin Wu, Craig Files).
4. A* algorithm of Artificial Intelligence (Haomin, Perkowski). This can be seen as the generalization to the Best Bound method.
5. Genetic Search (Mike Noviskey, Prof. Frenzel from Idaho [?]).
6. Search through all subsets of a set of variables, (Shen/Kellar, Varma/Trachtenberg).
7. predetermined order based on a heuristic of evaluating pairs of variables (cube based approach in presented by Wei Wan and Perkowski [?]).
8. Add and subtract variables from and to bound and free sets - Steinbach.
9. Select a single variable, based on complex block separation criteria based on partition calculus heuristics (Luba).
10. Other BDD variable selection methods (Friedman).
11. Other decision tree like variable selection methods (C4.5 - Quinlan).

17.10 Decomposition Strategy

Now, that we are familiar with several kinds of decompositions, we will combine all these decompositions to a single comphrehensive strategy.

First, we will define the concepts of *Immediate* and *Goal-Oriented* Decompositions.

17.10.1 Immediate Decompositions

Immediate Decompositions are those that are very good, happen relatively rarely, and if encountered, should be immediately executed. The Immediate Decompositions are: Strong Basic Gate Decompositions (Strong EXOR Decomposition, Strong AND Decomposition, Strong OR Decomposition), Strong Complex Gate Decompositions, Strong PUB Decompositions, and the Ashenhurst Decomposition. (The PUB decompositions will be not discussed because of lack of space). All these decompositions can be efficiently found in CDBs using cofactors and set-theoretical operations.

17.10.2 Goal-Oriented Reduction Decompositions

These decompositions are used in two cases:

1. When some already realized function F_G is close to the function F to be decomposed (we mean by this a high correlation of variables F_G and F in CDB). This function is then treated as a goal function F_G.

2. When some of the previously attempted (Immediate or Basic) decompositions of F was "nearly possible", which means, many columns had patterns corresponding to the given type of decomposition. In such case, the method from [?] is used to create the goal function F_G and select the Reduction_Type_Operator.

In both above cases, the procedure Reduction is next called, to execute the reduction, possibly also the decomposition, and put all newly created subfunctions to stack $OPEN$ of functions to realize.

Algorithm 17.10.1

Procedure *Reduction*(F, F_G, Reduction_Type_Operator).
Function F_G is a goal function, F is the decomposed function. F_C is the correcting function.

1. If Reduction_Type_Operator = 'OR'

   then $\mathrm{ON}(F_C) := \mathrm{ON}(F) \oplus \mathrm{OFF}(F_G)$; $\mathrm{OFF}(F_C) := \mathrm{OFF}(F)$.

   If Reduction_Type_Operator = 'AND'

   then $\mathrm{ON}(F_C) := \mathrm{ON}(F)$; $\mathrm{OFF}(F_C) := \mathrm{ON}(F_G) \oplus \mathrm{OFF}(F)$.

   If Reduction_Type_Operator = 'EXOR'

   then $\mathrm{ON}(F_C) := \mathrm{ON}(F_G) \wedge \mathrm{OFF}(F) \vee \mathrm{OFF}(F_G) \wedge \mathrm{ON}(F)$;

   $\mathrm{OFF}(F_C) := \mathrm{ON}(F_G) \wedge \mathrm{ON}(F) \vee \mathrm{OFF}(F_G) \wedge \mathrm{OFF}(F)$.

2. If F_G was a decomposable function, execute its corresponding decomposition and put the correcting function F_C to $OPEN$ stack. Otherwise, put F_C to $OPEN$ stack.

3. Put expression ($F := F_G$ Reduction_Type_Operator F_D) to $DONE$ stack, that collects pieces of the entire circuit for solution.

17.10.3 Last-Resort Methods

If all the above methods fail to give a good decomposition, the so-called "Last-Resort methods" are used. There are two types of Last-Resort design methods: *Last-Resort Decompositions*, and *Few-level-circuit Syntheses*. A decomposition for F creates the function: $F = F_G$ OPERATOR F_C, where OPERATOR $\in$ {AND,OR,EXOR}, and puts F_C and F_G to $OPEN$ stack. A few-level-circuits synthesis realizes F completely in a few level circuit.

The Last-Resort Decompositions include Weak AND and Weak OR Decompositions, as introduced in section 3 and section 21. Other weak decompositions include Weak PUB Decompositions from [?, ?]. Although these decompositions occur quite often, there exist still functions, for which none of these weak decompositions exists. (The weak EXOR decomposition from [?] exists for all functions, but we do not use it in GUD-2).

The few-level-circuits synthesis methods include the Sum-of-Products (SOP) synthesis, and the Product of Sums (POS) synthesis. Greedy algorithms are used to create candidate solutions of low complexity. All these methods are always applicable, but are not actually applied, if the circuit realization is too complex.

Other weak decompositions are similar to "multiplexer-like" or "Shannon-like" decompositions from [?, ?, ?]. These decompositions are really the "absolutely last resort" strategies, when no better methods were found. Such decompositions can be always found, but hopefully they are very rarely used in our overall top-down decomposition strategy.

17.10.4 The entire comprehensive search strategy

The comprehensive Decomposition strategy, Algorithm ??, includes all above partial decomposition schemes among its special cases. It is based on the following principles:

1. Several different fast and greedy synthesis methods are better for a very strongly unspecified function than a single method of high complexity. Try simple cases first. If several various previous attempts fail then try more complex circuit structures. This is a general ML philosophy.

2. Use DFC to measure the costs of partial solutions and be thus able to compare them.

3. The user can control the algorithm using several parameters.

Algorithm ?? uses three stacks:

1. The OPEN stack, which stores the sub-functions to be realized, from which the one evaluated as the easiest to realize is selected for the realization first.

2. The EVAL stack, which stores the decomposition attempts, in order to compare them one with other and select the best decomposition. The executed part of the decomposition is put to DONE stack, and new created by it subfunctions to be further decomposed are put to OPEN stack.

3. The DONE stack, which stores the completely realized subfunctions. These functions can be re-used during reduction decompositons. Finally, DONE is used to reconstruct the entire circuit when the OPEN stack becomes empty.

In making decisions, the following parameters of F_i are taken into account: number of true minterms, number of false minterms, number of true cubes, number of false cubes, sets A, B, C together with best patterns for them. number of input variables, % of ON Pattern columns, % of OFF Pattern columns, % of DC Pattern columns, % of $F/\overline{F}$ Pattern columns, % of Approximate ON Pattern columns, % of Approximate OFF Pattern columns, % of Approximate DC Pattern columns, % of Approximate $F/\overline{F}$ Pattern columns, cost parameters, distance parameters, number_of_bound_sets parameters, and other.

Algorithm 17.10.2

1. Put F to OPEN.
2. Take the easiest to realize function from OPEN. Call it FT.
3. Using $PAR1$ number of different sets A, B, C try Immediate Decompositions to FT in this order: OR, AND, EXOR, Complex_Gates, Ashenhurst.
   a) If the decomposition exists, execute it for FT, using stacks $OPEN$ and $DONE$.
   b) If there exist some close function F_G (of distance smaller than $DIST1$) to FT in DONE, then call Reduction(FT, F_G, Reduction_Type_Operator), to reduce function FT to F_G.
   c) If there exist a decomposition of function F_G, of distance $DIS2$ from FT then call Reduction(FT, F_G, Reduction_Type_Operator) to reduce function FT to F_G, and execute decomposition of F_G.
4. Using $PAR2$ number of different sets A, B, C, try Basic Decompositions in this order: Curtis ($\mu = PAR3$), PUB ($\mu = PAR4$).
5. If none of the above worked, and good weak patterns have been found in the previous stages, execute respective Weak Decompositions.
6. If none of the above worked, try $ESOP$, SOP, POS and $EXOR^2\_SOP$. Execute one that produces a SOLUTION with cost below $LAST\_COST$.
7. Execute Shannon Decomposition of FT.
8. If OPEN $= \phi$ then return the SOLUTION else go to 2 .

17.11 Other Applications of CDBs

Obviously, since CDB is a general-purpose representation of Boolean and multiple-valued functions, all logic synthesis and state machine algorithms can be programmed in them. Below we will list only those problems that in our opinion will especially benefit from this new representation.

17.11.1 General Decompositions of Multiple-Valued-Output Functions with Standard Don't Cares

The method for Multiple-Valued-Output functions with standard don't cares [?] is very similar to the one presented in previous sections.

A first variant is for the general-purpose mv logic, realized using multiple-valued Lookup Tables. This variant uses only mv counterparts of Ashenhurst, and Curtis decompositions from Algorithm 3. Finding patterns is done analogously, the only difference is that patterns are multi-valued and higher values of multiplicity indices are used. The standard procedure is followed, with the exception that the encoding algorithms are not used and a standard k-ary encoding of internal signals is used for k-valued logic.

A second variant assumes mv logic realized with multiple-valued Lookup Tables, MINIMUM, and MAXIMUM gates. This variant uses mv counterparts of AND, OR decompositions, OR/AND, AND/OR, OR/EXOR, other complex gate decompositions, Ashenhurst and Curtis Decompositions.

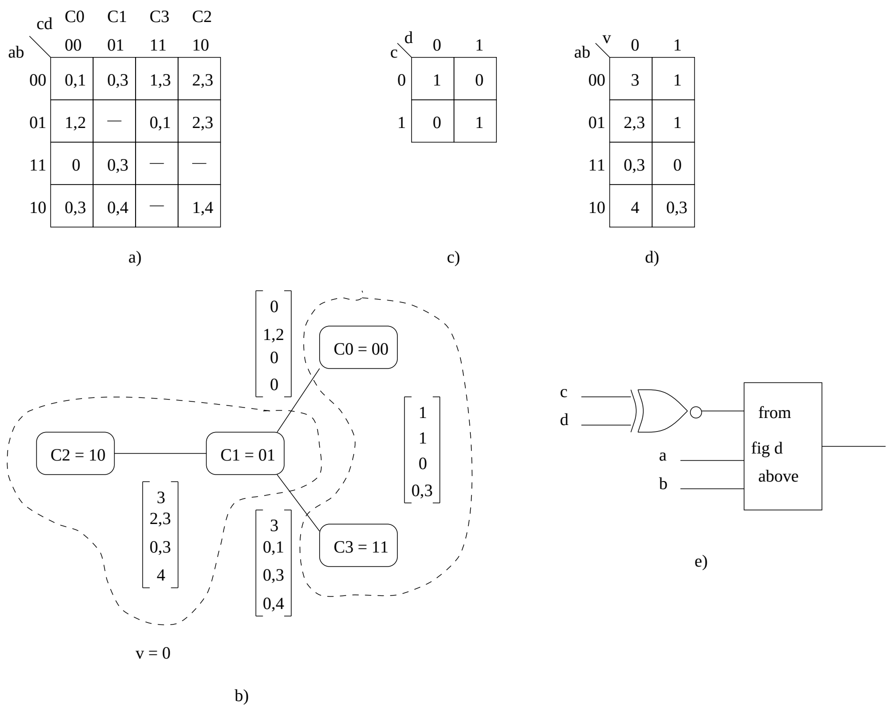


Figure 17.2: A Decomposition of a function with 4 binary variables, a 5-valued output variable and generalized don't cares from Table 1

17.11.2 Curtis-like Decomposition of Multiple-Valued-Output Functions with Generalized Don't Cares

Curtis Decomposition has been also generalized to Multiple-Valued-Output functions with Generalized Don't Cares [?]. When two columns (cofactors), f and g, are compared for compatibility in binary logic, the condition is: "for every two corresponding cells of these columns (in the same row of RVM), the cells must be the same, or one cell is a don't care." In case of generalized don't cares, this condition is generalized to: "for **every** row i of the RVM the set-intersection of values of the cells $C_i(f)$ and $C_i(g)$ of columns f and g of RVM must be non-empty, i.e.: $\forall\ i,\ C_i(f) \cap C_i(g) \neq \phi$." In such case, columns f and g are combined, and every combined cell $C_i(fg) := C_i(f) \cap C_i(g)$. Next, the Compatibility Graph is created with nodes for columns, and edges for pairs of compatible nodes. However, standard maximum cliques cannot be used, since, contrary to the standard column compatibility, it can happen that column C_1 is compatible with column C_2, column C_2 is compatible with column C_3, and column C_3 is compatible with column C_1, but columns C_1, C_2, and C_3 are not compatible all together as a set. Therefore, the cliques must be additionally checked for group compatibility [?]. Next, the standard procedure is followed, with the exception that the encoding algorithms are not used and an arbitrary k-ary encoding is used for k-valued logic.

Example 17.11.1

Given is a function with 4 binary variables and a 5-valued output variable from the map in Figure ??a (illustrated earlier in Table 1). Bound set is $\{c,d\}$. A don't care symbol, "-", stands for set of values $\{0,1,2,3,4\}$. Every cell that includes a set of values is a generalized don't care. The Column Compatibility Graph is presented in Figure ??b. The nodes represent columns from the map. A column near the edge between nodes C_i and C_j represents the combined column C_{ij}. As we see, nodes $C_0 = 00$ and

C_2 = 10 are not compatible, since for instance {0,1} $\cap$ {2,3} = ϕ. Although nodes C_0, C_1 and C_3 are pairwise compatible, the maximum clique from nodes C_0, C_1 and C_3 cannot be used, since $C_{01} \cap C_{03} \cap C_{13} = \phi$. The solution includes then the cliques {C_0,C_3} and {C_1, C_2}. The maps of functions G and H are shown in Figures ??c,d, respectively, and the decomposed circuit in Figure ??e. See [?] for applications to Attributes Reduction problem.

17.12 Applications to Machine Learning.

Recently, there has been an increasing interest in applying methods developed in design automation to other fields (see DAC'94 and Euro-DAC'94 panel discussions). Amazingly, the techniques developed in the last 15 years by the design automation community have been so universal and powerful, that they are also increasingly used in areas outside circuit design. For instance, they are now used in automatic theorem proving, robotics, industrial operations scheduling, stock market prediction, genetics research, and many others. It is then quite probable, that the design automation methods will lead to breakthroughs in these other fields.

One of the sub-areas of design automation that can find numerous external applications is logic synthesis. Unfortunately, the potential of logic synthesis for external applications is still less appreciated (by both the CAD and Machine Learning communities, and the general research and industrial circles) than the potentials of placement, routing, scheduling, placement, simulation, and database techniques of design automation.

Until very recently, the logic synthesis discipline formulated efficient methods and algorithms to minimize and optimize only the hardware of digital computers and other digital circuits. This goal was achieved very successfully, and the logic synthesis techniques developed in universities and industry in the last 15 years became one of the sources of the success of Design Automation in creation of such products as Intel's Pentium microprocessor. In the last few years, however, one can observe some increased trend to apply these methods also in image processing, machine learning, knowledge discovery, knowledge acquisition, data-base optimization, AI, image coding, automatic theorem proving and verification of software and hardware.

For instance, the ideas from logic synthesis and machine learning (ML) are starting to converge [?] [?]. The goals are similar. In Knowledge Discovery and Databases (KDD) and ML, at the core of every problem is how to effectively generalize concepts from data [?]. In the circuit design community, the end product is the realization of a minimum complexity circuit with respect to the number of gates, depth, inputs, literals, or product terms. The problems are analogous. In KDD, we have a database with fields and records. A set of records defines the concept. In logic synthesis, the fields of the records are the binary variables (inputs) of the circuit and each record is a specification of a minterm (true or false). The entire set of these records defines the circuit.

This part of the report is related to the use of the decomposer in ML and KDD applications. We will examine the performance of the decomposer on typical machine learning problems and compare the results with a standard decision tree learner, C4.5, with a multi-variant functional decomposer AFD, with Espresso, and with Exorcism. We will look at their performance on several small, noise free, binary functions without any "missing values" (i.e. completely specified functions, using circuit design terminology). We will also explore their performance on the heart disease database from the University of California-Irvine[1] (UCI). The comparison will serve to highlight the strengths and the shortcomings of all approaches, on a well-known and realistic problem. In this part, we will not present algorithms, but rather to demonstrate the applicability of logic synthesis, and specifically EXOR-based synthesis, to KDD and ML. All four algorithms share the common goal of a consistent, minimal complexity solution. All four differ in their method to find it.

17.13 The Basic Research Ideas of the PTG.

The Pattern Theory Group (PTG) in the Avionics Directorate at Wright Laboratory develops new system-level concepts for military applications, mostly based on image processing and machine learning technologies. Since 1988, the PTG developed a radically new approach to machine learning, where by "machine learning" we understand any method of teaching a computer to recognize any kind of patterns. Specifically, we are examining Supervised Classification Learning paradigms. The machine

[1] At present, the point of contact for this database is David W. Aha (aha@ics.uci.edu) and it is important to the authors that he is recognized here for his efforts to maintain it.

learning technologies most influential in military applications until now have been the neural nets and the bayesian methods. If successfully completed, the new approach of PTG will allow both automatic learning of any kind of images, and automatic creation of algorithms. Interestingly, in contrast to most of the well-known approaches, the approach of the PTG is based on logic synthesis methods: the so-called Curtis Decomposition of Boolean functions is applied here as the main component[?]. Many decomposition ideas have been implemented in the programming system FLASH (the Function Learning and Acquition Synthesis Hotbed) developed by this group[?]. This system is a Testbed for machine learning based on the logic synthesis approach. The group also compares FLASH to other logic optimizers and machine learning programs, (such as Espresso and C4.5, respectively) from the point of view of the robustness of learning[?, ?, ?, ?].

Simplifying, the main difference of the logic approach and the previous approaches to machine learning is that in these previous methods, the recognizing network had some assumed structure which was next only "tuned" by the learning process (for instance, by decreasing and increasing the numerical values of some coefficients). Thus, creating a new structure is acomplished by setting some coefficients to zero. All "new" structures are in a sense "hidden" in the assumed mother-structure. Also the type of an element, such as a formal neuron in Neural Nets, or a polynomial in Abductory Inference Mechanism (AIM[2]). [?] is decided before the learning takes place.

In contrast, in the Curtis Decomposition approach, there is no apriori assumption of structure of the learning network, nor on the type of the elements. The elements are arbitrary discrete mappings (functions). Both the structure and the elements are **calculated** in the learning process, and this process is entirely based on finding patterns in data.

The central concept of Pattern Theory is a "pattern." In Pattern Recognition and Knowledge Discovery the problems with nice representations based on "features" belong to a more general class of problems with strong "patterns." Pattern finding is, therefore, a generalization and formalization of feature extraction. The goal of the Pattern Theory is to support "automating" the pattern finding process, and to construct a representation of a function based on examples; therefore, this is a method for constructive induction. Furthermore, it constructs this representation by minimizing complexity as in Occam-based learning. The Ashenhurst Function Decomposition (AFD) method based on Curtis Decomposition implemented in FLASH is unusual in that it learns both the architecture of the combinational representation and the component functions **from** the examples.

Induction is often modeled as the process of extrapolating samples of a function. This extrapolation requires both the samples and the "inductive bias." The bias towards low complexity, as in Occam's Razor, is particularly important. There is a strong theoretical basis for Occam-based learning, see for example [?, ?]. Kolmogorov complexity was developed specifically for induction (reference [?]); however, it has been proven that its exact computation is not tractable. There have been some tractable measures of complexity used in actual implementations of Occam-based learning, such as the Abductory Inference Mechanism [?] which uses polynomial networks and C4.5 [?] which uses decision trees. However, the measures of complexity used in these applications are relatively narrow, which implies some compromise in the robustness of the learning; for example, neither of these methods would find the parity function to have low complexity even though it is highly patterned and can be easily computed. The challenge is to develop robust *and* tractable measures of complexity.

Pattern Theory [?] treats robust complexity determination as the problem of finding a pattern. Pattern theory uses Decomposed Function Cardinality (DFC), proposed by Y. S. Abu-Mostafa as a general measure of complexity [?, p.128]. DFC is based on the cardinality of a function. After all, a function is a *set* of ordered pairs and, as with any set, has a definite property in its number of elements or cardinality. DFC is defined as the sum of the cardinalities of the components of a combinational representation of a function, when that sum has been minimized. DFC is especially robust in the sense that it reflects patterns of many different kinds. Its robustness is supported by its relationship to more conventional measures of complexity, including circuit size complexity, time complexity, decision tree

[2]Trademark of AbTech Corp.

or diagram size and Kolmogorov complexity. If a problem has low complexity by any of these measures then it will also have a low DFC [?, Chapter 4]. The PTG work has concentrated on functions with binary inputs, but the concept is easily extended to continuous and multiple-valued functions [?].

The decompositions are evaluated based on the sum of the function cardinality of the decomposed partial blocks. Of course, the real DFC of f is less than or equal to this sum in any particular (usually approximate) realization. The objective of the decomposition is to search through partitions of input variables to find one that produces the smallest total DFC. In other words, the "cost" of the feature is measured in terms of DFC and that cost is minimized. Therefore, features are constructed to minimize their computational complexity. The use of DFC as the measure of complexity allows for robustness in the type of feature that can be generated.

Let us observe that all presented in previous sections methods: and in particular, the Curtis decomposition, look for certain patterns in data: in Curtis decomposition these patterns are in columns of the map corresponding to the cofactors of the bond set of variables [?]. Let us also observe that in all cases we minimize a certain cost of the circuit. Traditionally in circuit minimization one calculates the number of terms and the number of literals. In our case, we calculated additionally the total DFC as a sum of function cardinality of all non-decomposable blocks (this is an upper bound approximation of the minimum DFC). For an arbitrary non-decomposable block in Curtis Decomposition, the DFC of the block is calculated as 2^k where k is the number of inputs to the block. In "gate-based" minimizers such as Espresso and EXORCISM it is then fair to assume that a DFC of a decomposable gate (such as AND, OR or EXOR) is equal to the total DFC of a circuit equivalent to this gate, that is constructed from two-input gates. The DFC of a four input AND gate, OR gate, or EXOR gate is then $2^2 + 2^2 + 2^2 = 12$, since such gates can be decomposed to balanced trees of three two-input gates.

The AFD algorithms that we discuss all have the above in common. They differ in the number of partitions explored. The simplest method chooses one partition at random. It then tries to merge columns, encode new states, and then pick another partition at random from this new state until the one it picks does not decompose to a smaller number of unique columns. We will call this version of AFD "lotto." The reader will note this is not even a hill-climbing algorithm because only one partition at a given level is explored.

A more complex algorithm looks at all partitions at a given level and 40 random partitions one level deeper. Essentially, this is a 2-ply look ahead search where we only explore 40 grand children. It chooses a particular partition over another based on the function's DFC.

17.14 Summary of DFC Measurements and Applications

A number of experiments have been conducted to assess the generality of DFC across different problem domains (see [?, ?, ?]. The DFC of over 800 non-randomly generated functions was measured, including many *classes* of functions (numeric, symbolic, chaotic, string-based, graph-based, images and files). Roughly 98 percent of the non-randomly generated functions had low DFC (versus less than 1 percent for random functions). The 2 percent that did not decompose were the more complex of the non-randomly generated functions rather than some class of low complexity that AFD could not deal with. It is important to note that when AFD says the DFC is low, which it did some 800 times, it also provides an algorithm (or a description of the pattern or features found). AFD found **the** classical algorithms for a number of functions. Each of these algorithms are represented in a combinational form that includes intermediate states. These intermediate states are features in the sense of concentrating information.

There is also high correlation between DFC and a person's judgment of the strength of a pattern in an image, the degree of compression by a data compression program, and the Lyapunov exponents of logistic functions. This shows DFC's ability to reflect patterns within each domain, despite their different nature.

In learning experiments, AFD has been compared to back-propagation trained Neural Networks (NN), AIM, C4.5, and standard logic minimization tools. These comparisons used a broad range of function types (from the 800 mentioned above). For each of the test functions, AFD performed near the best of any method, while the other methods generalized well on some functions but not on others.

In the context of ML, when one talks about "noise," it is assumed that he is refering to the situation where you have some training data classified correctly as a "1" or a "0" but then "noise" flips that bit to the incorrect entry. Another situation is that the value of some minterm is or becomes unknown - a "don't care". This would be refered to as "unknown" in the ML community. "Noise" and "unknowns" are related - ML techniques can be used to restore noisy images [?]. Of course, we do not know which pixels (or feature values) are "noisy" - so we treat those that are most suspicious as don't cares.

The more robust generalization performance of AFD, including dealing with noise and unknown data, is a reflection of its robust measure of complexity, DFC. It is also an indication that the useful inductive bias of these various standard methods results from their parsimonious tendencies rather than their particular representation schemes (be they thresholded weighted sums, polynomials, decision trees or Boolean operators).

17.15 Towards Improved Approaches to Logic Minimizers for Machine Learning

Although the AFD approach of FLASH gives very robust results, it is slow, which essentially restricts its practicality to 20 or even less, variables. It is, therefore, important, to be able to compare the FLASH decompositional logic approach to other logic approaches that use the same DFC measure, but introduce some restricted bias resulting from the assumed network's structure. Such approaches are then faster and can be used for larger variable functions.

In this respect, the well-known circuit minimizer Espresso, the standard machine-learning program C4.5, and ESOP minimizer EXORCISM will tested together with our decomposer. These programs have the following structure/gate-type biases: Espresso assumes a two-level AND-OR network, EXORCISM assumes a two-level AND-EXOR network, C4.5 assumes an ordered tree. (The input variables can be multiple-valued).

The questions arise:

- How much of the network's simplicity is lost by assuming these structures?
- How much is gained in the speed of the program with respect to a bias-free decomposer? Is this speedup worth an increased DFC and thus a more limited extrapolation capability?
- Is the method with a biased structure still robust enough for practical applications?

Other important question that must be faced with while developing improved minimizers for machine learning applications is the following:

- What are the reasons that machine learning using logic synthesis is not exactly the same as circuit design using logic synthesis?

This question is very important practically. Improving the performance of the FLASH system orders of magnitude without sacrificing much of its robustness (DFC) is required for making it useful for such important military applications as High Resolution Radar, for example.

The data (switching functions) used in learning and algorithm design applications by the PT group are *arbitrary switching functions*. Thus, the standard and generally applicable minimization procedures of "logic synthesis" can be applied. An extremely important observation is that these functions have *quite different properties* than the data taken from industrial companies on which the programs are tested in the "logic synthesis" community (MCNC benchmarks).

In theory, the algorithm should work well on any type of data. However, since all practical network minimization problems are NP-hard, all practical algorithms, by neccessity, are heuristic in nature. Thus, they are very dependent on the type of data. Taking into account the data characteristics (such as closeness to unate Boolean functions) was, in principle, the main reason of the commercial success of two-level logic minimizers in circuit-design applications.

What is it that distinguishes the machine learning data from the circuit design data? Our preliminary answer is the following:

1. ML problems have an extremely high percent of don't cares (Don't cares are combination of argument value for which the function value is not specified.) The missing data can be represented as don't cares.

2. Arguments (variables) in ML problems are naturally multiple-valued, which means that they take any discrete number of values and not only true (1) or false (0). For instance, the size can be large (2), medium (1) or small (0).

3. The ML problems have some kind of "counting properties" (one counts the numbers of repetitions of some patterns, like the well-known problem of learning even/odd properties in perceptrons).

4. ML problems have various kinds of symmetries and generalized symmetries.

Decomposer is able to find a pattern of EXOR (parity) or similar functions, even when it is corrupted by "unknowns". This is a difficult problem in machine learning. To explain this case on an example, let us assume that we recognize the even/odd parity function. For a completely specified function and a relatively small (less than 16) number of variables, the AFD minimizer finds the exact minimal result (EXOR of input variables) quite fast. When we add some "unknowingness" to this function by replacing some ones and zeros with "don't cares", we should still be able to find the EXOR of inputs solution, since the underlying principle function did not change, only its pattern has been corrupted, "hidden" by the unknown values. This seemed to work, but when the percentage of don't cares increases and the number of variables inverases, the method yields poor results. First it ceases to recognize the "EXOR of variables" pattern, and second, on 96-variable functions, it finds no EXOR's at all and looses track of any patterns (so does a human on this case).

Another positive property of the decomposer is simultaneous classification of patterns to more than two categories (you want not only to distinguish a "friend from foe" airplane, but you want to learn its orientation, speed, etc.). In terms of logic synthesis, this property corresponds to concurrent minimization of switching functions with many outputs. Currently FLASH operates on single-output functions, but Decomposer works with multi-output functions. There are many decomposers that decompose multi-output functions, but all of them have been designed for circuit design. One needs a minimizer for **strongly unspecified, multi-valued input, multi-output functions.**

What is also missing in both "industrial circuit" and "machine learning" decomposing systems, is the **decomposition of multiple-valued input, multiple-valued-output functions.**

Why is this important? In theory, which is also the approach of the PT group, any multiple-valued variable can be encoded by a vector of binary variables. What happens, however, in learning situations is, that the learning system inferences rules that depend on the encoding of multiple-valued variables with binary variables. To give an example, if the system would infer from a large set of data that people who live close to power lines develop cancer, we would perhaps treat such "invention" with due care. If the system would, however, infer that people whose third bit of encoded distance from the line is 1 develop cancer, we would treat such inference as a "coding-related" artifact. Therefore, the best approach to the learning system would be not to use coding at all, but perform the inference on the variables that are natural for any given feature; e.g. either binary (man, woman), or multiple-valued (distance in yards).

Original Function	Actual DFC	Average Error		
		C4.5	Decomposition	Espresso
$kdd4.efr$	2	0	0	0
$kdd1.efr$	8	0.32	0	0.96
$kdd8.efr$	8	6.35	0	5.64
$kdd6.efr$	12	2.48	3.72	2.64
$kdd3.efr$	12	1.28	2.72	3.52
$kdd2.efr$	16	2.76	2.4	12.86
$kdd10.efr$	20	17.52	8.18	17.16
$kdd9.efr$	20	13.79	6.55	16.54
$kdd7.efr$	28	20.69	10.53	5.69
$kdd5.efr$	36	10.52	11.11	8.44
Average		7.57	4.52	7.345

Table 17.1: DFC and Average Error

Original Function	Actual DFC	# of Samples		
		C4.5	Decomposition	Espresso
$kdd4.efr$	2	8	7	9
$kdd1.efr$	8	31	25	40
$kdd8.efr$	8	83[3]	25	51
$kdd6.efr$	12	74	67	76
$kdd3.efr$	12	61	76	54
$kdd2.efr$	16	97	126	113
$kdd10.efr$	20	200	60	181
$kdd9.efr$	20	224	104	205
$kdd7.efr$	28	256	126	51
$kdd5.efr$	36	249	251	229
Average		128.3[4]	86.7	100.9

Table 17.2: DFC and # of Samples Needed to Learn

17.16 Small Problems

17.16.1 Overview

This section describes the first set of test functions. Each of these functions consist of 8 binary variables with a single binary output. All of the functions are void of noise and missing values. Each function was chosen as to in some way represent a "database-like" concept. It is noted to the reader that the logic synthesis approaches generalize to discrete and continuous data fields [?] as well as multi-valued outputs.

17.16.2 Experimental Design

Here, we show an abridged version of our test results with brute-force decomposition (AFD), Espresso, C4.5, and EXORCISM. We used the well-known KDD functions: kdd1.efr - kdd10.efr. The comparison of results of Decomposer, Espresso and EXORCISM on these functions is in Table 6, part 1. The functions are explained in section 10.

[3] C4.5 with pruning learned this function with 46 samples.
[4] The average with the better score for F_8 is 124.6.

The tests on the individual functions were as follows. First, each method was given a random set of data to train on ranging from 25 to 250 out of a total of 256 possible cases. Once the method was trained, the entire 256 cases were tested and the number of differences were recorded as errors. This procedure was repeated 10 times for a given sample training size in intervals of 25. Thus, the total number of runs for each function was 100 of varying sample size. None of the learning was incremental. All of the runs were independent.

For brevity, the comparison graphs were omitted in lieu of a condensed summary of each of the learning curves. For each function, the average number of errors for the entire run was recorded in Table 1. It was then possible to compare brute-force decomposition, Espresso, C4.5, and EXORCISM with these averages for a given function. Table 2 shows the number of samples necessary before the learning method obtains a concept such that in all ten separate runs, the number of errors was 0. The value at the bottom of each table is the average over all of the functions. The smaller the number here, the better the performance.

17.17 Future Research: Towards a Unified Approach to Decomposition

For the purposes of this research, we have created the following hierarchical classification of decompositions:

1. The decompositions that assume special types of blocks:
   (a) Dietmeyer's decomposition.
   (b) Wojcik's decomposition.
   (c) Davidson decomposition.
2. The decompositions that do not assume special types of blocks:
   (a) serial decompositions.
   (b) parallel decomposition.
   (c) generalized decomposition of Jozwiak.

The serial Decompositions are the following:

1. Non-Encoded Decompositions.
2. Encoded Decompositions - Curtis. (Until now only one type of encoded decompositions is known - Curtis).

