UNIFIED APPROACH TO FUNCTIONAL DECOMPOSITIONS OF SWITCHING FUNCTIONS

Marek A. Perkowski, Tadeusz Luba +, Stanislaw Grygiel,
Paul Burkey, Michael Burns, Nick Iliev, Marcel Kolsteren †, Robert Lisanke,
Rahul Malvi, Zhi Wang, Hongfei Wu, Feng Yang, Sida Zhou, and Jin S. Zhang
Department of Electrical Engineering,
Portland State University,
P.O. Box 751, Portland, Oregon 97207
tel. (503) 725-5411

+ Warsaw University of Technology,

† Eindhoven University of Technology,

December 29, 1995

Abstract

In this report, we present a new unified approach to decompose Boolean functions. It includes Steinbach's, Luba's, and Perkowski/Uong/Brown/Wan methods as special cases. The approach can be applied to binary, and multiple-valued functions, both completely specified, incompletely specified, and with generalized don't cares.

Contents

1 Introduction. -6

2 Towards a Unified Approach to Decomposition. -11

3 Cube Diagram Bundles to Represent Discrete Mappings. -13

4 Designs Realized With Multiplexers. -20

5 Basic Patterns for Decompositions. -26
5.1 Definitions of Patterns . . . . -27

6 Ashenhurst Decompositions -30
6.1 Disjoint Ashenhurst Decomposition of Completely Specified Functions. . . . . -30
6.2 Non-disjoint Ashenhurst Decomposition of Completely Specified Functions. . . . . -32

7 Ashenhurst Decomposition of Incompletely Specified Functions. -35
7.1 Disjoint Ashenhurst Decomposition of Incompletely Specified Functions. . . . . -35
7.2 Non-disjoint Ashenhurst Decomposition of Incompletely Specified Boolean Functions. . -37
7.3 Non-disjoint Ashenhurst Decomposition of Incompletely Specified Multiple-Valued Functions. . . . . -37

8 Introduction to Analytical Methods for Column Minimization for Incompletely Specified Single-Output Boolean Functions. -40
8.1 Exact Algorithm for Ashenhurst Decomposition. . . . . -42

9 PUB Decompositions. -45
9.1 Disjoint PUB Decomposition. . . . . -45
9.2 Non-disjoint PUB Decomposition. . . . . -51

10 Curtis Decomposition -55
10.1 Disjoint Curtis Decomposition. . . . . -55
10.2 Non-disjoint Curtis Decomposition. . . . . -55

11 Programs for Curtis Decomposition. -58

12 Decomposition of Multiple-Valued Logic Functions and Information Systems. -61

13 Fundamental Formalisms for Partition-Based Decomposition. -63

14 BDD Representation for Integer Sets Representing Blocks of Partitions. -68

15 Basic Theorem. -72

16 Decomposition of Decision Tables. -76

17 Approaches to Column Compatibility Problem. -79
17.1 Troubles with the Column Compatiblity Problem. . . . . -80

18 The Set Covering Approach to the Column Compatibility Problem. -81
18.1 The compatibility relation. . . . . -81

19 Graph Coloring Approach to the Column Compatibility Problem. -85

20 Strong AND/OR/EXOR Decompositions. -88
20.1 Boolean Operations . . . . . -88
20.2 Two-input Decompositions . . . . . -89
20.3 EXOR Decompositions . . . . . -97

21 Weak AND/OR/EXOR Decompositions. -108

22 Decomposition Strategy. -119
22.1 Immediate Decompositions. . . . . -119
22.2 Goal-Oriented Reduction Decompositions. . . . . -119
22.3 Last-Resort Methods. . . . . -120
22.4 The entire comprehensive search strategy . . . . . -120

23 Other Applications of CDBs. -121
23.1 General Decompositions of Multiple-Valued-Output Functions with Standard Don't Cares.-121
23.2 Curtis-like Decomposition of Multiple-Valued-Output Functions with Generalized Don't Cares. . . . . -121

24 Future Research: Towards a Unified Approach to Decomposition -123

25 Conclusions -130

List of Figures

1 *An example of a simple disjunctive decomposition* . . . -11
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* . . . -17
3 *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* -20
4 *Cofactors from Fig. 3 shown as separate functions of variables c and d* . . . -21
5 *Cofactors from Figure 4 rewritten to a more familiar form of K-maps* . . . -21
6 *Realization with address variables a, b* . . . -22
7 *Realization with address variables c and d* . . . -24
8 *Realization with address variables a, c* . . . -24
9 *Patterns of cofactors for various combinations of variables; (a) variables ab, (b) ac, (c) ad, (d) bc, (e) bd, (f) cd* . . . -25
10 *Basic patterns of OR, AND and EXOR decompositions* . . . -28
11 *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* . . . -30
12 *The SOP realization of the image function H to Example 6.2* . . . -31
13 *An example of a non-disjoint decomposition using a K-map to Example 6.3* . . . -32
14 *A different way of using a K-map to make the non-disjoint decomposition from Fig. 13* . -33
15 *Function $f2(a, b, c, d, e, f)$ to Example 7.1* . . . -35
16 *Loops $\bar{a}\bar{b}$ and $a\bar{b}$ from Fig. 15 redrawn to another form* . . . -36
17 *Function for Example 7.2* . . . -36
18 *A loop x z as a loop that can be completed to 1's is found in the function from Fig. 17* . -36
19 *The remaining loops are found for function from Fig. 18* . . . -37
20 *The decomposition for free set $\{x, z\}$* . . . -38
21 *A multiple-valued function that is Ashenhurst decomposable in Example 7.3* . . . -38
22 *A Repeated Variable Map for function from Example 7.3 with repeated variable b* . . . -39
23 *A Karnaugh map to Example 8.1* . . . -40
24 *A loop compatibility graph to Example 8.1* . . . -41
25 *Illustration of the 2-coloring algorithm: (a) the compatibility graph G, (b) the corresponding incompatibility graph $\bar{G}$, (c) - (k) stages of coloring* . . . -44
26 *A Karnaugh map for Example 9.1* . . . -46
27 *The loop pair incompatability graph for Example 9.1* . . . -47
28 *The graph from Fig. 26 after coloring* . . . -47
29 *The function after the completion of loop functions, the columns colored with the same colors are shown (the column multiplicity of 3)* . . . -48
30 *The PUB decomposition with column multiplicity of 3 realized with a multiplexer and gates* -49
31 *Different views at the Disjoint PUB Decomposition* . . . -50
32 *K-map of a function to a non-disjoint PUB decomposition from Example 9.2* . . . -51
33 *Incompatibility graph for non-disjoint PUB decomposition* . . . -52
34 *The K-map coresponding to the completed function with coloring of compatible columns shown and the multiplicity index of 2* . . . -53
35 *Realization of the PUB decomposition of function from Example 9.2 using a multiplexer* -53
36 *Comparison of the disjoint and non-disjoint Curtis and PUB decompositions* . . . -54
37 *Comparison of PUB and Curtis decompositions for small k* . . . -56

38 *K-map of function f to Example 10.1* . . . -56
39 *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)* . . . -59
40 *The Karnaugh Map to Example 13.2* . . . -67
41 *Standard enumeration of cells of a 3-variable Boolean function* . . . -68
42 *Karnaugh map for $P_1(X_0)$ for Example 14.1* . . . -69
43 *Karnaugh map for $P_2(X_0)$ for Example 14.1* . . . -69
44 *The BDDs for functions $P_1(X_0)$ and $P_2(X_0)$* . . . -69
45 *Partitions Π_1 and Pi_2 to Example 14.1:* $\Pi_1 = \bar{a}\bar{b} + ab$, $\Pi_2 = a\bar{b} + bc + \bar{a}b$ . . . -70
46 *Partitions to Example 14.1:* $P_1\Pi_1 = \bar{a}\bar{b}$, $P_2\Pi_2 = a\bar{b}+bc+\bar{a}b$, $P_2\Pi_1 = ab$, $P_1\Pi_2 = a\bar{b} + \bar{a}b\bar{c}$ . . . -70
47 *Decomposition of function f from Example 15.3* . . . -75
48 *Block Diagram of the Decomposition of Decision Tables* . . . -77
49 *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* . . . -79
50 *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* . . . -84
51 *The incompatibility graph for coloring* . . . -85
52 *Pseudo-code of graph coloring* . . . -87
53 *Graphical illustration of operations derivative, minimum and maximum for minterms* . . -90
54 *Strong OR (Disjunctive), AND (Conjunctive) and EXOR (Linear) Decompositions.* . . . -90
55 *The Care Functions f_{ON} and f_{OFF} of function $f(a, b, c, d)$ to Example 20.2* . . . -91
56 *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* . . . -92
57 *Function f to Example 20.4 - graphical calculation of max and intersection* . . . -93
58 *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* . . . -97
59 *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* . . . -99
60 *Function to Example 20.7* . . . -100
61 *Calculation of EXOR decomposability to Example 20.8* . . . -100
62 *Illustration of using Theorem 20.4 in Example 20.9* . . . -101
63 *Weak AND and OR Decompositions.* . . . -108
64 *Steps to OR decomposition to Example 21.1* . . . -109
65 *Function to Example 21.2 with enumeration of minterms* . . . -110
66 *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* . . . -111
67 *Graphical illustration of using the formulas in weak OR decompositions to Example 21.5* -113
68 *Graphical illustration of using the formulas in weak AND decompositions to Example 21.6*-113
69 *Graphical illustration of using the formulas in weak EXOR decomposition* . . . -114
70 *A Decomposition of a function with 4 binary variables, a 5-valued output variable and generalized don't cares from Table 1* . . . -122
71 *The types of decomposition* . . . -125
72 *Curtis decompositions* . . . -125

73 *Column Multiplicity Based Curtis Decompositions* . . . . -128
74 *A layered construction of GUD system* . . . . -129

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 report, this is still a *Curtis Decomposition Type*. We will talk, however, about the *Roth-Karp decomposition method.*

The goal of this report 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.

For instance, 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, [12]. 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 [165, 166]. 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 [130]. 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 [72]). A generalized decomposition, as discussed in [89], but for incompletely specified functions, was also developed.

Independently, Luba et al created a complete decomposition theory based on partition calculus [145]. 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 [106]; other approaches to reduction of an instance space can also be found in [204]. 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*}>

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 report, some of them quite complex, and we would like our report to be understood by non-specialists in logic synthesis, we will apply a tutorial approach below. The report will be full of examples and graphical illustrations. This report 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 report 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 a separate report, [222] we discuss the very important problem of finding good free and bound sets of variables. (This report is devoted also to solving the column compatibility problem). Report [222] 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 in a separate, accompanying report [223].

The presented report is organized as follows. Section 2 defines the decomposition problem. In section 3 we introduce a new representation of Boolean and Multi-valued Functions - *Cube Diagram Bundles (CDBs)*. This general-purpose representation combines Cube Calculus [53], Decision Diagrams [76] and Rough Partitions [145, 69, 92], 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 4 explains the design with multiplexers and section 5 introduces several basic types of decompositions based on patterns found in this representation. Ashenhurst decomposition [26, 27, 29] for completely and incompletely specified functions is introduced in sections 6 and 7, respectively. A new approach to Ashenhurst Decomposition is also presented in section 8 - it is shown that contrary to the more general case of Curtis Decomposition [37], 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 10. 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 [130], 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 12 presents the decomposition of Multiple-Valued functions and information systems. Section 13 presents fundamental formalisms for Partition-Based Decompositions. Section 14 shows how BDDs can be used to encode these partitions for data compression. Section 15 presents a basic theorem for Curtis decomposition. Section 16 discusses decomposition of decision tables. Section 17 presents approaches to the Column Minimization Problem. Section 18 presents the set covering approach to this problem. Section 19 presents the graph coloring approach to this problem. Section 20 discusses AND, OR and EXOR strong decompositions. Section 21 discusses AND, OR and EXOR weak decompositions. Section 22 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 [130]. 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 23 we present an even more comprehensive unified approach to decompositions, which we even did not start to implement. Section 24 presents some other applications of CDBs. Section 25 concludes our report.

When reading this report, 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.

2 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

$$F = H(A, G(B, C)) = H(A, Z) \tag{1}$$

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. 1.

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.

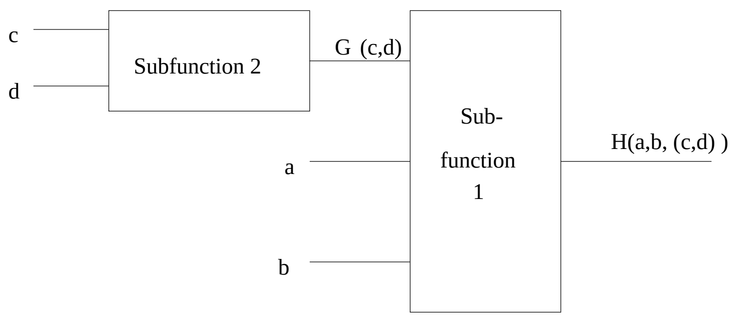


Figure 1: *An example of a simple disjunctive decomposition*

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 [112, 111, 108]. 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.

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 23.

Boolean and multiple-valued functions that include very many don't cares are becoming increasingly important in several areas of applications [173]. 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 [107] 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 [44]. Kohavi uses them under the name of *covers* and Luba et al call them *Rough Partitions.* Luba et al [145] 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 [188, 191].

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 [170], Three level AND/NOT Networks with True inputs (TANT) networks synthesis [167], 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 [53],
- Decision Diagrams [76],
- Rough-Partitions [145],
- and Boolean Relations [189].

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) [115, 173], 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 1, 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 1: 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 [145]. 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 1 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

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

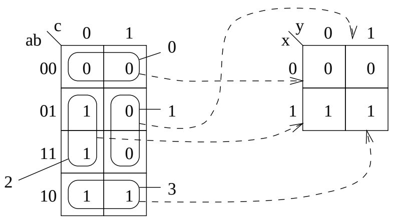
represents the function:

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 [188, 191]. While the authors from [188, 191] solve it as a truth-table permutation problem, we solve it as a cube encoding problem, which is more general.

Example 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. 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. 2b is created. Figure 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. 2a is shown in Fig. 2c, and the encodings of its rows to secondary input variables is shown in Fig. 2d.



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)

ab \ c	0	1	2
00	2	2	—
01	—	1	2
11	—	1	—
10	0	—	—

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 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*

The function is specified as the following CDB.
$Var(F) = \{a,b,c\}$.
$Inp(F) = \{$ [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 3.2

The second example, see Table 1, 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

The third example presents a table, Fig. 2f, for an incompletely specified function with standard don't cares from Fig. 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. 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), $\quad Var(CF) := Var(F)$ - $Var(C)$.

   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 [107] 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.

4 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 4.1

Let us assume that we want to realize the four-input function $f1(a, b, c, d)$, shown in Fig. 3a, using a 4×1 multiplexer with two select lines (2 bit address).

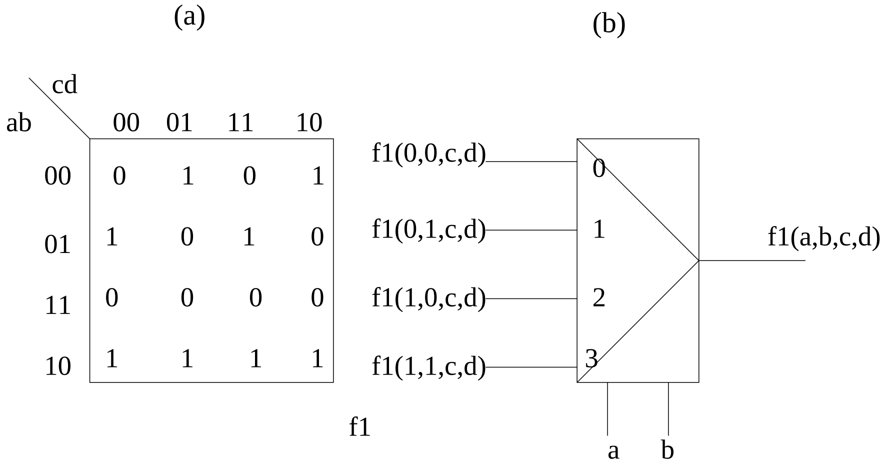


Figure 3: *Explanation of the concept of cofactors in the Kmap and their relation to the data inputs of a multiplexer, (a) A K-map of function $f1$, (b) realization of $f1$ using a multiplexer with address variables whose combinations of values correspond to the rows of the K-map*

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 3b.

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. 4.

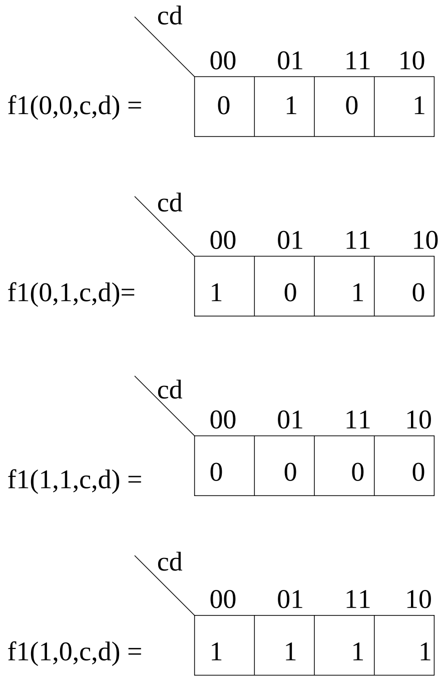


Figure 4: *Cofactors from Fig. 3 shown as separate functions of variables c and d*

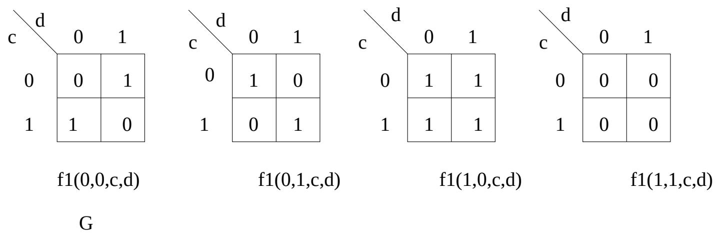


Figure 5: *Cofactors from Figure 4 rewritten to a more familiar form of K-maps*

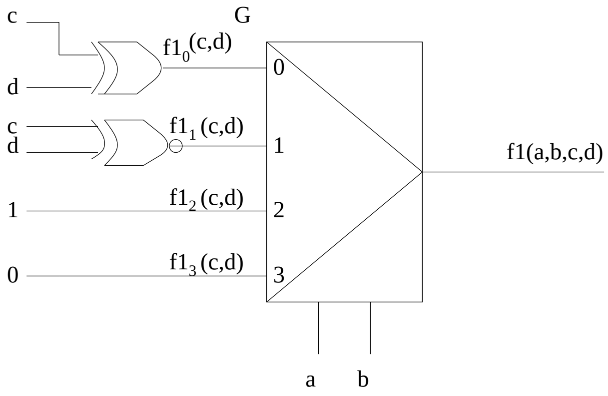


Figure 6: *Realization with address variables a, b*

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. 5. 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. 6.

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. 7.

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. 8a), the functions $f1_0(b,d)$, $f1_1(b,d)$, $f1_2(b,d)$, $f1_3(b,d)$ will be as shown in Fig. 8b and the realization as shown in Fig. 8c. Note the inverse order of variables in the headings of some of maps from Fig. 8b. If the designer has difficulties in using the "inverse" K-maps, he can rewrite some of them as shown in Fig. 8d. 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. 8c has the same cost as one from Fig. 6.

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. 9 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. 9a). 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. 9a 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. 9.

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.

