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.

June 17, 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, multiple-valued, fuzzy, and continuous functions, both completely specified, incompletely specified, and with generalized don't cares.

Contents

List of Figures

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 larger dimension is decomposed to several problems of smaller dimensions, which can be solved separately and with a smaller effort. The problem of decomposition of a Boolean function F is to find a 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 onset and offset of the function. This method used cube calculus instead decomposition charts, and generalized the Curtis method for incompletely specified functions. Although popularly called the *Roth-Karp decomposition,* this is basically a 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 create 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 below 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 blocks,
- *the representation of functions* in the decomposition, which is, what kind of data structure is the function represented with (arrays, lists, trees, graphs, spectra, etc.),
- the type of functions - the functions may be: binary, multiple-valued, fuzzy, and 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. Until now, only completely specified fuzzy and continuous functions have been presented in the literature.
- the method to solve the column multiplicity problem.

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. [156]. The special emphasis there was on the decomposition types, and decomposition representations. In the present report, however, we will concentrate on these methods and ideas only, 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 [?, 148]. The method had two variants of the decomposition type. One was based on a bond 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 has been next programmed by Wei Wan

[?]. Another variant, which we will call here the Perkowski-Uong-Brown Decomposition type (*PUB Decomposition, for short*), was based on minimizing the Row Multiplicity Index, and the so-called *trivial functions.* (All notions used here will be explained in the next sections.) If a cofactor can be realized in an easy way (be one of 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 bond set. In addition, PUB Decomposition does not require to encode the bond set, as Curtis 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, one can talk about the PUB decomposition type. While for a random function F the number of all Curtis and Roth-Karp decompositions is the same, the number of PUB decompositions is different. Therefore, we can talk about different decomposition type here.

Concluding, the decomposition methods presented by Perkowski, Uong, Brown and Wan included Curtis types and PUB types, and can be characterized as follows:

<{*disjoint Curtis, non-disjoint Curtis, disjoint PUB, non-disjont PUB, column multiplicity based EXOR*}, {*minterms, cube arrays*},{*binary, multiple valued; complete, incomplete*}, {*clique partitioning, graph coloring*} >

(Of course, the Ashenhurst decomposition is a special cases of the Curtis decomposition, so it is not separately mentioned here.)

A similar variant of a method was also created by these authors to be applied for the design of PLAs with pair decoders (such PLAs were presented by T. Sasao [48]). A generalized decomposition, as discussed in [63], but for incompletely specified functions, was also developed.

Independently, Luba et al created a complete decomposition theory based on partition calculus [169]. 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 Luba et al work comes also from the data compressing problems in machine learning, pattern recognition and in other areas of AI. In machine learning the idea of reduction of 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 [83]; or other appproaches to reduction of an instance space can also be found in [142]. 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)**, is currently being designed with the goal of outperforming all the existing approaches to those decompositions that are not technology-related but minimize certain general cost functions related to function complexity. Creating such a superior program was requested for high demand applications that are of interest to Pattern Theory group at Avionics Labs of Wright Laboratories, U.S. Air Force.

In a similar ways we generalized the decompositions for multiple-valued logic, fuzzy and continuous logic, and information systems 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 decomposition.

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

In this report we totally unify the methods developed in past by four aforementioned research groups: Portland State University, Warsaw Technical University, XBOOLE group at the Freiberg University in Germany, and Wright Laboratories, and we add several new methods not known from the literature. Therefore, the GUD program will be also able to solve problems 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 presented in this paper 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 minterns, partitions of cubes*}, {*binary, multiple valued output, fuzzy, continuous; complete, incomplete, generalized don't cares*}, {*Boolean function manipulation, weighted clique partitioning, weighted graph coloring*}>

While creating the GUD 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 bond and free sets of input variables; heuristic approaches and their comparison,
3. column compatibility problem,
4. subfunction encoding problem (column encoding problem),
5. combined column compatibility and subfunction encoding problems,
6. use of partial and total symmetries in data to decrease the decomposition search space,
7. methods of dealing with strongly unspecified functions which are typical for machine learning applications,
8. other decomposition types, especially XBOOLE decompositions and their generalizations,

9. controlling 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 on the other hand we would like this report to be understood with less than ordinary effort also 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 criticizm.

The plan of this "white paper" report is the following:
First we introduce, in an elementary way, various types of decomposition known from the literature. 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 decomposition of Wei Wan and Perkowski.