The Non-Encoded Decompositions are:

1. Ashenhurst Decompositions,
2. PUB Decompositions,
3. Weak Decompositions,
4. Column-Multiplicity-Based Decompositions.

The Ashenhurst Decompositions are the following:

1. Disjoint Ashenhurst Decompositions.
2. Non-Disjoint Ashenhurst Decompositions.

The Disjoint Ashenhurst Decompositions are the following:

1. Strong Disjoint Decompositions (decompositions in which both column and row multiplicity indices are 2.)
2. General Disjoint Ashenhurst Decompositions.

The Strong Disjoint Ashenhurst Decompositions are the following:

1. Strong AND Disjoint,
2. Strong OR Disjoint,
3. Strong EXOR Disjoint,

The Non-Disjoint Ashenhurst Decompositions are the following:

1. Strong Non-Disjoint Ashenhurst Decompositions,
2. General Non-Disjoint Ashenhurst Decompositions.

The Strong Non-Disjoint Ashenhurst Decompositions are the following:

1. Strong AND Non-Disjoint Ashenhurst,
2. Strong OR Non-Disjoint Ashenhurst,
3. Strong EXOR Non-Disjoint Ashenhurst.

The PUB Decompositions are the following:

1. Disjoint PUB Decomposition.
2. Non-Disjoint PUB Decomposition.

The Weak Decompositions are the following:

1. Weak AND Decomposition,
2. Weak OR Decomposition,
3. Weak EXOR Decomposition.

The Column-Multiplicity-Based Decompositions are the following:

1. Column-Multiplicity-Based AND Decomposition.
2. Column-Multiplicity-Based OR Decomposition.
3. Column-Multiplicity-Based EXOR Decomposition.

The Encoded Decompositions are the following:

1. Disjoint Curtis Decompositions.
2. Non-disjoint Curtis Decompositions.

Fig. ?? presents the above decompositions, Fig. ?? presents Curtis Decompositions, and Fig. ?? presents the Column-Multiplicity-Based Decompositions.

Three approaches have been discussed in more detail in this report:

- The graphical approach using Karnaugh maps is very easy to understand and can be successfully used for maps of up to six variables. It relates to the most familiar concepts of elementary logic design.
- The analytical approach using the well-known calculus of arrays of cubes (cube calculus formalism is presented in more detail for instance in [?, ?]). Cube Calculus is much used in CAD algorithms and its use illustrates also the indendence of the decomposition concepts from the representation of the data. Learning of Cube Calculus is useful because of its use in encoding.
- The partition method provided an approach to large functions, especially strongly unspecified functions, and respective efficient computer algorithms, [?]. The blocks of partitions were encoded as BDDs for additional improvement of efficiency; leading to the concept of CBDDs.

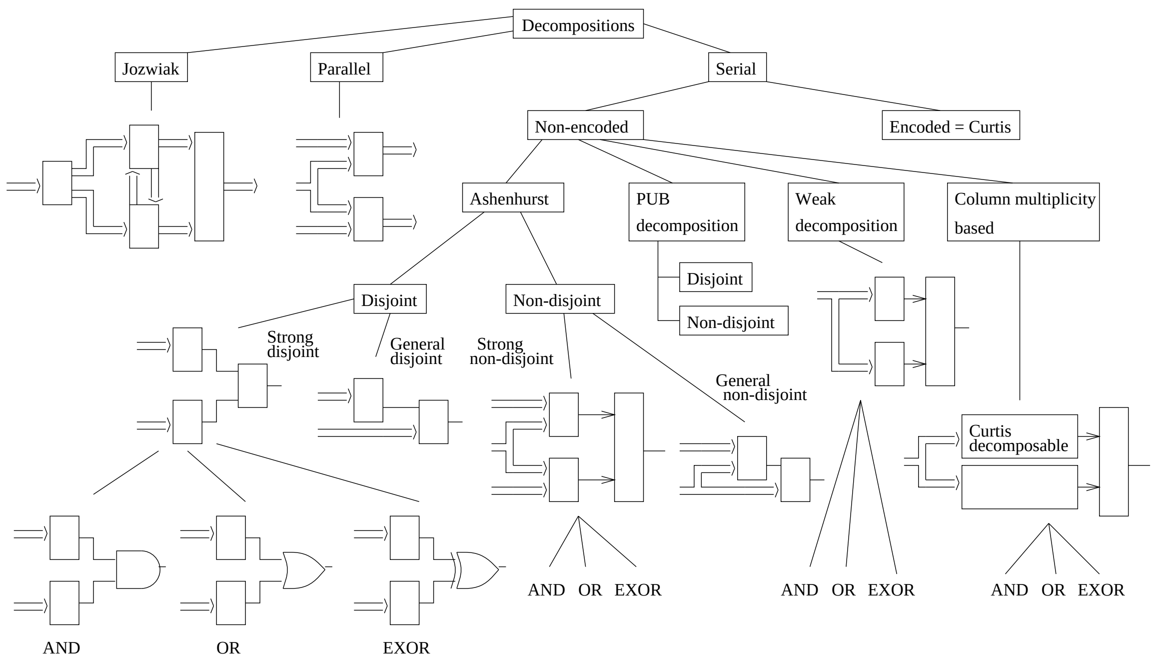


Figure 17.3: *The types of decomposition*

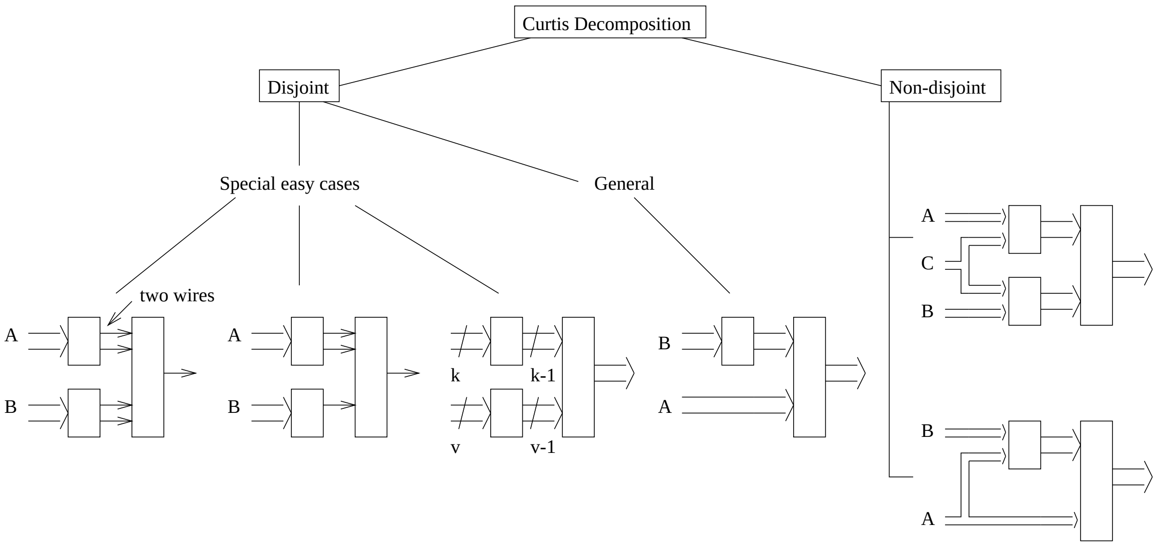


Figure 17.4: *Curtis decompositions*

Based on the analysis of the literature and our own experience, the following observations were crucial to the development of the presented system.

Representation.

1. The most important aspect of a successful Functional Decomposer is the representation of data. From all representations in the literature, the most successful programs use Binary Decision Diagrams and Edge-Valued Decision Diagrams [?], lists of disjoint ON and OFF cubes [?], and variable-partitions on ON and OFF minterms or cubes [?]. The last representation has very interesting properties. It allows to define, on incompletely specified functions, several operators, that have not been defined before. This leads to efficient variable partitioning and column compaction algorithms, as well as to formulating new kinds of decompositions, including a new method of "variable re-using". Additional advantage of this representation is that it can be easily generalized to multiple-valued logic. On the other hand, the disadvantage of this approach is the use of long lists, that are necessary to represent sets of minterms. For large examples, like an EXOR of 100 variables or very large sets of samples from Machine Learning applications, there is not enough space in computer memory to create such lists.

   Here came our idea of encoding these large data (originally represented as netlists, sets of samples, sets of cubes, or BDDs) as sets of BDDs for input and output variables. Each variable (input or output) creates a partition of elementary subfunctions (these subfunctions are for instance minterms or cubes) on the set of variable's values. For instance, for binary variable there are two values, and therefore two BDDs: the ON BDD, and the OFF BDD. Similarly, for a ternary variable there will be three BDDs; one for each value.

   All such sets, for all input and output variables, can be represented together as a "Shared" BDD. Such CDB representation is totally new in the literature, and has all advantages of the previous representations used in decomposition, and none of their respective disadvantages.

   A ready public domain software package can be used to represent "reduced, ordered, shared BDDs (possibly, with negated edges"), since the only operations that we use are the standard ones: intersection, union and complement. Similarly, any other decision diagrams such as FDDs, KFDDs, ADD, EVDDs or other can be used, whichever will be found to represent and manipulate the sets most efficiently.

2. The CDB representation allows to treat all subfunctions, newly created in the decomposition process, as the additional inputs variables. Partitions done by these functions on the function's cares can be calculated and used in the decomposition. Therefore, all new subfunctions (dependent variables) can be treated in the same way as the original initial input variables (independent variables) - the analysis of vacuous variables and variable partitioning procedures can be standardly applied to these new variables. This allows for powerful function sharing and investigating a wider space of decompositions.

3. Another advantages of some successful modern decomposers that are missing in most other are: multi-output functions, and multiple-valued functions. A careful analysis of all operations used in the respective systems, suggests that the proposed above representation will allow easy generalization to multi-output and multiple-valued functions. In essence, it is based on approaches that have been already created or generalized for these cases. Moreover, our representation is well suited not only for AC decomposition, but for many other decomposition variants [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]. Especially, it allows to take into account the partial symmetry of a function, as a most natural design bias. Moreover, this realization allows easily to represent and utilize the "generalized don't cares" for multiple-valued logic, that we invented and found useful in the decompositional approach to Machine Learning.

4. One more advantage of the representation is that it has been created to solve efficiently two of the most important problems in decomposition: selecting the best bound sets, and mimimizing the number of columns/encoding problem. The sets of minterms are kept separated in the decomposition; for each input variable, output variable, and sub-function. This representation allows for easy formulating of many partial problems associated with decomposition. Many heuristic or methodical criteria to select variables in order to achieve some minterm separation and variable correlation goals can be easily defined. All information related to variable partitioning, that is lost and repeatedly found in the classic decomposition algorithms, is retained efficiently here for subsequent use.

Search for Good Bound and Free Sets.

Several search strategies have been used in the literature to find the best partitions for both the disjoint and non-disjoint cases. A parameterized program can be created that would allow to easily create and compare several of them: depth-first search, depth-first-search-with-one-successor, breadth-first search (a variant in Flash by Mark Axtell), best-bound (Haomin Wu/Perkowski [?, ?]), limited best-bound search (Haomin Wu, Craig Files [?]), A* algorithm of Artificial Intelligence (Haomin Wu, Perkowski); searching subsets of a subset of variables (Shen/Kellar [?, ?, ?], Varma/Trachtenberg [?]), using predetermined order based on a heuristic of evaluating pairs of variables (cube based approach is presented by Wei Wan and Perkowski [?]), search by "add and subtract" variables from and to bound and free sets - Steinbach [?], search by selecting a single variable based on complex block separation criteria of partition calculus heuristics (Luba [?, ?, ?, ?]), other BDD variable selection and ordering methods [?, ?, ?, ?], and other decision tree like variable selection methods (such as in the C4.5 program by Quinlan). The search methods based on correlations of input and output variables should be investigated, since their results can be used for AC decomposition. Such ideas appeared in papers of Oliviera and Zaki [?], Mike Noviskey [?], and Grygiel and Perkowski [?].

Column Multiplicity Minimization.

In the past, the Column Multiplicity Minimization Problem was reduced either to the set covering with maximum cliques problem, [?, ?], or to the graph coloring problem. The advantage of the first approach is that very efficient algorithms for maximum clique and unate covering exist that can be adapted. The disadvantages are: the number of cliques can be prohibitely large, the size of the covering table or other data can be prohibitely large, the cliques in the solution are overlapping - so that an additional algorithm is required to remove their overlapping parts. The advantage of the graph coloring approach is that very fast approximate algorithms have been created for it, and that require little memory. The disadvantage of both approaches is that they combine in a standard way columns of don't cares with other columns, which introduces too many cares in the process. This is especially bad from the Machine Learning point of view. Another disadvantage from the Machine Learning point of view is that the columns, although compatible with respect to don't cares, would not be compatible in a completely specified function (a complete set of cares for the problem, if such are available) from which an incompletely specified function was selected.

To solve the above issues, we created four algorithms for GUD-2:

- graph coloring of care columns,
- clique partitioning of care columns,
- graph coloring of care columns with similarity constraints (weighted graph coloring),
- clique partitioning of care columns with similarity constraints (weighted clique partitioning).

Excellent algorithms exist for the first two problems [?, ?, ?] and many publications are devoted entirely to them. We have some initial algorithms and results for the other two problems that will be included in a separate forthcoming report.

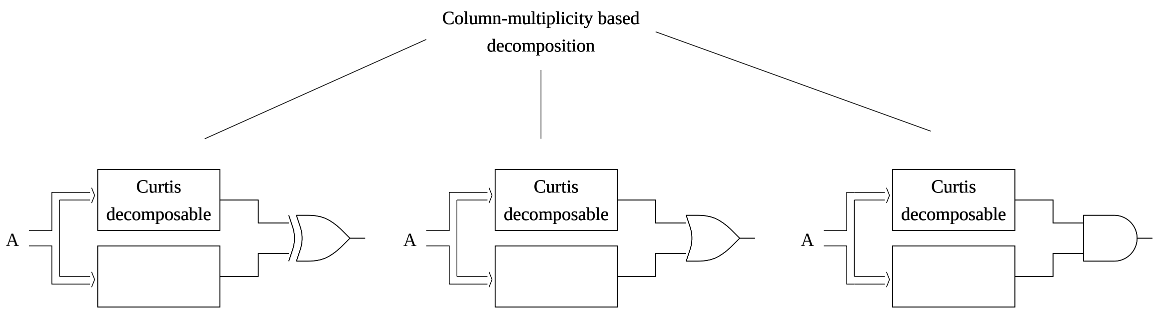


Figure 17.5: Column Multiplicity Based Curtis Decompositions

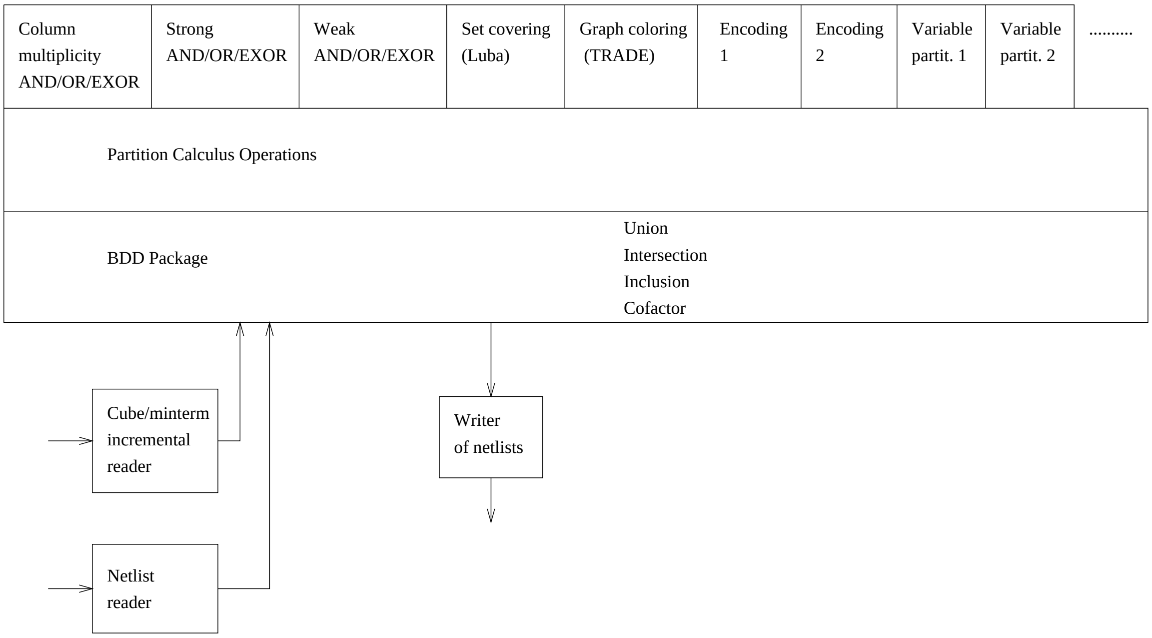


Figure 17.6: A layered construction of GUD system

We will investigate also new decomposition structures. For instance, a class of iterative circuits uses decompositions in which the number of outputs from a block is often the same or larger than the number of inputs, but the circuit has high regularity. Although technically this is not an AC decomposition, we believe it is worth investigations. Other interesting circuits are tree, cascades and tandem networks. The circuits that include blocks that have more outputs than inputs are quite common in practical design, so the respective new kinds of non-Curtis decompositions should also be studied.

Currently, a symbol in MV decomposition is encoded in binary. It would be advantageous to be able to use the variables used to encode the symbol all together as a group, while performing a decomposition. The property of grouping variables to bound sets would also be useful for decomposition of fuzzy functions, where variables x and $\overline{x}$ should always come together in a block.

DD-based methods may provide better ways of sharing sub-functions during decomposition. It seems that the problem of variable sharing is important, and although discussed in the literature on design with multiplexers, it was not investigated in the decomposition literature.

Part of the theory presented here was completed as program GUD-1, fully documented as part of MULTIS in [?]. Currently we are working on program GUD-2, presented in the entire body of the report. In future, we plan to work on program GUD-3, some ideas for which are in this section and in our several forthcoming papers.

Figure ?? presents a layered construction of GUD system. The lowest layer is created by a standard BDD, zero-supressed BDD, KFDD or any other DD package which efficiently executes set-theoretical operations. The next level from the bottom are all partition-calculus operations such as:

- creation of a partition from the (explicit or implicit) "input table".
- sum of partitions (r-partitions, covers).
- intersection of partitions (r-partitions, covers).
- inclusion test of partition in partition (r-partitions, covers).
- test if partition is a zero-partition (empty partition).
- test if partition is a one-partition (full partition).
- calculation of volumes (minterm sizes) of partitions, r-partitions and their blocks.
- printout of partitions, r-partitions and their blocks in various formats.

The next layer are all the application algorithms. They are of several types:

- decompositions (such as Curtis, PUB, strong OR, EXOR, weak AND, etc),
- realizations of combinatorial algorithms in Curtis decomposition (such as graph-coloring and set-covering).
- bound set encoding algorithms.
- variable partitioning algorithms (r-admissability, entropy, search,etc).

Chapter 18

History of Decompositions

18.1 Historical Overview of the Research on Decomposition.

Below we will list papers published every calendar year, to show the dynamics of the area, as well as the historical influences among the papers. In this section, only a very brief information about each paper is given. The papers that we consider to be the most important, or that have been the most influential ones, will be presented in more detail in the next sections.

The study of the historical literature is nevertheless fruitful, since it helps us to see the most important issues in their development perspective and allows thus to find the most prospective directions of future research.

YEAR 1854. Boole published his fundamental book, [?]. He introduced there the expansion that was later on attributed incorrectly to Shannon. Thus Boole makes the beginning of the decomposition theory, both with the respect to FD, and to the orthogonal (non-singular) expansions, because the "Shannon Expansion" is the fundament of both the decomposition and the decision diagrams used in orthogonal expansions. The so-called "Shannon Expansion" is THE most fundamental formula of Boolean logic. Boole's book is referenced in all American and Russian early research on decomposition.

YEAR 1927. Zhegalkin published his famous, but not known in Western Hemisphere paper [?], where he introduced what was next called "Reed-Muller" transform, and Reed-Muller Canonical Form. Not only was this transform next used in efficient realizations of "Ashenhurst Decompositions", but it was a base to create all orthogonal transforms and decompositions. Moreover, it was also used in new kinds of decision diagrams that can be used as an efficient representation of functions in decomposing programs.

YEAR 1949. Shannon published his paper [?], which started the history of engineering-oriented logic synthesis theory outside of the USSR. This paper influenced also the early research in USSR (Gavrilov, Povarov, Yablonsky).

YEAR 1952. Ashenhurst published early versions of his most influential works [?], that defined what we understand now by the "Ashenhurst Decomposition". This year brought also another early paper on "Ashenhurst Decomposition", published by Semon, [?].

YEAR 1953. The paper by Singer was also written, [?], a member of the same Aiken's group in Harvard, where the whole research on decomposition originated in the USA. (The Bell Laboratories is another place of early research).

Although the report by T. Singer, "The decomposition chart as a theoretical aid", from Harvard Computation Labs is little known now, the introduced by him method became next very influential. This report has been therefore reprinted as an appendix in the book of Curtis, [?].

These papers constitute the classical Ashenhurst Decomposition. In his historical paper [?], Ashenhurst stated the *fundamental disjunctive decomposition theorem.* He used the so-called *"Harvard Decomposition Charts"* in it. The charts, invented at Harvard, perhaps by either Ashenhurst himself, H. Aiken or T. Singer, were used by most of the early researchers, but they are completely equivalent to collections of the more popular Karnaugh Maps, with all possible different selections of variables in their rows and columns. The individual map from the "decomposition chart" was known since the end of the XIX-th Century under the name of the Marquand Chart. In this survey the more familiar Karnaugh maps will be used for illustration. The Ashenhurst method is based on an arrangement of the Karnaugh map with rows corresponding to the variables in the free set and columns corresponding to the variables in the bound set. A simple disjunctive decomposition exists if and only if the map has at most two distinct types of patterns of 0's and 1's in columns. It can be observed that the rows in such case have also at most two types (the function, or the function and its negation) but not counting rows of 0s and rows of 1s. The general observation is that by reducing the column multiplicity one reduces often the row multiplicity as well. All observations and ideas of Ashenhurst were next developed by other researchers.

YEAR 1954. The following papers have been written: [?], [?], [?]. The paper by G.N. Povarov in the Soviet Union originated a controversy who really invented the decomposition. It is quite possible that the Boolean Decomposition has been invented not by Ashenhurst but by Povarov, who in 1953 published (in Russian) a paper "About Functional Separability of Boolean Functions". He formulated there what is popularly known as "fundamental Ashenhurst Theorem", as well as three other theorems useful in simple decompositions. In his paper he gave references to work of Shannon, Gavrilov, and Polya. The paper was recommended by Academician Keldysh on Dec.7, 1953, and published in May of 1954. The Povarov's paper has been cited since then widely in Soviet and European literature, and he was treated in the USSR as the originator of the decomposition idea.

Also this year, Reed and Muller publish separately their fundamental papers [?], [?], being not aware that more or less the same ideas have been published by Zhegalkin in 1927.

YEAR 1956. A more advanced paper of Ashenhurst was published [?], which has been next often referenced.

YEAR 1957. R.L. Ashenhurst published his widely acclaimed paper "The decomposition of switching functions", in Proceedings of International Symposium of Theory of Switching [?]. A paper of Huffman is also published [?], which proved influential in logic design and is often referenced by early decomposition papers.

YEAR 1958. The first in the series of important papers by Curtis is published: [?]. This paper started what is now known as Curtis Decomposition.

The Curtis Decomposition is a generalization of the Ashenhurst method. However, although Curtis deals with a more general type of a decomposition, the general problem representation and the plan of looking for solutions (going through all possible partitions of input variables) are in his method still the same as in the Ashenhurst papers. Therefore, we will call this family of decompositions the Ashenhurst-Curtis, or for short, the AC decomposition.

Curtis [?] extended Ashenhurst's results to multiple decomposition when F is expressed as
$F = H(A, G_1(B), G_2(B), ..., G_k(B))$.

The use of decomposition charts (or K-maps) is of course limited to only small number of variables. Therefore, very early, researchers concentrated on the representation issue in order to allow using computers to decompose larger functions. This was done especially by the IBM group under Karp and Roth.

YEAR 1959. S.B. Akers published a paper [?], "On a theory of Boolean functions", where he introduced the concept of a Boolean Difference, one of the most important concepts in the switching circuit theory, that has been next used much in conjunction with Boolean decomposition. This paper started one approach to decomposition that was next developed mostly in Europe: Belgium (Davio, Deschamps, Thayse), USSR (Gavrilov, Zakrevskij, Bibilo), and Germany (Bochmannn, Posthoff, Dresig, Baitinger, Steinbach).

The following important papers in the mainstream of "Ashenhurst-Curtis Decomposition" have been also published: [?], [?].

YEAR 1960. J.P. Roth from IBM publishes "Minimization over Boolean Trees," [?], a paper that started the sub-area of the Ashenhurst decomposition, called the "Roth-Karp Decomposition". This is the decomposition that is most extensively investigated in the USA - all recent papers from U.C. Berkeley refer to Roth-Karp rather than to other types of decomposition, since it was considered to be the most efficient one.

Instead of looking for compatible columns as in the AC decomposition, Roth and Karp proposed to extract common sub-functions. In their seminal paper [?], they presented an algorithm for identifying a common sub-function between two functions, based on the partitioning to compatible classes. This approach has two disadvantages; first it does not apply to more than two functions; and second, it does not identify more than one shared sub-function.

In a very important step, Roth and Karp proposed a new data representation for the problem, by using the cube covers of sets ON and OFF. This was one of the first ideas and applications of cube calculus. Let us observe that this early representation, or its close derivatives, are still being used in some of the most successful programs for decomposition. Cube represenation is compact, and it is also especially well suited for strongly unspecified functions of many variables [?, ?, ?, ?].

The Roth/Karp algorithm is based on the computation of compatible classes. Let $F(x, y)$ be a Boolean function with a bound set B and a free set A, and $\mathrm{CARD}(B) = m$ and $\mathrm{CARD}(A_1) = n$, where $\mathrm{CARD}(S)$ denotes cardinality of set S. Let $B_b = B^i$ and $B_f = B^{n-i}$ where $B = \{0,1\}$. Two variables x_1 and x_2 from B_b are said to be compatible exactly, if for each y from B_f, $F(x_1, y) = F(x_2, y)$.

Roth and Karp showed that a function has simple disjunctive decomposition with respect to given bound and free sets if and only if B_f can be partitioned into $k \leq 2$ classes consisting of mutually compatible elements.

When a function is completely specified, compatibility is an equivalence relation, and k is simply the number of equivalence classes.

The relation between a *distinct column* of a K-map and a compatible class is *bijective*. The basic difference between the AC and Roth-Karp methods is in the use of different function representations. One uses K-map to represent the function, while the other uses covers of the ON-set and OFF-set of the function. When functions are incompletely specified, the detection of decomposability becomes quite complicated. For example, the compatibility in the Roth-Karp method is no longer an equivalence relation. The determination of k compatible classes requires then solving the minimum clique covering problem for the compatibility graph, which problem is known to be NP-hard.

YEAR 1961. Further papers refine the decomposition approaches of both the Aiken's Lab, [?], and the IBM group [?, ?].

YEAR 1962. H. A. Curtis published a book, [?] - "A New Approach to the Design of Switching Circuits", where he formulated a generalized theory. The book is still one of the basic references to what is called in the Western Hemisphere by the "Ashenhurst-Curtis Theory of Decomposition".

The IBM group continues their line of breakthrough papers: [?], [?].

K.K. Maitra introduces for the first time some kind of a restricted decomposed network - a cascade that was next to become known as the "Maitra Cascade".

YEAR 1963. Book by Maley [?] included early ideas on NOR decomposition, mainly intuitive graphical methods of "inhibiting" functions by other functions. These ideas have been, however, very important, since they laid a groundwork for further research of Darringer, Stoffers, Dietmeyer, and Duley [?] on applying decompositional methods to multi-level circuit design.

Further important work of Karp: [?].

YEAR 1964. The earliest Russian paper other than Povarov that we were able to find is the one by Tumanyan, [?]. We have not yet, however, had a chance to look in full detail to the very early literature from the USSR and Eastern Europe.

YEAR 1965. The following related papers have been published in USA: [?], [?], [?], [?], [?].

The papers published in USSR are: [?], [?], [?], [?].

The paper of Akers [?] is the predecessor of the Decision Diagrams approach that took over the field of logic synthesis in the 90-ties. Interestingly, the *alphabeta* diagrams invented by Akers allow to solve graphically, and for few variables, many problems that remain still of big interest nowadays. The diagrams are represented as successive partitioning of the Karnaugh-like map into smaller and smaller rectangles. Akers points out to the importance of multi-level design; the visual interpretation of such circuit constrains as fan-in, fan-out, levels of logic, decomposition and two-level forms; the reduction to smaller diagrams by Shannon expansion; the dependence on the order of variables; and the merging of diagrams to larger diagrams (compositional synthesis). Another advantage of this approach is the ease with which these constraints may be rephrased in terms of the diagrams. The explanation of Ashenhurst-Curtis decomposition using the *alphabeta* diagrams can be made quite easily. Given the diagram of a function F, select a smaller rectangle in F and remove it as a separate diagram G. Label the resulting empty rectangle in F with G and call this diagram H. Now if G and H are synthesized separately, a realization of F may be obtained by simply connecting the output of G to the input of H. This approach allows to easily find patterns of EXOR, AND, majority or other simple functions in the map. Even now, after 30 years, the Aker's paper can be very useful as a heuristic representational and teaching aid. This is also a one more way of linking Karnaugh maps and decision diagrams to the concepts of decomposition and composition of Boolean functions.

YEAR 1967. P. Deschizeaux in Grenoble, France defended one of the first theses related to decomposition [?], "Synthese de fonction Booleennes generales".

YEAR 1968. R.E. Prather published one of the first, if not the first, general logic synthesis textbooks that included the Ashenhurst decomposition [?]: "Introduction to Switching Theory: A Mathematical Approach".

The following papers have been published: [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?].

The paper by Thelliez is perhaps the first paper ever that attempts to generalize the circuit decomposition ideas to logic with more than two values - he discussed the case of ternary logic.

YEAR 1969. The following papers have been published:

[?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?].

V.Y. Shen defended a Ph.D. thesis in the Department of Electrical Engineering at Princeton, where he used for the first time the idea of Reed-Muller form for the representation of Boolean function in the decomposition process in order to test bound sets more efficiently [?]. He showed thus for the first time, that the very idea of the decomposition is not related to tabular representations of Ashenhurst and Curtis, but can be expressed in any logic representation. This idea has been next carried out very successfully by other researchers.

YEAR 1970. The following papers have been published: [?], [?], [?], [?], [?].

Kellar and Shen published [?]. Their algorithm could provide a quick way to determine if the children decompose, thus restricting the number of attempted decompositions.

The article [?] dealt with functions with more than two possible output values. It was thus the first approach to general-purpose multiple-valued function minimization (logic with an arbitrary number of values).

YEAR 1971. The following papers have been published: [?] [?], [?], [?], [?], [?], [?].

Thayse [?] and Bankowski [?] are the authors of two earliest papers (independently), that attempted at purely algebraic approach to the decomposition, based on the concept of the Boolean derivative.

YEAR 1972. The following papers have been published: [?], [?].

E. Kjelkrud published a paper [?], where he considered for the first time incompletely specified functions in more detail.

YEAR 1973. The following papers have been published: [?], [?], [?], [?], [?], [?], [?], [?].

The paper by A. Thayse, and M. Davio, [?] "Boolean differential calculus and its application to switching theory," proved with time to be a very influential one.

YEAR 1974. The following papers have been published: [?] [?], [?], [?], [?].

G.V. Bochmann and W.W. Armstrong publish a paper "Properties of Boolean Functions with a Tree Decomposition".

YEAR 1975. The following papers have been published: [?], [?], [?], [?], [?], [?], [?], [?], [?].

J.T. Butler publishes first of his papers on the number of functions realized by cascades, disjunctive networks and other special kinds of decompositions.

YEAR 1976. The following papers have been published: [?], [?], [?], [?], [?], [?], [?].

H.A. Curtis published an important paper: "Simplified decomposition of Boolean functions," which influenced the line of research represented recently by Luba et al. [?].

M.G. Karpovsky [?], published the first complete exposition of the spectral approach to logic synthesis: "Finite orthogonal series in the design of digital devices." This book presented also a new approach to decomposition, and introduced the linear (EXOR) pre- and post- processor based decompositions".