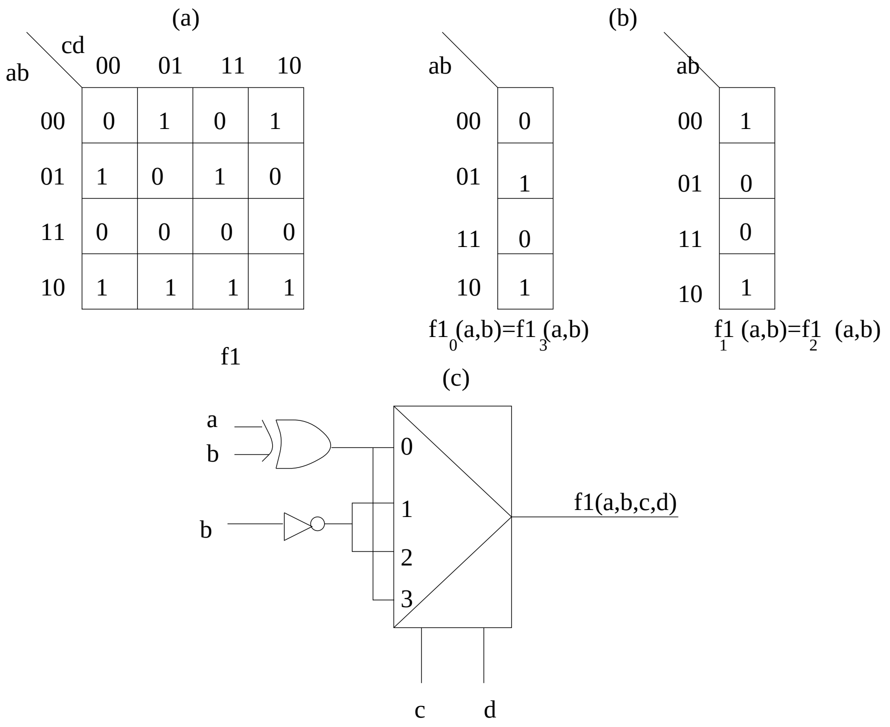


Figure 7: *Realization with address variables c and d*

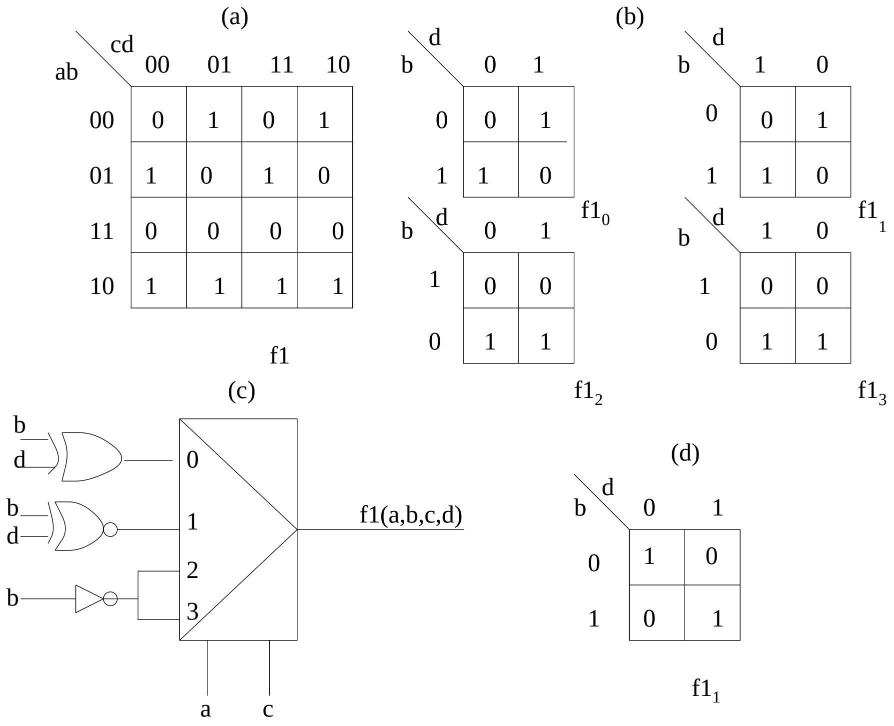


Figure 8: *Realization with address variables a, c*

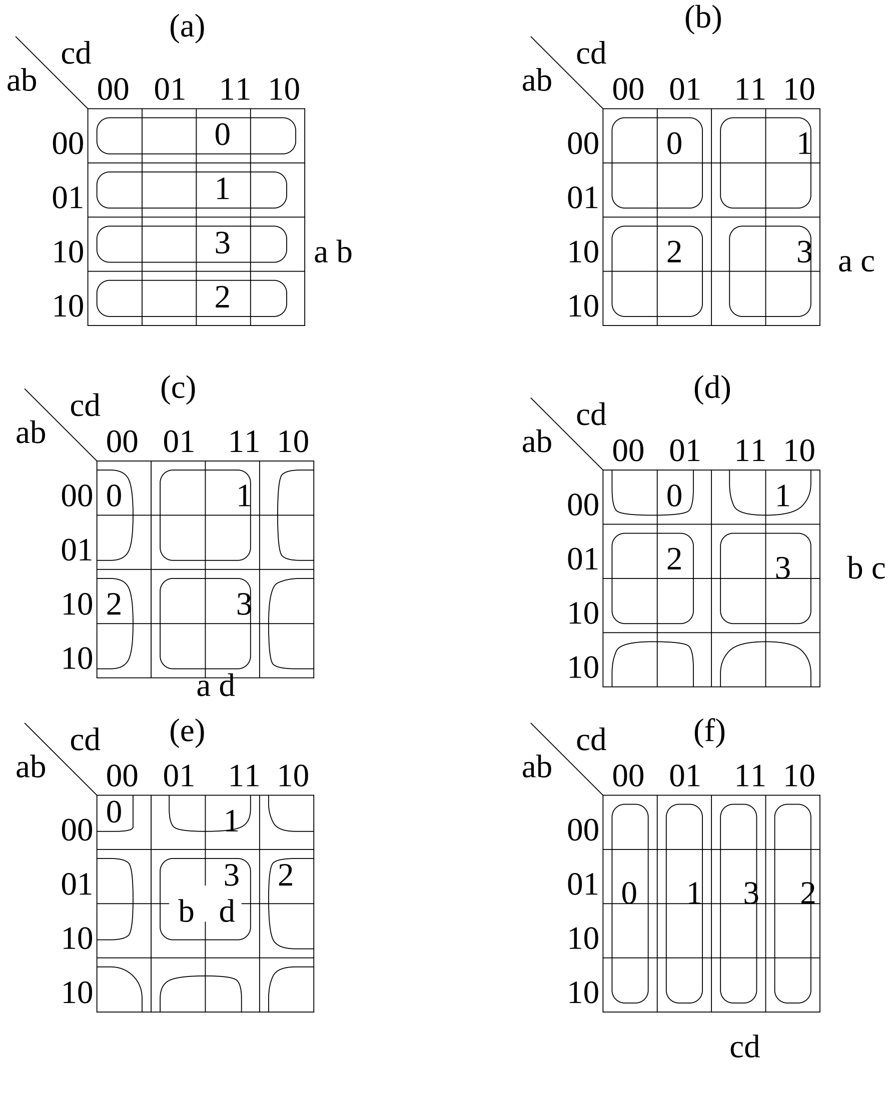


Figure 9: *Patterns of cofactors for various combinations of variables; (a) variables ab, (b) ac, (c) ad, (d) bc, (e) bd, (f) cd*

5 Basic Patterns for Decompositions.

The author is aware of 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 [37],
- Steinbach et al (XBOOLE) Decomposition [107],
- Perkowski/Uong/Brown (PUB) Decomposition [166, 165, 12].

The decompositions of Ashenhurst [26, 27, 29]; Luba et al [145]; Lai, Pedram et al [122], and many other are just special cases, or different realizations, of the Curtis decomposition [12]. 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 [107, 166, 165, 12]).

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 [37, 107, 166, 165, 12, 145] 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.

5.1 Definitions of Patterns

Definition 5.1 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 5.2 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 5.3 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 5.4 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 [107]. There exist then **Weak AND, Weak OR**, and **Weak EXOR** decompositions.

Definition 5.5 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 5.6 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 5.7 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 5.8 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.*

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 5.9 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 5.10 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 5.11 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**.

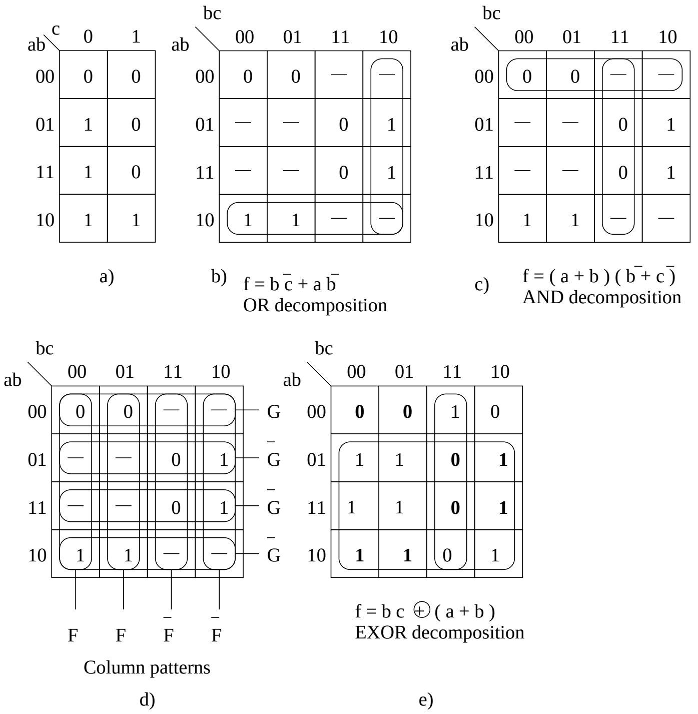


Figure 10: *Basic patterns of OR, AND and EXOR decompositions*

Example 5.1

An example of the RVM is shown in Figure 10. Figure 10a 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 10b. 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 10c, 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 10d, 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 10d. These patterns lead to Strong EXOR Decomposition: $(b\ c) \oplus (a\ +\ b)$ from Figure 10e. Figure 10e clearly shows the incomplete patterns from Figure 10d 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**.

6 Ashenhurst Decompositions

6.1 Disjoint Ashenhurst Decomposition of Completely Specified Functions.

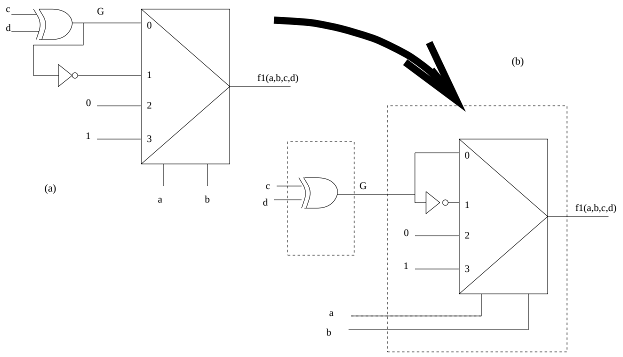


Figure 11: *Explanation of the concept of Ashenhurst decomposition for function* $f1(a, b, c, d)$*: (a) using a multiplexer and an inverter gate, (b) a corresponding block diagram of the Ashenhurst decomposition*

Example 6.1

Let us first observe that function $f1_0(c, d)$ in Fig. 6 is the negation of function $f1_1(c, d)$. Therefore, the realization from Fig. 6 can be re-drawn as shown in Fig. 11.

Next, the function from Fig. 11 can be represented in a block diagram form as in Fig. 1. 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 [180].

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. 11 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 [33].

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:

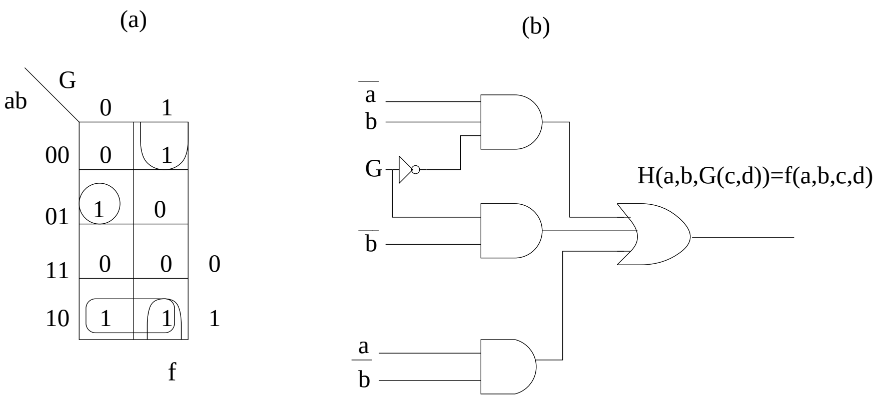


Figure 12: *The SOP realization of the image function H to Example 6.2*

- 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 6.2

Let us go back to function $f1(a, b, c, d)$ from Example 6.1. 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. 12a,b.

The K-map of successor is created from the K-map of Fig. 3, G, and $\overline{G}$, as in Fig. 12a. This function is minimized using two-level Boolean minimization to the circuit from Fig. 12b.

Example 6.3

For the same function let us now select variables c and d as the address variables (Fig. 7a). 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.

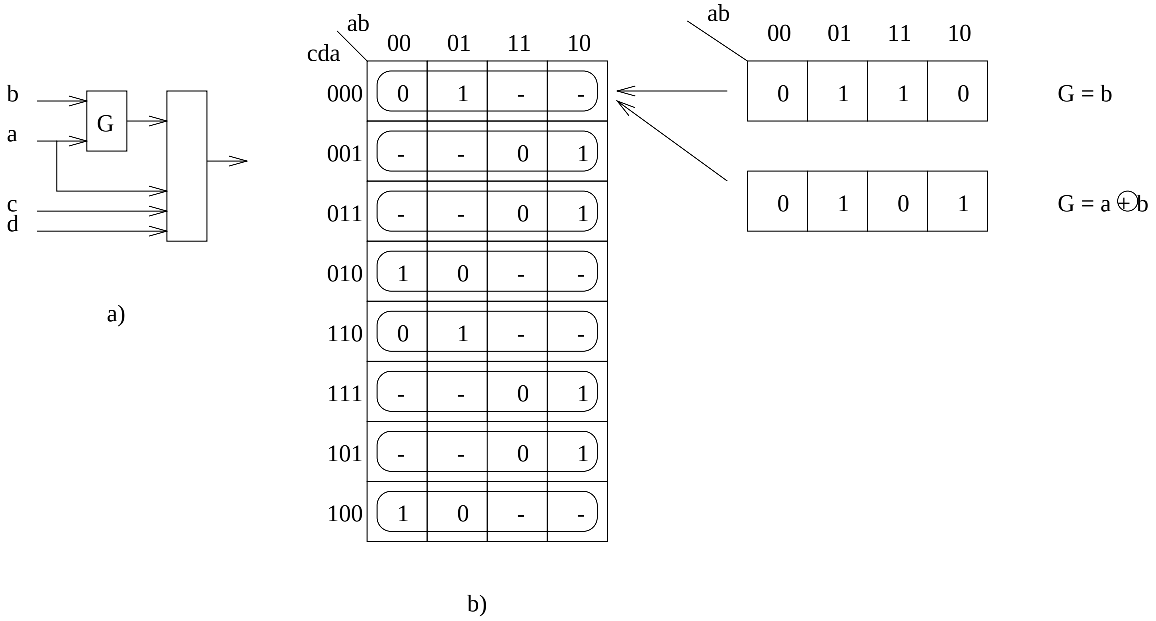


Figure 13: *An example of a non-disjoint decomposition using a K-map to Example 6.3*

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 7). Such functions are more likely to be found when the function is strongly unspecified.

6.2 Non-disjoint Ashenhurst Decomposition of Completely Specified Functions.

Example 6.4

Let us again consider the function from Fig. 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. 13a 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. 13b. 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. 14 shows a different way of using K-map to make the same decomposition. Decomposition scheme from Fig. 14a is equivalent to the scheme from Fig. 13a. K-map is shown in Fig. 14b. The loops drawn in K-map correspond to free set variables $A = \{c, d, a\}$.

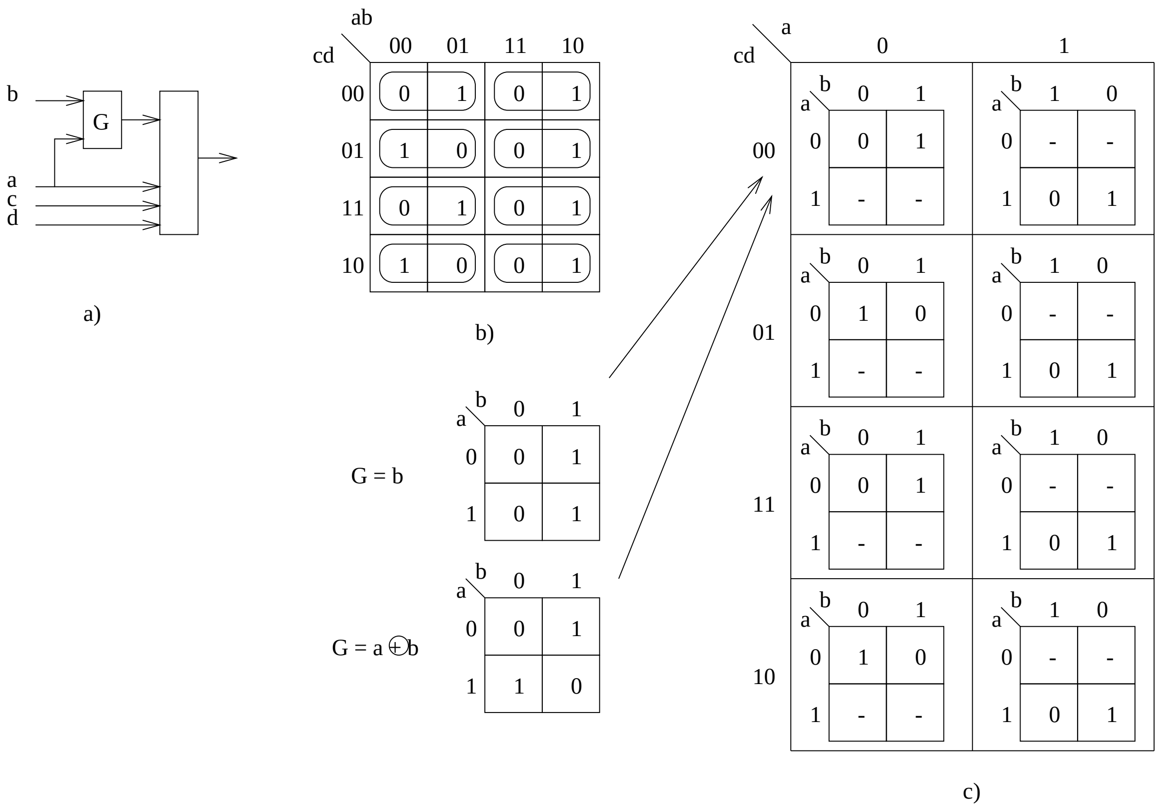


Figure 14: *A different way of using a K-map to make the non-disjoint decomposition from Fig. 13*

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. 14c. 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 $G = a \oplus b$ ($A = \{c, d, a\}$, $B = \{a, b\}$).

7 Ashenhurst Decomposition of Incompletely Specified Functions.

7.1 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 7.1

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

loop $\overline{a}\,\overline{b}$

loop $\overline{a}\,b$

loop $a\,b$

loop $a\,\overline{b}$

f2

Figure 15: *Function $f2(a,b,c,d,e,f)$ to Example 7.1*

Given is function $f2(a,b,c,d,e,f)$ from Fig. 15.

Assuming address variables a and b, the loops are as shown in Fig. 15. 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. 16. 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 7.2

We will solve the example from p. 171 in [180].

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

The K-map for this function is shown in Fig. 17.

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 16: *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 17: *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 18: *A loop $x\ z$ as a loop that can be completed to 1's is found in the function from Fig. 17*

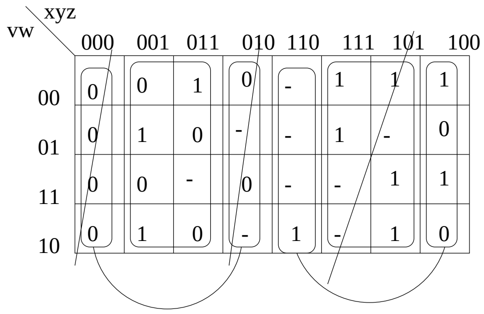


Figure 19: *The remaining loops are found for function from Fig. 18*

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. 18. 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. 19). 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. 20a. The realization of function $G(v, w, y)$ is shown in Fig. 20d. 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. 20a. Fig. 20b, c presents two methods for realization of function $f(x, z, G)$. The final circuit is presented in Fig. 20e.

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

See section 6, Example 6.4.

7.3 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 7.3

Multiple-valued function (three possible values: 0, 1, 2) which is defined in Fig. 21 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
- {1,2} - function may take either 1 or 2 value
- {0,2} - function may take either 0 or 2 value

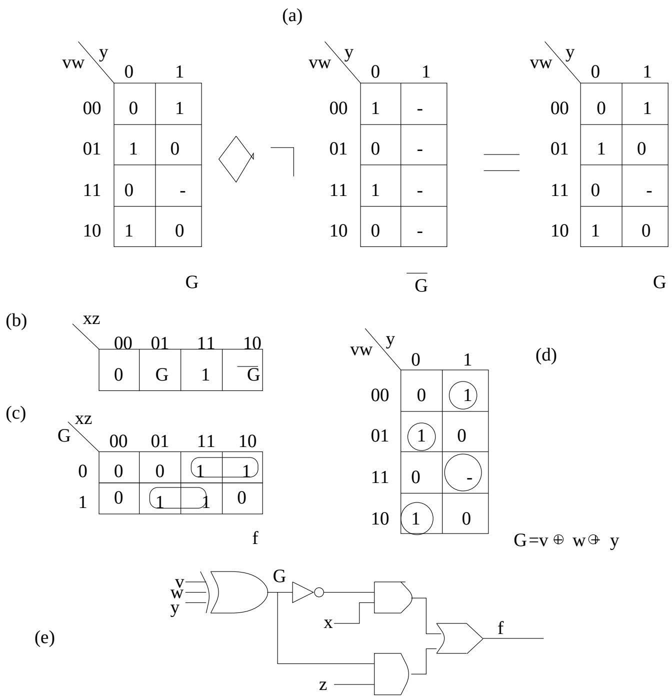


Figure 20: *The decomposition for free set* $\{x, z\}$

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 21: *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 22: *A Repeated Variable Map for function from Example 7.3 with repeated variable b*

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

As an example, let us again consider the function from Fig. 21 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. 21 we will come out with K-map in Fig. 22. 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.

8 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 8.1 *By a loop pair we will understand a pair:*

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

Definition 8.2 *Two functions are called the* compatible functions *when they can be completed to the same function.*

Definition 8.3 *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 8.4 *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 8.5 *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.