Next we introduce the Partition Calculus of Luba and our new *Cube Diagram Bundles* (CDBs for short), and demonstrate how all these decompositions can be implemented using CDBs.
In a separate paper, [] we discuss the very important problem of finding good free and bond sets of variables. We discuss a new method and compare it 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 of Quinlan, the method of Zwick).
- reuse of the previously found partitions (Ashenhurst, Curtis, Walliuzzaman).

The problem of bond set encoding, and the related problem of function reusing, will be described in a separate, accompanying paper []. One more paper describes the concurrent column minimization/encoding approach []. Finally, a paper is devoted to solving the column compatibility problem [].
This report is organized as follows.
In section 2 we introduce a new representation of Boolean and Multi-valued Functions - *Cube Diagram Bundles (CDBs).* This general-purpose representation combines Cube Calculus [?], Decision

Diagrams [?] and Rough Partitions [?, 45, 67], 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.

Sections 3 to 7 introduce several types of decompositions based on patterns found in this representation.

In section 3 general decomposition patterns with respect to EXOR, OR and AND gates are presented.

Section 4 presents the *Immediate Decompositions* that happen rarely but are of a good quality: Strong Gate Decompositions, and the Ashenhurst Decomposition. A new approach to Ashenhurst Decomposition [?] is also presented - it is shown that contrary to the more general case of Curtis Decomposition [?], the column minimization problem is polynomially complete, and we give an efficient algorithm to solve it.

Section 5 presents a new approach to Curtis Decomposition, which belong to the *Basic Decompositions* of the system. Although in some respects similar to the approach from [?], we use the new representation, and several of its partial problems are significantly improved. For instance, a new very efficient algorithm for coloring of the Column Incompatibility Graph is proposed, that utilizes the similarity of the graph coloring and the set covering problems, and thus gives an exact minimal coloring for any graph that corresponds to a non-cyclic set covering problem.

Section 6 introduces another new concept in logic synthesis: goal-oriented reduction schemes, which generalize the EXOR transformation of Curtis-nondecomposable functions from [?]. Any function can serve as a goal function, and three reduction types (EXOR, OR and AND) of reducing a given function to a goal function are presented.

Section 7 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, $EXOR^2\_SOP$ synthesis (this is a two-input EXOR gate of two SOP circuits used in several PLDs), ESOP, and other.

Section 8 presents the search strategy of using all the above decompositions and reductions.

In section 9 we present several applications of the presented approach to various known logic synthesis problems.

When reading this report, the reader has to bear the following in mind.

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

1.1 Towards a Unified Approach to Decomposition.

Among several types of decomposition of single-output Boolean functions the most familiar is a simple disjoint decomposition. Let F be a multiple-value or binary function representing functional dependence $Y = F(X)$, where X is a set of multiple-value or binary variables and Y is a set of binary output variables. Let $X = A \cup B, A \cap B = \oslash$ 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 $C = \oslash$ then H is called a simple *disjoint decomposition* of F. B is called a *bond 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:

- 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 bond 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.

Since 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 higher than one fan-out - 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 leads to optimal solutions sooner. Sometimes, the decomposition becomes the only way to obtain any results from the minimization programs.

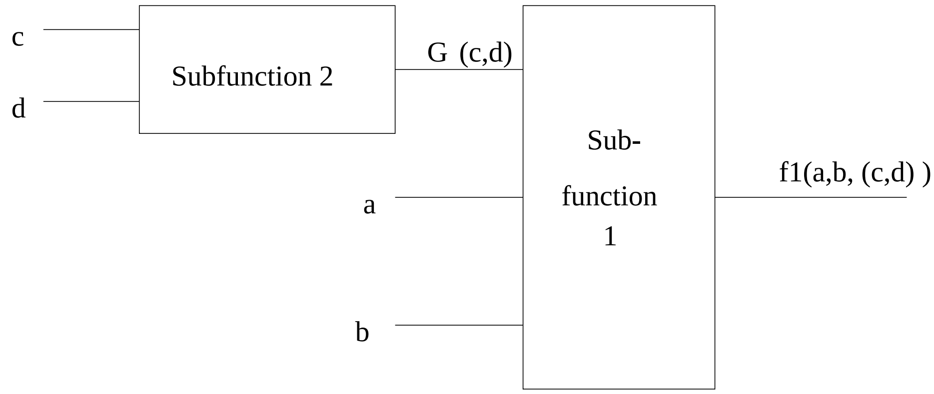


Figure 1:

Decomposition can also assume specific kind of realization of a block: PLA, CLB of FPGA or PLD, standard cell libraries, etc.