A. Kandel, [?], published the first paper that used the idea of functional decomposition to Fuzzy Logic circuits.

YEAR 1977. The following papers have been published: [?], [?], [?], [?], [?].

Stoffers wrote one of the early papers in "Gate Decomposition", [?]. He depended heavily on Karnaugh maps for realizing NOR kind of decomposition.

The method of Walczak [?] decomposes Boolean functions described by Boolean expressions and hence is limited to completely specified unitary functions and is even inferior to Roth-Karp method. We are not familiar, however, with all the references from his literature.

YEAR 1978. The following papers have been published: [?], [?], [?], [?], [?], [?], [?], [?], [?].

Butler [?] used partition matrix to design "tandem networks", as opposed to cascade and disjunctive nets. This is a new special architecture devised for realizing decomposed functions. "Tandem Network" is a one more idea of a restricted decomposed structure that can be further investigated in a practical program.

Kodadapani [?] gave interesting theorems for a, related to decomposition, topic of a design with a limited fan-out constraint.

YEAR 1979. The following papers have been published [?], [?], [?], [?], [?], [?], [?], [?].

Armstrong [?] presented applications of decomposition to pattern recognition, learning, adaptation, extrapolation. This is perhaps the first paper that linked machine learning and functional decomposition, being thus the predecessor of the Pattern Theory research.

The first papers of the German school of functional decomposition arrive: [?], [?].

YEAR 1980. The following papers have been published: [?] [?], [?], [?], [?], [?], [?], [?], [?] [?], [?], [?], [?].

Crist [?] proved that for the uniformly distributed (completely specified) functions, the non-trivial decompositions are very improbable for $n \geq 5$. He gave probabilities that a function will decompose. Much is worked out when the number of variables is small.

The first papers of the Zakrevskij/Bibilo's school in the Soviet Union have been published.

YEAR 1981. The following papers have been published: [?], [?], [?], [?], [?], [?], [?], [?].

D. Bochmann, and Ch. Posthoff publish in Germany "Binaere Dynamische Systeme," one of the first textbooks containing the German school's approach to decomposition.

YEAR 1982. The following papers have been published: [?], [?], [?], [?], [?], [?].

R.K. Brayton, and C. Mc. Mullen publish "The Decomposition and Factorization of Boolean Expressions," which becomes widely referenced by MIS II literature and recent U.S. industrial papers.

A.W. Biermann et al publish "Signature Table Systems and Learning," where several links of learning and decomposition are discussed.

YEAR 1983. The following papers have been published: [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?].

The first work of the Polish school by Luba was published, a monograph [?]. This approach uses partitions and Rough Partitions (set systems) to represent functions, and to solve all related decomposition problems.

Francioni and Kandel [?] publish first correct approach to Ashenhurst-like fuzzy logic decomposition.

YEAR 1984. The following papers have been published: [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?].

T. Sasao introduces the concept of Input Variable Assignment which was next used to create classes of non-FD decompositions.

YEAR 1985. The following papers have been published: [?], [?], [?], [?], [?], [?].

S.L. Hurst, D.M. Miller, J.C. Muzio publish one of modern textbooks which emphasize the spectral approach to logic synthesis - "Spectral Techniques in Digital Logic."

YEAR 1986. The following papers have been published: [?], [?], [?], [?], [?], [?], [?], [?], [?], [?].

Luba published his second monograph book on decomposition in Polish, [?]. R.E. Bryant publishes "Graph-Based Algorithms for Boolean Function Manipulation", one of the most important papers in logic synthesis which invigorated research in decision diagrams, and thus led also to the best recent decomposers.

J.C. Muzio and T.C. Wesselkamper publish "Multiple-Valued Switching Theory", one of very few textbooks on the topic of multiple-valued logic.

YEAR 1987. The following papers have been published: [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?].

This is the first year that the increased interest in Boolean decomposition has been reflected in the number of published papers. Several groups that are active in future years published their first papers.

A book by the Belorussian group headed by Bibilo is published, [?].

R.K. Brayton and U.C. Berkeley group publish the first journal paper on MIS II system - "MIS - a Multiple-Level Logic Minimization System."

T. Sasao published "Functional Decomposition of PLAs" where the ideas of PLA decomposition were first discussed.

Perkowski et al [?] is the full version of Perkowski/Brown 1988 paper [?], with several other extensions that were not included in [?]. It included the generalization to multi-output functions, the non-disjoint decompositions, the decompositions with one-hot encodings, the methods to minimize multiplicity index, the extension to multiple-valued input cubes, and the extension to the multiple-valued logic. Generalized don't cares and "multiple-valued relations" have been introduced here for the first time for multiple-valued logic. This paper introduced also the PUB decomposition. Some of this material has been next used in Wei Wan/Perkowski's 1992 paper, [?], but the original version has been never published. A renewed interest in decomposition causes that a modified version of this paper has been included in our recent report [?].

YEAR 1988. The following papers have been published [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?].

Paper [?] is a good reference to modular and multiple-valued decompositions.

Perkowski and Brown explained the PUB and Ashenhurst/Curtis decompositions in a simple and unified way, using the concepts of synthesis with multiplexers. A new graphical decomposition method is shown that uses Karnaugh maps instead of Ashenhurst's decomposition maps. Next, a cube-based method for strongly unspecified functions is given. The column minimization problem has been reduced here to a graph coloring problem for the first time in a published paper (this was perhaps also done in one of early Bibilo that we have no access to).

YEAR 1989. The following papers have been published: [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?].

S. Devadas et al present an early approach to Boolean decomposition from U.C. Berkeley "Boolean Decomposition in Multilevel Logic Optimization." T.T. Hwang et al present a new EXOR-like decomposition in a paper "Multi-Level Logic Synthesis Using Communication Complexity."

B.G. Kim defends his Ph.D. Dissertation "Multilevel Logic Synthesis Using Extended Array Implementation" at the University of Wisconsin-Madison which includes the ideas of using symmetry in the decomposition.

D. Varma and E.A. Trachtenberg publish a new spectral approach to decomposition in a paper "Design Automation Tools for Efficient Implementation of Logic Functions by Decomposition."

YEAR 1990. The following papers have been published: [?], [?].

The first paper of R. Murgai et al on "Logic Synthesis for Programmable Gate Arrays" has been published, where U.C. Berkeley's approach to lookup-table based FPGAs has been presented.

YEAR 1991. The following papers have been published: [?], [?].

Varma and Trachtenberg [?] discussed the problem of evaluating the complexity of a Boolean function, which is very important in decomposition. As observed already by Yablonsky in 1959, [?], an accurate evaluation of complexity may be as expensive as minimization of the function with the respective method. There exist functional decomposition procedures by Karpovsky 1977 [?] that use abstract complexity criteria based on functional approximations to logic complexity, and that arrive at optimal solutions by analytical methods.

An important paper by Yang and Ciesielski [?] was devoted to a Boolean decomposition of PLAs. Although Ashenhurst/Curtis work was never mentioned here, the paper introduced a powerful method to find variables' partitioning that is not exhaustive. This method has been applied here to PLA decomposition but it can be adapted to general functional decomposition as well. Especially, the presented by them ideas may be useful to find best encodings during the decomposition process.

A paper of Berkeley group by Murgai, Nishizaki, Shenoy, Brayton, and Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays," signified an increased interest of this group in Boolean methods of this group (while the group has been predominantly occupied with algebraic methods earlier). This was the second of their several important papers, that have been mostly devoted to Xilinx's Look-up Table model.

A book with an original approach to decomposition was published in Germany - D. Bochmann, B. Steinbach, "Logikentwurf mit XBOOLE."

YEAR 1992. The following papers have been published: [?], [?] [?], [?], [?], [?], [?].

Y.T. Lai, M. Pedram and S. Vrudhula in their report "BDD-based Logic Decomposition: Theory" presented a new approach to Curtis decomposition of multi-output incompletely specified functions based on Binary Decision Diagrams.

W. Wan and M.A. Perkowski published "A New Approach to the Decomposition of Incompletely Specified Multi-Output Function Based on Graph Coloring and Local Transformations and Its Application to FPGA Mapping" where several new heuristics for variable partitioning, encoding and column compaction were introduced, as well as EXOR decomposition of Curtis-nondecomposable functions.

One of the approaches to improve the general Ashenhurst/Curtis approach to Boolean decomposition is to apply some kind of transformation to non-decomposable blocks. Since many functions are non-decomposable, Wei Wan and Perkowski developed an approach to Ashenhurst/Curtis decomposition, that is particularly useful for new Lookup-table and fine-grain FPGAs [?]. One of the design parameters of their program, TRADE, is the number of inputs to the non-decomposable block (for instance, a CLB of Xilinx). In case of Xilinx, this parameter is 4 or 5. For Algotronix or Atmel, the parameter is set to 2, which produces two-input gates, in practice, AND, OR and EXOR gates. In TRADE, several carefully designed heuristic techniques are used to obtain high quality solutions without actually searching too large a space of candidate partitions, column groupings or column encodings. Firstly, instead of searching all possible bound sets, a cube-based heuristic to generate a limited sequence of "good" bound sets is used. Secondly, for an incompletely specified functions, a fast graph-coloring algorithm to find a minimum partitioning of columns to disjoint sets of compatible columns is applied. (Because of the constraint of 4 or 5 input CLBs, the graph for coloring is small and has not more than 32 nodes). Thirdly, the compatible columns are encoded with a fast encoding algorithm that sorts columns according to their "Hamming distance" similarities. Next, the existing CLB output functions are re-used in encodings of new symbolic functions, which decreases the total number of newly created binary functions. Finally, non-decomposable blocks are recursively transformed to EXORs of smaller functions on which the standard Curtis decomposition attempts are repeated. This approach leads to high quality decompositions on Xilinx and other FPGAs, and can be also used to create multi-level DAG circuits of AND, OR, EXOR and NOT gates (or equivalently, OR, AND, NOT and EQUIVALENCE gates). TRADE is superior on many examples (especially symmetric functions) to other FPGA minimizers.

TRADE tries to construct the network in such a way that:

- the decomposed network is technology feasible (for Xilinx architecture, the input variables of each node is up to five),
- the number of nodes in the network is as small as possible,
- the connections between the nodes are as simple as possible, and
- the path from the input to output, which is measured by the number of CLB layers, is as short as possible.

This approach generates circuits which fit better to Xilinx technology and have less CLBs, less connections and less layers. Thus the circuit is faster and easier to place and route.

The presented methods have the following assets:

- The input data to the program is an incompletely specified Boolean function described by the sets of ON and OFF cubes. It is the property of this method that the more DC cubes exist, the more efficient the method becomes. This makes our approach particularly powerful for strongly unspecified Boolean functions.
- The decomposition methods are specifically adapted to the lookup-table based FPGA architectures.
- A fast variable partitioning method is used to quickly find the good quality partitions for decomposition, avoiding the thorough test of all possible decomposition charts.
- In order to simplify the decomposed blocks, the column multiplicity minimization and the quasi-optimum don't care assignment are performed, they are achieved through a fast graph coloring algorithm. A bound set encoding algorithm is used to further simplify the decomposed blocks.
- A local transformation method is used to make the decomposition possible for all Boolean functions.
- A CLB reusing algorithm is used to decrease the number of CLBs used in the final mapped circuit.

YEAR 1993. The following papers have been published: [?], [?].

As evidenced by the amount of papers, in the last few years the increase of interest in Boolean Decomposition has been very significant.

A paper by Y.T. Lai, M. Pedram and S. Sastry, "BDD-based Decomposition of Logic Functions With Application to FPGA Synthesis," presents what is currently the published binary decomposer that is able to decompose the largest functions.

YEAR 1994. The following papers have been published: [?], [?], [?], [?], [?], [?], [?], [?], [?], [?].

T. Luba, and R. Lasocki in their paper "Decomposition of Multiple-valued Boolean Functions" introduce what is now perhaps the most efficient decomposer for multiple-valued functions. It is especially efficient for strongly incompletely specified functions. It seems that this is the best multiple-valued minimizer available, but not enough sufficient documentation is available.

B. Steinbach and M. Stoeckert in paper "Design of Fully Testable Circuits by Functional Decomposition and Implicit Test Pattern Generation" present the numerical results of Ternary-Vector-List based (a kind of cube calculus) universal decomposer, and its applications to testable circuit design.

Selvaraj in his Ph.D. Thesis defended at the Technical University of Warsaw introduced a program that combines parallel and serial decompositions.

18.2 Conclusions

Based on the surveyed literature, the following observations seem to be crucial:

1. The most important aspect of a successful Functional Decomposer is the representation of data. From all representations in the literature, the most successful programs use Binary Decision Diagrams and Edge-Valued Decision Diagrams [?], lists of disjoint ON and OFF cubes [?], and variable-partitions on ON and OFF minterms or cubes [?]. The last representation has very interesting properties. It allows to define, on incompletely specified functions, several operators, that have not been defined before. This leads to efficient variable partitioning and column compaction algorithms, as well as to formulating new kinds of decompositions, including a new method of "variable re-using". Additional advantage of this representation is its easy generalization for multiple-valued logic. On the other hand, the disadvantage of this approach is the use of large lists, that are necessary to represent the ON and OFF sets of minterms or cubes. Hence comes our idea of encoding these large sets as two BDDs: the ON BDD, and the OFF BDD. All such sets for all input and output variables, together with the newly created sub-functions, can be represented as a "Shared" BDD. Such representation is totally new in the literature, and has all advantages of the previous ones, and none of their respective disadvantages. A ready public domain software package can be used to represent "reduced, ordered, shared BDDs with negated edges", since the only operations that we use are the standard ones: intersection, union and complement.

2. The representation allows to treat all newly created sub-functions as additional inputs variables, by calculating the partitions done by these functions on ON and OFF cares. Then, all new variables can be treated in the same way as the original initial input variables - the analysis of vacuous variables and variable partitioning are applied to them. This allows for powerful function sharing and investigating a wider space of decompositions.

3. Another advantages of successful modern decomposers that are missing in Flash are: multi-output functions, and multiple-valued functions. A careful analysis of all operations used in the respective systems, suggests that the proposed above representation will allow easy generalization to multi-output and multiple-valued functions. In essence, it is based on approaches that have been already created or generalized for these cases. Moreover, our representation is well suited not only for AC decomposition, but for many other decomposition variants [?, ?], [?, ?, ?, ?], [?, ?], [?]. Especially, it allows to take into account the symmetry, as a most natural design bias. Moreover, this realization easily allows us to represent the "generalized don't cares", that we invented [?] and found useful in the decompositional approach to Machine Learning.

4. One more advantage of the representation is that it has been created to solve efficiently the two most important problem in decomposition: selecting the best bound sets, and minimizing the number of columns/encoding problem. The sets of minterms are kept separated in the decomposition; for each input variable, output variable and sub-function. This representation allows for convenient formulation of many partial problems associated with decomposition. Many heuristic or methodical criteria to select variables in order to achieve some separation goals can be easily defined. All information that is lost and repeatedly found in classic decomposition algorithms, is retained here for further use.

5. All NP-hard combinatorial problems have been reduced here to **a single one**: the weighted covering problem. Excellent algorithms exist [?, ?] to solve this problems, and many publications are devoted to it.

6. Several search strategies have been used in the literature to find the best partitions for both the disjoint and non-disjoint cases. A parameterized program can be created that would allow to easily create and compare several of them:

the depth-first search;

the depth-first-search-with-one-successor;

the breadth-first search (a variant in Flash by Mark Axtell);

the best-bound (Haomin Wu/Perkowski [?]);

the limited best-bound search (Haomin Wu, Craig Files [?]);

the A* algorithm of Artificial Intelligence (Haomin Wu, Perkowski);

the searching subsets of a subset of variables (Shen/Kellar [?, ?, ?], Varma/Trachtenberg [?]);

using the predetermined order based on a heuristic of evaluating pairs of variables (cube based approach in presented by Wei Wan and Perkowski [?]);

the search by "add and subtract" variables from and to bound and free sets - Steinbach [?, ?];

the search by selecting a single variable, based on complex block separation criteria of partition calculus heuristics (Luba [?, ?, ?, ?]);

other BDD variable selection and ordering methods [?, ?, ?];

and

other decision tree like variable selection methods (such as in the C4.5 program by Quinlan).

7. It would be very useful for future research to use a tool (from Mentor, Synopsys, or MIS II) to draw the structures created in decompositions. We would be able to find symmetries, linearities, etc. in some known functions of some known properties. This would perhaps help to find good heuristics for future decomposer versions.

8. One should investigate a relation between functional decompositions and iterative circuits. An iterative circuit can be often created by a class of AC decompositions (since the number of outputs from a block in an iterative circuit is often the same or larger than the number of inputs). The iterative circuit has, however, a high regularity, which can be used to increase the search efficiency of the decomposition.

9. Circuits that include blocks which have more outputs than inputs are quite common in practical design, so the respective new kinds of non-Curtis FD decompositions should also be interesting.

10. Currently, a symbol is encoded in binary. It would be useful to be able to use the variables from the symbol all together as a group, while performing a decomposition. The property of grouping variables to bond sets would be also useful for decomposition of fuzzy functions, where variables x and $\overline{x}$ should always come together.

11. Decision Diagram-based methods may provide better ways of sharing sub-functions during decomposition. It seems that the problem of variable sharing is important and it was not investigated in the literature.

12. We should investigate the complexity of connections - the "communication" complexity, versus the "computation" complexity calculated now, as related to various decomposition methods. Is it true that the communication complexity is close to DFC? Facts of this kind should be verified experimentally.

13. The synthesis methods based on correlations of input and output variables should be investigated, since their results can be used for AC decomposition. Such ideas appeared in papers of Oliviera and Zaki [?] and Mike Noviskey [?].

18.3 Conclusions

We introduced a new representation, CDBs, and a new general decomposition/synthesis approach for strongly unspecified multiple-valued multi-output functions. This approach can be used in several variants, and it opens several new research areas, such as: combining multi-level decomposition with such few-level synthesis methods as ESOP minimization; creating Curtis-like decomposers for strongly unspecified functions of many variables; input variable re-encoding problem to simplify DDs; using logic synthesis in Machine Learning; efficient solving of combinatorial problems (such as graph coloring); and other. We implemented GUD-1 and we are currently in the process of implementing GUD-2, the decomposition algorithm from section ??. This new representation allows also to utilize Sum-of-CDEC, TANT, ESOP, $EXOR^2\_SOP$, and other similar circuits for Machine Learning applications (forthcoming papers). The advantages and disadvantages of such circuits, versus those of the better known SOPs, trees, and Curtis decompositions in the area of Machine Learning should be investigated (see [?] for the first results). We developed also a new approach to decompose Fuzzy Logic circuits [?], it will be presented in a separate report.

GUD-2 will be implemented by expanding GUD-1. More testing results will be also available, as well as comparisons with other Machine Learning programs on larger benchmarks. For instance, the Heart Disease Problem Database. Our goal is the creation of a **practical** "Machine Learning" algorithm that will give useful results on military data. Which means, at the minimum, 30 binary input variables but more likely, about 100 multiple-valued variables. As presented above, in Machine Learning, with the increase in the number of input variables there is only a small increase in the number of both positive and negative samples, but a dramatic increase in the number of don't cares. For instance, it is reasonable to expect that for a function of 100 variables there will not be more than 10,000 cares. To be able to handle this kind of functions still remains a long-term goal of our research.

Acknowledgment

The author would like to thank Professor Robert Brayton from University of California in Berkeley, Professor Maciej Ciesielski from University in Massachussets in Amherst, and Professor Fabio Somenzi from the University of Colorado in Boulder for helping to find the respective literature. Thanks are also to Prof. Somenzi for the access to his FSM minimizer STAMINA, and to Prof. Ciesielski for the set of his synthesis tools. He would like to acknowledge Dr. Tim Ross, Mr. Mark Axtell, and Professors Malgorzata Marek-Sadowska, Tsutomu Sasao, Tadeusz Luba, Bernd Steinbach, Bernd Becker, Radomir Stankovic, and Randall Bryant for their help in finding some of the surveyed materials. He would also like to thank the Air Force Office of Scientific Research for providing the opportunity and support to do this research.

The members of the Functional Decomposition Group at PSU since 1994 until present were: Ralph Almeria, Paul Burkey, Michael Burns, Karen Dill, Craig Files, Robert Gatlin, Stanislaw Grygiel, Nick Iliev, Feng Yang, Robert Lisanke, Jing Lu, Rahul Malvi, Sanof Mohammad, Dhinesh Manoharan, Robert Price, Roger Shipman, Sridhar Srinivasan, Cynthia Stanley, Zhi Wang, Hongfei Wu, Sida Zhou, and Jin S. Zhang.

Bibliography

[1] H. Abelson, A. Ehrenfeucht, J. Fickett, J. Mycielski, "Compositional Complexity of Boolean Functions," *Discrete Applied Mathematics,* Vol. 4, No. 1, pp. 1-10, 1982.

[2] P. Abouzeid, et. al., "Multilevel Synthesis Minimizing the Routing Factor," *Proc. 27th ACM/IEEE Design Automation Conference,* pp. 365-368, June 1990.

[3] P. Abouzeid, et. al., "Lexicographical Expression of Boolean Functions for Multilevel Synthesis of High Speed Circuits," *VLSI Logic Synthesis and Design,* pp. 31-39, 1990.

[4] P. Abouzeid, et. al., "Lexicographical Factorisation Minimizing the Critical Path and the Routing Factor for Multilevel Logic," *IFIP Workshop on Logic and Architecture Synthesis,* pp. 219-228, Grenoble 1990.

[5] P. Abouzeid, B. Babba, M. Crastes de Paulet, and G. Saucier, "Input-Driven Partitioning Methods and Application to Synthesis on Table-Lookup-Based FPGAs," *IEEE Tr. on CAD* Vol. 12, No. 7, pp. 913–925.

[6] Abtech Corporation, company materials, 1994.

[7] S.B. Abugharbieh and S.C.Lee, "A Fast Algorithm for the Disjunctive Decomposition of m-Valued Functions, Part I: The Decomposition Algorithm," *Proc. 23-rd ISMVL,* pp. 118-125, 1993.

[8] S.B. Abugharbieh and S.C.Lee, "A Fast Algorithm for the Disjunctive Decomposition of m-Valued Functions, Part II: Time Complexity Analysis," *Proc. 23-rd ISMVL,* pp. 126-131, 1993.

[9] Y.S. Abu-Mostafa, *"Complexity in Information Theory",* Springer-Verlag, New York, 150pp, ISBN 0-387-96600-5, 1988.

[10] G.P. Agibalov, "Problemy Sinteza Cifrovych Automatov," pp. 96-100, Moscow, 1967, (in Russian).

[11] M.O. Albertson, and K.L. Collins, "Duality and perfection for edges in cliques," *J. Comb. Theory.* Ser. B., Vol. 36, No. 3., pp. 298–309, June 1984.

[12] C. Aleksander, "Notes on the Synthesis of the Form," Cambridge, MA, *Harvard Univ. Press,* 1967.

[13] I. Aleksand., *Radio El. En.* 38,28, 1969.

[14] C.M. Allen, and D.D. Givone, "A Minimization Technique for Multiple-Valued Logic Systems," *IEEE TC*, Febr. 1968, pp. 182–184.

[15] S.B. Akers, "On a Theory of Boolean Functions," *Journal of the Society for Industrial and Applied Mathematics,* 7, 4, 1959.

[16] S.B. Akers, "A Diagrammatic Approach to Multilevel Logic Synthesis," *IEEE Trans. on Comput.,* Vol. 14, pp. 174-181, April 1965.

[17] S.B. Akers, "Decompositions of Logical Functions Using Majority Decision Elements,"

[18] A. E. A. Almaini, M. E. Woodward, "An Approach to the Control Variable Selection Problem for Universal Logic Modules," *Digital Processes,* Vol. 3, pp. 189-206, 1977.

[19] A.E.A. Almaini, "Sequential Machine Implementations Using Universal Logic Modules," *IEEE Trans. on Comput.* Vol. 27, No. 10, pp. 951-960, 1978.

[20] A. Apostolico, and E. Hambrusch, "Finding maximum cliques on circular-arc graphs," *Inf. Proc. Letters*, Vol. 26., No. 4, December 1987, pp. 209–215.

[21] C. Apte, S. J. Hong, S. Prasad, and B. Rosen. Ramp: Rules abstraction for modeling prediction. *Technical report*, 1995.

[22] T. Araki, T. Kameda, "A Theory of Coteries. Mutual Exclusion in Distributed Systems," *IEEE Trans. on Parallel and Distributed Systems,* Vol. 4, No. 7, pp. 779-794, July 1993.

[23] H. Arango, J. Santos, A. Chacur, "Ternary Cyclo Decompositions," *IEEE Trans. Comput.*, Vol. C-17, No 12, pp. 1175-1176.

[24] C. Arcelli, *Kybernetik,* Vol. 13, pp. 155-?, 1973.

[25] T. Arciszewski and R.S. Michalski and J. Wnek, "Constructive induction: the key to design creativity," *Reports on Machine Learning and Inference Laboratory, MLI 95-6*, George Mason University, April, 1995.

[26] D.B. Armstrong, "A Programmed Algorithm for Assigning Internal Codes to Sequential Machines," *IRE Transactions Electr. Comp.,* Vol. EC-11, pp. 466-472, August 1962.

[27] W.W. Armstrong, and G. Godbout, "Properties of Binary Trees of Flexible Elements Useful in Pattern Recognition," *Proc. IEEE 1975 Int. Conf. Cybern. and Society*, San Francisco, pp. 447-450.

[28] W.W. Armstrong, J. Gecset, "Adaptation Algorithms for Binary Tree Networks," *IEEE Trans. on Systems, Man and Cybern.*, Vol. 9, No. 5, pp. 276-285, 1979.

[29] A.R.A. Asaad, L.A.M. Bennett, "Synthesis of Tributary Switching Networks by Spectral Means," *IEE Proc.* Vol. 137, Pt.E, No. 5, September 1990.

[30] P. Ashar, S. Devadas, and A.R. Newton, " A Unified Approach to the Decomposition and Redecomposition of Sequential Machines," *Proc. of the 27th A.C.M./I.E.E.E. Design Automation Conference,* pp. 601-606, 1990.

[31] P. Ashar, S. Devadas, and A.R. Newton, "Testability Driven Synthesis of Interacting Finite State Machines," *Proc. of the International Conference on Computer Aided Design,* pp. 273-276, 1990.

[32] R.L. Ashenhurst, "The Decomposition of Switching Functions," *Bell Laboratories Report,* No. BL-1(11), pp. 541-602, 1952.

[33] R.L. Ashenhurst, "Non-disjoint Decomposition," *Bell Laboratories Report,* No. 4, pp. 1V-1 - 1V-12, 1953.

[34] R.L. Ashenhurst, "The decomposition of switching functions," *pp. 571-603 of Curtis'62.*, 1952.

[35] R.L. Ashenhurst, "The Decomposition of Switching Functions," *Bell Telephone Labs Report,* 16, pp. III1-III72, 1956.

[36] R.L. Ashenhurst, "The Decomposition of Switching Functions," *Proceedings of International Symposium of Theory of Switching,* 1957.

[37] R.L. Ashenhurst, "The Decomposition of Switching Fuctions," *Proceedings of an International Symposium on the Theory of Switching Functions,* April 2-5, 1957, Ann. Computation Lab., Harvard Univ., Vol. 29, pp. 74-116, 1959.

[38] F.F. Atstonas, and K.I. Plukas, "A Method of Minimization of Microprograms for Electronic Digital Computers," *Avtomatika i Vycisl. Tehn,* Vol. 4., pp. 10–16, 1971.

[39] J.G. Auguston, and J. Minker, "An Analysis of Some Graph Theoretical Cluster Techniques," *JACM,* Vol. 17, No. 4, pp. 571–588, October 1970.

[40] M.J. Avedillo, J.M. Quintana, and J.L. Huertas, " State Reduction of Incompletely Specified Finite Sequential Machines," *Proc. Workshop on Logic and Architecture Synthesis,* Paris, May 1990, pp. 107-115.

[41] M.L. Axtell, "Partition Selection Algorithms: Row/Column Ratio Experiment," *Technical Report,* Veda Inc., c/o WL/AART-2, W/P AFB, OH 45433-7408, April 1993.

[42] M.L. Axtell, T.D. Ross, M.J. Noviskey, "Performance Comparison Between Occam-Based, Back-Propagation, and Abduction Learning Networks," In *IEEE Int. Conf. on Neural Network Applications in Signal Processing,* 1993.

[43] L. Babai, P. Hajnal, E. Szemeredi, G. Turan, "A Lower Bound for Read-once-only Branching Programs," *Journal of Computer and System Sciences,* Vol.35, pp. 153-162, 1987, QA76.5 .J7.

[44] B. Babba, M. Crastes, "Automatic Synthesis on Table Lookup-Based PGAs," *Proc. Euro-ASIC,* 1992.

[45] U.G. Baitinger, H.M. Lipp, D.A. Mlynski, D.A. K. Reiss, "MEGA-Ein Modulares Entwurfssystem fuer Gate-Arrays," *Fortschritt-Berichte VDI,* Reihe 9, Nr. 65, VDI-Verlag, Duesseldorf 1987, (in German).

[46] E. Balas, and C. Yu, "Finding a maximum clique in an arbitrary graph," *SIAM, J. Comput.*, Vol. 15, No. 4., pp. 1036, November 1986.

[47] E. Balas, V. Chvatal, and Nesetrily, "On the maximum weight clique problem," *Math. Oper. Res.*, Vol. 12., No. 3, pp. 522-535, August 1987.

[48] R.B. Banerji, *Pattern Recognition,* 1, 63, 1968.

[49] J. Bankowski, "Difference Theorems on Decomposition of Boolean Functions," not published manuscript, *Warsaw Technical University.* 1971, (in Polish).

[50] S.I. Baranov, V.N. Siniev, *Avtomaty i Programmirujemyje Matricy,* Minsk 1980, (in Russian).

[51] D.F. Barnard and D.F. Holman, "The Use of Decomposition Algorithm in Multilevel Design of Circuit," *Computer J.*, Vol. 11, pp. 269-276, 1968.

[52] K.A. Bartlett, R.K. Brayton, G.D. Hachtel, R.M. Jacoby, C.R. Morrison, R.L. Rudell, A.L. Sangiovanni-Vincentelli, A.R. Wang, "Multilevel Logic Minimization Using Don't Cares," *IEEE Trans. on CAD,* Vol. 7, pp. 723-740, June 1988.

[53] A.R. Barron, and R.L. Barron, "Statistical Learning Networks: A Unifying View," *Symposium on the Interface: Statistics and Computing Science,* 1988. page 12. craig.

[54] N.R. Bell, *IEEE Educat.* 11, pp. 173-?, 1968.

[55] L.A.M. Bennett, "The Application of Map-Entered Variables to the Use of Multiplexers in the Synthesis of Logic Functions," *Int. J. Electronics,* Vol. 45, No. 4, pp. 373-379, 1978.

[56] R.G. Bennetts, "An Improved Method of Prime C-Class Derivation in the State Reduction of Sequential Networks," *IEEE Trans. Computers,* Vol. C-20, pp. 229–231, 1971.

[57] R.G. Bennetts, J.L. Washington, and D.W. Lewin, "A Computer Algorithm for State-Table Reduction," *Radio Electron. Engn.* Vol. 42, pp. 513–520, 1972.

[58] E.A. Bender, and J. Butler, "Enumeration of functions realized by fanout-free networks of general multi-valued gates," *Proc. 1979 ISMVL*, pp. 94-103, May 1979.

[59] E.A. Bender, E.R. Canfield, "Fanout-Free Functions," *American Mathematical Monthly,* Vol. 88, No. 4, pp. 278-280, 1981.

[60] E.A. Bender, E.R. Canfield, "Fanout-Free Networks of Symmetric Gates: Synthesis and Canonical Forms," *preprint Dept. of Math.* University of California at San Diego, La Jolla, CA 94093.

[61] B. Becker, R. Drechsler, and R. Wechner, "On the Relation Between BDDs and FDDs," *Technical Report*, University of Frankfurt, 12, 1993.

[62] E.A. Bender, "Asymptotic Methods in Enumeration," *SIAM Rev.*, Vol. 16, pp. 485-515, Oct. 1974.

[63] C. Berge, "Theory of Graphs," Methuen, London, 1962.

[64] E. R. Laboratory. *OCTOOLS-5.2 Users guide.* University of California at Berkeley, May 1993.

[65] T. Besson, H. Bousouzou, M. Crastes, and G. Saucier, "Synthesis on Multiplexer-based Programmable Devices Using (Ordered) Binary Decision Diagrams," *Proc. EURO-ASIC,* pp. 8-13, Paris, France, June 1992.

[66] T. Besson, H. Bousouzou, M. Crastes, and G. Saucier "Synthesis on Multiplexer-based F.P.G.A. Using Binary Decision Diagrams," *Proc. of IEEE ICCD*, pp. 163-167, 1992.