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 23: *A Karnaugh map to Example 8.1*

Example 8.1

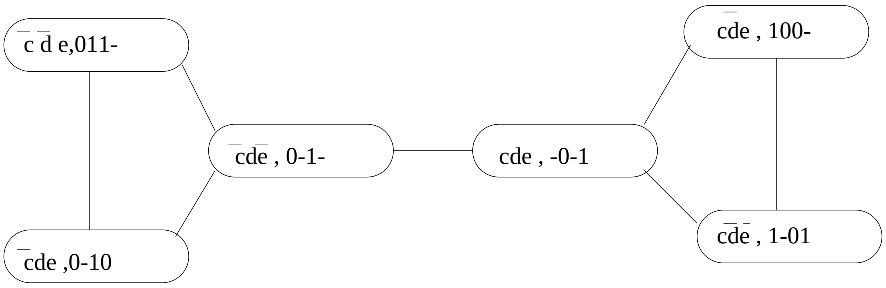


Figure 24: *A loop compatibility graph to Example 8.1*

Let us consider a function given in a Karnaugh map of Fig. 23, 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 $\bar{c}\ \bar{d}\ \bar{e}$ of 1's and the loop $c\ d\ \bar{e}$ of 0's, the *loop pair Compatibility Graph* of Fig. 24 is created with the remaining loop pairs as nodes.

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 $\bar{c}\ d\ e$ and $\bar{c}\ d\ \bar{e}$). 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 $\bar{c}\ d\ e$) 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:

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

Compatibility of loop pairs is denoted as an edge between nodes

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

In our notation, the nodes: [$\bar{c}\ d\ e$, 0-10] and [$\bar{c}\ d\ \bar{e}, 0-1-$] (see Fig. 24).

Similarly, loop pairs [$\bar{c}\ d\ \bar{e}$, 0-1-] and [$c\ d\ e$, -0-1] are compatible, since 0-1- $\diamond$ -0-1 = 0011. However loop pairs [$\bar{c}\ d\ e$, 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. 24. 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.

8.1 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 [145, 69, 92, 122, 165, 166, 131, 130]. 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 8.1

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 8.2

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 8.2

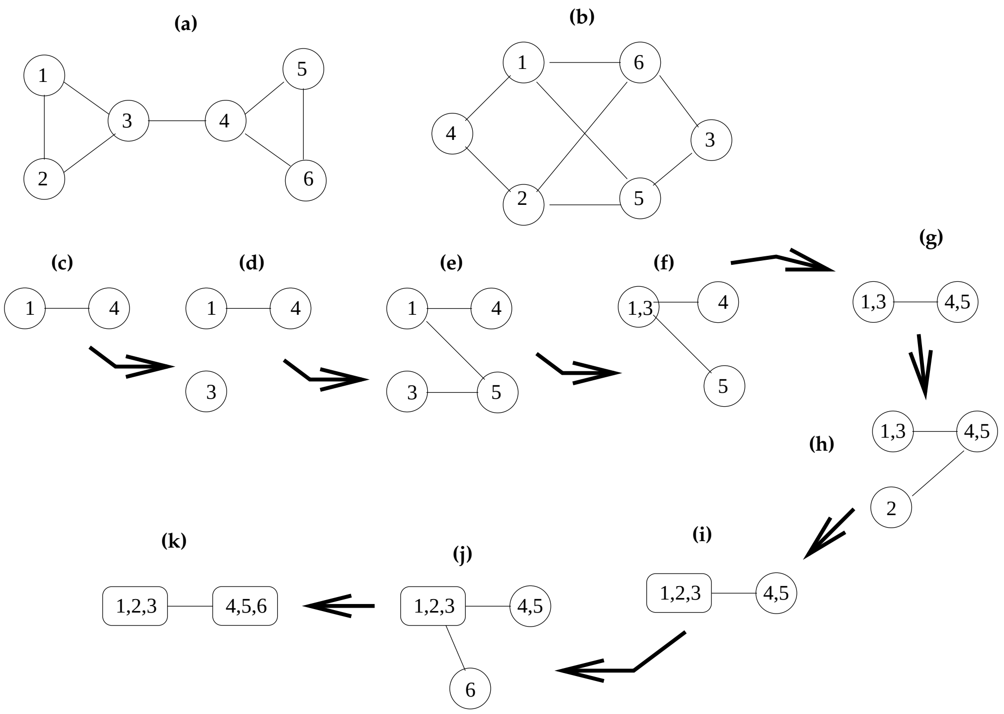


Figure 25: *Illustration of the 2-coloring algorithm: (a) the compatibility graph G, (b) the corresponding incompatibility graph $\bar{G}$, (c) - (k) stages of coloring*

Given is the graph from Figure 25a, that is the same graph as in Example 8.1. Its complement graph, the incompatibility graph is shown in Figure 25b. Let us assume that a pair of incompatible nodes initially selected is [{1}, {4}], Figure 25c. Now, node 3 is selected, Figure 25d. It is compatible with both nodes 1 and 4. Now, node 5 is selected, Figure 25e. It is incompatible with both nodes 1 and 3, but compatible with node 4. Thus, nodes 1 and 3 are combined, Figure 25f, and next nodes 4 and 5 are combined, Figure 25g, leading to graph with a pair $[\{1,3\},\ \{4,5\}]$. Now, node 2 is selected, Figure 25h. It is incompatible with nodes {4,5} but compatible with node {1,3}. Nodes 2 and {1,3} are combined, Figure 25i, leading to graph with a pair $[\{1,2,3\},\ \{4,5\}]$. Now, node 6 is selected, Figure 25j. It is incompatible with nodes {1,2,3} but compatible with node {4,5}. Nodes 6 and {4,5} are combined, Figure 25k, 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.

9 PUB Decompositions.

9.1 Disjoint PUB Decomposition.

The method shown here was first introduced by Perkowski, Uong and Brown in 1987, [165, 166]. 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 9.1 *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 9.1 *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,*
- *$(c_1,\ c_2) \in RS \iff$ 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 9.1

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 26: *A Karnaugh map for Example 9.1*

The function is presented in a Karnaugh map from Fig. 26.
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.

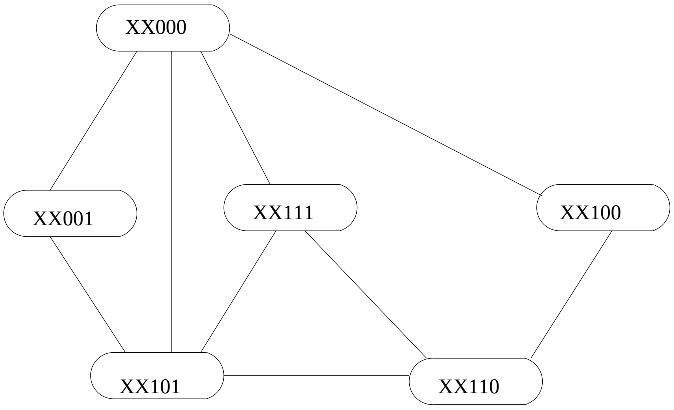


Figure 27: *The loop pair incompatability graph for Example 9.1*

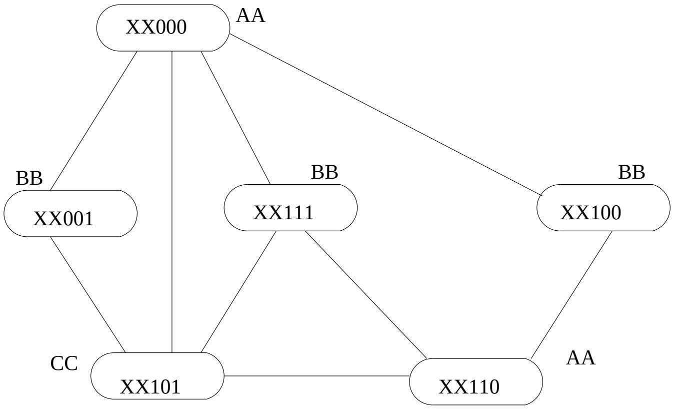


Figure 28: *The graph from Fig. 26 after coloring*

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. 27 is created.

4. Find minimal coloring of the loop pair incompatibility graph.

   After coloring with a minimum number of colors the graph from Fig. 27 becomes as in Fig. 28.

   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:

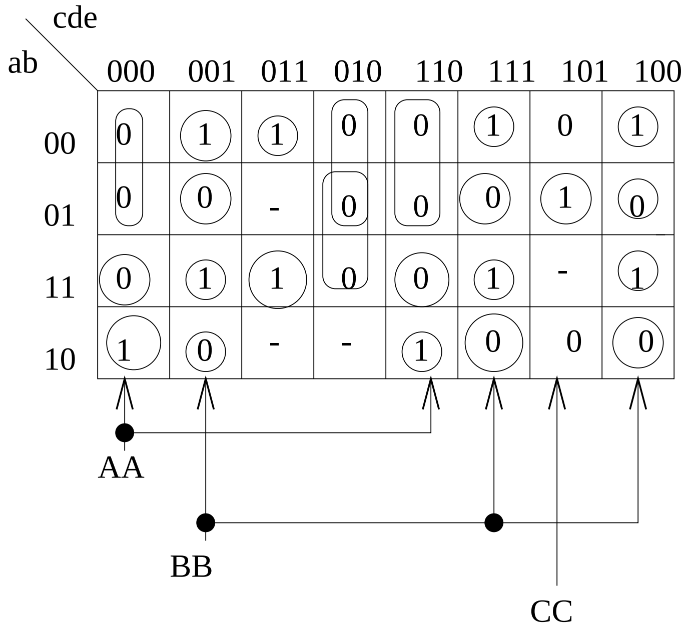


Figure 29: *The function after the completion of loop functions, the columns colored with the same colors are shown (the column multiplicity of 3)*

ON(XX000) $\cup$ ON(XX110) = {10XXX} $\cup$ {10XXX} = {10XXX}
OFF(XX000) $\cup$ OFF(XX110) = {00XXX, 11XXX} $\cup$ {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) $\cup$ ON(XX111) $\cup$ ON(XX100) =
{11XXX} $\cup$ {00XXX} $\cup$ {11XXX} = {11XXX, 00XXX}
OFF(XX001) $\cup$ OFF(XX111) $\cup$ OFF(XX100) =
{01XXX} $\cup$ {10XXX} $\cup$ {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. 29.
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

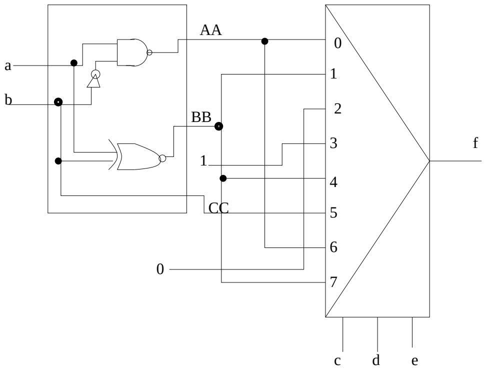


Figure 30: *The PUB decomposition with column multiplicity of 3 realized with a multiplexer and gates*

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. 30. 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 [165].

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 [165]:

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:

$$EXOR\_EXPR = f_{COF_i} \oplus x_j \oplus x_k \oplus x_l \tag{2}$$

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).

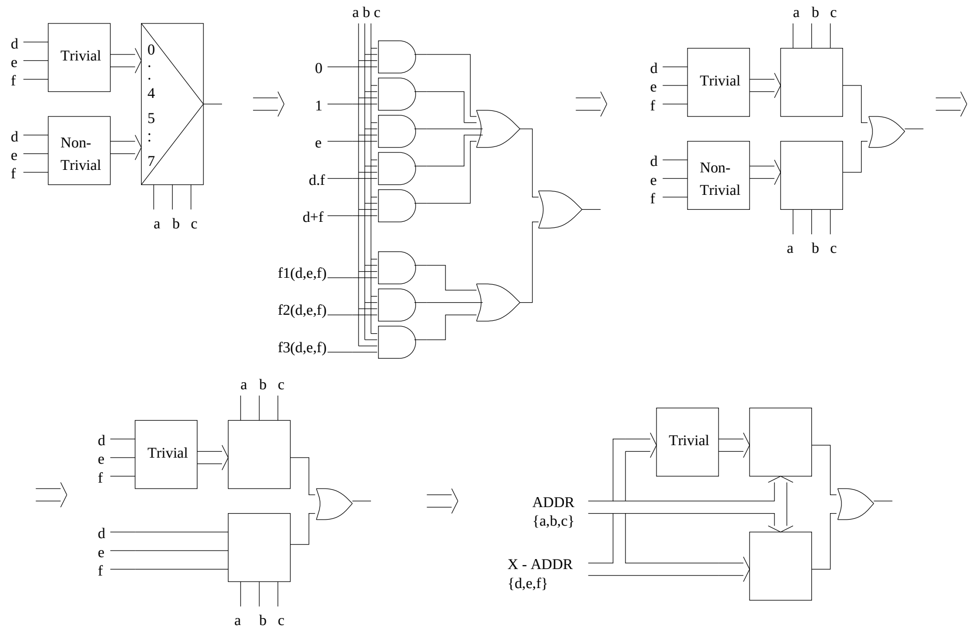


Figure 31: *Different views at the Disjoint PUB Decomposition*

Figure 31 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.

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 32: *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 2: *ON and OFF cofactor cubes of a function from Example 9.2*

9.2 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:

Example 9.2

Let us analyse the function from Example 9.1 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. 32 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 9.1 we then find cofactors of columns (Table 2). 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. 33. 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):

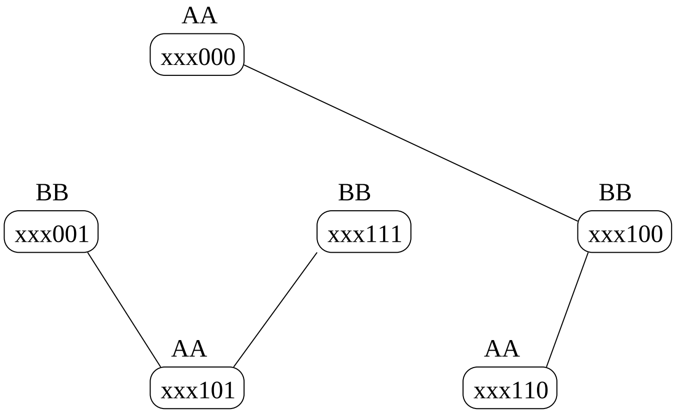


Figure 33: *Incompatibility graph for non-disjoint PUB decomposition*

AA:
ON(XXX000) ∪ ON(XXX110) ∪ ON(XXX101) = {100XXX, 011XXX}
OFF(XXX000) ∪ OFF(XXX110) ∪ OFF(XXX101) = {0X0XXX, 110XXX, 001XXX, 101XXX}

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. 34. 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. 35.

Fig. 36 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

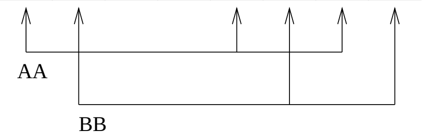


Figure 34: *The K-map coresponding to the completed function with coloring of compatible columns shown and the multiplicity index of 2*

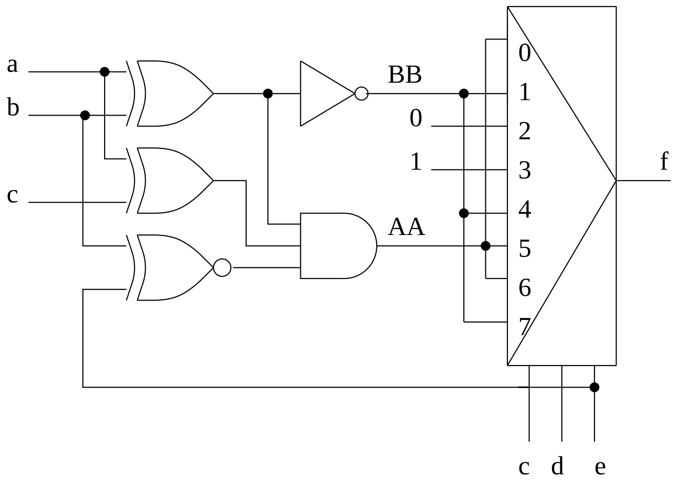


Figure 35: *Realization of the PUB decomposition of function from Example 9.2 using a multiplexer*

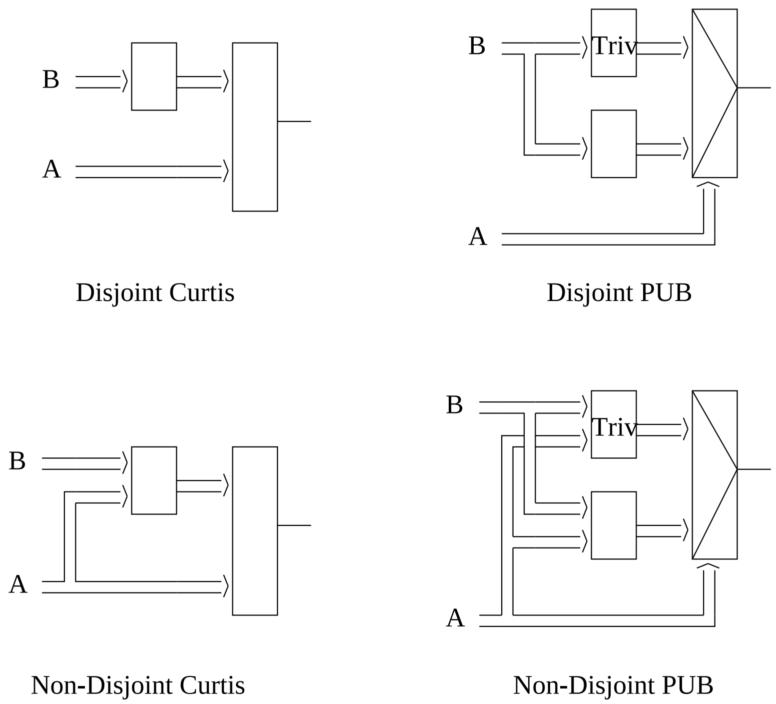


Figure 36: *Comparison of the disjoint and non-disjoint Curtis and PUB decompositions*

10 Curtis Decomposition

10.1 Disjoint Curtis Decomposition.

A decomposition approach based on Graph Coloring, very similar to the one from section 9, can be applied to Curtis Decomposition. It will not be repeated it here, since it was presented in detail in [130, 131]. 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 37. 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.

10.2 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 10.1

Let us consider the function f from Example 9.2 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. 38a. 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 column multiplicity of 2 is shown in Fig. 38b. Loops 000, 010 and 101 contain only 0s and loop 111 contains only

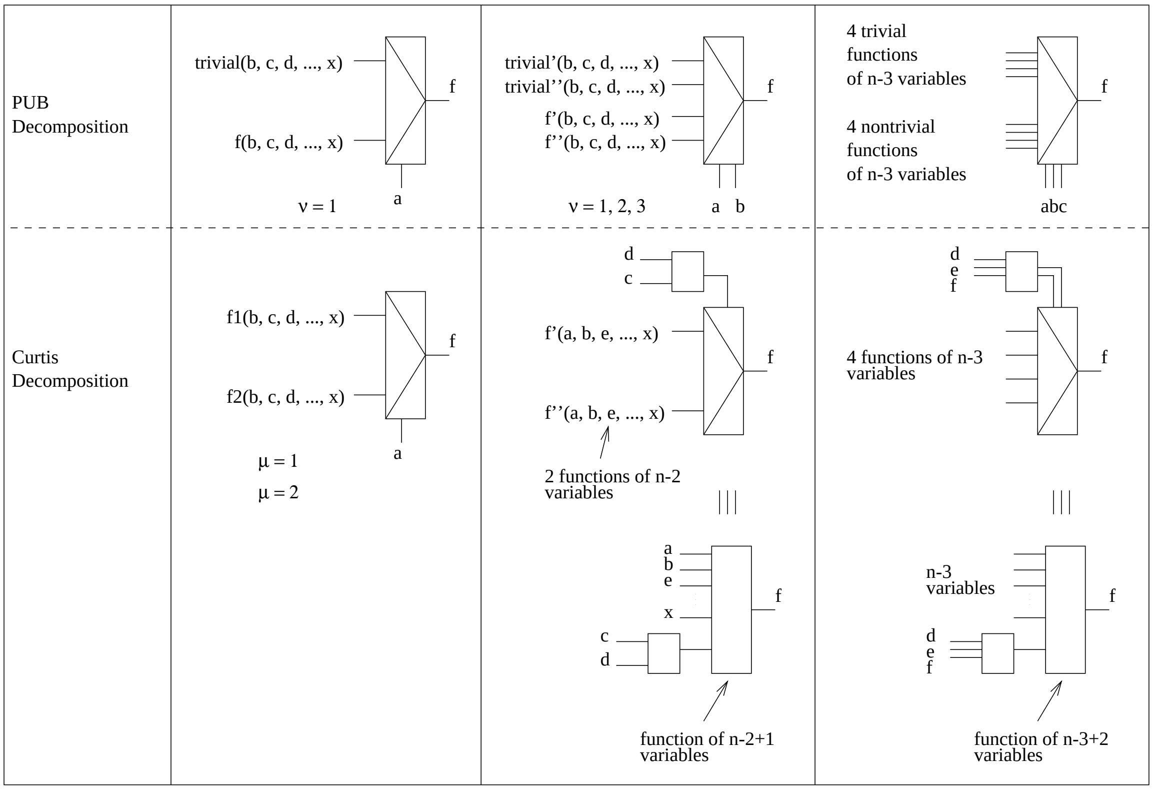


Figure 37: *Comparison of PUB and Curtis decompositions for small k*

a)

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

b)

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

Figure 38: *K-map of function f to Example 10.1*

1s. If we define the loop function corresponding to $a = 0$, $b = 0$, $e = 1$ as G we will see from Fig. 38b 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: $f(a, b, c, d, e) = H[G(a, b, e), c, d, e]$.

11 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 [222] and [223]). 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 [130].

   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, [130].

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* [167]. 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 [130]. 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 11.1

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 39a. 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 39b. The stages of coloring the graph are shown in Figure 39b - e. In Figure 39b 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.

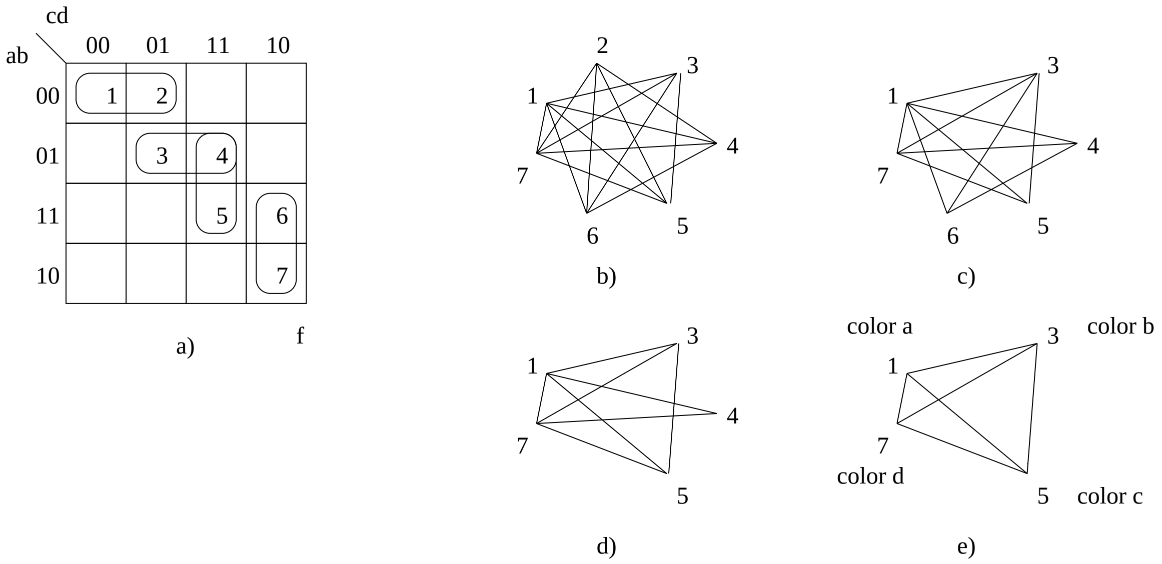


Figure 39: *Proper coloring of a non-cyclic graph: (a) a non-cyclic covering problem formulated as SOP minimization, (b) - (e) stages of coloring of the graph corresponding to the covering problem from (a)*

3c. Now node 6 has neighbors 1,3,4 and node 7 has neighbors 1,3,4,5. Thus node 7 dominates node 6 and node 6 is removed. This leads to the graph from Figure 39d. 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 39e, 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 [190]. 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 23.2.

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 [171]. 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 [192]. 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 [223].

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

Decompositions of Multiple-Valued Functions have been discussed by:

- Thelliez [17] (only for ternary logic).
- Walliuzzaman and Vranesic [51].
- Fang and Wojcik [18].
- Luba, Mochocki and Rybnik [144, 142, 141].
- Abugharrbieh and Lee [19, 20].

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, [51], and Thelliez, [17], 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, [18]. 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" [145, 124, 123, 126]. 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 [179]. 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, [71, 106, 119]. 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 [21] or as a base for design a new one [22, 23]. Quality of decomposition depends mainly on the analyst's experience. Some authors try to provide a formal approach to decomposition [24]. Decompositional approach similar in essence to the one proposed earlier by Armstrong [25] 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 [24]: "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.

13 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 13.1 *A Multiple-Valued input, binary output function with n input, m output variables is defined as a mapping:*

$$F(x_1, ..., x_n) : C_1 \times C_2 \times \ldots \times C_n \longrightarrow D^m \tag{3}$$

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

$$F(x_1, ..., x_n) : C_1 \times C_2 \times \ \ldots \times C_n \longrightarrow D_1 \times D_2 \times \ldots \times D_m \tag{4}$$

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

$$F(x_1, ..., x_n) : C_1 \times C_2 \times \ \ldots \times C_n \longrightarrow 2^{D_1} \times 2^{D}_{2} \times \ \ldots \times 2^{D_m} \tag{5}$$

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.:

$$a : M \longrightarrow V_a \tag{6}$$

  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* [121].

Definition 13.4 *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

$$IND(B) = \bigcap_{x \in B} IND(x) \tag{7}$$

Thus, the relation IND partitions M into equivalence classes $M/IND(B)$. Such partitions are of primary importance in logic synthesis [162]. 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

$$P(B) = \prod_{x \in B} P(x) \tag{8}$$

where $\prod$ denotes the product of partitions.

Definition 13.5 *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: *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 [44], 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 13.1

Consider a multiple-valued function F given in Table 3.

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 4: *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 5: *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 4, 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 13.6 *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 13.2

Let us analyze one more example (Table 5).

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 40: *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 40.

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.

14 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 14.1

For instance, in Example 13.2, we have:

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

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

Figure 41: *Standard enumeration of cells of a 3-variable Boolean function*

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 41 we obtain the Karnaugh maps as in Figure 42, Figure 43:

The function from Fig. 42 is: $P_1(X_0) = \bar{b} + \bar{a}\bar{c}$. The function from Fig. 43 is: $P_2(X_0) = a + b$. Their respective BDDs are shown in Figure 44.

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 42: *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 43: *Karnaugh map for $P_2(X_0)$ for Example 14.1*

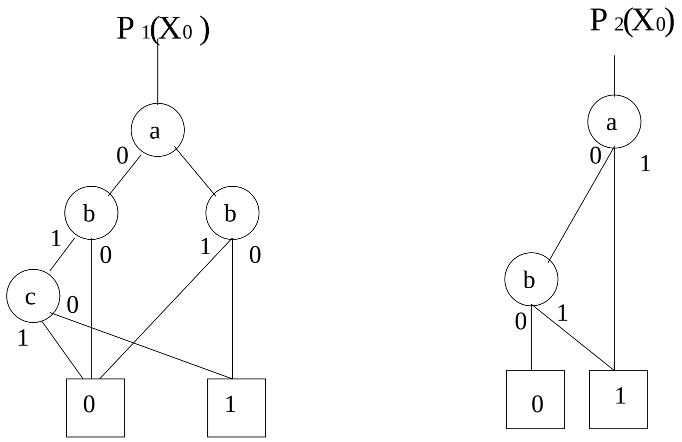


Figure 44: *The BDDs for functions $P_1(X_0)$ and $P_2(X_0)$*

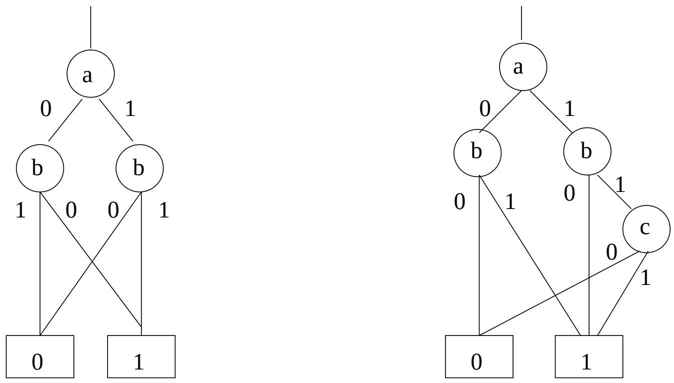


Figure 45: *Partitions* Π_1 *and* Pi_2 *to Example 14.1:* $\Pi_1 = \bar{a}\bar{b} + ab$, $\Pi_2 = a\bar{b} + bc + \bar{a}b$

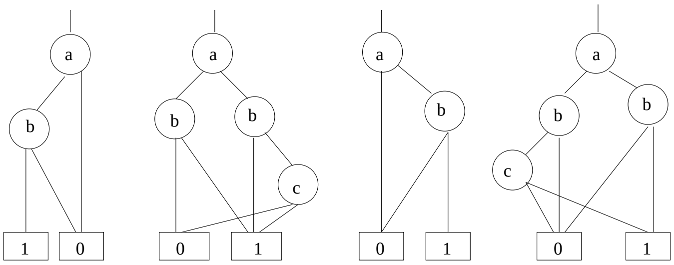


Figure 46: *Partitions to Example 14.1:* $P_1\Pi_1 = \bar{a}\bar{b}$, $P_2\Pi_2 = a\bar{b} + bc + \bar{a}b$, $P_2\Pi_1 = ab$, $P_1\Pi_2 = a\bar{b} + \bar{a}b\bar{c}$

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. 45).

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. 46).

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.

15 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

$$F = H(A, G(B, C)) = H(A, Z) \tag{9}$$

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 [118], 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 [72], 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 [145] states the sufficient condition for the existence of a serial decomposition.

Theorem 15.1 *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*

$$P(A) \cdot \Pi_G \leq P_F \tag{10}$$

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 $\lceil \log_2 L(\Pi_G) \rceil$, here $L(\Pi)$ denotes the number of blocks of partition Π, and $\lceil x \rceil$ denotes the smallest integer equal to or large than x.

Here $\lceil \log_2 L(\Pi_G) \rceil$ 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 23, 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 15.1

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 6: *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 7: *A Truth Table of output functions*

Let us decompose the function F of Table 3. 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 6. 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 7.

Example 15.2 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 $\lceil \log_2 2 \rceil = 1$.

The questions are:

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

Definition 15.1 *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 15.2 *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 15.3 *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 15.3

For example, in Table 5, repeated in Table 8, 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 8: *A Repeated Table for Example 15.3*

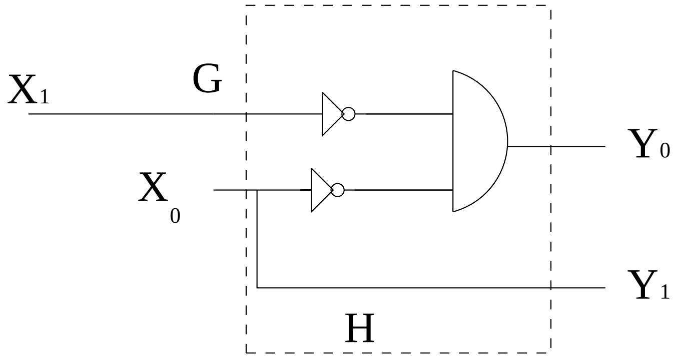


Figure 47: *Decomposition of function f from Example 15.3*

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 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 47. 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.

16 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 [106, 204].

Let F be a mapping function

$$F : c_1 \times \ldots \times c_n \longrightarrow D \tag{11}$$

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 16.1 *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*

$$P(A) \cdot \Pi_G \leq P(D) \tag{12}$$

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 48. 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 9).

Example 16.1

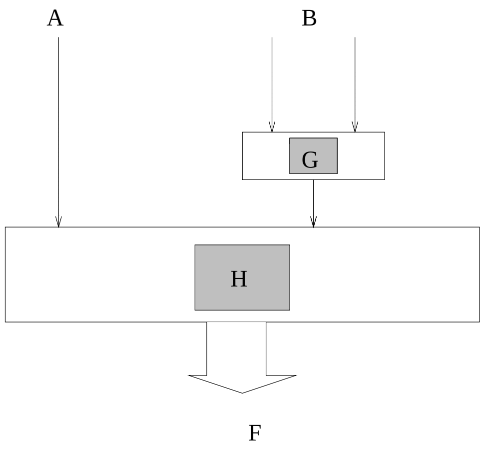


Figure 48: *Block Diagram of the Decomposition of Decision Tables*

	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 9: *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 10: *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 11: *A Truth Table of function H*

Let us decompose the function F of Table 9 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 [145].

17 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.

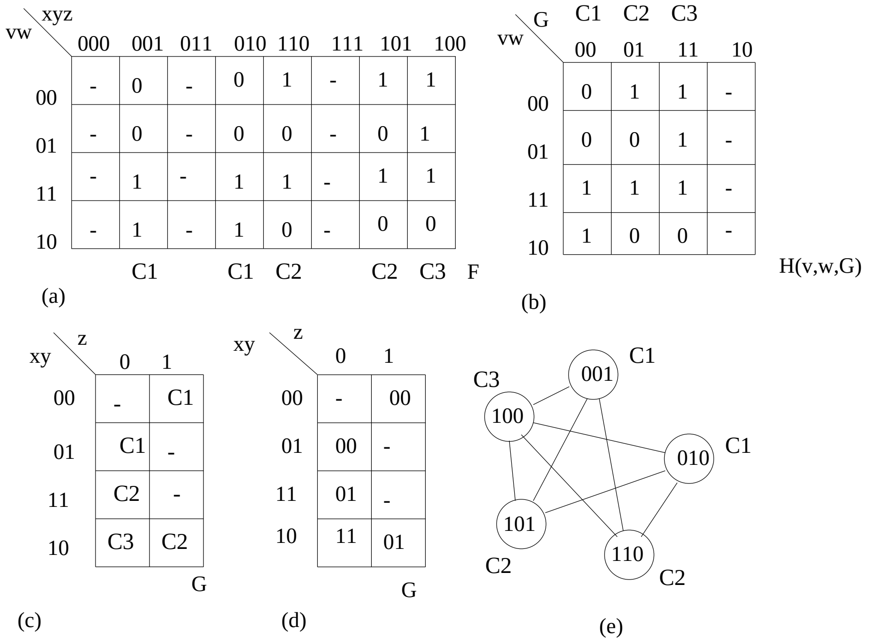


Figure 49: *Example of treating columns of don't cares: (a) Function $F(x, y, z, v, w)$, (b) Function H with a column of don't cares, (c) Function G with symbolic values, (d) Function G with binary values, (e) The incompatibility graph*

In Figure 49, 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 18. The Graph Coloring Problem will be discussed in section 19.

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. 49. 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.

17.1 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. 49.

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 our forthcoming report.

18 The Set Covering Approach to the Column Compatibility Problem.

18.1 The compatibility relation.

As it is seen from 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 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 18.1 *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 15.1, i.e., if and only if:*

$$P(A) \cdot P_{ij}(B) \leq P_F \tag{13}$$

Definition 18.2 *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 18.3 *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 18.1

For the function of Example 13.1, and the same sets A, B as in 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

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

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

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

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:

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

The Compatibility Graph drawn from this relation is shown in Figure 50. 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 50b.

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:

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

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 [80, 70], 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:

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

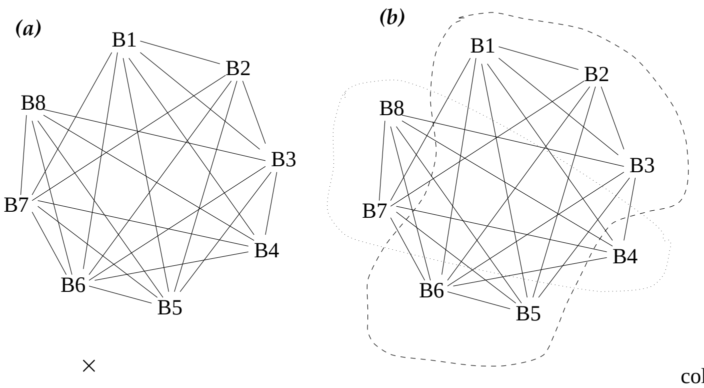
is a block of Π_G if and only if $\beta \subseteq$ MCC and there is no β' such that $\beta \cap \beta' \neq \emptyset$. Thus blocks of Π_G can be created from MCC by eliminating the repeated elements of B in the minimal cover MIN(M). The final Π_G is a result of the union of objects forming a set of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

Example 18.2

For the MCCs from Example 18.1, the covering table is shown in Figure 50c. 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 15.1.

Example 18.3

The coloring formulation of the same problem, using the Incompatibility Graph, is shown in Figure 50d. 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 50d 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.



(c)

	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	×		×	×		×		

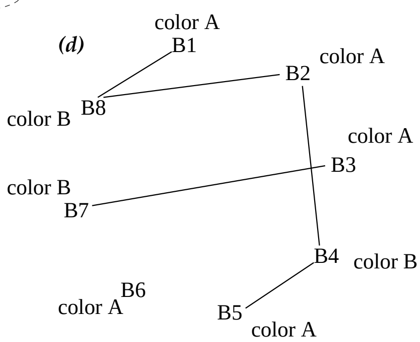


Figure 50: *Comparison of techniques to solve the Column Minimization Problem: (a) The Compatibility Graph, (b) The clique partitioning of the Compatibility Graph, (c) The covering table with maximum cliques from the Compatibility Graph as rows and its nodes as columns, (d) The graph coloring formulation of the same problem using the Incompatibility Graph being the complement of the Compatibility Graph*

19 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*, [13, 14, 15, 16] 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 [13] 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 51. 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.

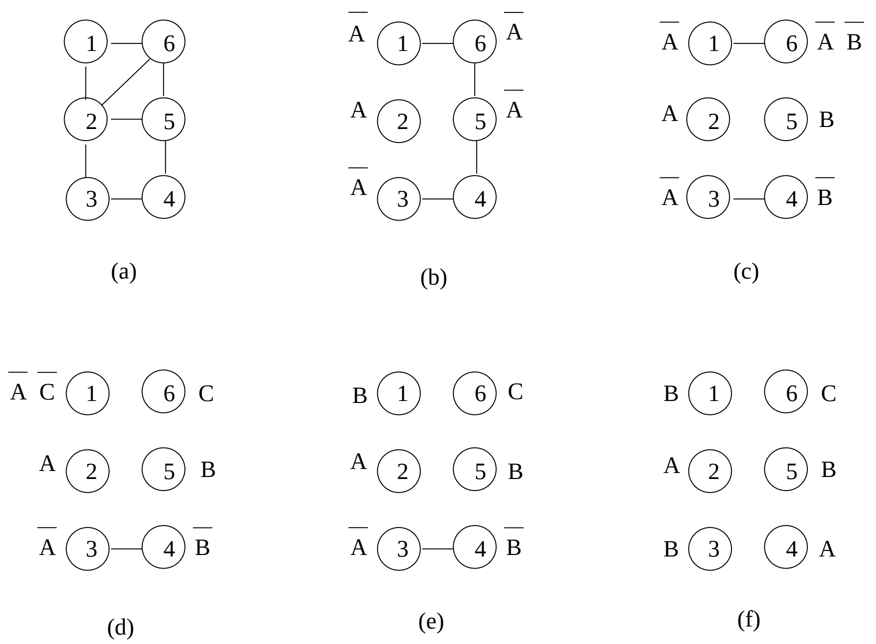


Figure 51: *The incompatibility graph for coloring*

Figure 51a 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 51b. 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 51c. The same way, color C is assigned to node 6, as shown in Figure 51d. Next, color B is assigned to node 1, as shown in Figure 51e. 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 51f.