[67] P.N. Bibilo, S.V. Yenin, "Vychislitelnaia Tekhnika v Mashinostroenii," ed. 1, pp. 100-108, Minsk 1979, (in Russian).

[68] P.N. Bibilo, S.V. Yenin, "Decomposition of a Boolean Function With Minimum Number of Significant Arguments of the Subfunctions," *Engineering Cybernetics,* Vol. 18, No. 3, pp. 75-81, 1980.

[69] P.N. Bibilo, S.V. Yenin, "Joint Decomposition of a System of Boolean Functions," *Engineering Cybernetics,* Vol. 18, No. 2, pp. 96-102, 1980.

[70] P.N. Bibilo, *Izv. A.N. BSSR Ser. fiz-mat nauk,* No. 4, pp.33-38, 1980, (in Russian).

[71] P.N. Bibilo, S.V. Yenin, "Avtomatizacjia Logiceskogo Proektirovanija Diskretnych Ustrojstv," ed. 2., pp. 33-46, Minsk, 1980, (in Russian).

[72] P.N. Bibilo, S.V. Yenin, *Izvestia Academy of Sciences of USSR,* Series Technical Cybernetics, No. 2., pp. 108-113, 1980, (in Russian).

[73] P.N. Bibilo, *Avtomatizacija tekhn. podgotovki proizvodstva,* pp. 86-93, Minsk 1982, (in Russian). XXXXXXX

[74] P.N. Bibilo, *Avtomatizacija Tekhniceskoj Podgotovki Proizvodstva,* pp. 40-49, Minsk 1983, (in Russian).

[75] P.N. Bibilo, *Avtomatika i Vycislitelnaya Tekhnika,* No. 1, pp. 91-92, 1984 (in Russian).

[76] P.N. Bibilo, *Proektirovanije Ustrojstv Logiceskogo Upravlenija,* pp. 106-126, Moscow 1984, (in Russian).

[77] P.N. Bibilo, S.V. Yenin, "Synthesis of Combinational Networks Using the Method of Functional Decomposition," ("Sintez Kombinacjionnych Schem Metodami Funkcijonalnoj Dekompozicji,") *Nauka i Tekhnika,* Minsk, 1987, (in Russian).

[78] A.W. Biermann, J.R.C. Fairfield, T.R. Beres, "Signature Table Systems and Learning," *IEEE Trans. on Syst. Man and Cybern.*, Vol. 12, No. 5, pp. 635-648, 1982.

[79] A.J. Billionnet, "An upper bound on the size of the largest clique in a graph," *Graph Theory,* Vol. 5., No. 2., pp. 165-169, Summer 1981.

[80]

[81] N.N. Biswas, "Identification of Totally Symmetric Boolean Functions," *IEEE Trans. on Comput.*, Vol. C-70, pp. 645-648, July 1973. Correction IEEE TC, pp. 863-864, September 1973.

[82] N.N. Biswas, " State Minimization of Incompletely Specified Sequential Machines," *IEEE Trans. on Computers,* Vol. C-23, pp. 80-84, 1974.

[83] N.N. Biswas, "Implication Trees in the Minimization of Incompletely Specified Sequential Machines," *Int. J. Electron. (GB),* No. 2, pp. 291-298, August 1983.

[84] R.E. Bixby, *J. Comb. Th. B* 18, pp. 59, 1975,

[85] V.D. Bliznyuk, M.F. Kholodny, "Decomposition of Boolean Functions Applying the Apparatus of Boolean Derivatives," *Automation and Remote Control,* USSR, Vol. 45, No.5, pp. 636-643, 1984.

[86] E. Bloedorn and R.S. Michalski, "Data Driven Constructive Induction in AQ17-PRE: A Method and Experiments," *Proc. of the Third International Conference on Tools for AI*, San Jose, CA, 1991.

[87] A. Blumer, A. Ehrenfeucht, D. Haussler, and M.K. Warmuth, "Occam's Razor", *Information Processing Letters*, Oct. 1987, pp. 377-380.

[88] D. Bochmann, "Zerlegung von Blockschaltungen," *Messen Steuern Regeln*, Vol. 11, No. 3, 1968, (in German).

[89] D. Bochmann, "Einfuehrung in die Strukturelle Automatentheorie," *VEB Verlag Technik,* Berlin 1975; Hanser Verlag, Muenchen-Wien 1975, (in German).

[90] G.V. Bochmann, W.W. Armstrong, "Properties of Boolean Functions with a Tree Decomposition," *BIT*, Vol. 13, pp. 1-13, 1974.

[91] D. Bochmann, Ch. Posthoff, "Binaere Dynamische Systeme," *Akademie Verlag,* Berlin 1981; Oldenbourg-Verlag, Muenchen 1981, (in German).

[92] D. Bochmann, F. Dresig, B. Steinbach, "A New Decomposition Method for Multilevel Circuit Design," *Proc. of the European Conference on Design Automation (EDAC),* Amsterdam, 1991.

[93] D. Bochmann, B. Steinbach, “Logikentwurf mit XBOOLE,” *Verlag Technik,* Berlin, 1991.

[94] D.Bochmann, F. Dresing, B. Steinbach, “New Decomposition Method for Multilevel Circuit Design,” *Proc. EDAC'92*, 1992.

[95] M. Bohanec and V. Rajkovic, “Knowledge acquisition and explanation for multi-attribute decision making,” *Proc. of the 8th Int. Workshop on Expert Systems and their Applications*, pp. 59-78, Avignon, France, 1988.

[96] R. Bolton, ‘Digital Systems Design with Programmable Logic,” *Addison-Wesley Publishing Company,* 1990.

[97] B. Bollobas, and E.J. Cockayne, “Graph-theoretic parameters concerning domination, independence and irredundance,” *J. Graph Theory*, Vol. 3., No. 3., pp. 241–250, Fall 1979.

[98] J.A. Bondy, “Bounds for the Chromatic Number of a Graph,” *J. of Combinatorial Theory,* 7, p. 96, 1969.

[99] G. Boole, “An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities,” *London*, 1854.

[100] A. Bouchet, “An Algebraic Method for Minimizing the Number of States in an Incomplete Sequential Machine,” *IEEE Trans. Computers*, Vol. C-17, pp. 795–798, 1968.

[101] E. Boros, V. Gurvich, P.F. Hammer, T. Ibaraki, and A. Kogan, “Decomposition of Partially Defined Boolean Functions,” *DIMACS Technical Report 94-9*, March 1994.

[102] R.K. Brayton, and C. Mc. Mullen, “The Decomposition and Factorization of Boolean Expressions,” In *Proc. of the Intern. Symp. on Circuits and Systems,* pp. 49-54, May 1982.

[103] R.K. Brayton, G.D. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli, “Logic Minimization Algorithms for VLSI Synthesis,” *Kluwer Academic Publishers,* Boston, 1984.

[104] R.K. Brayton, R. Rudell, A. Sangiovanni-Vncentelli, A.R. Wang, “MIS - a Multiple-Level Logic Minimization System,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 6. No. 6, pp. 1062-1081, 1987.

[105] R.K. Brayton, “Factoring Logic Functions,” *IBM J. Res. Develop.,* Vol. 31, No. 2, March 1987.

[106] R. Brayton and F. Somenzi, “An Exact Minimizer for Boolean Relations,” *Proc. of ICCAD,* pp. 316-320, 1989.

[107] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, “Multilevel Logic Synthesis,” *Proceedings of the IEEE*, Vol. 78, No. 2, pp. 264-300, February 1990.

[108] R.K. Brayton, P.C. McGeer, J.V. Sanghavi, A.L. Sangiovanni-Vincentelli, “A New Exact Minimizer for Two-Level Logic Synthesis,” *in T. Sasao, (ed), “Logic Synthesis and Optimization,* Kluwer 1993, pp. 1 –31.

[109] M.A. Breen, T.D. Ross, M.J. Noviskey, M.L. Axtell, “Pattern Theoretic Image Restoration,” *Proc. SPIE '93 Nonlinear Image Processing IV* 1902-12. January 1993.

[110] M. Breen, T.D. Ross, M.L. Axtell, “Computing Column Multiplicity in Function Decomposition,” *report, WPAFB*, 1994.

[111] D. Brelaz, “New Methods to Color Vertices of a Graph,” *Communications ACM,* Vol. 22, pp. 251–256, 1979.

[112] Breuer, M.A. (ed.), "Design Automation of Digital Systems," Vol. 1, Prentice Hall, Englewood Cliffs, New Jersey, 1972.

[113] R.C. Brigham, and R.D. Dutton, "Graphs which with their complements have certain covering numbers," *Discrete Math.*, Vol. 34, No. 1., pp. 1-7, April 1981.

[114] C. Bron, and J. Kerbosch, "Algorithm 457 - Finding all cliques of an undirected graph," *CACM,* Vol. 16., No. 9., pp. 575 – 1973.

[115] I.I. Brona, "On a Kind of Decomposition of Boolean Functions," *Kibernetika,* 1970, No.3.

[116] R.L. Brooks, "On Colouring the Nodes of a Network," *Proc. Cambridge Philosophical Soc.*, 37., p. 194, 1941.

[117] J.R. Brown, "Chromatic Scheduling and the Chromatic Number Problem," *Management Science,* Vol. 19, pp. 456-463, 1972.

[118] F.M. Brown, "Weighted Realizations of Switching Functions," *IEEE Trans. on Comput.* Vol. 24, No. 11, pp. 1217-1221, 1975.

[119] F.M. Brown, "Equational Realizations of Switching Functions," *IEEE Trans. on Comput,*, Vol. 24, No. 11, pp. 1054-1066, 1975.

[120] F.M. Brown, "Constrained-Input Problem," *IEEE Trans. on Comput.*, Vol. 24, No. 1, pp. 102-106, 1975.

[121] F.M. Brown, "Boolean Reasoning. The Logic of Boolean Equations," *Kluwer Academic Publishers,* 1990.

[122] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *Trans. on Comput.*, Vol. C-35, No. 8, pp. 667-691, 1986.

[123] R. Bryant, "Binary Moment Diagrams," *Carnagie Mellon University,* CMU-CS-94-160 Report, May 31, 1994.

[124] J.A. Brzozowski, and T. Luba, "Decompositions of Boolean Functions Specified by Cubes," *Report CS-97-01,* January 1997, University of Waterloo, Computer Science Department.

[125] E. Buehl, "Beitrag zur Theorie der Dynamischen Fehler-erscheinungen in Kombinatorischen und Sequentiellen Schaltnetzwerken," *Dissertation,* T.H. Karl-Marx-Stadt (Chemnitz), 1975, (in German).

[126] P. Burkey, "Constructive Induction by Decomposition of Noisy Multiple-Valued Functions, Relations and Fuzzy Functions in Machine Learning and Knowledge Discovery from Data Bases," *forthcoming M.S. Thesis*, PSU, 1998.

[127] M. Burns, "New Approaches to the Problems of Incompatibility Graph Creation and Column Encoding in Functional Decomposition of Boolean Function," *M.S. Thesis*, PSU, 1997.

[128] M. Burns, M. Perkowski, L. Jozwiak, "An Efficient Approach to Decomposition of Multi-Output Boolean Functions with Large Sets of Bound Variables," Accepted to Euro-Micro'98.

[129] M. Burns, M. Perkowski, L. Jozwiak, "An Efficient and Effective Approach to Column-Based Input/Output Encoding in Functional Decomposition," Accepted to Third Intern. Symposium on Boolean Problems, Freiberg, September 1998.

[130] J. Butler, and K.J. Breeding, "Some Characteristics of Universal Cell Nets," *IEEE Trans. on Comput.*, Vol. C-22, pp. 897-903, October 1973.

[131] J.T. Butler, "On the Number of Functions Realized by Cascades and Disjunctive Networks," *IEEE Trans. on Comput.*, Vol. 24, No. 7, pp. 681-690, July 1975.

[132] J.T. Butler, "Fanout-Free networks of multi-valued gates," *Proc. 1976 ISMVL*, pp. 39-46, May 1976.

[133] J.T. Butler, "Tandem Networks of Universal Cells," *IEEE Trans. on Comput.*, Vol. 27, No. 9, pp. 785-799, 1978.

[134] J.T. Butler, "Analysis and Design of Fanout-Free Networks of Positive Symmetric Gates," *Journal of the Association for Computing Machinery,* Vol. 25, No. 3, pp. 481-498, 1978.

[135] J. Butler, K.J. Breeding, "Some Characteristics of Universal Cell Nets," *IEEE Trans. on Comput.*, Vol. C-22, pp. 897-903, October 1973.

[136] J. Butler and J. Schueller, "Worst Case Number of Terms in Symmetric Multiple-Valued Functions," *Proc ISMVL 1991.*

[137] 23. A.A. Butov, *Avtomatika i telemekhanika,* No. 9, pp. 121-125, 1978, (in Russian).

[138] A.A. Butov, *Avtomatizacjia Log. Proektirovanija Diskret. Ustroystv,* pp. 72-77, Minsk, 1980, (in Russian).

[139] A.A. Butov, *Kibernetika,* No.3, pp. 63-70, 1980, (in Russian).

[140] R.W. Butterworth, "A Set Theoretic Treatment of Coherent Systems," *SIAM JAMA,* Vol. 22, No. 4, pp. 590–598, 1972. QA1 .S73.

[141] D. De Caen, P. Erdos, N. Pullman, and N. Wormeld, "Extremal Clique Coverings of Complementary Graphs," *Combinatorica,* Vol. 6., No. 4., pp. 309–314, 1986.

[142] P. Calazans, Ph.D. Thesis, 1994

[143] J. Carroll, *Final Technical Report*, Project GEMMA, Wright Lab/WPAFB, 8/9/94, 1994.

[144] P.A. Catlin, "On the Hajnal-Szemeredi theorem on disjoint cliques," *Util. Math.*, Vol. 17, pp. 163-178, May 1980.

[145] J. Catlett, "On changing continuous attributes into ordered discrete attributes," *Proc. of the European Working Session on Learning*, Y. Kodratoff (ed.), *Springer-Verlag*, pp. 164-178, Berlin, Germany, 1991.

[146] E. Cerny, D. Mange, E. Sanches, "Synthesis of Minimal Binary Decision Trees," *IEEE Trans. on Comput.*, Vol. 28, No. 7, pp. 472-482, July 1979.

[147] E.M. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large boolean functions with applications to technology mapping," *Proc. 30th DAC*, June 1993.

[148] M. Chabinyc, "Pattern Based Machine Learning: A Comparison of Ada Function Decomposer Versions," *Final Report,* AFOSR School Apprentice Program, August 1990.

[149] G. J. Chaitin, "On the Length of Programs for Computing Finite Binary Sequences," *J. ACM*, Vol. 13, p. 547, 1966.

[150] G. J. Chaitin, "Algorithic Entropy of Sets," *Comput and Math. Appls.*, Vol. 2, p. 233, 1976.

[151] K. Chakrabarti, O. Kolp, "Fan-in in Constrained Tree Networks of Flexible Cells," *IEEE Trans. on Comput.*, Vol. C-23, pp. 1238-1249, December 1974.

[152] A. H. Chan, "Using decision trees to derive the complement of a binary function with multiple-valued inputs," *IEEE Trans. on Comp.,* Vol. C-36, pp. 212 - 214, Febr. 1987.

[153] , S.C. Chang, M. Marek-Sadowska "Technology Mapping via Transformations of Function Graph," *Proc. Intern. Conf. on Computer Design,* pp. 159-162, 1992.

[154] M. Chein, H. Habib, M.C. Maurer, "Partitive Hypergraphs," *Discr. Math,* Vol.37, pp.35-50, 1981. QA1 .D52

[155] K.C. Chen et al, "Graph Based FPGA Technology Mapping for Delay Optimization," *IEEE Design and Test of Computers,* pp. 7-20, Sept. 1992.

[156] K-C. Chen and S. Muroga, "Input Assignment for Decoded PLA's with Multi-Input Decoders," *IEEE International Conference on Computer-Aided Design,* Digest of Technical Papers, pp. 474-477. YEAR*******.

[157] C.C. Cheung, "A Trimmed Tree Realization of Boolean Functions Using Universal Logic Modules," M.S. Thesis, *Dept Elec. and Comput. Eng. Univ Mass, Amherst,* 1973.

[158] C.C. Cheung, and B.W. Ehrich, "Minimization of Tree-Type Universal Logic Circuits," *IEEE Trans. on Comput,*, Vol. 24, No. 11, pp. 1110-1113, 1975.

[159] N. Christofides, "An Algorithm for the Chromatic Number of a Graph," *Comp. J.,* Vol. 14, pp. 38-39, 1971.

[160] N. Christofides, "Graph Theory," *Academic Press*, New York, 1975.

[161] M. J. Ciesielski, S. Yang, and M. Perkowski, "Multiple-Valued Minimization Based on Graph Coloring," *Proc. ICCD'89,* pp. 262 - 265, October 1989.

[162] M. Ciesielski, and S. Yang, "PLADE: A Two Stage PLA Decomposition," *IEEE Trans. on CAD.* Vol. 11, pp. 943-954, 1992.

[163] E.J. Cockayne, O. Favaron, C. Payan, and A.G. Thomason, "Contributions to the theory of domination, independence and irredundance in graphs," *Discrete Math.*, Vol. 33., No. 3., pp. 249–258, March 1981.

[164] T.F. Coleman, and J.J. More, "Estimation of Sparse Jacobian Matrices and Graph Coloring Problems," ANL-81-39, Argonne Nat'l. Lab., June 1981.

[165] R.C. Conant, "Set-Theoretic Structure Modeling," *International Journal of General Systems*, 1981, No. 38, Vol. 7., pp. 93-107.

[166] Concurrent Logic Inc, "CLi6000 Series Field Programmable Gate Array," *Preliminary Information,* December 1991.

[167] J. Cong, A. Kahng, P. Trajmar, and Kuang-Chien Chen, "Graph Based FPGA Technology Mapping for Delay Optimization," *Proc. First Int'l ACM/SIGDA Workshop on Field Programmable Gate Arrays,* Berkeley, CA, February, 1992, pp. 77-82.

[168] S.C. Crist, "Synthesis of Combinational Logic Using Decomposition and Probability," *IEEE Trans. on Comput.,* Vol.C-29, No.11, pp.1013-1016, November 1980.

[169] L. Csanky, M. Perkowski, I. Schaefer, "Canonical Restricted Mixed-Polarity Exclusive Sums of Products," *Proc. of the IEEE ISCAS'92,* Intern. Symp. on Circ. and Systems, Sheratons on Harbor Island, San Diego, CA, May 10-13, 1992, pp. 17-20.

[170] L. Csanky, M. Perkowski, I. Schaefer, "Canonical Restricted Mixed-Polarity Exclusive-Or Sums of Products and the Efficient Algorithm for their Minimization," *IEE Proceedings, Pt.E,* Vol. 140, No. 1, January 1993.

[171] W.H. Cunnigham, "A Combinatorial Decomposition Theory," *Can. J. Math.,* Vol.32, pp.734-765, 1980. QA1 .C36

[172] R. Cummings, "The nature of psychological explanation," *MIT Press*, Cambridge, MA, 1983.

[173] H.A. Curtis, "Non-Disjunctive Decompositions," *Theory of Switching Bell Labs Report,* No. 19, Section II, 1, 49, 1958.

[174] H.A. Curtis, "A Functional Canonical Form," *J. ACM,* 6, pp. 245-258, 1959.

[175] H.A. Curtis, "A Generalized Tree Circuit," *J. ACM,* pp. 484-496, 1961.

[176] H.A. Curtis, "A New Approach to the Design of Switching Circuits," em Princeton, N.J., Van Nostrand, 1962.

[177] H.A. Curtis, "Generalized Tree Circuit - The Basic Building Block of an Extended Decomposition Theory," *J. Assn. Comput. Mach,* 1963, 10, pp. 562-581.

[178] H.A. Curtis, "Simplified Decomposition of Boolean Functions," *IEEE Trans. on Comput.*, Vol. 25, pp. 1033-1044, October 1976.

[179] J. Darringer et al, "Logic Synthesis Through Local Transformations," *IBM J. Res. Develop.,* pp. 272-280, July 1981.

[180] J.A. Darringer, D. Brand, W.H. Joyner, J.V. Gerbi, L. Trevillyan, "LSS: A System for Production Logic Synthesis," *IBM J. Res. Develop.* Vol. 28, July 1984.

[181] S.R. Das, C.L. Sheng, "On Detecting Total or Partial Symmetry of Switching Functions," *Trans. on Comput.* pp. 352-355, March 1971.

[182] S.R. Das, "On a new approach for finding all the modified cut-sets in an incompatibility graph," *IEEE Trans. Comput.*, Vol. C-22, pp. 187-193, February 1973.

[183] E.S. Davidson, "An Algorithm for NAND Decomposition Under Network Costraints," *IEEE Trans. on Comput.*, pp. 1098-1109, Dec. 1969.

[184] M. Davio, J. P. Deschamps, and A. Thayse, "Discrete and Switching Functions," *Mc.Graw-Hill,* 1978.

[185] D. DeMicheli, M. Santomauro, "Topological Partitioning of Programmable Logic Arrays," *Proc. Intern. Conference on Computer Aided Design,* ICCAD'83, Santa Clara 1983.

[186] De Michelli, G., Sangiovanni-Vincentelli, A., Villa, T., "Computer-Aided Synthesis of PLA-Based Finite State Machines," *Proc. IEEE 1983 Intern. Conf. on Computer Aided Design,* pp. 154-156, September 1983. CONFLICT OF NAMES.

[187]

[188] G. DeMicheli, R.K. Brayton and A. Sangiovanni–Vincentelli, "Optimal State Assignment for Finite State Machines," *IEEE Transactions on Computer–Aided Design,* Vol. CAD-4, No. 3, pp. 268–284. July 1985.

[189] G. DeMicheli, "Symbolic Design of Combinational and Sequential Logic Circuits Implemented by Two-Level Logic Macros," *IEEE Transactions on Computer–Aided Design,* Vol. CAD-5, No. 4, pp. 597–616, Oct. 1986.

[190] G. De Micheli, and T. Klein, "Algorithms for Synchronous Logic Design," *Proc. of the Intern. Symp. on Circuits and Systems,* pp. 756-761, 1989.

[191] G. deMicheli, D. Ku, F. Mailhot, T. Truong, "The OLYMPUS Synthesis System," *IEEE Trans. on Design and Test of Computers,* Oct. 1990.

[192] J. Demsar, B. Zupan, M. Bohanec, and I. Bratko. Constructive induction by decomposition of real functions. In G. W. M. Someren, editor, *Proceedings of ECML-97,* pages 94–107, 1997.

[193] L. De Raedt and M. Bruynooghe, "Constructive Induction by Analogy," *Proceedings of EWSL-89,* pp. 189-200, Pitman, Montpellier, France, 1989.

[194] S.C. deSarkar, A.K. Basu, and A.K. Choudhury, "Simplification of Incompletely Specified Flow Tables With the Help of Prime Closed Sets," *IEEE Trans. Computers,* Vol. C-18, pp. 953–955, 1969.

[195] S.C. deSarkar, A.K. Basu, and A.K. Choudhury, "On the Determination of Irredundant Prime Closed Sets," *IEEE Trans. Computers*, Vol C-20, pp. 933–938, 1971.

[196] P. Deschizeaux , "Synthese de Fonction Booleennes Generales," Dr.Ing. Thesis, *Inst. Math. Appl. Grenoble,* France, 1967.

[197] M. DesJardin and D. F. Gordon, "Evaluation and Selection of Biases in Machine Learning," *Machine Learning Journal*, Vol. 20, pp. 5-21, 1995.

[198] S. Devadas, A.R. Wang, A.R. Newton, A. Sangiovanni-Vincentelli, "Boolean Decomposition in Multi-Level Logic Optimization," *Proc. IEEE International Conf. on Computer-Aided Design,* pp. 290-293, 1988.

[199] S. Devadas, A.R. Wang, A.R. Newton, and A. Sangiovanni-Vincentelli, "Boolean Decomposition of Programmable Logic Arrays," *Proc. IEEE Custom Integrated Circuit Conference,* pp. 2.5.1-2.5.5, 1988.

[200] S. Devadas, A. Wang, A.R. Newton, and A. Sangiovanni-Vincentelli, "Boolean Decomposition in Multilevel Logic Optimization," *IEEE Journal of Solid-State Circuits,* Vol. 24, pp. 399-408, April 1989.

[201] S. Devadas, " Approaches to Multi-Level Sequential Logic Syntesis," *Proc. of the 26th A.C.M./I.E.E.E. Design Automation Conference,* pp. 270-276, 1989.

[202] S. Devadas, and A.R. Newton, "Exact Algorithms for Output Encoding, State Assignment and Four-Level Boolean Minimization," *Proc. Hawaii Intern. Conf. on System Sciences,* pp. 387-396, 1990.

[203] S. Devadas and A. R. Newton, "Exact Algorithms for Output Encoding, State Assignment, and Four-Level Boolean Minimization," *IEEE Trans. on Computer Aided Design,* Jan 1991.

[204] S.Devadas, A.R. Newton, P. Ashar "Sequential Logic Synthesis". pp. 60-64, 1992, *Kluwer Academic Publishers.* CONFLICT NAMES.

[205] L. Diaz-Olavarrieta, and S. G. Zaky, " A New Synthesis Technique for Multilevel Combinational Circuits," *Proc. IEEE European Design Automation Conference*, pp. 222-227, Glasgow, Scotland, March 1990.

[206] D.L. Dietmeyer, P.R. Schneider, "Identification of Symmetry, Redundancy and Equivalence of Boolean Functions," *IEEE Trans. Electron. Comput.*, Vol. 16, pp. 804-817, December 1967.

[207] D.L. Dietmeyer, "Logic Design of Digital Systems," *Boston: Allyn and Bacon,* 1971.

[208] Dolotta, T.A., McCluskey, E.G., "The Coding of Internal States of Sequential Machines," *IEEE Trans. Electr. Comp.,* Vol. EC-13, pp. 549-562, October 1964.

[209] S. Dormino, M.A. Canto, "Generalization of the Voith Method for Minimization of Incompletely Specified Function With Universal Logic Modules," *Intern. J. Electr.*, Vol. 54, No. 2, pp. 271, 1983.

[210] J. Dougherty and R. Kohavi and M. Sahami, "Supervised and unsupervised discretization of continuous features," *Machine Learning: Proc. of the 12th International Conference,* Morgan Kaufmann, San Francisco, CA, 1995.

[211] D. Drake, "Embedding maximal cliques of sets in maximal cliques of bigger sets," *Discrete Math.,* Vol. 58., No. 3., pp. 229-242, March 1986.

[212] G. Drastal and G. Czako and S. Raatz, "Induction in an Abstraction Space: A Form of Constructive Induction," *Proc. of the IJCAI-89*, pp. 708-712, Morgan Kaufmann, Detroit, MI, 1989.

[213] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, M.A. Perkowski, "Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams," *Proceedings of DAC '94*, San Diego, CA, June 1994, pp. 415-419.

[214] F. Dresig, "Synthese Kombinatorischer Schaltnetzwerke mit Arbeitsplatzcomputern," *Dissertation*, T.U. Karl-Marx-Stadt (Chemnitz), 1988, (in German).

[215] F. Dresig, C. Zimmermann, "ALOE - Logiksynthese mittels XBOOLE," *27. Arbeitstagung des FA Schaltkreissysteme und CAD-Komponenten der WS Computer- und Mikroprozessortechnik der KdT,* Eisenach, 1989.

[216] F. Dresig, U. Landmann, B. Meyer, and C. Zimmermann, "Von DDL zu NBS - Beitraege zum Logikentwurf mittels XBOOLE," *Tagungsmaterialien der 4. Tagung Schaltkreisentwurf,* Dresden 1990, (in German).

[217] F. Dresig, O. Rettig, and U.G. Baitinger, "Logic Synthesis for Universal Logic Cells," *Intern. Workshop on Field Programmable Logic and Applications,* Oxford, in: Moore, W.R., Luk, W.: FPGAs. Abington EE and CS Books, Abington, England, 1991.

[218] F. Dresig, Ph. Lanches, O. Rettig, U.G. Baitinger, "Functional Decomposition for Universal Logic Cells Using Substitution," *Proc. of the European Conference on Design Automation (EDAC),* Brussels, 1992.

[219] D. Drusinsky, and D. Harel, "State Charts as an Abstract Model for Digital Control Units," *Dept. Applied Math. and Comp. Sci., The Weizmann Institute of Science,* Rehovot, Israel, 1986.

[220] E.V. Dubrova, J.C. Muzio, B. Von Stengel, "Finding Composition Trees for Multiple-Valued Functions," *Proc. ISMVL'97*, pp. 19 - 26.

[221] T. Duempfert, "Synthese von Schwellwertfunktionen," *Diplomarheit,* TU Chemnitz, 1992, (in German).

[222] J.R. Duley, D. L. Dietmeyer, "A Digital System Design Language (DDL)," *IEEE Trans. on Computers,* 17 (1968), pp. 850-861.

[223] F.D.J. Dunstan, "Sequential Colorings of Graphs," *Cong. Num.,* Vol. 15, pp. 151-158, 1976.

[224] A. Dunworth, and H.V. Hartog, "An Efficient State Minimization Algorithm for Some Special Classes of Incompletely Specified Sequential Machines," *IEEE Trans. Comp.*, Vol. C-28, No. 7, pp. 531–535, July 1979.

[225] Y. Durand, "Logic Synthesis Based on Unification and Rewriting Rules," in: *Design Methodologies for VLSI and Computer Architecture,* D.A. Edwards (Editor), Elsevier Science Publishers B.V. (North-Holland), 1989.

[226] V.F. D'yachenko, "On an Algorithm for Determining the Maximum of a Boolean Function," *Problemy Peredachi Informatsii,* No. 1, 1966.

[227] C.R. Edwards, "The Application of the Rademacher-Walsh Transform to Boolean Function Classification and Threshold Logic Synthesis," *IEEE Trans. Comput.*, Vol. C-24, pp. 48-62, 1975.

[228] C.R. Edwards, S.L. Hurst, "Digital Synthesis Procedure Under Function Symmetries and Mapping Methods," *IEEE Trans. on Comput.*, Vol. 27, No. 11, pp. 985-997, 1978.

[229] C.S. Edwards, and C..H. Elphick, "Lower bounds for the clique and chromatic number of a graph," *Discrete Appl. Math.,* Vol. 5., No. 1., pp. 541-64, January 1983.

[230] O. Ehrsam, "EKSIS - ein Programm zum Entwurf Kombinatorischer Schaltungen durch Iterative Ausnutzung von Symmetrieeigenschaften," *Nachrichtentechnik Elektronik* 12, pp. 460-462, 1984, (in German).

[231] A.B. Ektare, D.P. Mital, "An Algorithm for Designing Multiplexer Circuits," *Intern. J. of Electronics,* Vol. 49, No. 2, 1980.

[232] M. Elmasry, "Logic Partition for Multi-Emitter 2-Level Structures," *IEEE Trans. on Circuits and Systems,* Vol. 21, No. 3, pp. 354-359, 1974.

[233] W. Emde and C.U. Habel and C.R. Rollinger, "The Discovery of the Equator or Concept Driven Learning," *Proc. of IJCAI-83,* pp. 455-458, Karlsruhe, Germany, Morgan Kaufmann, 1983.

[234] G. Epstein, "Synthesis of Electronic Circuits for Symmetric Functions," *IRE Trans. Elec. Comp.* Vol. EC-7, March 1958, pp. 57-60.

[235] G. Epstein, D.M. Miller, J.C. Muzio, "Some Preliminary Views on the General Synthesis of Electronic Circuits for Symmetric and Partially Symmetric Functions," *Proc. 7th Intern. Symp. on Multiple-Valued Logic*, pp. 29-34, May 1977.

[236] G. Epstein, D.M. Miller, J.C. Muzio, "Selecting Don't Care Sets for Symmetric n-Valued Functions: a Pictorial Approach Using Matrices," *Proc. 10th Intern. Symp on Multiple-Valued Logic,* May 1980, pp. 219-225.

[237] G. Epstein. *Multiple-valued Logic Design: An Introduction.* Institute of Physics Publishing, Bristol, 1993.

[238] P. Erdos, A.M. Hobbs, and C. Payan, "Disjoint cliques and disjoint maximal independent sets of vertices in graphs," *Discrete Math.*, Vol. 42., No. 1., pp. 57–61, Nov. 1982.

[239] A.P. Ershov, "Vvedenije v Teoreticeskoye Programmirovanije," Moscow, 1977 (in Russian).

[240] S. Even, "Algorithmic Combinatorics", Macmillan, New York, 1973.

[241] H. Enrich, " A Note on State Minimization of a Special Class of Incomplete Sequential Machines," *IEEE Trans. on Computers,* C-21, pp. 500-502, 1972.

[242] I.L. Fadeev, *Tr. Sib fiz-tehn. in-ta,*

[243] I.L. Fadeev, "Logiceskij Jazyk Dlja Predstavlenija Algoritmov Sintieza Relejnych Ustroistv," pp. 184-193, Moscow, 1966, (in Russian).

[244] I.L. Fadeev, "Avtomatizacjia Logiceskogo Proektirovanija Cifrovych Ustrojstv," Kiev, 1974, pp. 48-61 (in Russian).

[245] I.L. Fadeev, *Algoritmy resenija zadac diskret. matematiki,* pp. 139-149, Tomsk 1979, (in Russian).

[246] B.J. Falkowski, and M. Perkowski, "Algorithm for the Generation of Disjoint Cubes for Completely and Incompletely Specified Boolean Functions," *Intern. J. of Electronics,* Vol. 70, No. 3, pp. 533 - 538, March 1991.

[247] B. Falkowski, I. Schaefer, M. Perkowski, "Computer Algorithms for the Calculation of Rademacher-Walsh Spectrum for Completely and Incompletely Specified Boolean Functions", *Proc. of the IEEE EURO-DAC '92, European Design Automation Conference,* Sept. 7-10, Hamburg, 1992.

[248] B. Falkowski, I. Schaefer, M. Perkowski, "Effective Computer Methods for the Calculation of Rademacher-Walsh Spectrum for Completely and Incompletely Specified Boolean Functions", *IEEE Trans. on Computer-Aided Design,* October 1992, pp. 1207 - 1226.

[249] B. Falkowski, I. Schaefer, and Ch-H. Chang, "An Effective Computer Algorithm for the Calculation of Disjoint Cube Representation of Boolean Functions", *Proc. of the 36-th Midwest Symposium on Circuits and Systems,* pp. 1308-1311, 1993.

[250] B.C. Falkenhainer and R.S. Michalski, "Integrating Quantitative and Qualitative Discovery in the ABACUS System," *Machine Learning: An Artificial Inteligence Approach*, Vol. 3, Y. Kodratoff and R.S. Michalski, Morgan Kaufmann, Palo Alto, CA, 1990.

[251] K.Y. Fang, A.S. Wojcik, "Modular Decomposition of Combinational Multiple-Valued Circuits," *IEEE Trans. on Comput.*, Vol. 37, No. 10, pp. 1293-1301, 1988.

[252] S. Fastowicz, "Independence, Clique size and maximum degree," *Combinatorica,* Vol. 4., No. 1., pp. 35–38, 1984.

[253] U.M. Fayyad and K.B. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," *Proc. of the 13th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, pp. 1022-1027, 1993.

[254] U. Fayyad and G. Piatetsky-Shapiro and P. Smyth, "The KDD Process for Extracting Useful Knowledge from Volumes of Data," *Communications of the ACM*, Vol. 39, no. 11, pp. 27-34, November, 1996.

[255] U. Fayyad and G. Piatetsky-Shapiro and P. Smyth, "From Data Mining to Knowledge Discovery: An Overview," *Advances in Knowledge Discovery and Data Mining*, Fayaad at al. (eds), *MIT Press*, 1996.

[256] E. Fehlauer, D. Garte, C. Mehlhorn, S. Ruelke, "AZUR - Ein Werkzeug zur Synthese Mehrstufiger Logik," *E.I.S. Workshop,* Dresden, April 1991, (in German).

[257] B. Fei, Q. Hong, H. Wu, M.A. Perkowski, and N. Zhuang, "Efficient Computation for Ternary Reed-Muller Expansions under Fixed-Polarities", *Int. J. Electronics,* Vol. 75, No. 4, pp. 685 - 688, 1993.

[258] C. Files, "Using a Search Heuristic in an NP-Complete Problem in Ashebhurst-Curtis Decomposition," Final Report for Graduate Summer Research Program, Wright Laboratory, Sponsored by Air Force Office of Scientific Research, Bolling Air Force Base, DC and Wright Laboratory, August 1994.

[259] C. Files, "A New Approach to Paritition Selection in Ashenhurst-Curtis Functional Decomposition," *M.S. Thesis, College of Graduate Studies,* University of Idaho, August 1995.

[260] C. Files and R. Drechsler and M. Perkowski, "Functional Decomposition of MVL Functions using Multi-Valued Decision Diagrams," *Proc. of ISMVL'97*, pp. 27-32, Halifax, Nova Scotia, Canada, May 28-30, 1997.

[261] C. Files, "Decomposition of Multiple-Valued Functions for Machine Learning, VLSI layout generators and FPGAs," *forthcoming Ph.D. Thesis,* PSU, 2000.

[262] C. Files, and M. Perkowski, "An Error Reducing Approach to Machine Learning Using Multi-Valued Functional Decomposition," Proc. ISMVL'98, Fukuoka, Japan, May 1998.

[263] C. Files, and M. Perkowski, "Multi-Valued Functional Decomposition as a Machine Learning Method," Proc. ISMVL'98, Fukuoka, Japan, May 1998.

[264] D. Filo, J. Yang, F. Mailhot, G. de Micheli, "Technology Mapping for a Two-Output RAM-Based Field Programmable Gate Array," *Proc. of the European Conf. on Design Automation (EDAC)*, Amsterdam, pp. 534-538, 1991.

[265] M. Findler, "Neural Networks and Machine Learning," *Graduate Student Research Program,* Final Report, AFOSR, August 1989.

[266] N.S. Flann, "Improving Problem Solving Performance by Example Guided Reformulation of Knowledge," in *Change of Representation and Iductive Bias*, D.P. Benjamin, Kluwer Academic, Boston, MA, 1990.

[267] H. Fleisher, and L.I. Maissel, "An Introduction to Array Logic," *IBM J. R&D*, Vol. 19, pp. 98-109, March 1975.

[268] J.M. Francioni, "Imprecise Decision Tables and Their Optimization," *Master Thesis,* Department of Mathematics and Computer Science, Florida State University, Tallahassee, Florida, June 1979.

[269] J.M. Francioni, A. Kandel, "Decomposable Fuzzy-valued Switching Functions," *Fuzzy Sets and Systems,* Vol. 9, No. 1, pp. 41-68, 1983.

[270] R. J. Francis, J. Rose, K. Chung, "Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Array," *Proc. 27th ACM/IEEE Design Automation Conf.*, 1990, pp. 613-619.

[271] R. Francis, J. Rose, Z. Vranesic, "Chortle-crf: Fast Technology Mapping Look-up Table-Based FPGAs," *Proc. 28-th ACM/IEEE Design Automation Conf.,* pp. 227-233, 1991.

[272] R. Francis, J. Rose, Z. Vranesic, "Technology Mapping of Look-up Table-Based FPGAs for Performance", *Proc. IEEE Intern. Conf. on CAD,* pp. 568-571, 1991.

[273] R.J. Francis, J. Rose, Z. Vranesic, "Technology Mapping for Delay Optimization of Lookup Table-Based FPGAs," *MCNC Logic Synthesis Workshop,* 1991.

[274] J.F. Frenzel, "Application of Genetic Algorithms to Pattern Theory," *Final Report,* Summer Faculty Research Program, July 1993.

[275] J. Fricke, "Decomposition of Multiple-Valued Logic Functions," *Proc. 8th ISMVL,* pp. 208-212, 1978.

[276] P. Fricnovics, "Coding of the Internal States of Asynchronous Finite Automata by a p-code of Minimal Length," *Theory of Finite Automata and Its Applications*, Vol. 1, pp. 22-34, pp. 64–65, 1973.

[277] A.D. Friedman, P.R. Menon, "Theory and Design of Switching Circuits." 1975.

[278] M. Fujita, Y. Matsunaga, T. Kakuda, "On Variable Ordering of Binary Decision Diagrams for the Application of Multi-Level Logic Synthesis," *Proc. of 2nd EDAC,* 1990.

[279] M. Fujita, Y. Matsunga, and T. Kakuda, "On Variable Ordering of Binary Decision Diagrams for the Application of Multi-Level Synthesis," *European Conf. on Design Automation*, pp. 50-54, 1991.

[280] M. Fujita, Y. Kukimoto, and R. Brayton, "BDD Minimization by Truth Table Permutations," *Proc. IWLS '95.*

[281] T. Funke, "Vergleich der Ergebnisse unterschiedlicher Synthesewerkzeuge," *Groeer Beleg*, T.U. Chemnitz, Juni 1991, (in German).

[282] Z. Furedi, "The number of maximal independent sets in connected graphs," *J. Graph Theory,* Vol. 11, No. 4, pp. 463–470, 1980.

[283] F. Furtek, G. Stone, I. Jones, "Labyrinth: A Homogeneous Computational Medium," *IEEE Custom Integrated Circuit Conference,* Boston, 1990.

[284] Galois

[285] M.R. Garey, and D.S. Johnson, "The Complexity of Near Optimal Graph Coloring," *JACM,* Vol. 23, pp. 43-49, 1976.

[286] M.R. Garey, D.S. Johnson, and H.C. So, "An Application of Graph Coloring to Printed Circuit Testing," *IEEE Trans. on Circuits and Systems,* Vol. 23, 1976.

[287] M. Garey and D. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," *W. H. Freeman and Co. Publ.*, San Francisco, 1979.

[288] R. Gatlin, "Fast Software Prototyping for Functional Decomposition," *forthcoming M.S. Thesis*, PSU, 1998.

[289] F. Gavril, "Algorithms for maximum k-coloring and k-covering of transitive graphs," *Networks*, Vol. 17, No. 4, pp. 465–470, Winter 1987.

[290] M.A. Gavrilov, "Teorija Avtomatov," Moscow, 1976, pp. 34-71 , (in Russian).

[291] M.A. Gavrilov, "Teorija Relejnych Ustrojstv i Konecnych Avtomatov: Izbrannyje Trudy," Moscow, 1983, (in Russian).

[292] T.K. Gearhart, "Investigations of a Lower Bound on the Error in Learned Functions," *Final Report,* USAF-UES Summer Faculty Research Program, July 1990.

[293] L. Geehaeds, and W. Lindenberg, "Clique detection for nondirected graphs: two new algorithms," *Computing,* Vol. 21, No. 4., pp. 295–322, 1979.

[294] A.J. de Geus, "Logic Synthesis and Optimization Benchmarks for the 1986 Design Automation Conference," *Proc. of the 23rd DAC,* 1986.

[295] A.J. de Geus, W. Cohen, "A Rule-Based System for Optimizing Combinational Logic," *IEEE Design and Test of Computers,* Vol. 2, No. 4, August 1986.

[296] S. Ghosh, A.K. Choudhury, "Partition of Boolean Functions for Realization with Multi-threshold Threshold Logic Elements," *IEEE Trans. Comput.*, Vol. C-22, pp. 204-215, 1973.

[297] M. Gondran, M. Minoux, "Graphs and Algorithms," *John Wiley & Sons,* New York, 1984.

[298] S.C. Gupta, "Decompositions of 4-Variable Boolean Functions," *Computers and Electrical Engineering,* Vol. 8, No. 1, pp. 41-48, 1981. S

[299] A. Gibbs, "Algorithmic Graph Theory," *Cambridge University Press*, 1985.

[300] K.G. Gilmanov, "Synthesis of Logical Circuits on the Basis of Universal Formal Neurons with a Small Number of Inputs", *Engineering Cybernetics*, Vol.9, No.1, pp.93-97, 1971, (in English). TJ212 .A413.

[301] Gimpel, J.F., "The Minimization of TANT Networks," IEEE TEC, Vol. EC-16, pp. 18-38, February 1967.

[302] S. Ginsburg, "A Synthesis Technique for Minimal State Sequential Machines," *IRE Trans. Electron. Computers,* Vol. EC-8, No. 1, pp. 13–24, March, 1959.

[303] S. Ginsburg, "On the Reduction of Superfluous States in a Sequential Machine," *J. Assoc. Computing Machinery,* Vol. 6, pp. 259–282, April, 1959.

[304] J.A. Goldman, "Pattern Theoretic Knowledge Discovery", *Proc. 6th IEEE International Conference on Tools with Artificial Intelligence,* IEEE, November 1994.

[305] J. A. Goldman, "Machine Learning: A Comparative Study of Pattern Theory and C4.5," Wright Laboratory, USAF, Technical Report, WL-TR-94-1102, WL/AART, WPAFB, OH 45433-6543, August 1994.

[306] J. A. Goldman and M. L. Axtell, "On Using Logic Synthesis for Supervised Classification Learning," *7th IEEE International Conference on Tools with Artificial Intelligence,* IEEE, November 1995.

[307] J. A. Goldman, and M. L. Axtell, "On Using Logic Synthesis for Knowledge Discovery," submitted.

[308] J.A. Goldman, T.D. Ross, and D.A. Gadd, "Pattern Theoretic Learning", *AAAI Spring Symposium Series on Systematic Methods of Scientific Discovery,* AAAI, March 1995.

[309] S.W. Golomb, "On the Classification of Boolean Functions," *IRE Trans. Circ. Theory,* Spectral suppplement, CT-6, pp. 176-186, 1959.

[310] L.M. Goodrich, "A Map Technique for Identifying Variables of Symmetry", *Bell Syst. T,* Vol.53, No.5, pp.801-826, 1974. TK1 .B425

[311] Y.B. Gershkov, *Autom. Remot.* R. 753, 1967, (in Russian), TJ212 .A9135.

[312] A. Grasselli, and F. Lucio, "A Method of Minimizing the Number of Internal States in Incompletely Specified Sequential Networks," *IEEE Trans. Electr. Comp.,* Vol. EC-14, No. 3, pp. 330–359, June 1965.

[313] G.H. Greene, "The Abacus 2 System for Quantitative Discovery: Using Dependencies to Discover Non-Linear Terms," *Reports of Machine Learning and Inference Laboratory, MLI 88-4*, Center for Artificial Intelligence, George Mason University, Fairfax, VA, 1988.

[314] C.R. Greer, R.A. Thompson, "Combinational Logic With Decoder," *IEEE Trans. on Comput.*, Vol. C-27, No. 9, 1978.

[315] G.R. Grimmett, and C.J.H. McDiarmid, "On Colouring Random Graphs," *Math. Proc. Comb. Phil. Soc.* Vol. 77, pp. 314–324, 1975.

[316] S. Grygiel, M. Perkowski, M. Marek-Sadowska, T. Luba and L. Jozwiak, "Cube Diagram Bundles: a new representation of strongly unspecified multiple-valued functions and relations," *Proc. of ISMVL'97*, Halifax, Nova Scotia, Canada, pp. 287-292, May 28-30, 1997.

[317] S. Grygiel and M. Perkowski, "Multiple-Valued Incompletely Specified Functions and Relations. Part I: Representation," 1997.

[318] S. Grygiel, "Ovulocomputer: Inductive Learning Method based on Decomposition of Multi-Valued Relations in a Medical Application," *forthcoming Ph.D. Thesis,* PSU, 1999.

[319] , S. Grygiel and M. Perkowski, "New Method for Efficient Representation of Multiple-Valued Functions and Relations," Unpublished.

[320] S. Grygiel and M. Perkowski, "New Compact Representation of Multiple-Valued Functions, Relations, and Non-deterministic State Machines," Accepted to ICCAD'98.

[321] W. Grzymala-Busse, Z. Pawlak, "On Some Subset of the Partition Set," *Fundamenta Informaticae,* Vol. VII, No. 4, 1984. PWN, Warszawa, Poland.

[322] J.W. Grzymala-Busse, "On the Reduction of Instance Space in Learning from Examples," In *"Methodologies for Intelligent Systems,"* eds. Z.W. Ras, M. Zemankova, M.L. Emrich, Elsevier Sci. Publ., pp. 388-395, 1990.

[323] J.W. Grzymala-Busse, "LERS - A System to Learning from Examples Based on Rough Sets," In *"Intelligent Decision Support. Handbook of Application and Advances of the Rough Sets Theory",* ed. R. Slowinski, Kluwer Academic Publishers, Dordrecht 1992.

[324] M. Habib, M.C. Maurer, "1-Intersecting Families", *Discr. Math.* Vol.53, pp.91-101, 1985. QA1 .D52

[325] G.D. Hachtel, J.K. Rho, F. Somenzi, and R. Jacoby, " Exact and Heuristic Algorithms for the Minimization of Incompletely Specified State Machines," *European Design Automation Conference,* Amsterdam, The Netherlands, pp. 184-191, 1991.

[326] F. Haentemann, "Anwendungsspezifischer Entwurf von DOMINO-CMOS-Schaltnetzen im Gate-Matrix-Entwurfsstil," *Fortschritt-Berichte VDI,* Reihe 9, Nr. 107, VDI-Verlag, Duesseldorf 1990, (in German).

[327] C. Halatsis, N. Gaitanis, "Irredundant Normal Forms and Minimal Dependence Sets of Boolean Function," *IEEE Trans. on Comput.*, Vol. C-27, Nov. 1978.

[328] G. Hallbauer, "Procedures of State Reduction and Assignment in One Step in Synthesis of Asynchronous Sequential Circuits," *Proc. Intern. IFAC Symp. on Discrete Systems,* pp. 272-282, Riga, September 30-October 4, 1974.

[329] P.L. Hammer, and S. Rudeanu, "Boolean Methods in Operations Research," *Springer-Verlag,* New York 1968.

[330] F. Harrary, "Graph Theory," *Addison Wesley*, Reading, Mass., 1969.

[331] F. Harary, J.S. Maylee, "Graph and Applications," *Proc. First Colorado Symp. on Graph Theory,* John Wiley & Sons, New York, 1985.

[332] J. Hartmanis, "On the State Assignment Problem for Sequential Machines, I," *IRE Trans. Electr. Comp.,* Vol. EC-10, p. 157-165, June 1961.

[333] J. Hartmanis , R.E. Stearns, "Algebraic Structure Theory of Sequential Machines," Englewood Cliffs, N.Y., *Prentice Hall, Inc.,* 1966.

[334] K. Hauptmann, W. Schuefer, "Rapid Prototyping mit Programmierbaren Gate Arrays," *ITG-Fachbericht der ITG Fachtagung* vom 3.-5.10.1989, Stuttgart, VDE-Verlag GmhH, Berlin-Offenbach, in German.

[335] J.P. Hayes, "The Fanout Structure of Switching Functions," *JACM,* Vol. 22, No.4, pp. 551-571, Oct. 1975.

[336] J.P. Hayes, "Enumeration of Fanout-Free Boolean Functions," *JACM*, Vol. 23. pp. 700-709, Oct. 1976.

[337] S. He and M. Torkelson, "Disjoint Decomposition With Partial Vertex Chart," In *Proc. of the Intern. Workshop on Logic Synthesis,* Lake Tahoe, CA, pp. P2a-1-P2a-5, May 1993.

[338] B. Hedman, "The maximum number of cliques in dense graphs," *Discrete Math,* Vol. 54., No. 2., pp. 161–166, April 1985.

[339] B. Hedman, "Clique numbers of graphs," *Discrete Appl. Math.,* Vol. 15. No. 1, pp. 55–60, Sept. 1986.

[340] S.L. Height, "Complex Disjunctive Decomposition of Incompletely Specified Boolean Functions," *Ph.D. dissertation,* Univ. New Mexico, Albuquerque, July 1970.

[341] S.L. Height, "Minimal Input Solutions," *IEEE Trans. on Comput.*, pp. 923-925, August 1971.

[342] S.L. Height, "Complex Disjunctive Decomposition of Incompletely Specified Boolean Functions," *IEEE Trans. on Comput.*, Vol. C-22, No. 1, January 1973, pp. 103-110.

[343] M. Helliwell, M. A. Perkowski, "A Fast Algorithm to Minimize Multi-Output Mixed-Polarity Generalized Reed-Muller Forms," *Proc. of the IEEE/ACM 25-th Design Automation Conference,* pp. 427 - 432, Anaheim, CA, June 12-15, 1988.

[344] B. Heeb, "Logic Minimization for Novel Programmable Logic Architectures," *International Workshop on Field Programmable Logic and Applications,* Oxford, in: Moore, W.R., Luk, W.: FPGAs. Abington EE and CS Books, Abington, England, 1991.

[345] Ph. Ho, and M. A. Perkowski, "Minimization of Fine-Grain FPGAs Using Free Kronecker Decision Diagrams," *Report, Department of Electrical Engineering,* PSU, 1994.

[346] C. Hoede, "Semi-matchings and semi-stars, surroundings, decompositions and detachments. An analysis of the maximum clique problem," Memo 547, Dept. Applied Math., Twente Univ. of Techn., preprint, 1985.

[347] C. Hoede, "Hard Graphs for the Maximum Clique Problem," *Discrete Mathematics*, Vol. 72., pp. 175–179, 1988.

[348] S.J. Hong, R.G. Cain, and D.L. Ostapko, " MINI: A heuristic approach for logic minimization," *IBM J. Res. Develop.,* Vol. 18, pp. 443 – 458, Sept. 1974.

[349] S. J. Hong. A heuristic algorithm for generating minimal rules from examples. *Pacific Rim International Conference on Systematic Methods of Scientific Discovery*, 1994.

[350] G. Hopkins, and W. Staton, "Graphs with unique maximum independent sets," *Discrete Math.*, Vol. 57., No. 3, pp. 245–251. *Discrete Math.*, December 1985.

[351] E. Howorka, "On metric properties of certain clique graphs," *J. Comb. Theory* Series, Vol. B 27, No.1., pp. 67–74, Aug. 1979.

[352] S. J. Hong, "R-MINI: A Heuristic Algorithm for Generating Minimal Rules from Examples", *Pacific Rim International Conference on Artificial Intelligence,* PRICAI, 1994.

[353] R.W. House, and D.W. Stevens, " A New Rule for Reducing CC Tables, " *IEEE Transactions on Computers,* Vol. C-19, pp. 1108-1111, Nov. 1970.

[354] B. Hruz, "Unateness Test of a Boolean Function and Two General Synthesis Methods Using Threshold Logic Elements," *IEEE Trans. on Comput.,* pp. 122-131, Febr. 1969.

[355] W.L. Hsu, and G.L. Nemhauser, "Algorithms for minimum covering by cliques and maximum clique in claw-free perfect graphs," *Discrete Math.*, Vol. 37., No. 2,3, pp. 181–191, December 1981.

[356] J.D. Huang, J.Y. Jou, and W.Z. Shen, "Compatible Class Encoding in Roth-Karp Decomposition for Two-Output LUT Architecture," *Proc 1995 ICCAD*, pp. 359–363, 1995.

[357] D.A. Huffmann, "The design and use of hazard-free switching networks," *Journal Ass. Comp. Mach.* Vol. 4, No. 1, 1957.

[358] R.B. Hurley, "Decision Tables in Software Engineering," *Van Nostrand Reinhold,* New York, 1983.

[359] S.L. Hurst, C.R. Edwards, "Preliminary Considerations of the Design of Combinatorial and Sequential Digital Systems Under Symmetry Methods," *Int. J. Electron.*, 1976, 40, pp. 499-507.

[360] S.L. Hurst, "Detection of Symmetries in Combinatorial Functions by Spectral Means," *IEE J. Electron. Circ. and Syst.*, Vol. 1, No. 5, pp. 173-180, 1977.

[361] S.L. Hurst, "Logical Processing of Digital Signals," *Crane-Russak,* New York, Edward Arnold, London, 1978.

[362] S.L. Hurst, D.M. Miller, J.C. Muzio, "Spectral Techniques in Digital Logic," *Academic Press,* London, 1985.

[363] T.T. Hwang et al, "Multi-Level Logic Synthesis Using Communication Complexity," *Proc. 26th DAC,* pp. 215-220, 1989.

[364] T.T. Hwang, R.M. Owens, M.J. Irwin, "Exploiting Communication Complexity for Multilevel Logic Synthesis," *IEEE Trans. on CAD,* Vol. 9, No. 10, pp. 1017-1027, Oct, 1990.

[365] T.T. Hwang, et al., "Efficiently Computing Communication Complexity for Multilevel Logic Synthesis," *IEEE Trans. on CAD,* Vol. 11., No 5, pp. 545-554, May 1992.

[366] Y. Ikuea, "Algorithms for vertex packing and clique covering problems on some unimodular graphs," Cornell Univ., Ithaca, N.Y., 1981, 158 pp.

[367] H. Imai, and T.S. Asano, "Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane," *Algorithms,* Vol. 1, No. 4., pp. 310–323, Dec. 1983.

[368] K. Jasinski, T. Luba, J. Kalinowski, "CAD Tools for PLD Implementation of ASICs", *Proc. of Second Eurochip Workshop on VLSI Design Training,* pp. 225-230, Grenoble, 1991.

[369] G.H. John, R. Kohavi, and K. Pfleger, "Irrelevant Features and the Subset Selection Problem," In *Machine Learning: Proceedings of Eleventh International Conference,* 1994, July 1994.

[370] D.S. Johnson, "Worst Case Behavior of Graph Coloring Algorithms," *Proc. 5th S-E Conf. on Combinatorics, Graph Theory and Comput.,* Utilitas Mathematica, Winnipeg 1979, pp. 513–527.

[371] J.M. Joller, "Constructing Fault-Trees by Stepwise Refinement," *IEEE Trans. on Reliability,* Vol. 31, No. 4, pp. 333-338, 1982.

[372] L. Jozwiak, F. Volf, "An Efficient Method for Decomposition of Multiple-Output Boolean Functions and Assigned Sequential Machines", *Proc. European Conference on Design Automation,* pp. 114-122, 1992.

[373] L. Jozwiak, "General Decomposition and Its Use in Digital Circuit Synthesis," *Manuscript,* Eindhoven University of Technology, Faculty of Electrical Engineering, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, 1994.

[374] T. Kalganova, "Combinational multiple-valued circuit design by generalized disjunctive decomposition," *Proc. of the Europ. Conf. on Circuit Theory and Design, ECCTD'97,* Budapest, Hungary, 1996.

[375] T. Kalganova, "Functional decomposition methods for multiple-valued logic functions and its system," *Proc. of the 3rd Int. Conf. on Applications of Computer Systems*, pp. 75-82, Szczecin, Poland, 1996.

[376] T. Kalganova, "The studing of the finctional decomposition methods for r-valued logic functions in the logic design courses," *Proc. of the 3nd Int. Conf. on the New Information Technologies in Education*, Vol. 2, pp. 150-158, Minsk, Bielarus, 1996.

[377] A.Kandel, "Decomposition of Fuzzy Functions," *IEEE Trans. on Comput.*, Vol. 25, No. 11, pp. 1124-1130, 1976.

[378] A. Kandel, "Fuzzy Switching and Automata: Theory and Applications", *Crane-Russak,*, New York, 1979.

[379] L.N. Kannan, and D. Sarma, " Fast Heuristic Algorithms for Finite State Machine Minimization," *Proc. EDAC 91,* Amsterdam, pp. 192-196, Febr. 1991.

[380] M. Karnaugh, "The map method for synthesis of combinational logic circuits", *Trans. AIEEE 72,* pt. I, pp. 593-598, 1953.

[381] R.M. Karp, F.E. McFarlin, J.P. Roth, J.R. Wilts, "A Computer Program for the Synthesis of Combinational Switching Circuits," *In Proc. AIEE Annual Symposium on Switching Circuits Theory,* pp. 182-194, 1961.

[382] R.M. Karp, "Functional Decomposition and Switching Circuit Design," *J. Soc. Industr. Appl. Math.*, Vol. 11, No. 2, pp. 291-335, June 1963.

[383] R.M. Karp, "Some Techniques of State Assignment for Synchronous Sequential Machines," *IEEE Trans. Electr. Comp.,* Vol. EC-13 No. 5, pp. 507-518, October 1964.

[384] R.M. Karp, “Sintez Relejnych Struktur,” pp. 220-232, Moscow 1965, (in Russian),

[385] R.M. Karp, “Reducibility among Combinatorial Problems,” Complexity of Computer Computations, (R.E. Miller and J.W. Thatcher eds), Plenum Press, New York (1972), pp. 85-104.

[386] K. Karp, “Using if-then-else DAGS to do Technology Mapping for Field-Progammable Gate Arrays,” *Internal Report, No.: UCSC-CRL-90-43,* Univ. of California, Santa Cruz, Sept. 1990.

[387] K. Karplus, “XMAP: A Technology Mapper for Table-Lookup Field Programmable Gate Arrays,” *Proc. 28th ACM/IEEE Design Automation Conf.,* pp.240-243, 1991.

[388] M.G. Karpovsky, “Finite Orthogonal Series in the Design of Digital Devices,” *J. Wiley,* New York, 1976.

[389] M.G. Karpovsky, “Harmonic Analysis over Finite Commutative Groups in Linearization Problems for Systems of Logical Functions,” *Inform. Control,* Vol. 33. No.2, pp. 142-165, Febr. 1977.

[390] V. Ye. Katsman, ”Theory of Partition Models,” *Soviet Journal of Computer and Systems Sciences,* (Tekhnicheskaya Kibernetika), Vol. 29, No. 1, pp. 64-73, Jan-Febr 1991.

[391] U. Kebschull, E. Schubert, and W. Rosenstiel, ”Multilevel Logic Synthesis Based on Functional Decision Diagrams,” *Proc. IEEE Euro-DAC*, pp. 43-47, 1992.

[392] J. Kella, “State Minimization of Incompletely Specified Sequential Machines,” *IEEE Trans. Computers,* Vol. C-19, pp. 342–348, 1970.

[393] M.B. Kermani, M.H. Mickle, L.P. McNamee, “Identification of Disjunctively Decomposable Logic Functions Employing a Karnaugh Map,” *IEEE Trans. on Comput.*, pp. 277-279, Vol. 18, March 1969.

[394] Kerntopf, P., Michalski, A., “Selected Problems in Synthesis of Combinatorial Logic Circuits,” PWN, Warsaw, 1972 (in polish).

[395] B. Kick, “Logiksynthese - eine Uebersicht,” in: *Simulation und Synthese logischer Schaltungen,* (Hrsg: H. Khakzar), Expert-Verlag, 1991, (in German).

[396] B.G. Kim, “Multilevel Logic Synthesis Using Extended Array Implementation,” *Ph.D. Dissertation,* Univ. Wisconsin-Madison, 1989.

[397] B.G. Kim, D.L. Dietmeyer, “Multilevel Logic Synthesis of Symmetric Switching Functions,” *IEEE Trans. on CAD,* Vol. 10, No. 4, pp. 436-446, April 1991.

[398] E. Kjelkrud, “On the Decomposition of General Switching Functions,” *The Royal Institute of Technology, Division of Applied Electronics,* Stockholm, Sweden, 1971.

[399] E. Kjelkrud, “A Computer Program for the Synthesis of Switching Circuits by Decomposition,” WHERE?? June 1972.

[400] C.A. Knoblock, “A Theory of Abstraction for Hierarchical Planning,” *Change of Representation and Inductive Bias*, D.P. Benjamin, Kluwer Academic, Boston, MA, 1990.

[401] C.A. Knoblock, S. Minton and O. Etzioni, “Integrating Abstraction and Explanation Based Learning in PRODIGY,” *Proc. of AAAI-91*, pp. 541-546, AAAI Press/MIT Press, 1991.

[402] U. Knoth, C. Zimmermann, “Funktionelle Verifikation synchroner, digitaler Systeme,” *Dissertation,* TU Chemnitz, 1990, (in German).

[403] K.L. Kodandapani, S.C. Seth, "Combinational Networks With Restricted Fan-Out," *IEEE Trans. on Comput.*, Vol. C-27, No. 4, pp. 309-318, 1978.

[404] M. Koegst, Ch. Posthoff, B. Steinbach, "Beraecksichtigung von kombinatorischen Bluecken beim Logischen Entwurf Digitaler Systeme," *Nachrichtentechn. Elektronik,* Berlin 34, 2, 1984, (in German).

[405] Z. Kohavi, "Switching and Finite Automata Theory," *McGraw-Hill Book Company,* 1978.

[406] R. Kohavi, and B. Frasca, "Useful Feature Subsets and Rough Set Reducts", *Third International Workshop on Rough Sets and Soft Computing,* 1994.

[407] R. Kohavi, "Bottom-up Induction of Oblivious Read-Once Decision Diagrams," In *European Conference on Machine Learning,* 1994.

[408] R. Kohavi, "A Third Dimension to Rough Sets," *Proc. of The Third International Workshop on Rough Sets and Soft Computing,* pp. 244-251, San Jose, 1994.

[409] M.M. Kokar, "Discovering Functional Formulas Through Changing Representation Base," *Proc. of the AAAI-86*, pp. 455-459, Philadelphia, PA, 1986.

[410] A. N. Kolmogorov, "On Tables of Random Numbers," *Sankhya*, Vol. A25, pp. 369, 1963.