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 52.

```
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 52: *Pseudo-code of graph coloring*

20 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.

20.1 Boolean Operations

The following Boolean operations are necessary; *derivative, minimum* and *maximum.* They are defined as follows:

1. Derivative:
$$\frac{\partial f(a,b,c)}{\partial a} = f(a=0,b,c) \oplus f(a=1,b,c) \tag{14}$$
2. Minimum
$$\min_a f(a,b,c) = f(a=0,b,c) \cdot f(a=1,b,c) \tag{15}$$
3. Maximum
$$\max_a f(a,b,c) = f(a=0,b,c) + f(a=1,b,c) \tag{16}$$

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 20.1

For a given function:

$$f(a,b,c,d) \;=\; a\bar{b} \;+\; \bar{a}b \;+\; \bar{a}cd \tag{17}$$

Derivative:

$$\begin{aligned}\frac{\partial^2 f(a,b,c,d)}{\partial a \partial b} &= f(a=0,b=0,c,d) \oplus f(a=0,b=0,c,d)\\ &\oplus f(a=0,b=1,c,d) \oplus f(a=1,b=0,c,d) \oplus\\ &f(a=1,b=1,c,d) = cd \oplus 1 \oplus 1 \oplus 0 \;=\; cd\end{aligned} \tag{18}$$

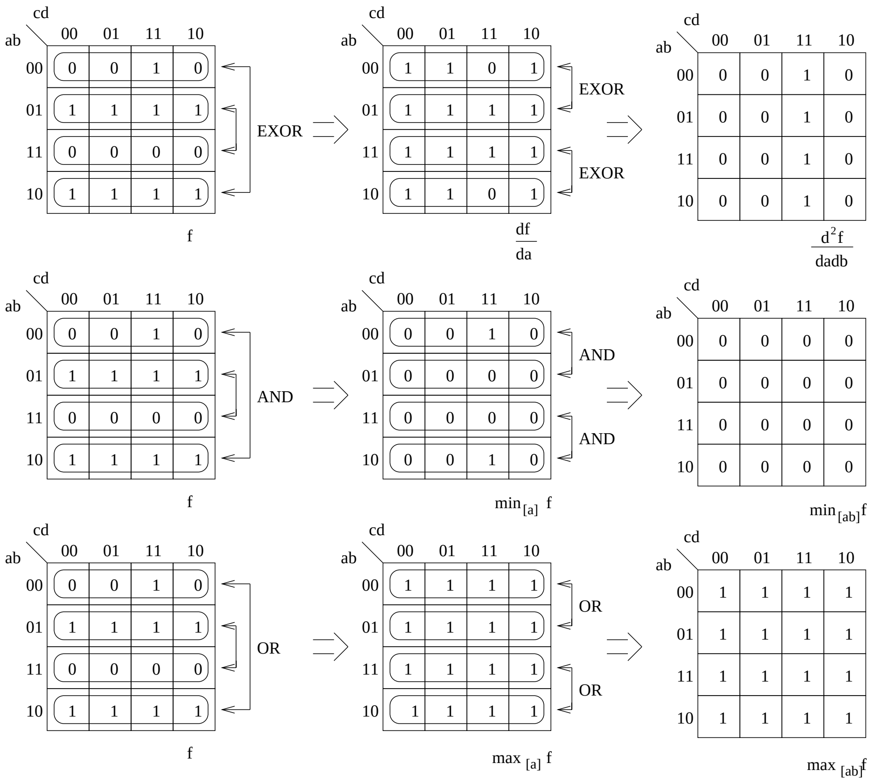


Figure 53: *Graphical illustration of operations derivative, minimum and maximum for minterms*

Minimum:

$$\begin{aligned} min^2_{[ab]} f(a,b,c,d) \ &= \ f(a=0,b=0,c,d) \cdot f(a=0,b=1,c,d) \cdot f(a=1,b=0,c,d) \cdot \\ & f(a=1,b=1,c,d) = cd \cdot 1 \cdot 1 \cdot 0 \ = \ 0 \end{aligned} \tag{19}$$

$$\begin{aligned} min^2_{[cd]} f(a,b,c,d) \ &= \ f(a,b,c=0,d=0) \cdot f(a,b,c=0,d=1) \cdot f(a,b,c=1,d=0) \cdot \\ & f(a,b,c=1,d=1) = (\overline{a}b + a\overline{b}) \cdot (\overline{a}b + a\overline{b}) \cdot (\overline{a}b + a\overline{b}) \cdot \\ & (\overline{a}b + a\overline{b} + \overline{a}) = \overline{a}b + a\overline{b} \end{aligned} \tag{20}$$

Maximum:

$$\begin{aligned} & max^2_{[ab]} f(a,b,c,d) \ = \ f(a=0,b=0,c,d) \ + \ f(a=0,b=1,c,d) \ + \\ & f(a=1,b=0,c,d) \ + \ f(a=1,b=1,c,d) = cd \ + \ 1 \ + \ 1 \ + \ 0 \ = \ 1 \end{aligned} \tag{21}$$

Graphical illustration of operations derivative, minimum and maximum for minterms in a completely specified function using K-maps are shown in Fig. 53.

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.

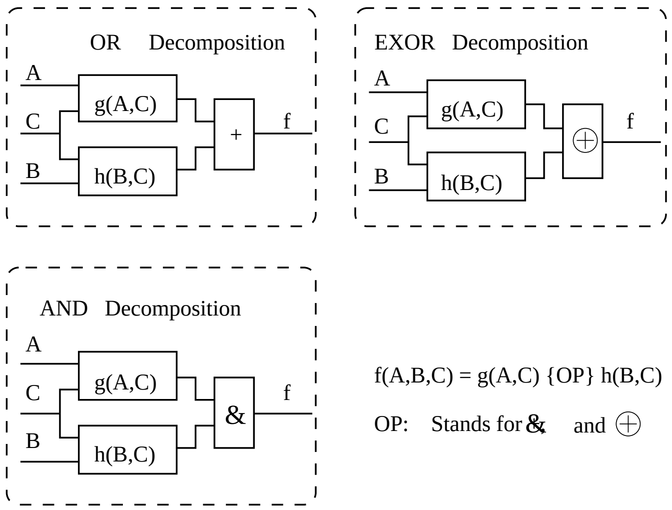


Figure 54: *Strong OR (Disjunctive), AND (Conjunctive) and EXOR (Linear) Decompositions.*

20.2 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 20.1 *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*

$$f(A,B,C) \;=\; g(A,C) \;\bigcirc\; h(B,C) \tag{22}$$

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.

Definition 20.2 *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*

$$f(A,B) \;=\; h(A,B) \;\bigcirc\; g(A) \tag{23}$$

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.

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 55: *The Care Functions f_{ON} and f_{OFF} of function $f(a,b,c,d)$ to Example 20.2*

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 20.2

Let us look at the example in Fig. 55, 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 20.3

Graphical ilustrations of *derivative, min*, and *max* operations for function from Figure 56a are shown in Fig. 56b, 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 (24):

$$(A,B,C) \ \preceq \ (A',B',C') \Longleftrightarrow (A \subseteq A') \ \wedge \ (B \subseteq B) \tag{24}$$

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 (24).

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.

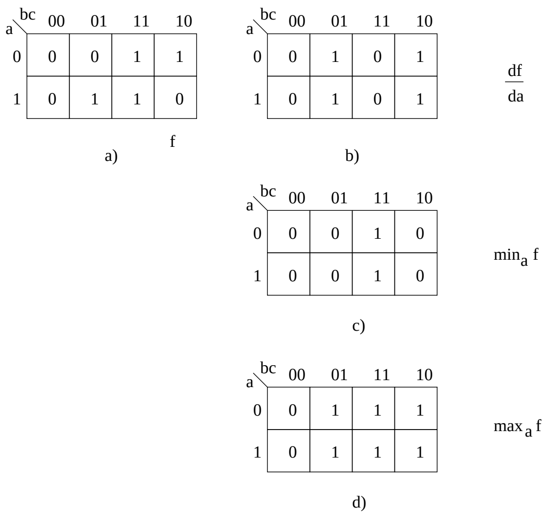


Figure 56: *Example 20.3: (a) function $f(a,b,c)$, (b) its derivative with respect to variable a, (c) its minimum with respect to variable a, (d) its maximum with respect to variable a*

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 20.1 *An incompletely specified function with functions $f(X)_{ON}$ and $f(X)_{OFF}$ for A and B is* OR decomposable, *if and only if*

$$f(X)_{ON} \;\cdot\; \max_A{}^k f(X)_{OFF} \;\cdot\; \max_B{}^k f(X)_{OFF} \;=\; 0 \tag{25}$$

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 20.2 *An incompletely specified function with functions $f(X)_{ON}$ and $f(X)_{OFF}$ for A and B is* AND decomposable, *if and only if*

$$f(X)_{OFF} \;\cdot\; \max_A{}^k f(X)_{ON} \;\cdot\; \max_B{}^k f(X)_{ON} \;=\; 0 \tag{26}$$

Example 20.4

Using the above theorem is illustrated in Figure 57.

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 $FB(X,f)$.

Algorithm 20.1

```
Subroutine find_A_B

GIVEN:
```

(a) f_{ON}

ab \ cd	00	01	11	10
00	0	0	1	0
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

(b) f_{OFF}

ab \ cd	00	01	11	10
00	1	1	0	1
01	0	0	0	0
11	1	1	1	1
10	0	0	0	0

(c) $\max_{[ab]} f_{ON}$

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

(d) $\max_{[cd]} f_{ON}$

ab \ cd	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

(e) $f_{OFF} * \max_{[ab]} f_{ON} * \max_{[cd]} f_{ON}$

ab \ cd	00	01	11	10
00	1	1	0	1
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

Figure 57: *Function f to Example 20.4 - graphical calculation of max and intersection*

```
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 $g(A, C)_{ON} = \text{M1g}$ and $g(A, C)_{OFF} = \text{M0g}$ as well as $FB(B, C, \{h\})$ with $h(B, C)_{ON} = \text{M1h}$ and $h(B, C)_{OFF} = \text{M0h}$ can be calculated for the OR decomposition with the formulas (27) to (30), if $FB(x, \{f\})$ is AND decomposable with respect to (A, B, C).

$$g(A, C)_{ON} \;=\; \max_B{}^k f(X)_{ON} \;\cdot\; \overline{\max_B{}^k f(X)_{OFF}} \tag{27}$$

$$g(A, C)_{OFF} \;=\; \max_B{}^k f(X)_{OFF} \tag{28}$$

$$h(B, C)_{ON} \;=\; \max_A{}^k [\; f(X)_{ON} \;\cdot\; \overline{g(A, C)_{ON}} \;] \tag{29}$$

$$h(B, C)_{OFF} \;=\; \max_A{}^k f(X)_{OFF} \tag{30}$$

For the AND decomposition one obtains the decomposed functions g, h using formulas (31) to (34).

$$g(A, C)_{ON} \;=\; \max_B{}^k f(X)_{ON} \tag{31}$$

$$g(A, C)_{OFF} \;=\; \max_B{}^k f(X)_{OFF} \;\cdot\; \overline{\max_B{}^k f(X)_{ON}} \tag{32}$$

$$h(B, C)_{ON} \;=\; \max_A{}^k f(X)_{ON} \tag{33}$$

$$h(B,C)_{OFF} \;=\; \max_{A}{}^{k}[\; f(X)_{OFF} \;\cdot\; \overline{g(A,C)_{OFF}}\;] \tag{34}$$

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 (27) to (30) or (31) to (34).

Algorithm 20.2

```
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;
```

20.3 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 20.1.

Lemma 20.1 The derivative of an incompletely specified function $f(X)$ with respect to variable x_a *is an incompletely specified function defined as follows:*

$$f_{x_a}(X \setminus x_a)_{ON} := \max_{x_a} f(X)_{ON} \cdot \max_{x_a} f(X)_{OFF} \tag{35}$$

$$f_{x_a}(X \setminus x_a)_{OFF} := \min_{x_a} f(X)_{ON} + \min_{x_a} f(X)_{OFF} \tag{36}$$

Example 20.5

Figure 58 illustrates calculation of the formulas from Lemma 20.1 using Kmaps.

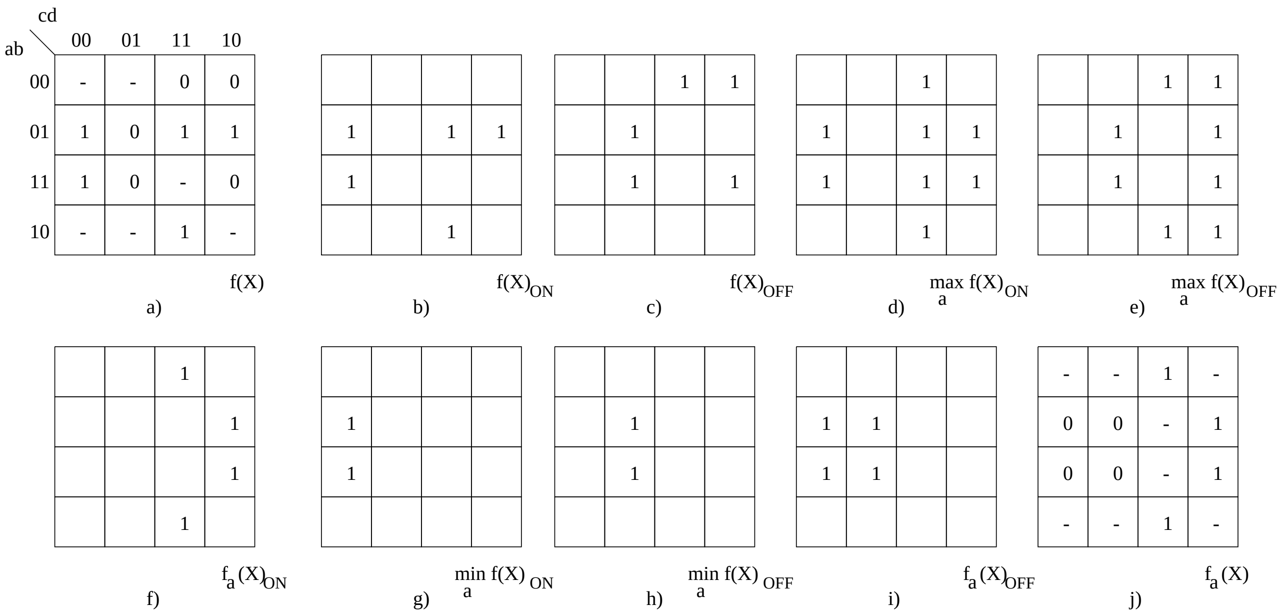


Figure 58: *Calculation of a derivative $f_a(X)$ of an incompletely specified function $f(X)$ from Example 20.5 using min and max formulas on Kmaps*

Figure 58f shows $f_a(X)_{ON} = \max_a\ f(X)_{ON} \cdot \max_a\ f(X)_{OFF}$, and Fig. 58i illustrates $f_a(X)_{OFF} = \min_a\ f(X)_{ON} + \min_a\ f(X)_{OFF}$. Finally, Fig. 58j shows the composite incompletely specified derivative composed from Fig. 58f and Fig. 58i.

The same map of incomplete derivative from Fig. 58j 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 (35) 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 (36), 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 20.3

```
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 20.3 Function $f(X)$ is EXOR decomposable *with respect to variable x_a and set of variables B if:*

$$\frac{\partial^2 f(X)}{\partial x_a \partial x_j} = 0 \tag{37}$$

for all variables x_j in set B.

Definition 20.4 *Function $f(X)$ is linearly groupable with respect to variable sets A and B if:*

$$\frac{\partial^2 f(X)}{\partial x_i \partial x_j} = 0 \tag{38}$$

for all variables x_j in set B, and for all variables x_i in set A.

Example 20.6

Given is function $f(X) = f(a, b, c, d)$ from Figure 59a. Its graphically obtained derivatives with respect to variables a and b are shown in Figure 59b and Figure 59c, respectively.
It is immediately seen in Figure 59b 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 59c 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

a) f(X)

1	1	1	1
0	0	0	0
0	0	0	0
1	1	1	1

c) $\frac{d\, f(X)}{d\, a}$

1	1	1	1
1	1	1	1
0	0	0	0
0	0	0	0

c) $\frac{d\, f(X)}{d\, b}$

Figure 59: *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:

$$g(A, C) \;=\; f(A, B = c0, C) \tag{39}$$

$$h(B, C) \;=\; \max_A{}^k \; [\; f(X) \;\oplus\; g(A, C)\;] \tag{40}$$

Example 20.7

Given is function from Fig. 60. $B = 00$ was selected (Fig. 60a) and respective $g(A, C) = f(a = 0, b = 0, c, d) = c \oplus \overline{d}$ is found, Fig. 60b. Next $h(B, C)$ is found (Fig. 60e) by EXOR-ing $f(X)$ (Fig. 60c) and $g(A, C)$ (Fig. 60d). Finally *max* operation with respect to set A is applied to remove vacuous variables (Fig. 60f). 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 20.3 *(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:*

$$\max_B{}^k f_a(X)_{ON} \;\cdot\; \max_B{}^k f_a(X)_{OFF} \;=\; 0 \tag{41}$$

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 20.8

Figure 61 illustrates the calculation of EXOR decomposability according to Theorem 20.3. Fig. 61 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 20.4.

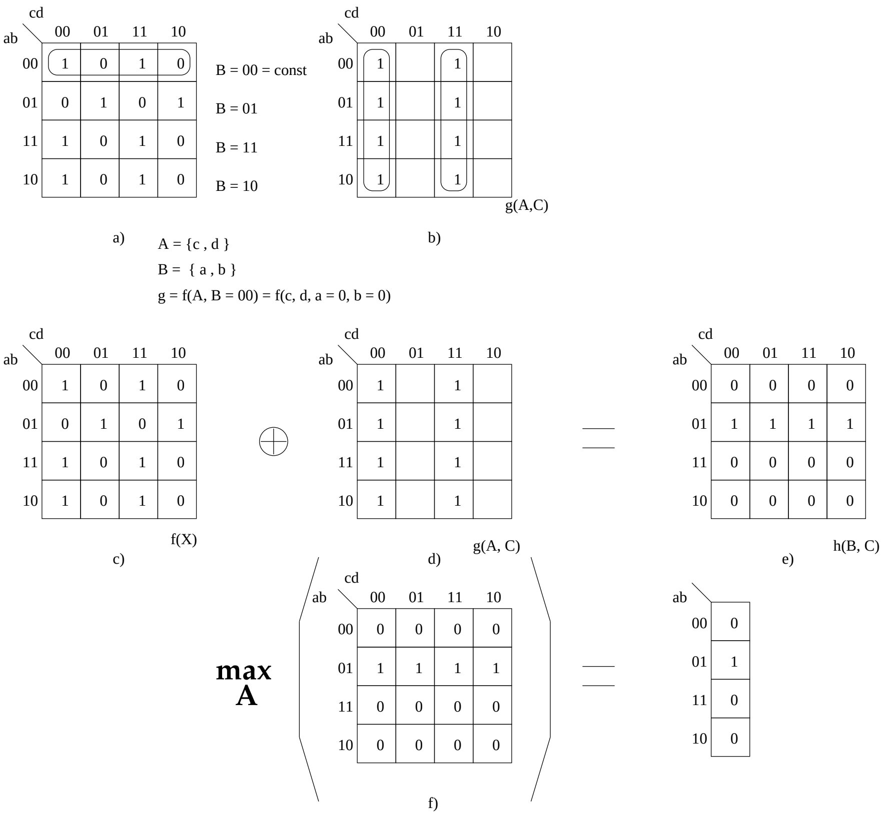


Figure 60: *Function to Example 20.7*

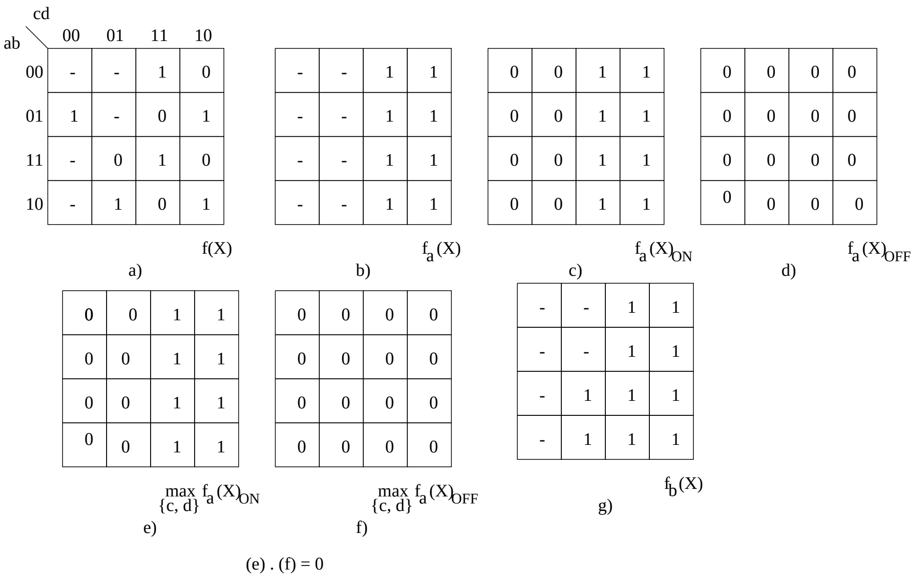


Figure 61: *Calculation of EXOR decomposability to Example 20.8*

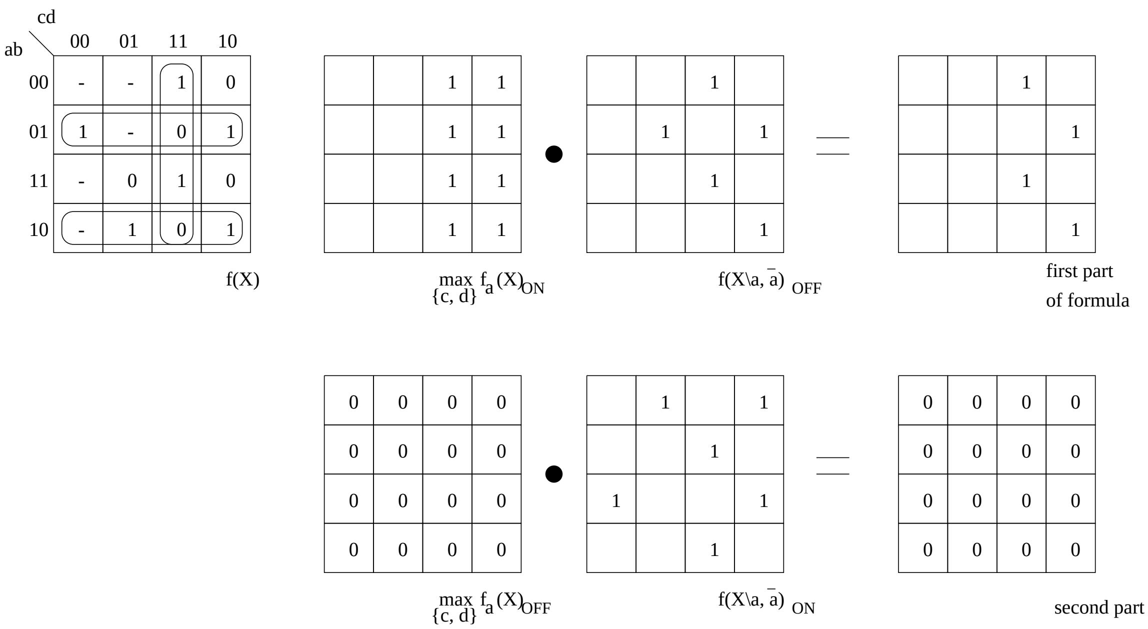


Figure 62: *Illustration of using Theorem 20.4 in Example 20.9*

Theorem 20.4 *The reduced incompletely specified function with care functions:*

$$f(X)_{ON} \;:=\; f(X)_{ON} + f(X \setminus x_a, \overline{x_a})_{OFF} \cdot \max_B{}^k f_a(X)_{ON} + f(X \setminus x_a, \overline{x_a})_{ON} \cdot \max_B{}^k f_a(X)_{OFF} \quad (42)$$

$$f(X)_{OFF} \;:=\; f(X)_{OFF} + f(X \setminus x_a, \overline{x_a})_{ON} \cdot \max_B{}^k f_a(X)_{ON} + f(X \setminus x_a, \overline{x_a})_{OFF} \cdot \max_B{}^k f_a(X)_{OFF} \quad (43)$$

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 20.9

Figure 62 illustrates the usage of Theorem 20.4.

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 60 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 (41) is used as a condition for the linear groupability (x_a, B, C).

While formula (41) 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 (41) 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 (41) to additionally satisfy formulas (42) and (43) 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 (42) and (43), 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 20.4

```
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 (42) and M0 according to equation (43).

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 (24), (35), (36), (41), (42), and (43) and uses the subroutines *derivative* and *check_EXOR_variable_set*. The function is constructed to find strong EXOR decomposability as efficiently as possible.

Algorithm 20.5

```
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 (44) the initial value of function $u(A,C)$ is calculated:

$$u(A,C) \ := \ \max_B{}^k[f(X)_{ON} \ + \ f(X)_{OFF}] \tag{44}$$

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 (45) and (46) are applied:

$$g(A,C)_{ON} := g(A,C)_{ON} + u(A,C) \cdot f(B = xb0)_{ON} \tag{45}$$

$$g(A,C)_{OFF} := g(A,C)_{OFF} + u(A,C) \cdot f(B = xb0)_{OFF} \tag{46}$$

Now, in order to exclude contradictions, the function $u(A,C)$ is shrinked by sharping the above selected cofactor care functions.

$$u(A,C) := u(A,C) \cdot \overline{g(B = xb0)_{ON}} \cdot \overline{g(B = xb0)_{OFF}} \tag{47}$$

This whole process of using equations (45) to (47) is iterated, and in each step (47) 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.

$$f(X)_{ON} := g(A,C)_{ON} \cdot f(X)_{OFF} + g(A,C)_{OFF} \cdot f(X)_{ON} \tag{48}$$

$$f(X)_{OFF} := g(A,C)_{ON} \cdot f(X)_{ON} + g(A,C)_{OFF} \cdot f(X)_{OFF} \tag{49}$$

The subfunction $g(A,C)$ is finally found, when in (47) $u(A,C) = 0$ results. Otherwise the iterative process is continued again with $xb0$ selected according to equations (45) and (46).

The EXOR operation of function $g(A,C)$ and $f(X)$ creates incompletely specified function $h(B,C)$. This is done in formulas (48) and (49). Now, we calculate the (more specified) function $h(B,C)$ on the basis of the incompletely specified function obtained in equations (48) and (49).

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 (50) 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 (53)), results in 0.

$$u(B,C) := \max_{A}{}^{k}[f(X)_{ON} + f(X)_{OFF}] \tag{50}$$

$$h(B,C)_{ON} := h(B,C)_{ON} + u(B,C) \cdot f(A = xa0)_{ON} \tag{51}$$

$$h(B,C)_{OFF} := h(B,C)_{OFF} + u(B,C) \cdot f(A = xa0)_{OFF} \tag{52}$$

$$u(B,C) := u(B,C) \cdot \overline{h(A = xa0)_{ON}} \cdot \overline{h(A = xa0)_{OFF}} \tag{53}$$

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 (44) to (53), 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 20.6

```
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;
```

21 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.63. Figure 63a presents the *Weak AND Decomposition*, and Figure 63b presents the *Weak OR Decomposition.*

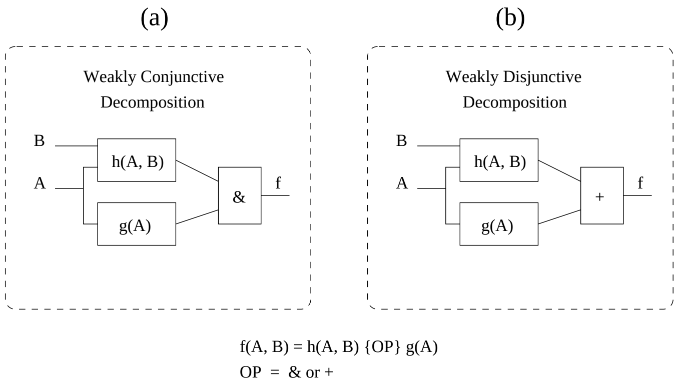


Figure 63: *Weak AND and OR Decompositions.*

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 21.1 or Theorem 21.2 are satisfied, respectively.

Theorem 21.1 *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:*

$$f(X)_{ON} \cdot \overline{\max_B{}^k \, f(X)_{OFF}} \neq 0 \tag{54}$$

Example 21.1

A completely specified function $f(A, B)$ is presented in Fig. 64a. 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. 64b. The condition in Theorem 21.1 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. 64c, 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.

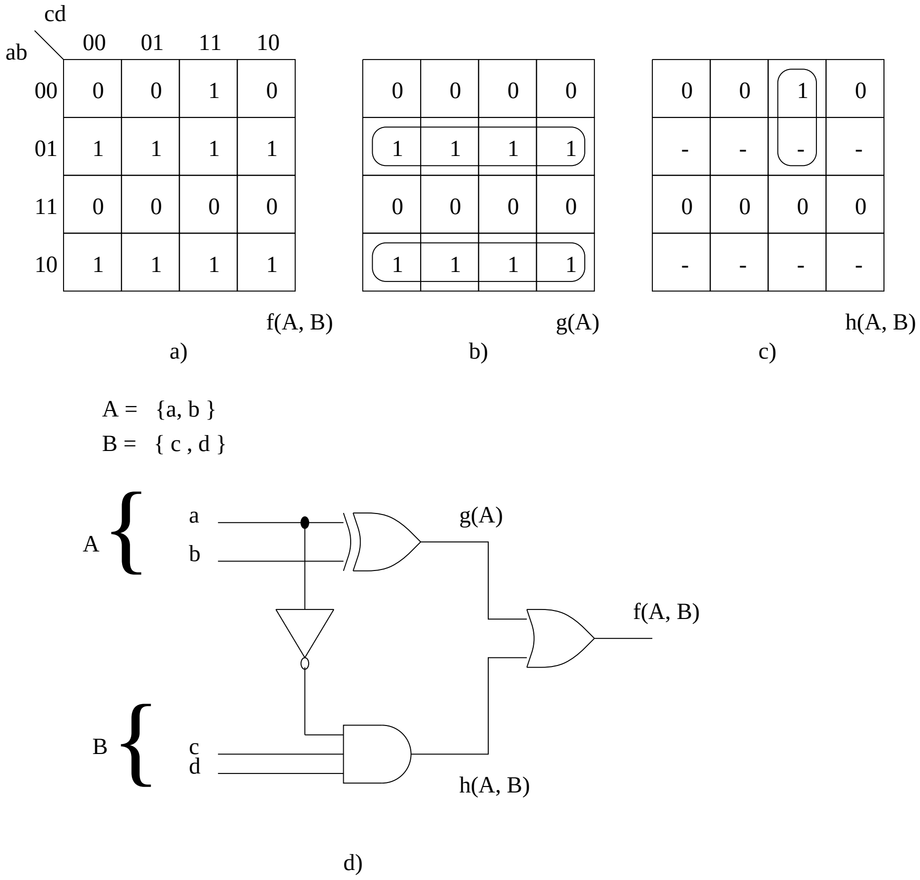


Figure 64: *Steps to OR decomposition to Example 21.1*

The final OR decomposition of function from Fig. 64a is shown in Fig. 64d.

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 21.2

Given is function $f(A, B)$ with $A = \{a, b\}$ and $B = \{c, d\}$ from Fig. 65. The enumeration of minterms for partitions is shown in Fig. 65.

Using partitions, the above presented steps from Example 21.1 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.

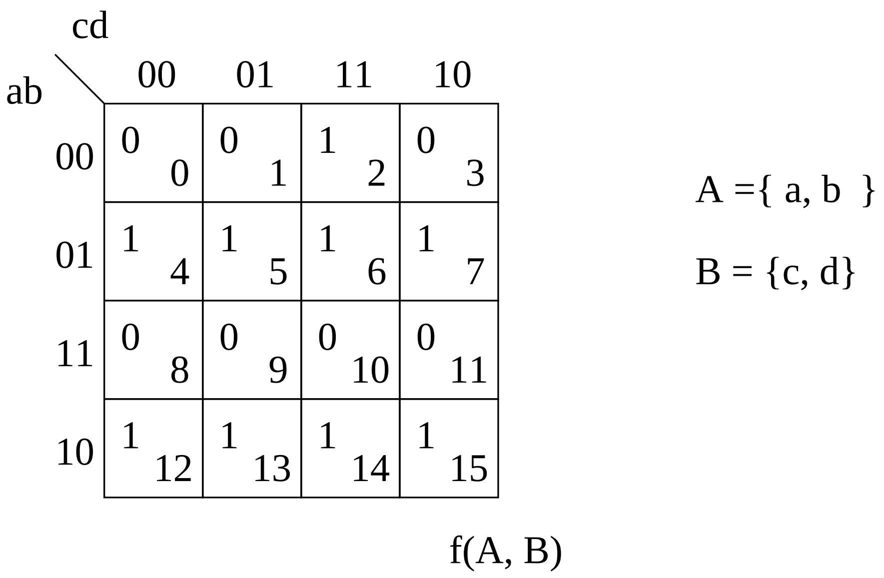


Figure 65: *Function to Example 21.2 with enumeration of minterms*

$$g(A) = \overline{a}\ b + a\ \overline{b}$$

$h(A, B) = \{\{0, 1, 3, 8, 9, 10, 11\}; \{2, 4, 5, 6, 7, 12, 13, 14, 15\} - \{4, 5, 6, 7\} - \{12, 13, 14, 15\}\} = \{\{0, 1, 3, 8, 9, 10, 11\};\ \{2\}\}$
Above, $\{2\}$ is the ON-part of $h(A, B)$.
A weak AND decomposition is a dual to the weak OR decomposition.

Theorem 21.2 *An incompletely specified function with functions* $f(X)_{ON}$ *and* $f(X)_{OFF}$ *is* weakly AND decomposable *with set* B*, if and only if:*

$$f(X)_{OFF} \cdot \overline{\max_B{}^k f(X)_{ON}} \neq 0 \tag{55}$$

Example 21.3

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. 66. 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. 66. 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. 66a is shown in Fig. 66d.

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 21.3 *Every Boolean function that is not weak OR decomposable and not weak AND groupable has the property of EXOR decomposability.*

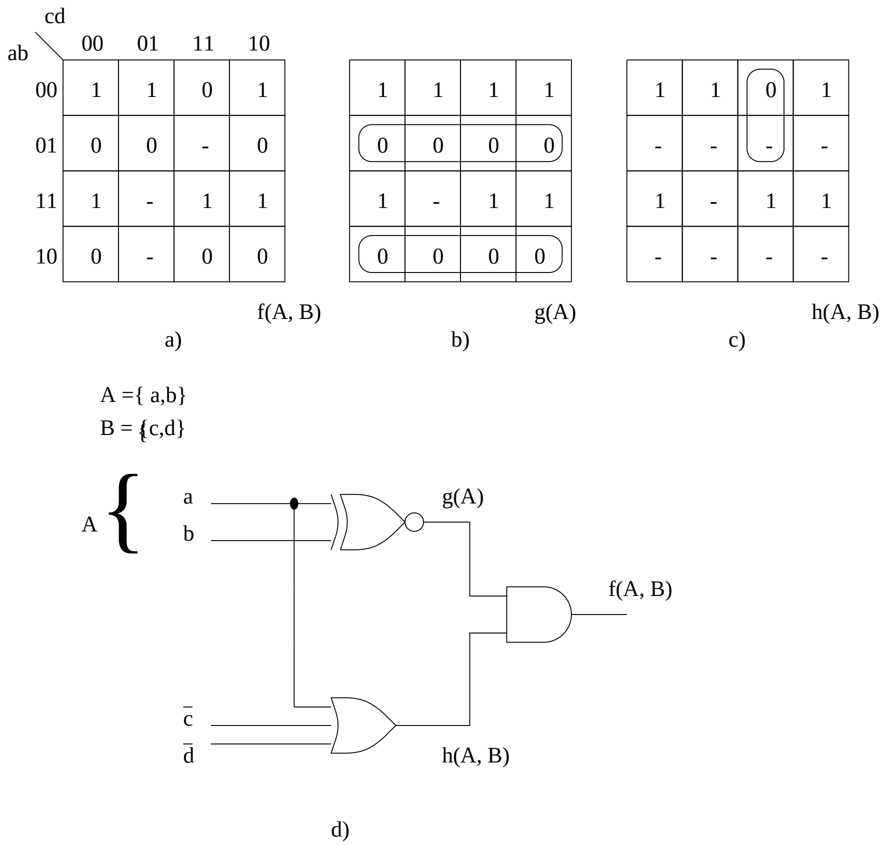


Figure 66: *Steps of AND decomposition: (a) function $f(A,B)$, $A = \{a,b\}$, $B = \{c,d\}$, (b) calculation of function $g(A)$, (c) calculation of function $h(A,B)$, (d) final circuit for AND decomposition*

If the variable set B cannot be further increased in Equation (54) or (55), without these inequalities changing to equalities, then weak decomposability exists. For each weak decomposability there are several weak partitionings. In the formulas (56) to (63) 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 (54) is satisfied, the *Weak AND Decomposition*, see Fig. 63b, can be made into bundles $BF(A,\{g\})$ with $g(A)_{ON} = \mathrm{M1g}$ and $g(A)_{OFF} = \mathrm{M0g}$ as well as $BF(A,B,\{h\})$ with $h(A,B)_{ON} = \mathrm{M1h}$ and $h(A,B)_{OFF} = \mathrm{M0h}$ after the formulas (56) to (59).

$$g(A)_{ON} = \max_B{}^k f(A,B)_{ON} \cdot \overline{\max_B{}^k f(A,B)_{OFF}} \tag{56}$$

$$g(A)_{OFF} = \max_B{}^k f(A,B)_{OFF} \tag{57}$$

$$h(A,B)_{ON} = f(A,B)_{ON} \cdot \overline{g(A)_{ON}} \tag{58}$$

$$h(A,B)_{OFF} = f(A,B)_{OFF} \tag{59}$$

If the inequality (55) is satisfied, then a *Weak AND Decomposition*, Fig. 63b, is obtained from formulas 60 to 63.

$$g(A)_{ON} = \max_B{}^k f(A,B)_{ON} \tag{60}$$

$$g(A)_{OFF} = \max_B{}^k f(A,B)_{OFF} \cdot \overline{\max_B{}^k f(A,B)_{ON}} \tag{61}$$

$$h(A,B)_{ON} = f(A,B)_{ON} \tag{62}$$

$$h(A,B)_{OFF} = f(A,B)_{OFF} \cdot \overline{g(A)_{OFF}} \tag{63}$$

The subroutine *nonlinear_weak* analyzes BDDs of a bundle for weak OR and AND decomposability, and forms (if possible) the weak subfunctions using (56) to (59), or (60) to (63), respectively.

Application of these formulas to calculate OR and AND decompositions is shown in the next examples.

Example 21.4

Figure 67 illustrates graphically using of the above formulas in a weak OR decomposition.

Example 21.5

Figure 68 illustrates graphically using of the above formulas in a weak AND decomposition.

Example 21.6

Figure 69 illustrates graphically the weak EXOR decomposition. Formulas (54) and (55) 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. 69a, and EXOR-ed from $F(A,B)$ as $g(A) = a\ b$, thus creating $h(A,B)$ from Fig. 69b. The final circuit is shown in Fig. 69c.

Algorithm 21.1

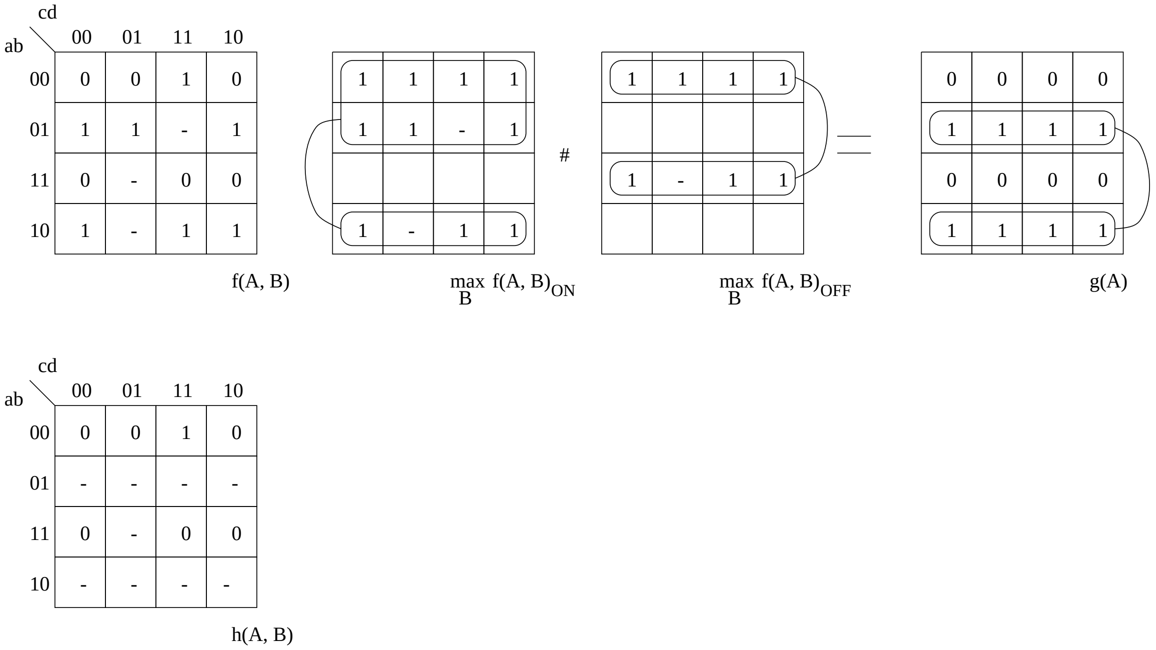


Figure 67: *Graphical illustration of using the formulas in weak OR decompositions to Example 21.5*

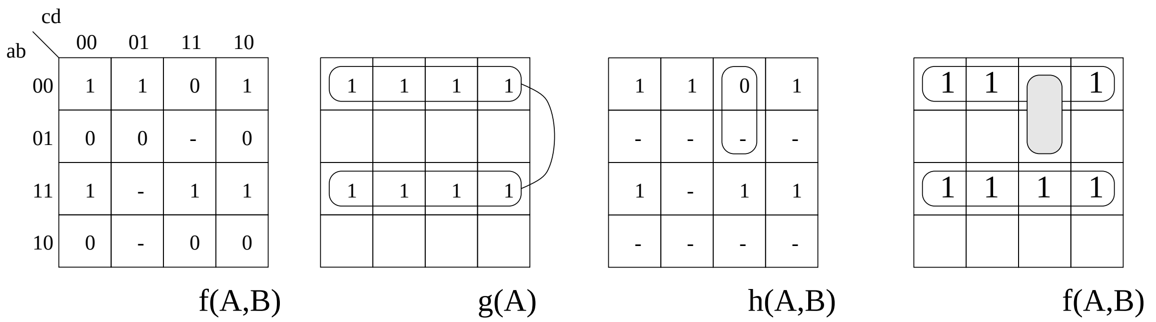


Figure 68: *Graphical illustration of using the formulas in weak AND decompositions to Example 21.6*

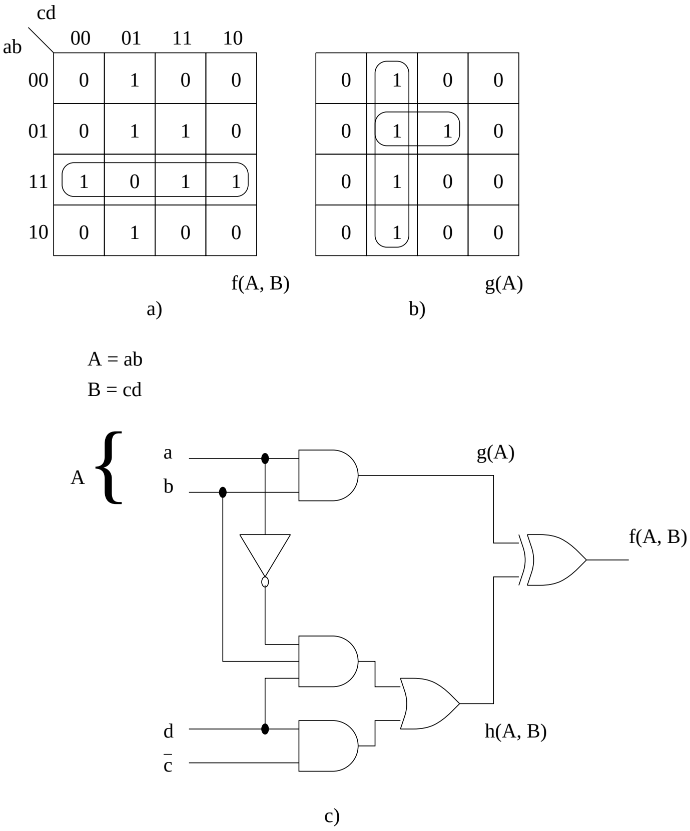


Figure 69: *Graphical illustration of using the formulas in weak EXOR decomposition*