[411] I. Kononenko,"Inductive and bayesian learning in medical diagnosis," *Applied Artificial Intelligence*, No. 7, pp. 317-337, 1993.

[412] I. Kononenko, E. Simec, and M. R. Sikonja. Experiments in automatic learning of medical diagnostic rules. Technical report, Jozef Stefan Institute, Ljubljana, 1996.

[413] L.T. Kou, L.S. Stockmeyer, and C.K. Wong, "Covering edges by cliques with regard to keyword conflicts and intersection graphs," *CACM,* Vol. 21, No. 2., pp. 135–139, Febr. 1978.

[414] A. Kowalczyk, "Decomposition of Strongly Unspecified Boolean Functions," *Przeglad Telekomunikacyjny,* No. 1, 1983, (in Polish).

[415] J. Koza, *"Genetic Programming,"* MIT Press, 1992.

[416] H. Kreutz, "Entwurf von Schaltnetzen aus Zuordnerbausteinen durch Dekomposition von Baende In unvollstaendiger Schaltfunktionen," *Dissertation,* Universitaet Karlsruhe 1980, (in German).

[417] Y.S. Kuo, and C. Chau, " Generating essential primes for a boolean function with multiple-valued inputs," *Proceedings of DAC' 86,* 1986.

[418] Y.S. Kuo, "Generating Essential Prime Implicants for a Boolean Function with Multiple–valued Inputs", *IEEE Trans. on Computers,* Vol.36, pp. 356-359, Mar. 1987.

[419] A.V. Kuznetsov, "On Nonrepeating Contact Networks and Nonrepeating Superpositions of Boolean Functions," *Trans. Steklov Math. Inst. AN SSSR*, Izd. AN SSSR, Vol. 51, (in Russian), 1958.

[420] A.V. Kuznetsov, "A Property of Functions Realized by Nonplanar Nonrepeating Circuits," *Trans. Steklov. Math. Inst. AN SSSR,* Izd. AN SSSR, Vol 51, (in Russian).

[421] H.C. Lai, "Design of diagnosable MOS - Networks," *Dissertation,* Dep. of Computer Science, University of Illinois, December 1979.

[422] Y.T. Lai, M. Pedram, S. Vrudhula, "BDD-based Logic Decomposition: Theory." *Technical Report,* Dept. of EE. Systems, University of Southern California, 1992.

[423] Y.T. Lai, M. Pedram, S. Sastry, "BDD-based Decomposition of Logic Functions With Application to FPGA Synthesis," *Proc. of 30th DAC,* pp. 642-647, 1993.

[424] Y.T. Lai, K.R. Pan, M. Pedram, S. Vrudhula, "FGMap: A Technology Mapping Algorithm for Look-up Table Type FPGA Synthesis," *Proc. of 30-th DAC*, pp. 642-647, 1993.

[425] Y.T. Lai, K.R. Pan, and M. Pedram, "FPGA Synthesis using Function Decomposition," 1994.

[426] P.K. Lala, "An Algorithm for the State Assignment of Synchronous Sequential Circuits," *Electr. Letters,* Vol. 14, No. 6., 16 March 1978.

[427] U. Landmann, "Preprozessing-Methoden fuer die Synthese Kombinatorischer Schaltnetzwerke," *Dissertation,* T.U. Chemnitz, 1992, (in German).

[428] C. Lang, "Glitchpower in CMOS-Schaltungen. Praktikumsbericht," *IBM Laboratorien* Bueblingen, 1992, (in German).

[429] G.G. Langdon, "Decomposition Chart Technique to Aid in Realizations With Multiplexers," *IEEE Trans. on Comput.*, Vol. 27, No. 2, pp. 157-159, 1978.

[430] J. Langenderfer, "A Study of the Computational Complexities of Functions and their Inverses," Memorandum. *Wright Research and Development Center,* WL/AART Wright Patterson AFB, OH, Sept. 1989.

[431] P. Langley and G.L. Bradshaw and H.A. Simon, "Rediscovering Chemistry with the BACON System", *Machine Learning: An Artificial Inteligence Approach*, R.S. Michalski and J.G. Carbonell and T.M Mitchell, eds., Morgan Kaufmann, Los Altos, CA, 1983.

[432] P. Langley and H.A. Simon and G.L. Bradshaw and J.M. Zytkow, "Scientific Discovery: Computational Explorations of the Creative Process," *MIT Press*, Cambridge, MA, 1987.

[433] F. Lapscher, "Decompositions Simples de Functions Booleans," *Automatisme*, No. 3, 14, 115, 1969.

[434] J.B. Larson and R.S. Michalski, "Inductive Inference of VL Decision Rules," *ACM SIGART Newsletter*, no. 63, pp. 38-44, 1977.

[435] T. Q. Le, "Testbarkeit Kombinatorischer Schaltungen - Theorie und Entwurf," *Dissertation,* T.U. Karl-Marx-Stadt (Chemnitz), 1989, (in German).

[436] T.Q. Le, B. Steinbach, "Effiziente Methoden zum Logikentwurf Testbarer Kombinatorischer Schaltungen und deren Impliziten Testsatzberechnung," *3. ITG/GI-Workshop,* Blomberg, Muerz 1991, (in German).

[437] T.Q. Le, "Logikentwurf Digitaler Schaltungen," *Habilitation,* TU Chemnitz, 1992, (in German).

[438] R. Lechner and A. Moezzi, "The Synthesis of Encoded Programmable Logic Arrays in Spectral Techniques and Fault Detection," M. Karpovsky (Ed.) Orlando, FL, *Academic Press*, 1985.

[439] E.B. Lee and M.A. Perkowski, "A New Approach to Structural Synthesis of Automata," University of Minnesota, Dept. of Electrical Engineering, Technical Report, 1982.

[440] E.B. Lee and M.A. Perkowski, "Concurrent Minimization and State Assignment of Finite State Machines," *Proc. IEEE Int. Conference on Systems, Man and Cybernetics,* pp. 248 - 260, Halifax, Nova Scotia, Canada, October 9 - 12, 1984.

[441] E. Lehman, Y. Watanabe, J. Grodstein, H. Harkness, "Logic Decomposition during Technology Mapping," *Proc. ICCAD'95*, pp. 264–271, 1995.

[442] F.T Leighton, "A Graph Coloring Algorithm for Large Scheduling Problems," *J. of Research. National Bureau of Standards,* Vol. 84, pp. 489-496, 1979.

[443] T.A. Leksich, S. Nikolich, "Computer-Systems and Automata Predictive Expansion of Switching Functions," *Engineering Cybernetics,* Vol. 18, No. 5, pp. 79-85, 1980.

[444] G.G. Lendaris and G.L. Stanley, "Structure and constraints in discrete adaptive networks," *National Electronics Conference*, Vol. XXI, 1965.

[445] G.G. Lendaris and G.L. Stanley, "Self-oranization: meaning and means," J. Spiegel and D. Walker, *Proceedings of the Second Congress*, Information System Sciences, *Spartan Books*, Baltimore, 1965.

[446] G.G. Lendaris and G.L. Stanley, "On the structure-dependant properties of adaptive logic networks," *GM Defense Research Laboratories*, Santa Barbara, California, July, 1963.

[447] D. Lewin, *Computer BU,* Vol. 13, pp. 382, 1974.

[448] D.B. Lenat, "On Automated Scientific Theory Formation: A Case Study Using AM Program," *Machine Intellience*, J.E. Hayes and D. Mitchie and L.I. Mikulich (eds), Vol. 9, Halsted Press, New York, 1977.

[449] D.B. Lenat, "Learning from Observation and Discovery," *Machine Learning: An Aritficial Intelligence Approach*, R.S. Michalski and J.G. Carbonell and T.M. Mitchell (eds), Morgan Kaufmann, Los Altos, CA, 1983.

[450] D. Lewin, "Outstanding Problems in Logic Design," *Radio and Electronics Engineer,* Vol. 44, No. 1, pp.9-17, 1974.

[451] S.Y. Levy, R.O. Winder, T.H. Mott, "A Note on Tributary Switching Networks," *IEEE Trans. on Electr. Comput.*, Vol. EC-13, pp. 148-151, 1964.

[452] H.F. Li, "Variable Selection in Logic Synthesis Using Multiplexers," *Intern. J. of Electronics,* Vol. 49, No. 3, September 1980.

[453] M. Li and P. M. B. Vitányi, "Inductive Reasoning and Kolmogorov Complexity", *Journal of Computer and System Sciences,* Vol. 44, pp. 343-384, 1992.

[454] B. Lin, and A.R. Newton, " Restructuring State Machines and State Assignment: Relationship to Minimizing Logic Across Latch Boundaries," *Proc. International Workshop on Logic Synthesis,* Research Triangle Park, North Carolina, May 23-26, 1989, paper 5.3.

[455] B. Lin, H.J. Touati, and A.R. Newton, " Don't Care Minimization of Multi-Level Sequential Logic Networks," *Proc. of the Intern. Conf. on Computer Aided Design,* pp. 414-417, 1990.

[456] B. Lin, and F. Somenzi, " Minimization of Symbolic Relations, " *Proc. of the Intern. Conf. on Computer Aided Design,* pp. 88-91, 1990.

[457] A. M. Lloyd, "Design of Multiplexer Universal-Logic-Module Networks Using Spectral Techniques," *IEE Proc. Pt. E.,* Vol. 127, pp. 31-36, January 1980.

[458] LogicBenchmarks, "Logic Benchmarks of International Workshop on Logic Synthesis," *MCNC*, North Carolina, May 1987.

[459] T. Luba, "Synthesis of Combinational Circuits Using Boolean Decomposition Method," *Publishers of Institute of Telecommunication,* Warsaw Technical University, No. 108, 1983, (in Polish).

[460] T. Luba, L. Wronski, "Decompositional Method of Realizing Systems of Switching Functions," *Review of Telecommunication,* No. 4, 1985, (in Polish).

[461] T. Luba, "New Approach to Boolean Function Decomposition for ROMs and PLAs," *Publishers of Institute of Telecommunication,* Warsaw Technical University, No. 123, 1985.

[462] T. Luba, "Synthesis of Combinational Circuits Using the Method of Boolean Decomposition," *Rozprawy Elektrotechniczne,* No. 2, 1985.

[463] T. Luba, L.Wronski, "Sequential Determination of Values of Boolean Expressions Using the Threshold Table Method," *Review of Telecommunication,* No.1, 1986, (in Polish).

[464] T. Luba, B. Zbierzchowski, "Topological Models of Boolean Functions and Their Application in the Synthesis of Complex Combinational Circuits," *Rozprawy Elektrotechniczne,* No.2, 1987.

[465] T. Luba, "A Uniform Method of Boolean Function Decomposition," *Rozprawy Elektrotechniczne,* No. 4, pp. 1041-1054, 1986.

[466] T. Luba, "Synthesis of Multi-Level Logic Circuits," *Prace Naukowe Politechniki Warszawskiej, Elektronika,* Vol. 79, 1988, Publishers of Warsaw Technical University, 00-665, Warszawa, Nowowiejska 24, Poland, (in Polish).

[467] T. Luba, J. Kalinowski, K. Jasinski, A. Krasniewski, "Combining Serial Decomposition with Topological Partitioning for Effective Multi-Level PLA Implementations", In *"Logic and Architecture Synthesis,"*, 1991. P. Michel and G. Saucier, (Editors), Elsevier Science Publisher B.V. (North Holland), pp. 243-252, 1991.

[468] T. Luba, J. Janowski, J. Rybnik, "Relations Between Multiple-Valued Logic and Decision Logic with Respect to Rough Set Theory Semantics," *Research Report, Institute for Telecommunications,* No. 139, 1991.

[469] T. Luba, J. Kalinowski, K. Jasinski, "PLATO - A Cad Tool for Logic Synthesis Based on Decomposition," *Proc. European Conference on Design Automation,* pp. 65-69, Amsterdam, February 1991.

[470] T. Luba, M.A. Markowski, B. Zbierzchowski, "Logic Decomposition for Programmable Gate Arrays," *Proc. of Euro-ASIC'92,* IEEE Computer Society Press, pp. 19-24, Paris 1992.

[471] T. Luba, J. Rybnik, "Algorithm for Elimination of Attributes and Arguments Based on Unate Complement Concept," *Bull. Pol. Ac. , Tech.,* No. 40, pp. 313-322, 1992.

D

[472] T. Luba, and J. Rybnik, "Rough Sets and Some Aspects in Logic Synthesis," In *Intelligent Decision Support - Handbook of Application and Advances of the Rough Sets Theory,* R.Slowinski (ed.), pp. 181-199, Kluwer Academic Publishers 1992.

[473] T. Luba, J. Rybnik, "Algorithmic Approach to Discernibility Function with Respect to Attributes and Object Reduction", *Int. Workshop on Rough Sets: State of the Art and Perspectives,* Poznan 1992.

[474] T. Luba, K. Gorski, L.B. Wronski, "ROM-Based Finite State Machines with PLA Address Modifiers," *Proc. European Design Automation Conf.*, pp. 272-277, 1992.

[475] T. Luba, H. Selvaraj, A. Krasniewski, "A New Approach to FPGA-based Logic Synthesis," *Workshop on Design Methodologies for Microelectronics and Signal Processing,* pp. 135-142, Gliwice-Cracow 1993.

[476] T. Luba, R. Lasocki, J. Rybnik, "An Implementation of Decomposition Algorithm and its Application in Information Systems Analysis and Logic Synthesis," *International Workshop on Rough Sets and Knowledge Discovery,* pp. 487-498, Banff 1993,

[477] T. Luba, J. Rybnik, "Algorithmic Approach to Discernibility Function with Respect to Attributes and Objects Reduction," *Foundation of Computing and Decision Sciences,* Vol. 18, No. 3-4, pp. 241-258, 1993.

[478] T. Luba, H. Selvaraj, A. Krasniewski, "A New Approach to FPGA-based Logic Synthesis", *Workshop on Design Methodologies for Microelectronics and Signal Processing,* Gliwice-Cracow, 1993.

[479] T. Luba, M. Mochocki, J. Rybnik, "Decomposition of Information Systems Using Decision Tables," *Bulletin of the Polish Academy of Sciences, Technical Sciences,* Vol. 41, No.3, 1993.

[480] T. Luba, R. Lasocki, "Decomposition of Multiple-valued Boolean Functions," *Applied Mathematics and Computer Science,* Vol.4, No.1, pp. 125-138, 1994.

[481] T. Luba,and R. Lasocki, "On Unknown Attribute Values in Functional Dependencies," *Proc. of The Third International Workshop on Rough Sets and Soft Computing,* pp.490-497, San Jose, 1994.

[482] T. Luba, "Decomposition of Multiple-Valued Functions," *Proc. 25th ISMVL*, pp. 256-261, 1995.

[483] T. Luba, "Multiple-Valued and Multiple-Level Decomposition and Its Applications in Information Systems Analysis and Logic Synthesis," *Report on Window-on-Science Project WOS-95-2155*, June 1995.

[484] T. Luba, M. Nowicka, "Balanced Multi-Level Decomposition and Its Applications in FPGA-based Synthesis," *Report*, Warsaw University of Technology, 1995.

[485] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM J. Comput.*, Vol. 15, No. 4., pp. 1036–1055, Nov. 1986.

[486] F. Luccio, " Reduction of the Number of Columns in Flow Table Minimization, " *IEEE Trans. on Electr. Computers,* Vol. EC-15, pp. 803-805, Oct. 1966.

[487] G. Luger, W. Stubbefield, "Artificial Intelligence. Structures and Strategies for Complex Problem Solving," *Benjamin/Cummings*, 1993.

[488] O.B. Lupanov, "On the Synthesis of Certain Classes of Control Devices," *Collection: Problems of Cybernetics,* in Russian, No. 10, 1961.

[489] E. Macii, T. Wolf, "Multiple Stuck-At Fault Test Generation Techniques for Combinational Circuits Based on Network Decomposition," *Proc. 36th Midwest Symposium*, pp. 465-467, 1993.

[490] V. Madhavan, "Approximation algorithm for maximum independent set in planar triangle-free graphs," *Foundations of software technology and theoretical Computer Science*, Proc. of 4th conference, Bangalore, India, Dec. 13-15, 1984, Joseph, M., Shyamasundar, R. (eds), Lecture Notes in Computer Science, No. 181, Springer Verlag, N.Y., 1984.

[491] C. Madre, Coudert, "New Ideas for Set Covering Problems," Proc. DAC 1995.

[492] K.K. Maitra, "Cascaded Switching Networks of Two-Input Flexible Cells," *IRE Trans. Electr. Comput.*, Vol. EC-11, pp. 136-143, Apr. 1962.

[493] G. Maley, J. Earle, "The Logic Design of Transistor Digital Computers," Englewood Cliffs, N.J. Prentice Hall, 1963.

[494] O. Maler, A.Pnueli, "Tight Bounds on the Complexity of Cascaded Decomposition of Automata," *Proc. of the 31t Annual Symposium on Foundation of Computer Science,* pp. 672-682., October 22-24 1990.

[495] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Proc. of. Int. Conf. on CAD*, pp. 6-9, 1988.

[496] R. Malvi, "Comparison of Exact and Approximate Coloring and Multi-Coloring Algorithms for Functional Decomposition," *M.S. Thesis,* Dept. EE., PSU, June 1997.

[497] R. Malvi, M. Perkowski, and L. Jozwiak, "Exact Graph Coloring for Functional Decomposition: Dow we Need it?," Accepted to Third Intern. Symposium on Boolean Problems, Freiberg, September 1998.

[498] B. Manvel, "Coloring Large Graphs," *Proc. of the 1981 S.E. Conf. on Graph Theory, Combin. and Computer Science,* 1981.

[499] B. Manvel, "Extremely Greedy Coloring Algorithms," in "Graphs and Applications", *Proc. of the FIrst Colorado Symposium on Graph Theory*, F. Harray, and J. Maybee, Eds. John Wiley, 1985.

[500] M.P. Marcus, "Derivation of Maximal Compatibles Using Boolean Algebra," *IBM J. Res.Dev.*, Vol 8, pp. 537–538, 1964.

[501] M. Marek-Sadowska, "Detecting Symmetric Variables in Boolean Functions using Generalized Reed-Muller Forms," *Proc. ISCAS'94*, pp. 287-290, 1994.

[502] P.N. Marinos, "Fuzzy logic and its application to switching systems", *IEEE Trans. on Comput.*, 18 (4), pp. 345-348, 1969.

[503] T. R. Martinez and D. M. Cambell. A self organizing binary decision tree for incrementally defined rule based systems. *IEEE Trans. on Systems, Man and Cybernetics*, 21, 1991.

[504] A. Maruoka, N. Honda, "Logical Networks of Flexible Cells," *IEEE Trans. Comput.*, Vol. C-22, pp. 347-358, Apr. 1973.

[505] A. Maruoka, N. Honda, "The Range of Flexibility of Tree Networks," *IEEE Trans. on Comput.*, Vol C-24, pp. 9-28, Jan. 1975.

[506] C. Matheus, "Feature Construction: An Analytic Framework and Application to Decision Trees," University of Illinois, PhD Dissertation, Urbana-Champaign, 1989.

[507] H.J. Mathony, "Zum Entwurf mehrstufiger Schaltnetze," *Interner Bericht* Bl-2.85, Univ. Karlsruhe, 1985, (in German).

[508] H.J. Mathony, U.G. Baitinger, "CARLOS: An Automated Multilevel Logic Design System for CMOS Semi - Custom Integrated Circuits," *IEEE Transaction on Computer-Aided Design,* Vol. 7, No. 3, March 1988.

[509] H.J. Mathony, "Algorithmische Entwurfsverfahren fuer Zwei- und Mehrstufige Schaltnetze," *Fortschritt-Berichte VDI,* Reihe 9, Nr. 79, VDI-Verlag, Duesseldorf 1988, (in German).

[510] D.W. Matula, "A Min-Max Theorem for Graphs with Application to Colouring," *J. of SIAM,* Review, 10., p.481, 1968.

[511] D.W. Matula, G. Marble, and J.D. Isaacson, "Graph Coloring Algorithms," *Graph Theory and Computing,* (R.C. Read, ed.), Academic Press, New York (1972), pp. 109-122.

[512] E. J. McCluskey, "Minimization of Boolean Functions," *Bell System Technical Journal,* Vol. 35, pp. 1417-1445, Nov. 1956.

[513] Ch. Meinel, J. Bern, and A. Slobodova, "Efficient OBDD-Based Boolean Manipulation in CAD Beyond Current Limits," *Proc. 32nd DAC,* San Francisco 1995.

[514] B. Melzer, "Dekompositorische Synthese Kombinatorischer Schaltungen mit Impliziter Testsatzberechnung," *Diplomarbeit,* TU Chemnitz, 1992, (in German).

D

[515] A.R. Meo, "Modular Tree Structures," *IEEE Trans. on Comput.*, Vol.17, 432-442, pp. 559-566, June 1968.

[516] R.D. Merril, "Symmetric Ternary Switching Functions: Their Detection and Realization with Threshold Logic," *IEEE Trans. on Electr. Comput.*, Vol. EC-16, No. 5, pp. 634-637,.

[517] J. Michael, "Validation, Verification, and Experimentation with Abacus 2," *Reports of Machine Learning and Inference Laboratory, MLI 91-8,* Center for Artificial Intelligence, George Mason University, Fairfax, VA, 1988.

[518] . A. Michalski, Poland

[519] R.S. Michalski, "Pattern Recognition as Knowledge-Guided Computer Induction," Technical Report No. 927, *Department of Computer Science, University of Illinois*, Urbana-Champaign, 1978.

[520] R.S. Michalski, "A Theory and Methodology of Inductive Learning," *Machine Learning: An Artificial Intelligence Approach*, R.S. Michalski and J.G. Carbonell and T.M. Mitchell, (eds.), Morgan Kaufmann, Los Altos, CA, 1983. pp. 93-134.

[521] R. Michalski, J. Carbonell, and T.M. Mitchell, *"Machine Learning: An Artificial Intelligence Approach," Morgan Kaufmann,* vol. 1, 1983, vol. 2. 1986.

[522] G.J. Minty, "On maximal independent sets of vertices in claw-free graphs," *J. Combin. Theory,* B., Vol. 28., pp. 284–304, 1980.

[523] R.E. Miller, "Switching Theory," Vol. 1 and 2, John Wiley, New York, 1965.

[524] D.M. Miller, "Decomposition in many-valued design," *Ph.D. Thesis*, Dept. of Computer Science, University of Manitoba, May 1976.

[525] R. Miller, "Teorija Pereklucatelnych Skhem, Vol. 1. Kombinacjionnyje Schemy," Moscow 1970 (in Russian).

[526] D.M. Miller, J.C. Muzio, "Two-place Decoposition and the Synthesis of Many-Valued Switching Circuits," *Proc. 6th ISMVL,* Logan, UT, pp. 164-168, May 1976.

[527] D.M. Miller, "The Fanout-Free Realization of Multiple-Valued Functions," *Proc. 11th ISMVL,* 1981, pp. 246–255.

[528] D.M. Miller, "Spectral Symmetry Tests," Proc. 11th ISMVL, 1981, pp. 130-134.

[529] Miller93, ISMVL.

[530] S. Minato, N. Ishiura, and S. Yajima, "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation," *Proc. 27th ACM/IEEE DAC,* pp. 52-57, 1990.

[531] S. Minato, "Graph-Based Representations of Discrete Functions," *Proc. Reed-Muller'95 Workshop*, Chiba, Japan, August 1995, pp. 1-10.

[532] R.C. Minnick, "A System of Magnetic Bubble Logic," *IEEE Trans. Comput.*, Vol. C-24, pp. 217-218, Febr. 1975.

[533] J. Mitchem, "On Various Algorithms for Estimating the Chromatic Number,"

[534] T.M. Mitchell and P.E. Utgoff and R. Banerji, "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics," *Machine Learning: An Artificial Intelligence Approach*, R.S. Michalski and J.G. Carbonell and T.M. Mitchell (eds), Morgan Kaufmann, Los Altos, CA, 1983.

[535] T. Mitchell, J. Carbonell, and R. Michalski, (eds), *Machine Learning: A Guide to Current Research. (Knowledge Representation, Learning, and Expert Systems).* Kluwer Academic Publishers, Nowell, MA, 1986.

[536] D. Mitchie, "Problem Decomposition and the Learning of Skills," *Proc. of the European Conference on Machine Learning, Lecture Notes in Artificial Intelligence*, Vol. 912, *Springer Verlag*, Berlin, Heidelberg, New York, Inc., pp. 17-31, 1995.

[537] S. Mohamed, M. Perkowski, and L. Jozwiak, "Fast Approximate Minimization of Multi-Output Boolean Functions in Sum-of-Product-Condition-Decoders Structures," Proc. Euro-Micro '97, September 1997.

[538] G.J. Montgomery, and K.C. Drake, "Abductive Networks", *SPIE Applications of Neural Networks Conference,* April 1990,

[539] J.W. Moon, and L. Moser, "On Cliques in Graphs," *Israel J. of Mathematics*, Vol. 3, pp. 23–28, March 1965,

[540] W.R. Moore, W. Luk, "FPGAs," *Abington EE and CS Books,* Abington, England, 1991.

[541] C. Moraga, "Spectral Analysis of Co-Symmetries", *Proc of 3rd Intern. Workshop of Spectral Techniques*, pp. 127-139, ISSN 0933-6192, Dortmund 1988.

[542] K. Morik, "Sloppy Modeling," in *Knowledge Representation and Organization in Machine Learning*, (K. Morik, ed.), *Springer-Verlag*, Berlin Heidelberg, 1989.

[543] Motorola MPA10XX Data Sheet, 1994.

[544] A. Muciek, "On Identification of Dimensional Function of Many Variables," *Proc. IEEE Instrumentation and Measurement Technology Conference,* pp. 209-212, 1991.

[545] K.D. Mueller, "Reduktion der Verlustleistung von CMOS-Schaltungen durch logisch-strukturelle Maenahmen," *Dissertation, Universitaet Karlsruhe,* 1977, (in German).

[546] A. Mueschwitzer, F. Rueler, "VLSI - Systeme," *VEB Verlag Technik,* Berlin 1988, (in German).

[547] S. Muggleton, "Duce, and Oracle-based Approach to Constructive Induction," *Proc. of IJCAI-87*, pp. 287-292, Milan, Italy, Morgan Kaufmann, 1987.

[548] S. Muggleton and W. Buntine, "Machine Invention of First Order Predictes by Inverting Resolution," *Proc. of the 5th International Conference on Machine Learning*, pp. 339-352, Ann Arbor, MI, Morgan Kaufmann, 1988.

[549] S. Muggleton and C. Feng, "Efficient induction of logic programs," *Proc. First Conf. on Algorithmic Learning Theory*, S. Arikawa, S. Goto, S. Ohsuga and T. Yokomori (eds), Tokyo, Japan, Soc. Art. Intell., pp. 368-381, 1990.

[550] S. Muggleton and L. D. Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 16(20):629–679, 1989.

[551] A. Mukhophadyay, "Symmetric Ternary Switching Functions," *IEEE Trans. on Comp.* Vol. EC-15, pp. 731-739, Oct. 1966.

[552] A. Mukhopadhyay, "Unate Cellular Logic," *IEEE Trans. Comput.*, Vol. C-18, pp. 114-121, Febr. 1969.

[553] A. Mukhopadhyay and Stone, "Cellular Logic," *Academic Press*, New York, 1971.

[554] G.D. Muligan, and D.G. Corneil, "Corrections to Bierstone's Algorithm for Generating Cliques," *JACM,* Vol. 19, No. 2, pp. 244–247, April 1972.

[555] D. E. Muller, "Application of Boolean Algebra to Switching Circuit Design and to Error Detection", *IRE Trans. on Electr. Comp.*, vol. EC-3, pp. 6-12, Sept. 1954.

[556] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, A. Sangiovanni - Vincentelli, "Logic Synthesis for Programmable Gate Arrays," *Proc. 27th ACM/IEEE Design Automation Conf.,* pp. 620-625, 1990.

[557] R. Murgai, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithm for Table Look Up Architectures," *Proc. IEEE Intern. Conf. on Computer Aided Design,* pp. 564-567, 1991.

[558] R. Murgai, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Performance Directed Synthesis for Table Look Up Programmable Gate Arrays," *Proc. ICCAD 1991*, Santa Clara, CA, Nov. 1991, pp. 572-575.

[559]

[560] R. Murgai, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Optimum Functional Decomposition Using Encoding," *Proc. 31st DAC*, pp. 408–414, June 1994.

[561] R. Murgai, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Optimum Functional Decomposition Using Encoding," *Proc. 31st DAC*, pp. 408–414, June 1994.

[562] R. Murgai, M. Fujita, F. Hirose, "Logic Synthesis for a Single Large Look-up Table," *Proc. ICCD'95*, pp. 415-424, 1995.

[563] D. Muroga, H.C. Lai, "Minimization of Logic Networks under a Generalized Cost Function," *IEEE Trans. on Comput.*, pp. 893-??, 1976.

[564] S. Muroga, "Logic Design and Switching Theory," *John Wiley and Sons,* 1979.

[565] J.C. Muzio, S.L. Hurst, "The Computation of Complete and Reduced Sets of Orthogonal Spectral Coefficients for Logic Design and Pattern Recognition Purposes," *Computers Electr. Engng.*, Vol. 5, pp. 231-249, 1978.

[566] J.C. Muzio, D.M. Miller, G.Epstein, "The Simplification of Multiple-Valued Symmetric Functions," *Proc. of the 13th Intern. Symp. on Multiple-Valued Logic*, pp. 111-119, May 1983.

[567] J.C. Muzio, T.C. Wesselkamper, "Multiple-Valued Switching Theory," *Adam Hilger,* Boston, MA, 1986.

[568] J.C. Muzio, "Concerning the Maximum Size of the Terms in the Realization of Symmetric Functions," *Proc. 20th Intern. Symp. on Multiple-Valued Logic,* May 1990, pp. 292-299.

[569] J. Neuschwander, J. Beister, "Uher die Ursachen des Realisierungsaufwands von Schaltfunktionen," *Informationstechnik it,* 28. Jg. No. 2, 1986, (in German).

[570] L. Nguyen, M.A. Perkowski and N.B. Goldstein, "PALMINI – Fast Boolean Minimizer for Personal Computers," Proc. 24th. ACM/IEEE Design Automation Conference, 1987, pp. 615–621.

[571] N.J. Nilsson, "Problem Solving Methods in Artificial Intelligence," *McGraw-Hill,* New York, 1971.

[572] E.A. Nordhous, and J.W. Gaddum, "On Complementary Graphs", *American Math. Monthly,* 63, p.175, 1956.

[573] C.B. Novikov, C.B. Oleksin, "Riealizacija Sistiemy Slabaopriedielionnych Bulievych Funkcji Bolszovo Czisla Pieremiennych Setiju iz PLM," *YC i M,* No. 2, 1984, (in Russian).

[574] M.J. Noviskey, "Correlation Partition Selection Algorithm," *Technical Report,* WL/AART-2, W/P AFB, OH 45433-7408, August 1992.

[575] M.J. Noviskey, "Row Identification for Function Decommposition in Pattern Theory," *Technical Report,* WL/AART-2, W/P AFB, OH 45433-7408, May 1993.

[576] M. J. Noviskey, T.D. Ross, D.A. Gadd, M. Axtell, "Application of Genetic Algorithms to Function Decomposition in Pattern Theory," *report WL-TR-94-1015,* 1994.

[577] B.R. Nyers and R. Lin , "A Lower Bound on the Chromatic Number of a Graph," *Networks,* Vol. 1., p. 273., 1972.

[578] A.L. Oliveira, and A. Sangiovanni-Vincentelli, "Constructive Induction Using a Non-Greedy Strategy for Feature Selection," *Proceedings on the Ninth International Conference on Machine Learning,* pp. 355-360, 1992.

[579] A.L. Oliveira, and A. Sangiovanni-Vincentelli, "Learning Complex Boolean Functions: Algorithms and Applications", *Implementation and Simulation - VLSI,* preliminary copy 6/10/93.

[580] A.L. de Oliveira, *"Inductive Learning by Selection of Minimal Complexity Representations,"* Ph.D. Thesis, University of California at Berkeley, Dec. 1994.

[581] A. Oliveira, 1994.

[582] G. Pagallo, and D. Hausler, "Boolean Feature Discovery in Empirical Learning", *Machine Learning,* Vol. 5, pp. 71-99, 1990.

[583] D. Pager, "Conditions for the Existence of Minimal Closed Covers Composed of Maximal Compatibles, " *IEEE Transaction on Computers,* Vol. C-20, pp. 450-452, 1971.

[584] A. Pal, "An Algorithm for Optimal Logic Design Using Multiplexers," *IEEE Trans. on Comput.,* Vol. C-35, No.8, August 1986.