```
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 21.2

```
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 21.4 *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 21.3

```
 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;
```

22 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.

22.1 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.

22.2 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 [130] 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 22.1

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.

22.3 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 [165, 166]. Although these decompositions occur quite often, there exist still functions, for which none of these weak decompositions exists. (The weak EXOR decomposition from [107] 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 [178, 165, 166]. 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.

22.4 The entire comprehensive search strategy

The comprehensive Decomposition strategy, Algorithm 22.2, 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 22.2 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 22.2

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 .

23 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.

23.1 General Decompositions of Multiple-Valued-Output Functions with Standard Don't Cares.

The method for Multiple-Valued-Output functions with standard don't cares [12] 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.

23.2 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 [12]. 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

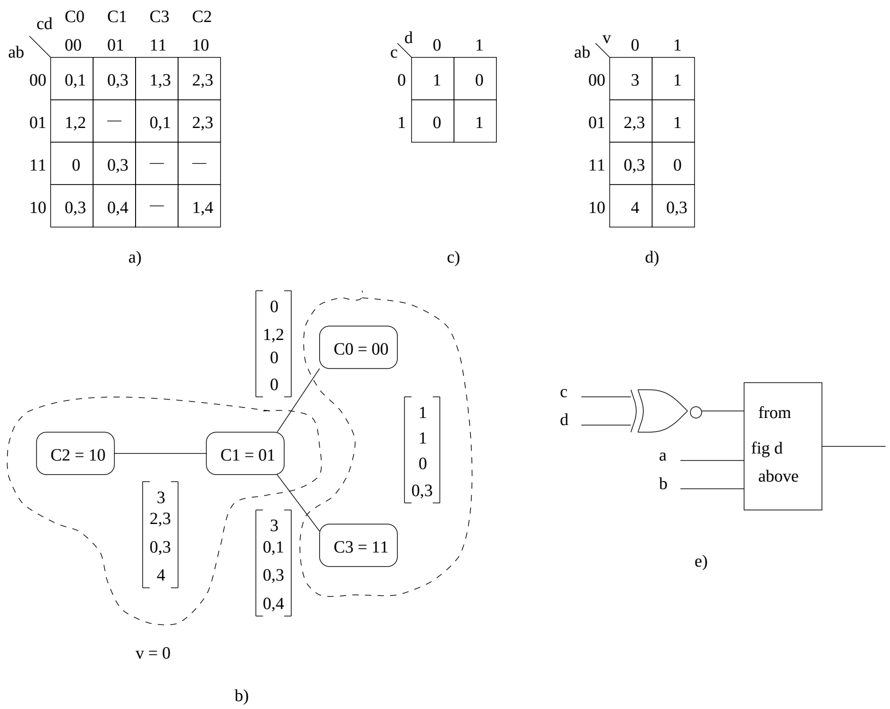


Figure 70: *A Decomposition of a function with 4 binary variables, a 5-valued output variable and generalized don't cares from Table 1*

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 [190]. 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 23.1

Given is a function with 4 binary variables and a 5-valued output variable from the map in Figure 70a (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 70b. 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 $\{0,1\} \cap \{2,3\} = \phi$. 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 70c,d, respectively, and the decomposed circuit in Figure 70e. See [145] for applications to Attributes Reduction problem.

24 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. 71 presents the above decompositions, Fig. 72 presents Curtis Decompositions, and Fig. 73 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 [89, 80]). 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, [145]. The blocks of partitions were encoded as BDDs for additional improvement of efficiency; leading to the concept of CBDDs.

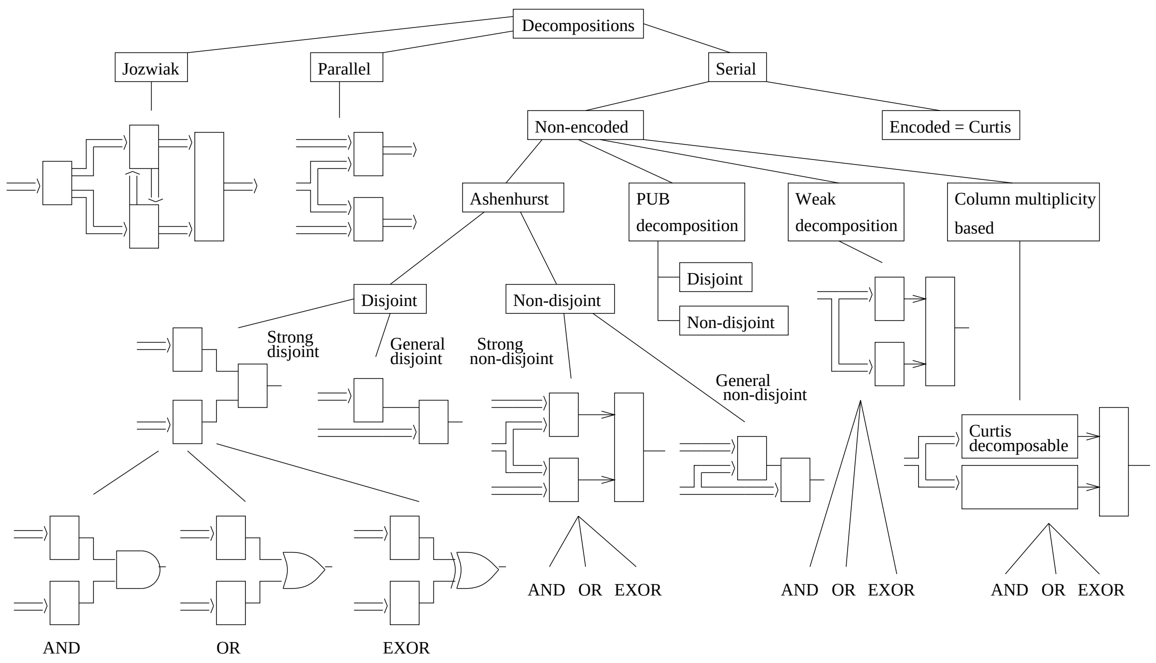


Figure 71: *The types of decomposition*

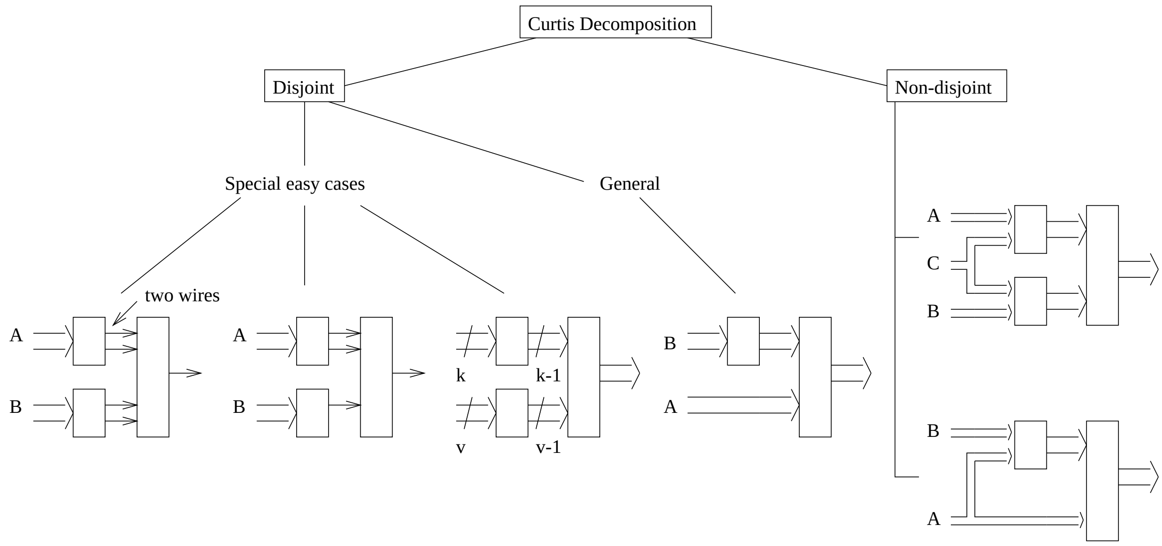


Figure 72: *Curtis decompositions*

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 [138], lists of disjoint ON and OFF cubes [107], and variable-partitions on ON and OFF minterms or cubes [161]. 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 [107, 138, 161, 79, 1, 2, 8, 9, 45, 53, 110]. 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 [155, 156]), limited best-bound search (Haomin Wu, Craig Files [158]), A* algorithm of Artificial Intelligence (Haomin Wu, Perkowski); searching subsets of a subset of variables (Shen/Kellar [49, 50, 55], Varma/Trachtenberg [5]), using predetermined order based on a heuristic of evaluating pairs of variables (cube based approach is presented by Wei Wan and Perkowski [130]), search by "add and subtract" variables from and to bound and free sets - Steinbach [107], search by selecting a single variable based on complex block separation criteria of partition calculus heuristics (Luba [69, 92, 140, 161]), other BDD variable selection and ordering methods [4, 3, 81, 82], 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 [10], Mike Noviskey [11], and Grygiel and Perkowski [222].

Column Multiplicity Minimization.

In the past, the Column Multiplicity Minimization Problem was reduced either to the set covering with maximum cliques problem, [92, 145], 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 [99, 161, 130] 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.

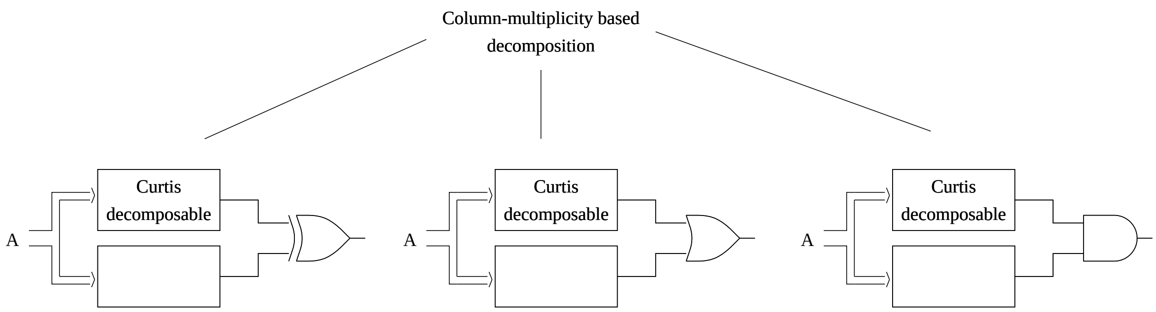


Figure 73: *Column Multiplicity Based Curtis Decompositions*

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 [224]. 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 74 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:

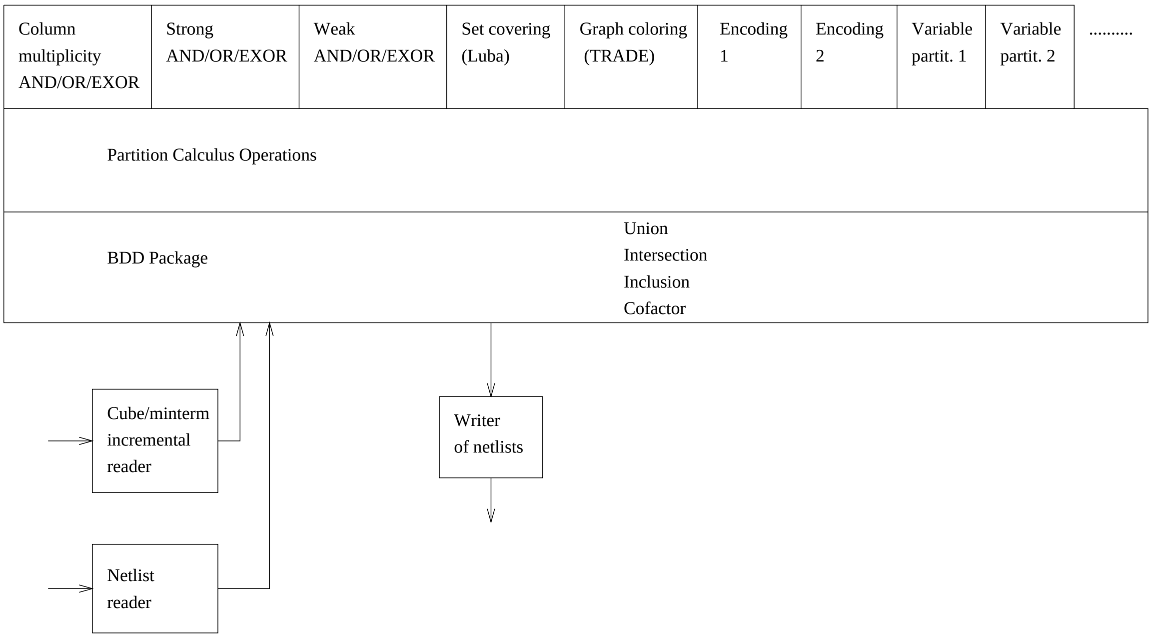


Figure 74: *A layered construction of GUD system*

- 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).

25 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 22. 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 [173] for the first results). We developed also a new approach to decompose Fuzzy Logic circuits [68], 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.

References

[1] J. Butler, and K.J. Breeding, "Some Characteristics of Universal Cell Nets," *IEEE Trans. on Comput.*, Vol. C-22, pp. 897-903, October 1973.

[2] 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.

[3] 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.

[4] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and 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.

[5] 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.

[6] 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.

[7] 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.

[8] J.T. Butler, "Tandem Networks of Universal Cells," *IEEE Trans. on Comput.*, Vol. 27, No. 9, pp. 785-799, 1978.

[9] 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.

[10] 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.

[11] M.J. Noviskey, "Correlation Partition Selection Algorithm," *Technical Report,* WL/AART-2, W/P AFB, OH 45433-7408, August 1992.

[12] M. Perkowski, "Survey of Literature on Functional Decomposition," *Dept. Electr. Engn.*, report, 1995.

[13] N. Christofides, "Graph Theory," *Academic Press*, New York, 1975.

[14] M. Gondran, M. Minoux, "Graphs and Algorithms," *John Wiley & Sons,* New York, 1984.

[15] R.C. Read, "Graph Theory and Computing," *Academic Press*, New York, 1972.

[16] F. Harary, and J.S. Maylee, "Graph and Applications," *Proc. First Colorado Symp. on Graph Theory,* John Wiley & Sons, New York, 1985.

[17] S. Thelliez, "Introduction to the Study of Ternary Switching Structures," *Gordon and Breach Science,* N.Y. 1973.

[18] K.Y. Fang, and A.S. Wojcik, "Modular Decomposition of Combinational Multiple-Valued Circuits," *IEEE Trans. on Comput.*, Vol. 37, No. 10, pp. 1293-1301, 1988.

[19] 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.

[20] 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.

[21] H.A. Simon, and A. Ando, "Aggregation of Variables in Dynamic Systems," *Econometrica,* 29, pp. 111-138, 1961.

[22] C. Aleksander, "Notes on the Synthesis of the Form," Cambridge, MA, *Harvard Univ. Press,* 1967.

[23] H.A. Simon, "The Sciences of the Artificial," Cambridge, MA, *MIT Press,* 1981.

[24] D. Paulson, and Y. Wand, "An Automated Approach to Information Systems Decomposition," *IEEE Trans. Soft. Engng.,* 18, pp. 174-189, 1992.

[25] W.W. Armstrong, and J. Gecset, "Adaptation Algorithms for Binary Tree Networks," *IEEE Trans. on Systems, Man and Cybern.*, Vol. 9, No. 5, pp. 276-285, 1979.

[26] R.L. Ashenhurst, "The Decomposition of Switching Functions," *Bell Laboratories Report,* No. BL-1(11), pp. 541-602, 1952.

[27] R.L. Ashenhurst, "Non-disjoint Decomposition," *Bell Laboratories Report,* No. 4, pp. 1V-1 - 1V-12, 1953.

[28] R.L. Ashenhurst, "The decomposition of switching functions," *pp. 571-603 of Curtis'62.*, 1952.

[29] R.L. Ashenhurst, "The Decomposition of Switching Functions", *Proceedings of International Symposium of Theory of Switching,* 1957.

[30] H.A. Curtis, "Non-Disjunctive Decompositions," *Theory of Switching Bell Labs Report,* No. 19, Section II, 1, 49, 1958.

[31] S.B. Akers, "On a Theory of Boolean Functions," *Journal of the Society for Industrial and Applied Mathematics,* 7, 4, 1959.

[32] H.A. Curtis, "A Functional Canonical Form," *J. ACM,* 6, pp. 245-258, 1959.

[33] R.L. Ashenhurst, "The Decomposition of Switching Fuctions," *Proceedings of an International Symposium on the Theory of Switching,* April 2-5, 1957, Ann. Computation Lab., Harvard Univ., Vol. 29, pp. 74-116, 1959.

[34] J.P. Roth, "Minimization over Boolean Trees," *IBM Journal.* 4, 5, pp. 543-555, 1960.

[35] H.A. Curtis, "A Generalized Tree Circuit", *J. ACM,* pp. 484-496, 1961.

[36] R.M. Karp, F.E. McFarlin, J.P. Roth, and 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.

[37] H.A. Curtis, "A New Approach to the Design of Switching Circuits", em Princeton, N.J., Van Nostrand, 1962.

[38] J.P. Roth, and R.M. Karp, "Minimization Over Boolean Graphs," *IBM J. Res. Develop.* pp. 227-238, April 1962.

[39] 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.

[40] K.K. Maitra, "Cascaded Switching Networks of Two-Input Flexible Cells," *IRE Trans. Electr. Comput.*, Vol. EC-11, pp. 136-143, Apr. 1962.

[41] H.A. Curtis, "Generalized Tree Circuit - The Basic Building Block of an Extended Decomposition Theory," *J. Assn. Comput. Mach,* 1963, 10, pp. 562-581.

[42] R.M. Karp, "Functional Decomposition and Switching Circuit Design," *J. Soc. Industr. Appl. Math.*, Vol. 11, No. 2, pp. 291-335, June 1963.

[43] A.D. Zakrevskij, "An Algorithm for Decomposition of Boolean Functions," *Trudy Sib. Fiz-tekhn In-ta*, Collection 44, pp. 5-16, 1964, (in Russian)

[44] J. Hartmanis, and R.E. Stearns, "Algebraic Structure Theory of Sequential Machines," Englewood Cliffs, N.Y., *Prentice Hall, Inc.,* 1966.

[45] D.L. Dietmeyer, and P.R. Schneider, "Identification of Symmetry, Redundancy and Equivalence of Boolean Functions," *IEEE Trans. Electron. Comput.*, Vol. 16, pp. 804-817, December 1967.

[46] 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.

[47] A.R. Meo, "Modular Tree Structures," *IEEE Trans. on Comput.*, Vol.17, 432-442, pp. 559-566, June 1968.

[48] P.R. Schneider, and D.L. Dietmeyer, "An Algorithm for Synthesis of Multiple-Output Combinational Logic," *IEE Trans. on Comput.*, pp. 117-128, Febr. 1968.

[49] V.Y. Shen, "On Simple Disjunctive Decompositions of Switching Functions," Ph.D. thesis, *Dept. Electr. Engng., Princeton University,* N.J., January 1969.

[50] V.Yun-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.

[51] K.M. Walliuzzaman, and Z.G. Vranesic, "On Decomposition of Multiple-Valued Switching Functions," *Computer Journal,* Vol. 13, pp. 359-362, 1970.

[52] S.S. Yau, and C.K. Tang, "Universal Logic Modules and Their Applications," *IEEE Trans. on Comput.*, Vol. C-19, No. 2, Febr. 1970.

[53] D.L. Dietmeyer, "Logic Design of Digital Systems," *Boston: Allyn and Bacon,* 1971.

[54] S.R. Das, and C.L. Sheng, "On Detecting Total or Partial Symmetry of Switching Functions," *Trans. on Comput.* pp. 352-355, March 1971.

[55] 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.

[56] S.S. Yau, and Y.S. Tang, "On Identification of Redundancy and Symmetry of Switching Functions," *IEEE Trans. on Comput.*, Vol. C-20, pp. 1609-1613, Dec., 1971.

[57] A. Thayse, "A Fast Algorithm for the Proper Decomposition of Boolean Functions," *Philips Res. Reports,* No. 27, 1972.

[58] 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.

[59] A. Maruoka, and N. Honda, "Logical Networks of Flexible Cells," *IEEE Trans. Comput.*, Vol. C-22, pp. 347-358, Apr. 1973.

[60] 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.

[61] A. Maruoka, and N. Honda, "The Range of Flexibility of Tree Networks," *IEEE Trans. on Comput.*, Vol C-24, pp. 9-28, Jan. 1975.

[62] H.A. Curtis, "Simplified Decomposition of Boolean Functions," *IEEE Trans. on Comput.*, Vol. 25, pp. 1033-1044, October 1976.

[63] A.Kandel, "Decomposition of Fuzzy Functions," *IEEE Trans. on Comput.*, Vol. 25, No. 11, pp. 1124-1130, 1976.

[64] Z. Kohavi, "Switching and Finite Automata Theory," *McGraw-Hill Book Company,* 1978.

[65] 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.

[66] A.W. Biermann, J.R.C. Fairfield, and T.R. Beres, "Signature Table Systems and Learning," *IEEE Trans. on Syst. Man and Cybern.*, Vol. 12, No. 5, pp. 635-648, 1982.

[67] R.K. Brayton, and C.T. Mc Mullen, "The Decomposition and Factorization of Boolean Expressions, International Symposium on Circuits and Systems," Rome, Italy, 1982, pp. 49 - 54.

[68] J.M. Francioni, and A. Kandel, "Decomposable Fuzzy-valued Switching Functions," *Fuzzy Sets and Systems,* Vol. 9, No. 1, pp. 41-68, 1983.

[69] T. Luba, "Synthesis of Combinational Circuits Using Boolean Decomposition Method," *Publishers of Institute of Telecommunication,* Warsaw Technical University, No. 108, 1983, (in Polish).

[70] R.K. Brayton, G.D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis," *Kluwer Academic Publishers,* Boston, 1984.

[71] W. Grzymala-Busse, and Z. Pawlak, "On Some Subset of the Partition Set," *Fundamenta Informaticae,* Vol. VII, No. 4, 1984. PWN, Warszawa, Poland.

[72] T. Sasao, "Input Variable Assignment and Output Phase Optimization of PLA's," *IEEE Trans Comput.* Vol. C-33. pp. 879-894, 1984.

[73] T. Luba, and L. Wronski, "Decompositional Method of Realizing Systems of Switching Functions," *Review of Telecommunication,* No. 4, 1985, (in Polish).

[74] T. Luba, "New Approach to Boolean Function Decomposition for ROMs and PLAs," *Publishers of Institute of Telecommunication,* Warsaw Technical University, No. 123, 1985.

[75] T. Luba, "Synthesis of Combinational Circuits Using the Method of Boolean Decomposition," *Rozprawy Elektrotechniczne,* No. 2, 1985.

[76] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *Trans. on Comput.*, Vol. C-35, No. 8, pp. 667-691, 1986.

[77] T. Luba, "A Uniform Method of Boolean Function Decomposition," *Rozprawy Elektrotechniczne,* No. 4, pp. 1041-1054, 1986.

[78] T. Luba, and L.Wronski, "Sequential Determination of Values of Boolean Expressions Using the Threshold Table Method," *Review of Telecommunication,* No.1, 1986, (in Polish).

[79] P.N. Bibilo, and 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).

[80] R. Rudell, and A. Sangiovanni-Vincentelli, "Multiple-Valued Minimization for PLA Optimization," Proceedings of 17th Intern. Symposium on Multiple-Valued Logic, Boston MA, May 26-28, 1987, pp. 198-208.

[81] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *Intern. Workshop on Logic Synthesis,* pp. 3a 1-12, 1993.

[82] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *Proc. Int. Conf. on Computer Aided Design*, pp. 42-47, 1993.

[83] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang, "MIS: Multiple-Level Logic Minimization System," *IEEE Trans. on CAD,* vol. CAD-6, pp. 1062-1081, 1987.

[84] Loc Bao Nguyen, M.A. Perkowski, and N.B. Goldstein, "PALMINI - Fast Boolean Minimizer for Personal Computers," *Proc. 24th ACM/IEEE Design Automation Conf.*, 1987, pp. 615-621.

[85] LogicBenchmarks, "Logic Benchmarks of International Workshop on Logic Synthesis," *MCNC*, North Carolina, May 1987.

[86] T. Luba, and B. Zbierzchowski, "Topological Models of Boolean Functions and Their Application in the Synthesis of Complex Combinational Circuits," *Rozprawy Elektrotechniczne,* No.2, 1987.

[87] M. Perkowski, and H. Uong, "Generalized Decomposition of Incompletely Specified Multioutput, Multi-Valued Boolean Functions," *Unpublished manuscript, Department of Electrical Engineering,* PSU 1987.

[88] 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.

[89] T.Sasao, "Functional Decomposition of PLAs", *Proc. of the Intern. Workshop on Logic Synthesis,* Research Triangle Park, North Carolina, May 12-15, 1987.

[90] S. Devadas, A.R. Wang, A.R. Newton, and A. Sangiovanni-Vincentelli, "Boolean Decomposition in Multi-Level Logic Optimization," *Proc. IEEE International Conf. on Computer-Aided Design,* pp. 290-293, 1988.

[91] F. Dresig, "Synthese Kombinatorischer Schaltnetzwerke mit Arbeitsplatzcomputern," *Dissertation*, T.U. Karl-Marx-Stadt (Chemnitz), 1988, (in German).

[92] 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).

[93] M. Perkowski, and J. Brown, "A Unified Approach to Designs with Multiplexers and to the Decomposition of Boolean Functions," *Proc.ASEE Annual Conference,* pp.1610-1619, 1988.

[94] J. Poswig, "Disjoint decomposition," *Proc. of 3rd. International Workshop on Spectral Techniques,* Dept. Comp. Science, University of Dortmund, Dortmund 1988.

[95] 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.

[96] T.D. Ross, "Pattern Representation and Recognition," *Research Prospectus,* Air Force Institute of Technology. 1988.

[97] T.D. Ross, "Elementary Theorems in Pattern Theory," *Ph.D. Thesis,* Air Force Institute of Technology. 1988.

[98] 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.

[99] M.A. Perkowski, J. Liu, and 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.

[100] 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.

[101] 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.

[102] S. Yang, and M. Ciesielski, "PLA Decomposition with Generalized Decoders," *Proc. of the IC-CAD,* pp. 312-315, 1989.

[103] F.M. Brown, "Boolean Reasoning. The Logic of Boolean Equations," *Kluwer Academic Publishers,* 1990.

[104] G. Saucier, P. Sicard, and L. Bouchet, "Multi-Level Synthesis on PALs," *Proc. European Design Automation Conf.*, pp. 542-546, 1990.

[105] R. J. Francis, J. Rose, and 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.

[106] 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.

[107] D. Bochmann, and B. Steinbach, "Logikentwurf mit XBOOLE," *Verlag Technik,* Berlin, 1991.

[108] D. Filo, J.C. Yang, F. Mailhot, and G.D. Micheli, "Technology Mapping for a Two-Output RAM-based Field-Programmable Gate Array," *Proc. European Conference on Design Automation,* pp. 534-538, 1991.

[109] T. Luba, J. Kalinowski,and K. Jasinski, "PLATO: A Cad Tool for Logic Synthesis Based on Decomposition," *Proc. of European Conference on Design Automation,* pp. 65-69, 1991.

[110] B.G. Kim, and D.L. Dietmeyer, "Multilevel Logic Synthesis of Symmetric Switching Functions," *IEEE Trans. on CAD,* Vol. 10, No.4, April 1991.

[111] 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.

[112] R. Murgai, N. Shenoy, R.K. Brayton, and A. Shangiovanni-Vincentelli, "Performance Directed Synthesis for Table Look Up Programmable Gate Arrays," *Proc. ICCAD 1991*, Santa Clara, CA, Nov. 1991, pp. 572-575.

[113] Z. Pawlak, "Rough Sets," *Theoretical Aspects of Reasoning about Data,* Kluwer Academic Publishers, Dordrecht, 1991.

[114] J. Poswig, "Disjoint Decomposition of Boolean Functions," *IEE Proceedings,* Vol. 138, No. 1, pp. 48-56, January 1991.

[115] T. D. Ross, M.J. Noviskey, T.N. Taylor, and 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.

[116] 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.

[117] A. Skowron, and C. Rauszer, "The Discernibility Matrices and Functions in Information Systems," *Res. Rep. 1/91,* Inst. Comp. Sci., Warsaw, 1991.

[118] M. Ciesielski, and S. Yang, "PLADE: A Two Stage PLA Decomposition," *IEEE Trans. on CAD.* Vol. 11, pp. 943-954, 1992.

[119] 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.

[120] L. Jozwiak, and 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.

[121] 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.

[122] Y.T. Lai, M. Pedram, and S. Vrudhula, "BDD-based Logic Decomposition: Theory." *Technical Report,* Dept. of EE. Systems, University of Southern California, 1992.

[123] 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.

[124] T. Luba, and J. Rybnik, "Algorithm for Elimination of Attributes and Arguments Based on Unate Complement Concept," *Bull. Pol. Ac. , Tech.,* Vol. 40, No.3, pp. 313-322, 1992.

[125] T. Luba, M.A. Markowski and, B. Zbierzchowski, "Logic Decomposition for Programmable Gate Arrays," *Proc. of Euro-ASIC'92,* Paris, 1992.

[126] T. Luba, and 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.

[127] T. Luba, K. Gorski, and L.B. Wronski, "ROM-Based Finite State Machines with PLA Address Modifiers," *Proc. European Design Automation Conf.*, pp. 272-277, 1992.

[128] T.D. Ross, M.L. Axtell, M.J. Noviskey, and M. Breen, "A Demonstration of a Robust Occam-Based Learner," In *Proc. IEEE Int. Symp. on Information Theory,* 1992.

[129] 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.

[130] W. Wan, and M.A. Perkowski, "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," *Proc. European Design Automation Conf.*, pp.230-235, 1992.

[131] W. Wan, "M.S. Thesis," *Department of Electrical Engineering*, Portland State University, 1992.

[132] 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.

[133] T. Luba, H. Selvaraj, and A. Krasniewski, "A New Approach to FPGA-based Logic Synthesis," *Workshop on Design Methodologies for Microelectronics and Signal Processing,* Gliwice - Cracow, 1993.

[134] M.L. Axtell, T.D. Ross, and 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.

[135] M.A. Breen, T.D. Ross, M.J. Noviskey, and M.L. Axtell, "Pattern Theoretic Image Restoration," *Proc. SPIE '93 Nonlinear Image Processing IV* 1902-12.

[136] J.F. Frenzel, "Application of Genetic Algorithms to Pattern Theory," *Final Report,* Summer Faculty Research Program, July 1993.

[137] Shusheng 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.

[138] Y.T. Lai, M. Pedram, and S. Sastry, "BDD-based Decomposition of Logic Functions With Application to FPGA Synthesis," *Proc. of 30th DAC,* pp. 642-647, 1993.

[139] Y.T. Lai, K.R. Pan, M. Pedram, and S. Vrudhula, "FGMap: A Technology Mapping Algorithm for Look-up Table Type FPGA Synthesis," *Proc. of 30-th DAC,* pp. 642-647, 1993.

[140] T. Luba, H. Selvaraj, and 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.

[141] T. Luba, R. Lasocki, and 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,

[142] T. Luba, and 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.

[143] T. Luba, H. Selvaraj, and A. Krasniewski, "A New Approach to FPGA-based Logic Synthesis", *Workshop on Design Methodologies for Microelectronics and Signal Processing,* Gliwice-Cracow, 1993.

[144] T. Luba, M. Mochocki, and J. Rybnik, "Decomposition of Information Systems Using Decision Tables," *Bulletin of the Polish Academy of Sciences, Technical Sciences,* Vol. 41, No.3, 1993.

[145] T. Luba, "Decomposition of Multiple-Valued Functions," Invited paper, *Proc. 25th ISMVL*, pp. 256-261, 1995.

[146] M.J. Noviskey, "Row Identification for Function Decommposition in Pattern Theory," *Technical Report,* WL/AART-2, W/P AFB, OH 45433-7408, May 1993.

[147] D. Patel, and T. Luba, "Dependence Sets and Functional Decomposition of Boolean Functions," *Intern. J. of Electr.*, Vol. 75, No. 2, pp. 177-198, August 1993.

[148] T.D. Ross, M.L. Axtell, and M.J. Noviskey, "Logic Minimization as a Robust Pattern Finder," *Intern. Workshop on Logic Synthesis,* Lake Tahoe, CA, May 23-26, 1993.

[149] T.D. Ross, M.L. Axtell, M.J. Noviskey, and D.A. Gadd, "Pattern Theory Paradigm for System Design," In *Proc. 36th Midwest Symposium on Circuits and Systems,"* 1993.

[150] T.D. Ross, M.J. Noviskey, and M.L. Axtell, D.A. Gadd, "Flash Software Description," *Technical report, Wright Laboratory,* USAF, WL/AART, WPAFB, OH 45433-6543, December 1993.

[151] T. Sasao (ed.), "Logic Synthesis and Optimization," *Kluwer Academic Publishers*, 1993.

[152] T. Sasao, "FPGA Design by Generalized Functional Decomposition," in "Logic Synthesis and Optimization, " T. Sasao. (Ed), Kluwer Academic Publishers, pp. 233-258, 1993.

[153] 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.

[154] H. Selvaraj, A. Czerczak, A. Krasniewski, and 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.

[155] H. Wu, and 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.

[156] 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.

[157] M. Breen, T.D. Ross, and M.L. Axtell, "Computing Column Multiplicity in Function Decomposition," *report, WPAFB*, 1994.

[158] 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.

[159] 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.

[160] 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.

[161] T. Luba, and R. Lasocki, "Decomposition of Multiple-valued Boolean Functions," *Applied Mathematics and Computer Science,* Vol.4, No.1, pp. 125-138, 1994.

[162] 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.

[163] M. Marek-Sadowska, "Detecting Symmetric Variables in Boolean Functions using Generalized Reed-Muller Forms," *Proc. ISCAS'94*, pp. 287-290, 1994.

[164] M. J. Noviskey, T.D. Ross, D.A. Gadd, and M. Axtell, "Application of Genetic Algorithms to Function Decomposition in Pattern Theory," *report WL-TR-94-1015,* 1994.

[165] M. Perkowski, and H. Uong, "Generalized Decomposition of Incompletely Specified Multioutput, Multi-Valued Boolean Functions," *Unpublished manuscript, Dept. Electr. Eng.,* PSU 1987.

[166] M. Perkowski, and J. Brown, "A Unified Approach to Designs with Multiplexers and to the Decomposition of Boolean Functions," *Proc. ASEE Ann. Conf.,* pp.1610-1619, 1988.

[167] 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.

[168] 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.

[169] 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.

[170] 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.

[171] 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.

[172] M. A. Perkowski, and M. Chrzanowska-Jeske, "Multiple-Valued-Input TANT Networks," *Proc. ISMVL'94,* pp. 334-341, Boston, MA, May 25-27, 1994.

[173] 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 Reed-Muller Workshop,* August 1995.

[174] E. Pierzchala, M.A. Perkowski, and S. Grygiel, "A Field Programmable Analog Array for Continuous, Fuzzy, and Multi-Valued Logic Applications," *Proc. ISMVL '94,* pp. 148 - 155.

[175] T.D. Ross, J.A. Goldman, M.J. Noviskey, and M.L. Axtell, D.A. Gadd, "Graph Coloring for Column Multiplicity: A Survey," WL WPAFB report, April 21, 1994.

[176] T.D. Ross, M.J. Noviskey, M.L. Axtell, D.A. Gadd, and J.A. Goldmann, "Pattern Theoretic Feature Extraction and Constructive Induction," *Technical report,* Wright Laboratory, USAF, WL/AART, WPAFB, OH 45433-6543, April 22, 1994.

[177] T.D. Ross, J.A. Goldmann, and D.A. Gadd, "On the Decomposition of Continuous Functions," *In Third Intern. Workshop on Post-Binary VLSI Systems,* 1994.

[178] 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 CAD,* Vol. 12, No. 11, pp. 1655 - 1664, Nov. 1993.

[179] H. Selvaraj, "FPGA-Based Logic Synthesis," *Ph.D. Dissertation,* Warsaw University of Technology, 1994.

[180] Z. Kohavi, "Switching Circuits and Finite Automata Theory,"

[181] 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.

[182] A.L. Oliveira, and A. Sangiovanni-Vincentelli, "Learning Complex Boolean Functions: Algorithms and Applications", *Implementation and Simulation - VLSI*, preliminary copy 6/10/93.

[183] A.L. de Oliveira, *"Inductive Learning by Selection of Minimal Complexity Representations,"* Ph.D. Thesis, University of California at Berkeley, Dec. 1994.

[184] A. Oliveira, 1994.

[185] R. Murgai, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Optimum Functional Decomposition Using Encoding," *Proc. 31st DAC*, pp. 408–414, June 1994.

[186] D.F. Barnard, and D.F. Holman, "The use of Roth's Decomposition Algorithm in Multi-Level Design of Circuits".

[187] L. B. Nguyen, M.A. Perkowski, and N.B. Goldstein, "PALMINI - Fast Boolean Minimizer for Personal Computers," *Proc. 24th DAC*, 1987, pp. 615-621.

[188] Ch. Meinel, J. Bern, and A. Slobodova, "Efficient OBDD-Based Boolean Manipulation in CAD Beyond Current Limits," *Proc. 32nd DAC,* San Francisco 1995.

[189] R. Brayton and F. Somenzi, "An Exact minimizer for Boolean Relations," Proc. of ICCAD, pp. 316-320, 1989.

[190] M. J. Ciesielski, S. Yang, and M. A. Perkowski, "Multiple-Valued Minimization Based on Graph Coloring," *Proc. ICCD'89,* pp. 262 - 265, October 1989.

[191] M. Fujita, Y. Kukimoto, and R. Brayton, "BDD Minimization by Truth Table Permutations," *IWLS '95.*

[192] E. B. Lee, and M. Perkowski, "Concurrent Minimization and State Assignment of Finite State Machines," *Proc. Int. Conf. on Syst. Man, and Cyb.,* pp. 248 - 260, Halifax, Nova Scotia, Canada, October 9 - 12, 1984.

[193] Y.S. Abu-Mostafa, *"Complexity in Information Theory",* Springer-Verlag, New York, 150pp, ISBN 0-387-96600-5, 1988.

[194] A.R. Barron, and R.L. Barron, "Statistical Learning Networks: A Unifying View", *Symposium on the Interface: Statistics and Computing Science,* 1988.

[195] A. Blumer, A. Ehrenfeucht, D. Haussler, and M.K. Warmuth, "Occam's Razor", *Information Processing Letters*, Oct. 1987, pp. 377-380.

[196] M.A. Breen, T.D. Ross, M.J. Noviskey, and M.L. Axtell, "Pattern Theoretic Image Restoration," *Proc. SPIE'93 Nonlinear Image Processing,* Intern. Soc. for Optical Engineering, January 1993.

[197] 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.

[198] J.A. Goldman, "Pattern Theoretic Knowledge Discovery", *Proc. 6th IEEE International Conference on Tools with Artificial Intelligence,* IEEE, November 1994.

[199] 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.

[200] J. A. Goldman, and M. L. Axtell, "On Using Logic Synthesis for Knowledge Discovery," submitted.

[201] 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.

[202] 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.

[203] R. Kohavi, and B. Frasca, "Useful Feature Subsets and Rough Set Reducts", *Third International Workshop on Rough Sets and Soft Computing,* 1994.

[204] R. Kohavi, "Bottom-up Induction of Oblivious Read-Once Decision Diagrams," In *European Conference on Machine Learning,* 1994.

[205] J. Koza, *"Genetic Programming,"* MIT Press, 1992.

[206] G.J. Montgomery, and K.C. Drake, "Abductive Networks", *SPIE Applications of Neural Networks Conference,* April 1990,

[207] Ming Li and Paul M. B. Vitányi, "Inductive Reasoning and Kolmogorov Complexity", *Journal of Computer and System Sciences,* Vol. 44, pp. 343-384, 1992.

[208] R. Michalski, J. Carbonell, and T.M. Mitchell, *"Machine Learning: An Artificial Intelligence Approach," Morgan Kaufmann,* vol. 1, 1983, vol. 2. 1986.

[209] G. Pagallo, and D. Hausler, "Boolean Feature Discovery in Empirical Learning", *Machine Learning*, Vol. 5, pp. 71-99, 1990.

[210] 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.

[211] 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.

[212] J. R. Quinlan, *"C4.5: Programs for Machine Learning",* Morgan Kaufmann, 1993, Palo Alto, Ca.

[213] 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.

[214] T.D. Ross, M.L. Axtell, and M.J. Noviskey, "Logic Minimization as a Robust Pattern Finder", *International Workshop on Logic Synthesis,* May 1993.

[215] T.D. Ross, M.J. Noviskey, and M.L. Axtell, D.A. Gadd, "Flash user's guide," *Technical report, Wright Laboratory,* USAF, WL/AART, WPAFB, OH 45433-6543, December 1993.

[216] 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.

[217] T.D. Ross, "Variable Partition Search for Function Decomposition," *Technical report, Wright Laboratory,* USAF, WL/AARA-3, WPAFB, OH 45433-6543, November 1994.

[218] T. Sasao, "Exmin2: A Simplification Algorithm for Exclusive-Or Sum-of-Products Expressions for Multiple-Valued-Input Two-Valued-Output Functions", *IEEE Trans. on CAD.*, Vol. 12, No. 5, pp. 621-632, 1993.

[219] N. Song, and M.A. Perkowski, "Minimization of Exclusive Sum of Products Expressions for Multi-Output Multiple-Valued Input Switching Functions," *accepted to IEEE Trans. on CAD.*

[220] S. B. Thrun and et. al., "The Monk's Problems - A Performance Comparison of Different Learning Algorithms", Carnegie Mellon University, Dec. 1991.

[221] 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.

[222] Perkowski et al, "Development of the Search Strategies for MULTIS," *report*, 1995.

[223] Perkowski et al, "Encoding Problem in Boolean Decomposition," *report*, 1995.

[224] Perkowski et al, "Documentation of MULTIS," *report*, 1995.