[585] C.H. Papadimitriou, "The clique problem for planar graphs," *Inf. Proc. Letters,* Vol. 13, No. 4,5, pp. 138–141, December 1981.

[586] C.A. Papachristou, "Implementation of Discrete an Residue-Based Functions via Optimal Encoding: a PAL Approach," *IEEE Trans. on Comput.*, Vol. C-32, October 1983.

[587] C. Papachristou, "Characteristic Measures of Switching Functions," *Information Sciences,* Vol. 13, No. 1, pp.51-75, 1977.

[588] Ch. A. Papachristou and D. Sarma, "An Approach to Sequential Circuit Construction in LSI Programmable Arrays," *IEE Proceedings,* Vol. 130, Pt. E, No. 5, pp. 159-164, September 1983.

[589] M.C. Paul, and S.H. Unger, " Minimizing the Number of States in Incompletely Specified Sequential Switching Functions, " *IRE Trans. on Electron. Computers,* Vol. EC-8, pp. 356-367, 1959.

[590] P. Paulin, " Horizontal Partitioning of PLA-based Finite State Machines," *Proc. of the A.C.M./I.E.E.E. Design Automation Conference,* pp. 333-338, 1989.

[591] Z. Pawlak, "Rough Sets," *Theoretical Aspects of Reasoning about Data,* Kluwer Academic Publishers, Dordrecht, 1991.

[592] D. Patel, T. Luba, "Dependence Sets and Functional Decomposition of Boolean Functions," *Intern. J. of Electr.*, Vol. 75, No. 2, pp. 177-198, August 1993.

[593] D. Paulson, Y. Wand, "An Automated Approach to Information Systems Decomposition," *IEEE Trans. Soft. Engng.,* 18, pp. 174-189, 1992.

[594] J.Peck and M. Williams, "Examination Scheduling," *Communication of ACM,* Vol. 9, pp. 433–434, 1966.

[595] M. Perkowski, "Synthesis of Multioutput Three Level NAND Networks," *Proc. of the Seminar on Computer Aided Design,* IFAC - Intern. Federation of Control, pp. 238 - 265, Budapest, Hungary, 3-5 November 1976.

[596] M.A. Perkowski, A. Rydzewski, and P. Misiurewicz, "Theory of Logic Circuits. Selected Problems," *Publishers of the Technical University of Warsaw* (book in Polish).

[597] M.A. Perkowski, and N. Nguyen, "Minimization of Finite State Machines in System SuperPeg," *Proc. of the Midwest Symposium on Circuits and Systems,* Luisville, Kentucky, pp. 139-147, 1985.

[598] M. Perkowski, H. Uong, "Generalized Decomposition of Incompletely Specified Multioutput, Multi-Valued Boolean Functions," *Unpublished manuscript, Department of Electrical Engineering,* PSU 1987.

[599] M. Perkowski, J. Brown, "A Unified Approach to Designs with Multiplexers and to the Decomposition of Boolean Functions," *Proc.ASEE Annual Conference,* pp. 1610-1619, 1988.

[600] M. A. Perkowski, M. Helliwell, P. Wu, "Minimization of Multiple-Valued Input, Multi-Output Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions," *Proc. of the 19th ISMVL, International IEEE Symposium on Multiple-Valued Logic,* pp. 256 - 263, Guangzhou, People's Republic of China, May 1989.

[601] M.A. Perkowski, P. Wu, and K.A. Pirkl, "KUAI-EXACT: A New Approach for Multi-Valued Logic Minimization in VLSI Synthesis," *Proc. 1989 IEEE International Symposium on Circuits and Systems*, pp. 401–404, 1989.

[602] M.A. Perkowski, J. Liu, J.E. Brown, "Rapid Software Prototyping: CAD Design of Digital CAD Algorithms," In G. W. Zobrist (ed), *Progress in Computer-Aided VLSI Design,* Vol. 1, pp. 353-401, 1989.

[603] M.A. Perkowski, P. Dysko, and B.J. Falkowski, "Two Learning Methods for a Tree-Search Combinatorial Optimizer," *Proc. of IEEE Int. Conf. on Comput. and Comm.,* pp. 606-613, Scottsdale, Arizona, 1990.

[604] M. A. Perkowski, M. Chrzanowska-Jeske, and T. Shah, "Minimization of Multioutput TANT Networks for Unlimited Fan-In Network Model," *Proc. of ICCD'90,* pp. 360 - 363, Boston, MA, September 1990.

[605] M. A. Perkowski, and M. Chrzanowska-Jeske, "An Exact Algorithm to Minimize Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions," *Proc. ISCAS'90,* pp. 1652 - 1655, New Orleans, 1-3 May 1990.

[606] M. A. Perkowski, and J. Liu, "A Program for Exact Synthesis of Three-Level NAND Networks," *Proc. ISCAS'90,* pp. 1118 - 1121, New Orleans, 1-3 May 1990.

[607] M.A. Perkowski, and L.B. Nguyen, "An Encoding Program for Concurrent Finite State Machines Realized in Constrained Logic," *Proc. of the Midwest Symp. on Circuits and Systems,* Calgary, Alberta, Canada, 1990.

[608] M.A. Perkowski, and P. D. Johnson, "Canonical Multi-Valued Input Reed-Muller Trees and Forms", *Proc. 3rd NASA Symposium on VLSI Design,* Moscow, Idaho, pp. 11.3.1 - 11.3.13, October 1991.

[609] M. A. Perkowski, and A. Coppola, "A State Machine PLD and Associated Minimization Algorithms," *Proc. FPGA'92,,* pp. 109 - 114, Berkeley, February 16-18, 1992.

[610] M. A. Perkowski, "The Generalized Orthonormal Expansions of Functions with Multiple-Valued Inputs and Some of its Applications," *Proc. 22nd ISMVL,* pp. 442-450, Sendai, Japan, May 1992.

[611] M. Perkowski, L. Csanky, A. Sarabi, I. Schaefer, "Fast Minimization of Mixed-Polarity ADN/XOR Canonical Forms", *Proc. of the IEEE Intern. Conf. on Comp. Design,* ICCD'92, Boston, October 11-13, 1992, pp. 32-36.

[612] M. A. Perkowski, W. Zhao, and D. Hall, "Concurrent Two-Dimensional State Minimization and State Assignment of Finite State Machines," *Proc. VLSI Design '92,* pp. 80 - 84, Bangalore, India, January 4-7, 1992.

[613] M. A. Perkowski, "A Fundamental Theorem for for EXOR Circuits," *Proc. of IFIP W.G. 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design",* Hamburg, Germany, September 16-17, 1993.

[614] M. A. Perkowski, A. Sarabi, and F. R. Beyl, "XOR Canonical Forms of Switching Functions," *Proc. of IFIP W.G. 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design",* Hamburg, Germany, September 16-17, 1993.

[615] M. A. Perkowski, and M. Chrzanowska-Jeske, "Multiple-Valued-Input TANT Networks," *Proc. ISMVL'94,* pp. 334-341, Boston, MA, May 25-27, 1994.

[616] M.A. Perkowski, Ph. Ho, "Free Kronecker Decision Diagrams and their Application to Atmel 6000 Series FPGA Mapping," *Proc. Euro-DAC'94,* Grenoble, Sept. 19-23, 1994, France.

[617] M. Perkowski, "A Survey of Literature on Functional Decomposition, Version I," *Avionics Directoriate, Wrights Laboratories, Wright-Patterson Air Force Base,* Dayton, Ohio, U.S. Government Publication, 1995.

[618] M. Perkowski, S. Grygiel, and the Functional Decomposition Group, "A Survey of Literature on Functional Decomposition, Version IV," November 20, 1995, *Report to Wright Labs,* Portland State University, Department of Electrical Engineering.

[619] M. A. Perkowski, T. Ross, D. Gadd, J. A. Goldman, and N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," *Proc. of the Second Workshop on Applications of Reed-Muller Expansion in Circuit Design,* Chiba City Japan, 27 - 29 August 1995, pp. 102 - 109.

[620] Perkowski et al, "Development of the Search Strategies for MULTIS," *report*, 1995.

[621] Perkowski et al, "Encoding Problem in Boolean Decomposition," *report*, 1995.

[622] M. Perkowski, et al, "Documentation of MULTIS," December 29, 1995, *Report to Wright Labs,* Portland State University, Department of Electrical Engineering.

[623] M. Perkowski, T. Luba, S. Grygiel, M. Kolsteren, R. Lisanke, N. Iliev, P. Burkey, M. Burns, R. Malvi, C. Stanley, Z. Wang, H. Wu, F. Yang, S. Zhou, and J. S. Zhang, "Unified Approach to Functional Decompositions of Switching Functions," *PSU Report*, 1995.

[624] M. Perkowski and M. Burns and T. Luba and S. Grygiel and C. Stanley and R. Price and Z. Wang and J. Lu and P. Burkey and D. Manoharan and S. Mohammad, "Development of search strategies for MULTIS," *Portland State University,* Electrical Engineering Department, Portland, OR, December, 1995.

[625] M. Perkowski, T. Luba, S. Grygiel, M. Kolsteren, R. Lisanke, N. Iliev, P. Burkey, M. Burns, R. Malvi, C. Stanley, Z. Wang, H. Wu, F. Yang, S. Zhou, and J. S. Zhang, "Unified Approach to Functional Decompositions of Switching Functions," *PSU Report*, unpublished, Version III, September 1995.

[626] M. Perkowski, "Survey of Literature on Functional Decomposition," *Dept. Electr. Engn.*, report, 1995.

[627] M. Perkowski, "A New Representation of Strongly Unspecified Switching Functions and its Application to Multi-Level AND/OR/EXOR Synthesis," *Proc. Reed-Muller'95 Workshop,* Chiba, Japan, August 1995, pp. 143-151.

[628] M. Perkowski, A. Sarabi, and I. Schaefer, "Multi-Level Logic Synthesis Based on Kronecker and Boolean Ternary Decision Diagrams for Incompletely Specified Functions," *Computer Aided Design and Test,* Dagstuhl-Seminar-Report, 105, 13 Febr - 17 Febr, 1995.

[629] M. Perkowski, M. Burns, R. Almeria, N. Iliev, "Approaches to the Input-Output Encoding Problem in Boolean Decomposition," *Report to Wright Labs*, Portland State University, Department of Electrical Engineering, January 1996.

[630] M. A. Perkowski, I. Schaefer, A. Sarabi, and M. Chrzanowska-Jeske, "Multi-Level Logic Synthesis Based on Kronecker Decision Diagrams and Boolean Ternary Decision Diagrams for Incompletely Specified Functions," *Journal of VLSI Design.*

[631] M. Perkowski, M. Marek-Sadowska, T. Luba, S. Grygiel, P. Burkey, R. Malvi, Z. Wang, J.S. Zhang, and C. Stanley, "Fundamental Operations on Strongly Unspecified Multi-Valued Functions and Relations," *Report,* PSU, 1996.

[632] M. Perkowski, M. Marek-Sadowska, T. Luba, S. Grygiel, P. Burkey, R. Malvi, Z. Wang, J.S. Zhang, and C. Stanley, "Cube Diagram Bundles: A New Representation of Multi-Valued Relations," *Proc. ISMVL'97.*

[633] M. Perkowski and M. Marek-Sadowska and L. Jozwiak and T. Luba and S. Grygiel and M. Nowicka and R. Malvi and Zhi Wang and Jin S. Zhang, "Decomposition of multiple-valued relations," *Proc. of ISMVL'97*, Halifax, Nova Scotia, Canada, pp. 13-18, May 28-30, 1997.

bitemPerkowski97a M. Perkowski and S. Grygiel, "Multiple-Valued Incompletely Specified Functions and Relations. Part II: Decomposition," 1997.

[634] M. Perkowski, L. Jozwiak, and S. Mohamed, "New Approach to Learning Noisy Boolean Functions," *Proc. ICCIMA'98*, pp. 693-706, World Scientific.

[635] M. Perkowski, L. Jozwiak, and C. Files, "Optimization of Continuous-Input Sum of AND-OR-IMPLICANT Expressions for Data Mining Applications," Accepted to Third Intern. Symposium on Boolean Problems, Freiberg, September 1998.

[636] O. Persin, "An Algorithm for Determining the Minimum Coloring of a Finite Graph," *Engineering Cybernetics,* Vol. 11, pp. 980–985, 1973.

[637] G. Piatesky-Shapiro, Ch. Matheus, P. Smyth, and R. Uthurusamy, "KDD-93: Progress and Challenges in Knowledge Discovery in Databases," *Artificial Intelligence Magazine,* Vol. 15, No. 3, pp. 77-82, Fall 1994.

[638] E. Pichat, "Decomposition d'une Fonction de Variables a Nombres Finites de Valeurs," *Compt. Rend. H. Acad. Sci,* Vol. 268, ser. A, pp. 761-763, Nov. 18, 1969, (in French).

[639] E. Pichat, "Decomposition d'une Fonction de Plusiers Variables," *Compt. Rend. H. Acad. Sci,* Vol. 268, ser. A, pp. 1523-1526, June 23, 1969, (in French).

[640] E. Pierzchala, M.A. Perkowski, S. Grygiel, "A Field Programmable Analog Array for Continuous, Fuzzy, and Multi-Valued Logic Applications," *Proc. ISMVL '94*, pp. 148 - 155.

[641] D.A. Pospelov, "Analyse und Synthese von Schaltsystemen," *VEB Verlag Technik,* Berlin 1973, (in German).

[642] Post

[643] C. Posthoff, B. Steinbach, "Binaere Dynamische Systeme - Algorithmen und Programme," *Wissenschaftliche Schriftenreihe der T.H. Karl-Marx-Stadt,* (Chemnitz), 8, 1979, (in German).

[644] C. Posthoff, B. Steinbach, "Binaere Gleichungen - Algorithmen und Programme," *Wissenschaftliche Schriftenreihe der TH Karl-Marx-Stadt,* (Chemnitz) 1, 1979, (in German).

[645] Ch. Posthoff, D. Bochmann, K. Haubold, "Diskrete Mathematik," *Teubner-Verlag*, Leipzig, 1986, (in German).

[646] J. Poswig, "Disjoint decomposition," *Proc. of 3rd. International Workshop on Spectral Techniques,* Dept. Comp. Science, University of Dortmund, Dortmund 1988.

[647] J. Poswig, "Disjoint Decomposition of Boolean Functions," *IEE Proceedings,* Vol. 138, No. 1, pp. 48-56, January 1991.

[648] G.N. Povarov, "About Functional Separability of Boolean Functions", May 1954, (in Russian).

[649] G.N. Povarov, "On Functional Decomposability of Boolean Functions," *Doklady AN SSSR*, 94, No. 5, 1958.

[650] R.E. Prather, "Three Variable Multiple-output Tree Circuits," *IEEE Comput.*, Febr. 1965.

[651] R.E. Prather, "Introduction to Switching Theory: A Mathematical Approach," Boston, MA, *Allyn and Bacon*, 1967.

[652] F.P. Preparata, "On the Design of Universal Boolean Functions," *IEEE Trans. on Comput.*, Vol. C-20, pp. 418-423, April 1971.

[653] F.P. Preparata, "Universal Logic Modules of a New Type," *IEEE Trans. Comput.*, Vol. C-21, pp. 585-588, June 1972.

S

[654] E.I. Pupyrev, "Realization of Control Algorithms in a Rearrangeable Automata Basis," *Automation and Remote Control,* USSR, Vol. 50, No. 3, pp. 411-421, 1989.

[655] L. Pyber, "Clique covering of graphs," *Combinatorica*, Vol. 6, No.4, pp. 393–398, 1986.

[656] Z.W. Pylyshyn, "Computation and cognition," *MIT Press*, Cambridge, MA, 1984.

[657] J. R. Quinlan, "Induction of Decision Trees," *Machine Learning*, No. 1, pp. 81-106, 1986.

[658] J. R. Quinlan, *"C4.5: Programs for Machine Learning",* Morgan Kaufmann, 1993, Palo Alto, Ca.

[659] J.R. Quinlan, "Comparing connectionist and symbolic learning methods," *S.J. Hanson and G.A. Drastal and R.L. Rivest*, Volume I: Constraints and Prospects, Computational Learning Theory, *MIT Press*, chpt. 15, pp. 445-456, 1994.

[660] C.V.S. Rao, and N.N. Biswas, "Minimization of Incompletely Specified Sequential Machines, " *IEEE Trans. on Computers,* Vol. C-24, No. 11, pp. 1089-1100, Nov. 1975.

S

[661] R.C. Read, "Graph Theory and Computing," *Academic Press*, New York, 1972.

[662] I.S. Reed, "A Class of Multiple-Error-Correcting Codes and the Decoding Scheme", *IRE Transactions on Information Theory,* Vol. IT-4, pp. 38-49, September 1954.

[663] E.M. Reingold E.M., J. Nievergelt, and N. Deo, "Combinatorial Algorithms: Theory and Practice," *Prentice-Hall*, 1977.

[664] N.P Redkin, "Avtomatika i Telemekhanika," No. 8, pp. 84-88, (in Russian), 1970.

[665] L. Rendell, "Substantial Constructive Induction Using Layered Information Compression: Tractable Feature Formation in Search," *Proc. of IJCAI-85*, pp. 650-658, 1985.

[666] D.C. Rine, "Computer Science and Multiple-Valued Logic," North Holland, 1977.

[667] T.K. Rosenfeld, V.N. Silayev, "Boolean Equations and Decomposition of Boolean Functions," *Engineering Cybernetics,* Vol. 17, No. 1, pp. 85-92, 1979.

[668] W. Rosenstiel (Ed.), *Proceedings of the IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design,* Hamburg, Germany, September 1993.

[669] T.D. Ross and A.V. Lair, "Definition and Realization in Pattern Recognition System Design," In *Proc. of 1987 IEEE Int. Conf. on Systems, Man and Cybernetics,* pp. 744-748, 1987.

[670] T.D. Ross, "Pattern Representation and Recognition," *Research Prospectus,* Air Force Institute of Technology. 1988.

[671] T.D. Ross, "Elementary Theorems in Pattern Theory," *Ph.D. Thesis,* Air Force Institute of Technology. 1988.

[672] T.D. Ross and A.V. Lair, "On the Role of Patterns in Recognizer Design," In *Josef Kittler, ed. Pattern Recognition,* pp. 193-202, Springer-Verlag, New York, 1988.

[673] T. D. Ross, M.J. Noviskey, T.N. Taylor, D.A. Gadd, "Pattern Theory: An Engineering Paradigm for Algorithm Design," *Final Technical Report WL-TR-91-1060,* Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543, August 1991.

[674] T.D. Ross, M.L. Axtell, M.J. Noviskey, M. Breen, "A Demonstration of a Robust Occam-Based Learner," In *Proc. IEEE Int. Symp. on Information Theory,* 1992.

[675] T.D. Ross, "Function Decomposition Strategy for the Function Learning and Synthesis Hotbed," *Technical Memorandum WL-TM-92-110,* Wright Laboratory, USAF, WL/AART, WPAFB, OH 45433-6543, August 1992.

[676] T.D. Ross, M.J. Noviskey, M.L. Axtell, D.A. Gadd, "Flash Software Description," *Technical report, Wright Laboratory,* USAF, WL/AART, WPAFB, OH 45433-6543, December 1993.

[677] T.D. Ross, M.J. Noviskey, M.L. Axtell, D.A. Gadd, "Flash user's guide," *Technical report, Wright Laboratory,* USAF, WL/AART, WPAFB, OH 45433-6543, December 1993.

[678] T.D. Ross, M.L. Axtell, M.J. Noviskey, "Logic Minimization as a Robust Pattern Finder," *Intern. Workshop on Logic Synthesis,* Lake Tahoe, CA, May 23-26, 1993.

[679] T.D. Ross, M.L. Axtell, M.J. Noviskey, D.A. Gadd, "Pattern Theory Paradigm for System Design," In *Proc. 36th Midwest Symposium on Circuits and Systems,"* 1993.

[680] T.D. Ross, J.A. Goldman, M.J. Noviskey, M.L. Axtell, D.A. Gadd, "Graph Coloring for Column Multiplicity: A Survey," WL WPAFB report, April 21, 1994.

[681] T.D. Ross, M.J. Noviskey, M.L. Axtell, D.A. Gadd, J.A. Goldmann, "Pattern Theoretic Feature Extraction and Constructive Induction," *Technical report,* Wright Laboratory, USAF, WL/AART, WPAFB, OH 45433-6543, April 22, 1994.

[682] T.D. Ross, M.J. Noviskey, J.A. Goldmann, D.A. Gadd, "On the Decomposition of Real-Valued Functions," *In Third Intern. Workshop on Post-Binary VLSI Systems,* 1994.

[683] T.D. Ross, "Variable Partition Search for Function Decomposition," *Technical report, Wright Laboratory,* USAF, WL/AARA-3, WPAFB, OH 45433-6543, November 1994.

[684] T.D. Ross, J.A. Goldman, D.A. Gadd, M.J. Noviskey, and M.L. Axtell, "On the Decomposition of Real-Valued Functions", *"Third International Workshop on Post-Binary ULSI Systems in affiliation with the Twenty-Fourth International Symposium on Multiple-Valued Logic",* 1994.

[685] J.P. Roth, "Minimization over Boolean Trees," *IBM Journal.* 4, 5, pp. 543-555, 1960.

[686] J.P. Roth, R.M. Karp, "Minimization Over Boolean Graphs," *IBM Journal Res. and Develop.* No.4, pp. 227-238, April 1962.

[687] J.P. Roth and E.G. Wagner, "Algebraic Topological Methods for the Synthesis of Switching Systems. Part III: Minimization of Non-Singular Boolean Trees," *IBM J. Res. Develop.* Vol. 3, Oct. 1962.

[688] D. Rottem, and J. Urrutia, "Finding maximum cliques in circle graphs," *Networks II,* Vol. 3., pp. 269–278, Fall 1981.

[689] R.L. Rudell, "Espresso IIC User's Manual," CAD Toolbox User's Manual, UC Berkeley, Jan. 1984.

[690] R.L. Rudell, and A. Sangiovanni–Vincentelli, "Espresso MV: Algorithms for Multiple-Valued Logic Minimization," *Proc. IEEE Custom Integr. Circ. Conf.,* 1985, pp. 230–234.

[691] R.L. Rudell, and A. Sangiovanni–Vincentelli, "Espresso MV: Algorithms for Multiple-Valued Logic Minimization," *Proc. IEEE Custom Integr. Circ. Conf.,* 1985, pp. 230–234.

[692] R.L. Rudell and A.L. Sangiovanni-Vincentelli, "Exact Minimization of Multiple–Valued Functions for PLA Optimization," Digest of Technical Papers, *IEEE Int. Conf. on Computer-Aided Design,* pp. 352-355, 1986.

[693] R.L. Rudell, *Multiple-Valued Logic Minimization for PLA Synthesis,* Masters Report, University of California, Berkeley, 1986.

[694] S

[695] R.L. Rudell, and A.L. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *Proc. Intern. Symp. on Multiple-Valued Logic,* pp. 198-208, May 26-28, Boston, MA, 1987.

[696] R.L. Rudell and A. Sangiovanni–Vincentelli, "Multiple-Valued Minimization for PLA Optimization," *IEEE Transactions on Computer–Aided Design,* Vol. CAD-6, No. 5, Sept. 1987, pp. 727–750.

[697] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *Intern. Workshop on Logic Synthesis,* pp. 3a 1-12, 1993.

[698] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *Proc. Int. Conf. on Computer Aided Design*, pp. 42-47, 1993.

[699] J. Rybnik, "Minimization of Partially Defined Switching Functions Using Rough Sets Theory," *Manuscript, Institute of Telecommunication*, Warsaw Technical University, 1990, (in Polish).

[700] A. Saldanha, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli, "Satisfaction of Input and Output Encoding Constraints," *IEEE Trans. on CAD,* Vol. 13, No. 5, pp. 589 - 602, May 1994.

[701] P. Sapiecha, M. A. Perkowski, and T. Luba, "Decomposition of Information Systems Based on Graph Coloring Heuristics," *Proc. Symposium on Modelling, Analysis and Simulation,* CESA'96 IMACS Multiconference, Lille, France, July 9-12, 1996.

[702] C.E. Shannon, and W. Weaver, "The Mathematical Theory of Communication," *University of Illinois Press*, 1975 (first published in 1949).

[703] V.Y. Shen, and A. C. McKellar, "An Algorithm for the Disjunctive Decomposition of Switching Functions," *IEEE Trans. on Comput.*, Vol. C-19, pp. 239-248, March 1970.

[704] V.Z. Shen, J.D. Huang, and S.M. Chao, "Lambda Set Selection in Roth-Karp Decomposition for LUT-Based FPGA Technology Mapping," *Proc. 32 DAC*, June 1995, pp. 65–69.

[705] A.A. Shneider, *Kibernetika,* No.4, pp. 15-22, 1984, (in Russian).

[706] B.N. Shneyder, "Algebra of Sets Applied to Solution of Graph Theory Problems," *Engineering Cybernetics*, Plenum Press p. 521, 1970.

[707] M.I. Shvarcman, *Avtomatika i Vycislitelnaya Tekhnika,* No. 6, pp. 12-17, 1981, (in Russian).
D

[708] P. Sicard, M. Crastes, K. Sakouti, and G. Saucier, "Automatic Synthesis of Boolean Functions on Xilinx and Actel Programmable Devices," *Proc. Euro ASIC '91,* pp. 142-145, 1991.

[709] H.A. Simon, A. Ando, "Aggregation of Variables in Dynamic Systems," *Econometrica,* 29, pp. 111-138, 1961.

[710] T. Singer, "The Decomposition Chart as a Theoretical Aid", *Harvard Computational Lab.,* Cambridge, Mass., Rept. BL-4, 1953.

[711] P.K. Sinha Roy, and C.L. Sheng, "A Decomposition Method of Determining Maximum Compatibles," *IEEE Trans. Computers*, Vol. C-21, pp. 309–312, 1972.

[712] J. Sklansky, "General Synthesis of Tributary Switching Networks," *IEEE Trans. on Electr. Comput.* Vol. EC -12, pp. 464-469, 1963.

[713] A. Skowron, C. Rauszer, "The Discernibility Matrices and Functions in Information Systems," *Res. Rep. 1/91,* Inst. Comp. Sci., Warsaw, 1991.

[714] Slagle, J.R., C.L. Chang, R.C.T. Lee, "A New Algorithm for generating Prime Implicants," IEEE TEC, Vol.EC-19, pp.304-311 April 70.

[715] N. Song and M. A. Perkowski, "EXORCISM-MV-2: Minimization of Exclusive Sum of Products Expressions for Multiple-Valued Input Incompletely Specified Functions," *Proc. ISMVL '93,* pp. 132-137, Sacramento, CA, May 24-27 1993.

[716] B.L. Sorokin, *Avtomatika i Vycislitelnaya Tehnika,*, No. 4, pp. 50-55, 1982, (in Russian).

[717] M. Stankovic, Z. Tosic, S. Nikolic, "Synthesis of Maitra Cascades by Means of Spectral Coefficients," *IEE Proc.* Vol. 130, Pt. E, No. 4, pp. 101-108, July 1983.

[718] R.S. Stankovic, "Matrix Relations for Symmetry Properties of Boolean Function Detection," *Proc. of XXXIII Yugoslav Conference on ETAN*, Novi Sad, (in Serbian), 1989.

[719] R.S. Stankovic, C. Moraga, M. Stankovic, K. Rangelov, "Simple Disjoint Decomposition of Multiple-Valued Functions," Research Reports in Applied Mathematics, Series: Multiple-Valued Logic, YU Report No. 4.

[720] R.S. Stankovic, N. Denic, "An Approach to the Detection of Symmetry and Co-Symmetry Properties of Switching Functions," Research Reports in Applied Mathematics, Series: Multiple-Valued Logic, Report No. 5.

[721] R.S. Stankovic, and C. Moraga, "Methods for Detection of Some Properties of Multiple-Valued Functions," *IEE Proceedings E,* Vol. 139, No. 5, Sept. 1992, pp. 421-429.

[722] S.M. Starobinets, "On an Algorithm for Finding the Greatest Internally Stable Sets of a Graph," *Engineering Cybernetics '73.*, Plenum Press, p. 873., 1973.

[723] B. Steinbach, "Theorie, Algorithmen und Programme fuer den Rechnergestaetzten Logischen Entwurf Digitaler Systeme," *Habilitation,* T.H. Karl-Marx-Stadt/Chemnitz 1983, (in German).

[724] B. Steinbach, T.Q. Le, "Tools fuer den Logikentwurf," *Wiss. Schriftenreihe der T.U. Karl-Marx-Stadt (Chemnitz)*, No. 9, 1988, (in German).

[725] B. Steinbach, R. Hilbert, "Schnelle Testdatengenerierung' Gestuetzt auf den Booleschen Differentialkalkuel," in: *Bochmann, D.; Uhar, R.: Fehler in Automaten. Verlag Technik,* Berlin, 1989, (in German).

[726] B. Steinbach, J. Fehmel, "Ausnutzung von Funktionseigenschaften beim Dekompositorischen Logikentwurf," *Proc. of 3rd "Tagung Schaltkreisentwurf"*, Tagungsmaterialien, Dresden, 1989, (in German)

[727] B. Steinbach, T.Q. Le, "Entwurf testbarer Schaltungen," *Wiss. Schriftenreihe der TU Chemnitz,* No. 12, 1990.

[728] B. Steinbach, "XBOOLE - A Toolbox for Modeling, Simulation and Analysis of Large Digital Systems," *System Analysis and Modelling, Gordon and Breach Science Publishers,* Vol. 9, No. 4, pp. 297-312, 1992.

[729] B. Steinbach, F. Schumann, M. Stoeckert, "Functional Decommposition of Speed Optimized Circuits," In *Auvergne, D., Hartenstein, R. (ed) "Power and Timing Modeling for Performance of Integrated Circuits,"* Proc. of the Third Intern, Workshop on Power and Timing Modeling and Optimization, IT Press Verlag, Bruchsal, 1993, pp. 65-77.

[730] B. Steinbach and M. Stoeckert, "Design of Fully Testable Circuits by Functional Decomposition and Implicit Test Pattern Generation," *Proc. of 12th IEEE VLSI Test Symposium*, Cherry Hill, New Jersey, 1994.

[731] B. Steinbach, and A. Wereszczynski, "Synthesis of Multi-Level Circuits Using EXOR-Gates," *Proc. Reed-Muller'95 Workshop,* Chiba, Japan, August 1995, pp. 161-168.

[732] K.E. Stoffers, "Partitioning of Separating Edges - New Approach to Combinational Logic Design," *IEEE Trans. on Comput.*, Vol. 26, No. 8, pp. 833-836, 1977.

[733] , I. Stojmenovic, M. Miyakawa, and R. Tosic, "On Spectra of Many-Valued Logic Symmetric Functions," *Proc. 18th ISMVL,* Palma De Mallorca, pp 285-292, May 24-26, 1988.

[734] S.Y.H. Su, and P.T. Cheung, "Computer Minimization of Multivalued Switching Functions," *IEEE TC*, Vol. 21, No. 9, pp. 995-1003, Sept 1972.

[735] B.A. Subbotovskaya, "Comparison of Bases in the Realization of Boolean Functions by Formulas," DAN SSSR, 149, No. 4, 1963.

[736] A. Suzuki, *Electr. Co. J.,* 52, pp. 121-?, 1969.

[737] A. Suzuki, *Electr. Co. J.,* 52, pp. 132-?, 1969.

[738] Svoboda, A., "The Concept of Term Exclusiveness and Its Effect on the Theory of Boolean Functionss," Journ. of the Assoc. for Comp. Mach., Vol. 22, No. 3, July 1975, pp. 425-440.

[739] *Synopsys Design Compiler V 1.3h.* Manual 1991.

[740] M.M. Syslo, N.Deo, and J.S. Kowalik, "Discrete Optimization Algorithms with Pascal Programs," *Prentice Hall,* Englewood Cliffs, N.J., 1983.

[741] Sakawa, Masatosh, "Fuzzy Sets and interactive multiobjective optimization," *Plenum Press,* 233 Spring Street, New York, NY 10013, 1993.

[742] A. Sangiovanni-Vincentelli, A. Gamal, J. Rose, "Synthesis Methods for Field Programmable Gate Arrays," *Proc. IEEE,* Vol.81, No.7, pp. 1057-1083, 1993.

[743] E.S. Santos, "Maxmin Automata," *Information and Control,* pp. 363-377, 1968.

[744] J. Santos, H. Arango, M. Pascual, G. Roing, "A Cyclic Algebra for the Synthesis of Ternary Digital Systems," *IEEE Trans. on Comput.*, Vol. C-19, pp. 651-653, July 1970.

[745] A.A. Sapozenko, L.M. Karahanijan, *Avtomatika i Vycislitelnaya Tekhnika,* No. 3, pp. 28-35, 1981, (in Russian).

[746] A. Sarabi, M.A. Perkowski: "Fast Exact and Quasi-Minimal Minimization of Highly Testable Fixed-Polarity AND/XOR Canonical Networks", *Proc. of the IEEE/ACM Design Automation Conference,* Anaheim, CA, June 8-12, 1992, pp. 30-35.

[747] A. Sarabi, P.F. Ho, K. Iravani, W.R. Daasch, and M. Perkowski, "Minimal Multi-level Realization of Switching Functions based on Kronecker Functional Decision Diagrams," *Proc. IWLS '93, Intern. Workshop on Logic Synthesis*, 1993.

[748] A. Sarabi, M.A. Perkowski, "Cube Based Method for Optimal and Quasi-Optimal Minimization of Consistent Generalized Reed-Muller Expansions", *submitted to IEEE Transactions on Computers,* 1994.

S

[749] T. Sasao, "Multiple-Valued Decomposition of Generalized Boolean Functions and the Complexity of Programmable Logic Arrays," *IEEE Trans. on Comput.*, Vol. 30, No. 9, pp. 635-643, 1981.

[750] T. Sasao, "Input Variable Assignment and Output Phase Optimization of PLA's," *IEEE Trans Comput.* Vol. C-33. pp. 879-894, 1984.

S

[751] T. Sasao, "An Algorithm to Derive the Complement of a Binary Function with Multiple-Valued Inputs," *IEEE Transactions on Computers,* Vol. C-34, No. 2, Feb. 1985, pp. 131–140.

S

[752] T. Sasao, "MACDAS: Multi-Level AND-OR Circuit Synthesis Using Two-Variable Function Generators," *Proc. 23rd Design Automation Conference*, 1986, pp. 86-93.

[753] T.Sasao, "Functional Decomposition of PLAs", *Proc. of the Intern. Workshop on Logic Synthesis,* Research Triangle Park, North Carolina, May 12-15, 1987.

[754] T. Sasao, "An application of multiple-valued logic to a design of Programmable Logic Arrays," *Proc. 8th Intern. Symp. on Multiple-Valued Logic (ISMVL),* 1978.

[755] T. Sasao, "Bounds on the average number of products in the minimal Sum-of-Products expressions for multiple-valued input two-valued output functions," *Proc. Intern. Symp. on Multiple-Valued Logic,* pp. 260-267, May 26-28, Boston, MA, 1987.

[756] T.Sasao, "Aplication of Multiple-Valued Logic to a Serial Decomposition of PLAs", *Proc. of the Intern. Symp. on Multiple-Valued Logic,* Zangzou, China, pp. 264-271, May 1989.

[757] T. Sasao, and Ph. Besslich, "On the Complexity of MOD-2 Sum PLAs," *IEEE Trans. on Comput.*, Vol. 39, No. 2, pp. 262-266, 1990.

[758] T. Sasao (ed.), "Logic Synthesis and Optimization," *Kluwer Academic Publishers,* 1993.

[759] T. Sasao (ed.), "An Exact Minimization of AND-EXOR Expressions Using BDDs," *in Proc. of IFIP Work. on Appl. of RM Exp. in Circ. Des.,* Hamburg, pp. 91-99, 1993.

[760] T. Sasao, "FPGA Design by Generalized Functional Decomposition," in "Logic Synthesis and Optimization, " T. Sasao. (Ed), Kluwer Academic Publishers, pp. 233-258, 1993.

[761] T. Sasao, "Exmin2: A simplification algorithm for exclusive-OR sum of products expressions for multiple-valued-input two-valued output functions," *IEEE Tr. on CAD*, Vol. 12, No. 5, pp. 621–632, 1993.

[762] D. Sass, H. Warmers, E.-H. Horneber, "Waveform Estimation and Hazard Detection in Digital MOS Circuits." *Proc. of the Ist Int. Conf. on Microelectro, Opto, Mechanic Systems and Components,* Berlin, Sept. 1990.

[763] G. Saucier, P. Sicard, L. Bouchet, "Multi-Level Synthesis on PALs," *Proc. of the European Conference on Design Automation,* Glasgow, pp. 542-546, March, 1990.

[764] H. Sawada, T. Suyama, and A. Nagoya, "Logic Synthesis for Look-Up Table Based FPGAs Using Functional Decomposition and Support Minimization," *Proc. 1995 ICCAD*, pp. 353–358. 1995.

[765] P. Sawkar, D. Thomas, "Area and Delay Mapping for Table-Lookup Based Field Programmable Gate Arrays," *Proc. of 29th ACM/IEEE Design Automation Conf.,* pp. 368-373, 1992.

[766] I. Schaefer Ph.D. Thesis, Portland State University, 1992.

[767] I. Schaefer, M.A. Perkowski, "Multiple-Valued Input Generalized Reed-Muller Expansions", *IEE Proceedings, Pt.E,* Vol. 139, No. 6, pp. 519-527, Nov. 1992.

[768] I. Schaefer, B.J. Falkowski, M.A. Perkowski, "Generation of Adding and Arithmetic Multi-Polarity Transforms for Incompletely Specified Boolean Functions", *Intern. J. of Electr.*, Vol. 73, No. 2, pp. 321-331, 1992.

[769] I. Schaefer, M. Perkowski, "Extended Spectral Techniques for Logic Synthesis", Submitted book chapter, *"Formal Methods in Spectral Analysis",* prof. Radomir Stankovic, Editor, 1992.

[770] I. Schaefer, M. A. Perkowski, and H. Wu, "Multilevel Logic Synthesis for Cellular FPGAs Based on Orthogonal Expansions," *Proceedings of the IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design,* Hamburg, Germany, September 1993.

[771] I. Schaefer, and M.A. Perkowski, "Synthesis of Multi-Level Multiplexer Circuits for Incompletely Specified Multi-Output Boolean Functions with Mapping Multiplexer Based FPGAs", *IEEE Trans. on Computer Aided Design,* Vol. 12, No. 11, Nov. 1993, pp. 1655 - 1664.

[772] I. Schaefer, M. Perkowski, "On the Minimal Multiple-Valued Input Kronecker Reed-Muller Form for Incompletely Specified Multiple-Valued Input Functions", *Submitted to IEEE Trans. on Comput.* , 1994.

[773] R. Scheuring, H. Wehlan, "On the Design of Discrete Event Dynamic Systems by Means of the Boolean Differential Calculus," *Prepr. of the IFAC Symposium: Design Methods of Control Systems,* Zuerich 1991, (in German).

[774] U. Schlichtmann, "Boolean Matching and Disjoint Decomposition for FPGA Technology Mapping, *IFIP Workshop on Logic and Architecture Synthesis,* pp. 83–102, 1993.

[775] J.C. Schlimmer, "Learning and Representation Change," *Proceedings of AAAI-87*, pp. 511-515, Morgan Kaufmann, 1987.

[776] Schmidt, D., C., Druffel, L.E., "An Extension of the Clause-Table Approach to Multi-Output Combinatorial Switching Networks," *IEEE TC,* Vol. C-23, April 1974.

[777] P.R. Schneider, D.L. Dietmeyer, "An Algorithm for Synthesis of Multiple-Output Combinational Logic," *IEE Trans. on Comput.*, pp. 117-128, Febr. 1968.

[778]) Schneider, new 1994 paper about improving power consumption by decompositional methods.

[779] C. Scholl, and P. Molitor, "Efficient ROBDD based computation of common decomposition functions for multioutput Boolean functions,“ In G. Saucier and A. Mignotte, editors, *Novel Approaches in Logic and Architecture Synthesis,* pp. 57 - 63, Chapman and Hall, 1995.

[780] E. Schubert, U. Kebschull, and W. Rosenstiel, "FDD Based Technology Mapping for FPGA," *Proc. EURO-ASIC*, pp. 14-18, Paris, France, June 1992.

[781] G.W. Schwede, and A. Kandel, "Fuzzy maps", *IEEE Trans. Syst. Man Cybernet.,* Vol. 7, No. 9, pp. 699-674, 1977.

[782] S.M. Selkow, "New bounds for the clique number of a graph," *Inf. Proc. Letters,* Vol. 7., No. 4., pp. 170–172, June 1978.

[783] H. Selvaraj, A. Czerczak, A. Krasniewski, T. Luba, "A Generalized Decomposition of Boolean Functions and its Application in FPGA-Based Synthesis," *IFIP Workshop on Logic and Architecture Synthesis,* pp. 147-166, Grenoble 1993.

[784] H. Selvaraj, "FPGA-Based Logic Synthesis," *Ph.D. Dissertation,* Warsaw University of Technology, 1994.

[785] W.L. Semon, "Characteristic Numbers and Their Use in the Decomposition of Switching Functions," *Proc. ACM,* Vol. 17, pp. 273-280, May 1952. (In Curtis'62).

[786] C.E. Shannon, "A symbolic analysis of relay and switching circuits," *Trans. AIEE 57,* pp. 713-723, 1938.

[787] C.E. Shannon, "The Synthesis of Two-Terminal Switching Circuits," *The Bell System Technical Journal,* 28, pp. 59-98, 1949.

[788] V.Y. Shen, "On Simple Disjunctive Decompositions of Switching Functions," Ph.D. thesis, *Dept. Electr. Engng., Princeton University,* N.J., January 1969.

[789] V.Y. Shen, A. C. McKellar, and P. Weiner, "An Fast Algorithm for the Disjunctive Decomposition of Switching Functions," *IEEE Trans. on Camput.*, Vol. C-20, No. 3, pp. 304-309, March 1971.

[790] H.A. Simon, "The Sciences of the Artificial," Cambridge, MA, *MIT Press,* 1981.

[791] A. Skowron, and C. Rauszer, "The Discernibility Matrices and Functions in Information Systems," *In Intelligent Decision Support - Handbook of Application and Advances of the Rough Sets Theory,* R.Slowinski (ed), *Kluwer Academic Publisher*, 1992.

[792] R.J. Solomonoff, "A formal theory of inductive inference, part 1 and part 2," *Inform. Contr.*, Vol. 7, pp. 1-22, 224–254, 1964.

[793] R. J. Solomonoff, "Complexity Based Induction Systems: Comparisons and Convergence Theorems," *Rockford Research*, Report RR-329, Cambridge, MA, August, 1976.

[794] R. J. Solomonoff, "Complexity Based Induction Systems: Comparisons and Convergence Theorems," *IEEE Trans. Inform. Theory,* Vol. IT-24, pp. 422-432, 1978.

[795] N. Song, and M.A. Perkowski, "Minimization of Exclusive Sum of Products Expressions for Multi-Output Multiple-Valued Input Switching Functions," *IEEE Trans. on CAD.*

[796] P. Szolgay, "On Algorithms in the Parallel design of Logic and Layout of Circuits with Functional Blocks", *Intern. J. of Circuit Theory and Applic.*, Vol.20, No.4, pp.411-429, 1992. TK454 .I58

[797] T. Stanion, and C. Sechen, “A Method for Finding Good Ashenhurst Decomposition and Its Application to FPGA Synthesis,” *Proc. 32nd DAC,* pp. 60 - 64, San Francisco, 1995.

[798] R.E. Stearns, and J. Hartmanis, “On the State Assignment Problem for Sequential Machines, II,” *IRE Trans. Electr. Comp.,* Vol. EC-10, No. 4, pp. 593-603, December 1961.

[799] K.E. Stoffers, “Sequential Algorithm for the Determination of Maximum Compatibles,” *IEEE Trans. on Computers,* pp. 95-98, January 1974.

[800] J.R. Story, H.J. Harrison, and E.A. Reinhard, “Optimum State Assignment for Synchronous Sequential Circuits,” *IEEE Trans. Comp.,* Vol. C-21, No. 12, p. 1365-1373, December 1972.

[801] T.F. Tabloski, F. J. Mowle, “A Numerical Expansion Technique and Its Application to Minimal Multiplexer Logic Circuits,” *IEEE Trans. on Comput.,* Vol. 25., No. 7, pp. 684-702, 1976.

[802] R.E. Tarjan, and A.E. Trojanowski, “Finding a maximum independent set,” SIAM J. Comput., Vol. 6, pp. 537–46, 1977.

[803] A. Tehrani, “Un Algorithme de Coloration,” *Cahiers Centre Etudes Recherche Oper.,* Vol. 17, pp. 395-398, 1975.

[804] A.K. Teslenko, *Vestnik Kijev politehniceskogo Instituta, seria avtomatiki i elektropriborostroenija,* coll. 12, pp. 59-61, 1975, (in Russian).

[805] A. Thayse, *Phil. Res. R.* Vol. 26, pp. 229, 1971.

[806] A. Thayse, “A Fast Algorithm for the Proper Decomposition of Boolean Functions,” *Philips Res. Reports,* No. 27, 1972.

[807] A. Thayse, and M. Davio, “Boolean Differential Calculus and its Application to Switching Theory,” *IEEE Trans. on Comput.*, Vol. C-22, No. 4, pp. 409-420, 1973.

[808] A. Thayse, “Boolean Differential Calculus and its Application to Switching Theory,” *IEEE Trans. on Comput.*, Vol. C-22, April 1974.

[809] A. Thayse, “Boolean Differential Calculus,” *Springer-Verlag,* Heidelberg, 1981.

[810] S. Thelliez, “Sur la Decomposition Disjonctive Complexe Arborescent d’une Fonction Ternaire en vue de son Application a la Synthese des Structures Combinatoires Ternaires,” *Compt. Rend. H. Acad. Sci.,* Vol. 264, ser. A, pp. 419-421, February 1967, (in French).

[811] S. Thelliez, “Introduction to the Study of Ternary Switching Structures,” *Gordon and Breach Science,* N.Y. 1973.

[812] R. Thiele, “Synthese von Verknuepfungsnetzen aus Universellen Logikbausteinen,” *Dissertation,* Darmstadt 1978, (in German).

[813] P. Tison, “Generalization of Consensus Theory and Application to the Minimization of Boolean Functions,” *IEEE TC*, Vol. EC-16, Aug. 1967, pp. 446–456.

[814] S. B. Thrun and et. al., “The Monk’s Problems - A Performance Comparison of Different Learning Algorithms”, Carnegie Mellon University, Dec. 1991.

[815] V.H. Tokmen, “Disjoint Decomposibility of Multi-Valued Functions by Spectral Means,” *Proc. IEEE 10th International Symp. on Multiple Valued Logic,* pp. 88-93, 1980.

[816] I. Tomescu, “The maximum number of cliques and recoveries by cliques of complete chromatic hypergraphs,” *Discrete Mathematics*, Vol. 37, No. 2,3, pp. 263–277, Dec. 1981.

[817] I. Tomescu, “Some properties of irreducible coverings by cliques of complete multipartite graphs,” *J. Comb. Theory,* Sec. B 27, No. 2., pp. 164–167, April 1980.

[818] A.J. Tosser, D. Aoulad-Syad, “Cascade Networks of Logic Functions Built in Multiplexer Units,” *IEE Proc. Pt. E,* Vol. 127, No. 2, pp. 64-68, March 1980.

[819] L. Trevillyan, “An Overview of Logic Synthesis Systems,” *Proc. of the 24th ACM/IEEE Design Automation Conference (DAC),* 1987.

[820] S. Dao Trong, “Ein Entwurfsverfahren fuer Mehrstufige Schaltnetzwerke unter Gate-Array Randbedingungen,” *Fortschritt-Berichte VDI,* Reihe 9, Nr. 46, VDI-Verlag, Duesseldorf 1984, (in German).

[821] S

[822] Chia-Jeng Tseng, and D.P. Siewiorek, “Facet: A Procedure for the Automated Synthesis of Digital Systems,” *Proc. 20-th Design Automation Conference,* p.566.

[823] S. Tsukijama, M. Ide, H. Ariyoshi, and I. Shirakawa, “A New Algorithm for Generating All the Maximal Sets,” *SIAM J. of Computing*, Vol. 6, No. 3., pp. 505–517, September 1977.

[824] A.C. Tucker, and L. Bodin, “A Model for Municipal Street Sweeping Operations,” *Case Studies of Applied Mathematics,* Math. Assoc. of America, Washington, pp. 251–295, 1976.

[825] G.B., Tumanyan, *Engng. Cybern.* 156, 525, 1964, (in Russian), TJ212 .A413

[826] G.B. Tumanyan, *Engng. Cybern.* 74, 1965, (in Russian). TJ212 .A413 5

[827] Tutte, W.T., (B. Descartes) “Solution of Advanced Problem No. 4526”, American Math. Monthly, 61., p.352.

[828] P. Uehlau, “Eine Dekompositionsstrategie fuer den Logikentwurf auf der Basis Funktionstypischer Eigenschaften,” *Dissertation,* T.U. Karl-Marx-Stadt (Chemnitz), 1987, (in German).

[829] S.H. Unger, “Flow Table Simplification - Some Useful Aids,” *IEEE Transaction on Electron. Computers,* Vol. EC-14, pp. 472-475, 1965.

[830] P.E. Utgoff, “Shift of Bias for Inductive Concept Learning,” Rutgers University, PhD Dissertation, 1984.

[831] P.E. Utgoff, “Shift of Bias for Inductive Concept Learning,” *Machine Learning: An Artificial Intelligence Approach*, Vol. 2, in R.S. Michalski and J.G. Carbonell and T.M. Mitchell (eds), Morgan Kaufmann, Los Altos, CA, 1986.

[832] K. Walczak, “Decompositional Method of Combinational Circuits Synthesis Free from Static Hazard for Adjacent Changes,” *Archiwum Automatyki i Telemechaniki,* No. 1-2, 1977.

[833] K.M. Walliuzzaman, Z.G. Vranesic, “On Decomposition of Multiple-Valued Switching Functions,” *Computer Journal,* Vol. 13, No.4, pp. 359-362, 1970.

[834] W. Wan, M.A. Perkowski, “A New Approach to the Decomposition of Incompletely Specified Functions based on Graph-Coloring and Local Transformations and Its Application to FPGA Mapping”, *Proc. of the IEEE EURO-DAC '92, European Design Automation Conference,* Sept. 7-10, Hamburg, 1992, pp. 230 - 235.

[835] W. Wan, “M.S. Thesis,” *Department of Electrical Engineering*, Portland State University, 1992.

[836] F.L. Wang, "Isomorphic Notation of Switching Circuits," IEEE Comp., 5., Dec. 1965.

[837] C.C. Wang, "An Algorithm for the Chromatic Number of a Graph," *JACM,* Vol. 21, pp. 385-391, 1974.

[838] Y. Watanabe, and R.K. Brayton, "Heuristic Minimization of Multiple-Valued Relations," *IEEE Trans. on CAD.*, Vol. 12, No. 10, pp. 1458-1472, October 1993.

[839] T. Wecker, R. Kumar, W. Rosenstiel, H. Kraemer, M. Neher, "CALLAS - ein System zur Automatischen Synthese Digitaler Schaltungen," *Informatik - Forschung und Entwicklung,* April 1989, (in German).

[840] D.J.A. Welsh, and M.B Powell, "An Upper Bound to the Chromatic Number of a Graph and its Application to Time-Table Problems," *Comput. J.* Vol. 10, pp. 85–86, 1967.

[841] C.D. Weiss, "The Characterization and Properties of Cascade Realizable Switching Functions," *IEEE Trans. on Comput.* pp. 624-633, July 1969.

[842] C.D. Weiss, "Optimal Synthesis of Arbitrary Switching Functions with Regular Arrays of 2-input 1-output Switching Elements", *IEEE Trans. on Comput.*, pp. 839-856, September, 1969.

[843] S. M. Weiss and C. A. Kulikowski, "Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems," *Morgan Kaufmann Publishers, Inc.*, San Mateo, California, 1991.

[844] S.H. Whitesides, "An algorithm for finding clique cut-sets," *Inf. Process. Letters*, Vol. 12., No. 1., pp. 31–32, Febr. 1981.

[845] H.J. Wilf, "Spectral bounds for the cliques and independence numbers of graphs," *Comb. Theory Series B,* Vol. 40, No. 1, pp. 113-117, Febr. 1986.

[846] M.R. Williams, "The Coloring of Very Large Graphs," *Combinatorial Structures and Their Applications,* Gordon and Breach, New York, pp. 477–478, 1970.

[847] J. Wnek and R.S. Michalski, "Hypothesis-Driven Constructive Induction in AQ17: A Method and Experiments," *Proc. of IJCAI-91 Workshop on Evaluating and Changing Representation in Machine Learning*, pp. 13-22, Sydney, Australia, 1991.

[848] S

[849] J. Wnek, and R.S. Michalski, "Hypothesis-driven Construction Induction in AQ17-HCI," *Machine Learning and Inference Laboratory, Center for Artificial Intelligence, School of Information Science and Technology, George Mason University,* Jan. 1992, Technical Report.

[850] J. Wnek and R.S. Michalski, "Hypothesis-driven Constructive Induction in AQ17-HCI: A Method and Experiments", *Machine Learning*, no. 14, pp. 139-168, 1994.

[851] W.S. Wojciechowski, A.S. Wojcik, "Automated Design of Multiple-Valued Logic Circuits by Automatic Theorem-Proving Techniques," *IEEE Trans. on Comput.*, Vol. 32, No. 9, pp. 785-798, 1983.

[852] Wojutynski, J., "Graph Coloring Algorithms," M.Sc. Thesis, Institute of Automatics, Warsaw Technical University, 1979 (in Polish).

[853] Nam-Sung Woo, "A Study on the Structure of the Intermediate Network in an FPGA Technology Mapping", *Proc. Field Programmable Logic and Applications,* Oxford, 1991.

[854] Nam-Sung Woo, "A Heuristic Method for FPGA Technology Mapping Based on the Edge Visibility," *Proc. 28th ACM/IEEE Design Automation Conf.* San Francisco, CA, June 1991, pp. 248-251.

[855] S. Wrobel, "Demand-Driven Concept Formation," in *Knowledge Representation and Organization in Machine Learning,* (K. Morik, ed), Springer-Verlag, Berlin Heidelberg, 1989.

[856] D.C. Wood, "A Technique for Coloring a Graph Applicable to Large-Scale Time-Tabling Problems," *Comp. J.,* Vol. 12, pp. 317-319, 1969.

[857] L-F. Wu, M.A. Perkowski, "Minimization of Permuted Reed-Muller Trees for Cellular Logic Programmable Gate Arrays", *Proc. of the 2nd Intern. Workshop on Field-Programmable Logic and Applications, FPL'92,* Vienna, Austria, August 31-September 2, 1992, pp. 7/4.1-7/4.4.

[858] L-F, Wu, and M. Perkowski, "Minimization of Permuted Reed-Muller Trees for Cellular Logic Programmable Gate Arrays," *invited chapter to a book edited by Prof. Reiner Hartenstein, "Lecture Notes in Computer Science",* Number 705, Springer Verlag, pp. 78-87, Berlin/Heidelberg, 1993.

[859] H. Wu, and M. A. Perkowski, "Synthesis for Reed-Muller Directed-Acyclic-Graph networks with applications to Binary Decision Diagrams and Fine Grain FPGA Mapping", *Proc. of IWLS '93,* Tahoe City, CA, May 1993.

[860] H. Wu, N. Zhuang, and M. A. Perkowski, "Synthesis for Reed-Muller Directed-Acyclic-Graph network," accepted to *IEE Proceedings, Pt. E.,* in June 1993.

[861] H. Wu, M. A. Perkowski, and N. Zhuang, "A New Easily Testable Canonical AND/XOR Circuit Form and its Minimization," submitted to *IEEE Trans. on Comput.*, June 1994.

[862] L.G. Valiant, "A theory of the learnable," *Comm. ACM,* no. 27, pp. 1134-1142, 1984.

[863] P. Vanoostende, "Power estimation," *Internal Report,* IMEC Leuven, April 1991.

[864] D. Varma, and E.A. Trachtenberg, "Design Automation Tools for Efficient Implementation of Logic Functions by Decomposition," *IEEE Trans. on CAD,* Vol. 8, No. 8, pp. 901-917, August 1989.

[865] D. Varma, E.A. Trachtenberg, "On the Estimation of Logic Complexity for Design Automation Applications," *1990 Conference of IEEE,* pp. 368-371, 1991.

[866] V.B. Varshavsky, *Engng. Cybern.* R 58, 1965, (in Russian).

[867] V.B. Varshavsky, Ovsievich, "Networks Composed of Ternary Majority Elements," October, *IEEE Comput.* 14, 730, 1965.

[868] V.B. Varshavsky, *Engng. Cybern.* R 39, 1965, (in Russian).

[869] Vashchenko, "A Method of Synthesizing a Class of Schemes on the Basis of Functional Decomposition," *Kibernetika,* No. 1, 1969.

[870] V.P. Vascenko, *Tr. Mosk. Energ. Instituta,* vol. 247, pp. 5-10, 1975, (in Russian)

[871] V.P. Vascenko, *Dokl. AN SSSR,* Vol. 234, No. 3, pp. 509-512, 1977, (in Russian).

[872] V.P. Vascenko, *Dokl. AN SSSR,* Vol. 238, No. 1, pp. 18-21, 1978, (in Russian).

[873] V.P. Vascenko, "On the Computation of all Nontrivial Simple Decompositions", Vol.20, No.4, pp.629-632, 1979, (in English). V.P. Vascenko, *Dokl. AN SSSR,* Vol. 247, No. 1, pp 15-18, 1979, (in Russian). QA1 .S78 Mathematics

[874] J. Vasudevamurthy, J. Rajski, "A Method for Concurrent Decomposition and Factorization of Boolean Expressions," *Proc. IEEE ICCAD,* Vol. 7, No. 12, December 1988, pp. 1290-1300.

[875] E.W. Veitch, "A chart method for simplifying truth functions", *Proc. ACM, Pittsburg, PA,* pp. 127-133, May 1952.

[876] Voigt

[877] E. Voigt, "Anwendung einer Ternaervektorlisten-Algorithmensprache auf die Struktursynthese von GATE-ARRAY-Schaltkreisen," *Dissertation,* T.H. Karl-Marx-Stadt (Chemnitz), 1985, (in German).

[878] Voith,R.P. "ULM Implicants for Minimization of Universal Logic Module Circuits," *IEEE Trans. on Comput.*, Vol. C-26, WHEN?????

[879] R.P. Voith, "Minimum Universal Logic Module Sequential Circuits with Decoders," *IEEE Trans. on Comput.*, Vol. C-26, No. 10, 1977.

[880] Xilinx Inc., "The Programmable Gate Array Data Book", "Xilinx Programmable Gate Array User's Guide," 1991.

[881] S.V. Yenin, P.N. Bibilo, *Avtomatizacjia Proektirovanija EVM,* pp. 55-65, Kiev 1979, (in Russian).

[882] S.V. Yenin, P.N. Bibilo, *Avtomatika i vycislitelnaya tekhnika,* No. 1, pp. 16-22, 1979, (in Russian).

[883] A.A. Ytkin, *Avtomatizacjia Logiceskogo Proektirovanija,* pp. 41-58, Minsk 1982, (in Russian)

[884] S.W. Yablonskii, "On Algorithmic Obstacles to the Synthesis of Minimal Contact Networks," *Problemy Kibernitiki,* No. 2, pp. 75-121, 1959, (in Russian).

[885] K. Yamamoto, "Design of Irredundant MOS - Networks: A Program Manual for the Design Algorithm DIMN," *Dept. of Computer Science,* University of Illinois, February 1976.

[886] Chao-Chih Yang, "On the Equivalence of Two Algorithms for Finding All Maximal Compatibles," *IEEE Trans. Comp.*, pp. 977–979, Oct. 1975.

[887] S. Yang, M. Ciesielski, "A Generalized PLA Decomposition With Programmable Encoders," In the *Proc. of the Intern. Workshop on Logic Synthesis,* May 1989, pp. 1-13.

[888] S. Yang, M. Ciesielski, "PLA Decomposition with Generalized Decoders," *Proc. of the ICCAD,* pp. 312-315, 1989.

[889] S. Yang and M.J. Ciesielski, "On The Relationship Between Input Encoding and Logic Minimization," *Proc. 23rd Hawaii International Conference on System Sciences*, pp. 377-386, Jan. 1990.

[890] S. Yang, M. Ciesielski, "Optimum and Suboptimum Algorithms for Input Encoding and its Relationship to Logic Minimization," *IEEE Transactions on Computer Aided Design,* Vol. 10, No. 1, January 1991.

[891] S.S. Yau, C.K. Tang, "Universal Logic Circuits and Their Modular Realizations," *1966 Spring Joint Comp. Conf.*, AFIPS Conf. Proc. Vol. 28, Washington DC, pp. 297-305, 1966.

[892] S.S. Yau, and M. Orsic, "Synthesis of Universal Logic Modules," *Proc. 3rd Annual Princeton Conf. Inform. Sci. and Syst.*, pp. 498-502, March, 1969.

[893] S.S. Yau, C.K. Tang, "Universal Logic Modules and Their Applications," *IEEE Trans. on Comput.*, Vol. C-19, No. 2, Febr. 1970.

[894] S.S. Yau, Y.S. Tang, "On Identification of Redundancy and Symmetry of Switching Functions," *IEEE Trans. on Comput.*, Vol. C-20, pp. 1609-1613, Dec., 1971.

[895] S. Yang, and M. Ciesielski, "A Generalized PLA Decomposition With Programmable Encoders," In the *Proc. of the Intern. Workshop on Logic Synthesis,* May 1989, pp. 1-13.

[896] S. Yang, and M. Ciesielski, "PLA Decomposition with Generalized Decoders," *Proc. of the IC-CAD,* pp. 312-315, 1989.

[897] S. Yang and M. Ciesielski, "Optimum and Suboptimum Algorithms for Input Encoding and Its Relationship to Logic Minimization," *IEEE Trans. on Computer Aided Design,* Jan 1991.

[898] M. Yamamoto, "A Method for Minimizing Incompletely Specified Sequential Machines, " *IEEE Trans. Comp.,* Vol. C-29, No. 8, p. 732-736, August 1980.

[899] L.A. Zadeh, "A Fuzzy-Set-Theoretic Interpretation of Liguistic Hedges", *Journal of Cybernetics,* Vol.2, No.4, pp.4-34, 1972.

[900] A.D. Zakrevskij, "An Algorithm for Decomposition of Boolean Functions," *Trudy Sib. Fiz-tekhn In-ta,* Collection 44, pp. 5-16, 1964, (in Russian)

[901] 39. A.D. Zakrevskij, "Algoritmy Sinteza Diskretnych Avtomatov", Moscow 1971, (in Russian)

[902] A.D. Zakrevskij, "Logicheskije yravnenija," Minsk, 1975, (in Russian)

[903] A. D. Zakrevskij, *Dokl. A.N. BSSR,* Vol. 21, No. 11, pp. 991-994, 1977, (in Russian).

[904] A.D. Zakrevskij, "Logiceskij Sintez Kaskadnych Schem," Moscow, 1981, (in Russian).

[905] A.D. Zakrevskij, P.N. Bibilo, A. A. Dudkin, A.A. Shneider, "Upravlajuscije Sistemy i Masiny, No.3, pp. 115-118, 1984, (in Russian).

[906] A.D. Zakrevskij, P.N. Bibilo, A. A. Dudkin, *Upravlajuscije sistemy i masiny,* 1985, No. 1, pp. 27-29, (in Russian).

[907] Zaky, Oliviera,

[908] H.J. Zander, "Logischer Entwurf binaerer Systeme," *VEB Verlag Technik,* Berlin 1989, (in German).

[909] W. Zhao, "Two-Dimensional State Minimization with Input Encoding," M.S. Thesis, PSU.

[910] I.I. Zhegalkin, "O Tekhnyke Vychyslenyi Predlozhenyi v Symbolytscheskoi Logykye," *Mat. Sb.,* Vol. 34, pp. 9-28, 1927 (in Russian).

[911] K. Zibert, "Entwurfstechnik fuer IS - Versuch einer Bestandsaufnahme," ITG-Fachbericht der ITG-Fachtagung vom 3.-5.10.1989, Stuttgart, *VDE-Verlag GmbH,* Berlin Offenbach, (in German).

[912] B. Zupan and M. Bohanec, "Learning concept hierarchies from examples by function decomposition," J. Stefan Institute, Ljubljana, 1996.

[913] B. Zupan, "Machine Learning based on function decomposition," Ph.D. Thesis, *University of Ljubljana*, Slovenia, 1997,

[914] M. Zwick, "Control Uniqueness in reconstructability analysis," *International Journal of General Systems*, Vol. 23, no. 2, 1995.

[915] A.A. Zykov, "On Some Properties of Linear Complexes," *Math. Sbornik,* 24/26, p. 163., Amer. Math, Soc. Translation No. 79.