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. Again the above methodology when compared with circuit design approaches suffers from absence of functional decomposition strategies, as those applied for binary functions, especially for FPGA-based implementations [112, 85, 88]. Their promising results 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, 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 paper will be applied by us as a dydactic method to simplify our explanations, and make them intuitive. Multiplexers will make also an easy link to multiple-valued, fuzzy and continuous logic, and the *if-then-else* rules.

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 important that the reader is 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 paper 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.

2 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 special use of EXOR gates and EXOR-based 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, multiple-valued, and fuzzy functions, both completely specified, incompletely specified, and with generalized multiple-valued don't cares. Other applications of the new representation, such as synthesis of EXOR_of_SOPs circuits, are presented briefly in [176].

Boolean and multiple-valued functions that include very many don't cares are becoming increasingly important in several areas of applications [155]. 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 contiuously improved with time, and several variants of them have been invented and proved superior in some applications. For instance, XBOOLE system of Bochmann and Steinbach [?] introduces Ternary Vector Lists (TVLs), a variant of Cube Calculus with disjoint cubes, new position encoding and new operations, and demonstrates its superiority 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. In other related efforts: Luba et al [?] introduced a new representation of Rough Partitions and used it in few successful programs for Boolean and multiple-valued decomposition; and Truth Table Permutations to create BDDs are recently investigated [171, 174].

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 [152], Three level AND/NOT Networks with True inputs (TANT) networks synthesis [149], and other.

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

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

CDBs are related to four representations known from literature:

- Cube Calculus [?],
- Decision Diagrams [?],
- Rough-Partitions [?],
- and Boolean Relations [172].

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) [93, 155], 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,
- it is difficult to add and remove input and output variables to the cubes dynamically in the synthesis process,

	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: Table 1. A Table of a mv-output Function with Generalized Don't Cares.

- in strongly unspecified functions we would need relatively few but very long cubes,
- column-based operations are global, and therefore slow.

Luba invented a new function representation called Rough Partitions (r-partitions, or RP) [?]. R-partition is also called a *cover*. 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.

Our other source of inspiration while creating this representation was the concept of the Generalized Don't Cares [156]. 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 has also some link to Boolean Relations. 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".

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). 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 [171, 174]. While the authors from [171, 174] solve it as a truth-table permutation problem, we solve it as a cube encoding problem, which is more general.

Example 1: Tables and encodings of functions. 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. reffig:fig1ma. This table has four disjoint cubes. Two are OFF cubes, enumerated 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, a new map from Fig. ??b is created. Figure 1a,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. ??a is shown in Fig. ??c, and the encodings of its rows to secondary input variables is shown in Fig. ??d.

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.

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. 5a. In this case:

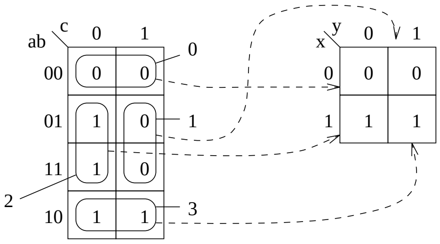
$Out(F) = \{$ [pointer to BDD for {0,2,3,4,5,6,8}, pointer to BDD for {0,1,7,8,10}, pointer to BDD for {1,9,11}, pointer to BDD for {3,4,6,7,9,11}, pointer to BDD for {5,10}] }
;;;;;; for 5-valued output variable f.

The third example presents a table, Fig. ??f, for an incompletely specified function with standard don't cares from Fig. ??e. 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 inptut variables x and y is shown in Fig. ??g.

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.



a) b)

row (cube)	input variables			input variable
	a	b	c	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 1. Mapping from primary to secondary variables: (a) original function with primary input
(b) swcondary space with secondary input variables x and y
(c) table of function f
(d) encoding of primary cubes

MV function: (e) K-map
(f) table
(g) encoding to secondary variables

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

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

The following points about CDBs are important to note:

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

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

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

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

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

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

6. There is no difference in the representation of primary input variables, auxiliary variables and output variables. For instance, CDBs are good to represent functions defined on Boolean operators on input variables, one can just add new "columns" representing these operators 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.

7. There exist one more variant of CDBs, that we call *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 more detail in one of subsequent paragraphs.

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.

3 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 3.1 Let us assume that we want to realize the four-input function $f1(a, b, c, d)$, shown in Fig. 2a, using a 4×1 multiplexer with two select lines (2 bit address).

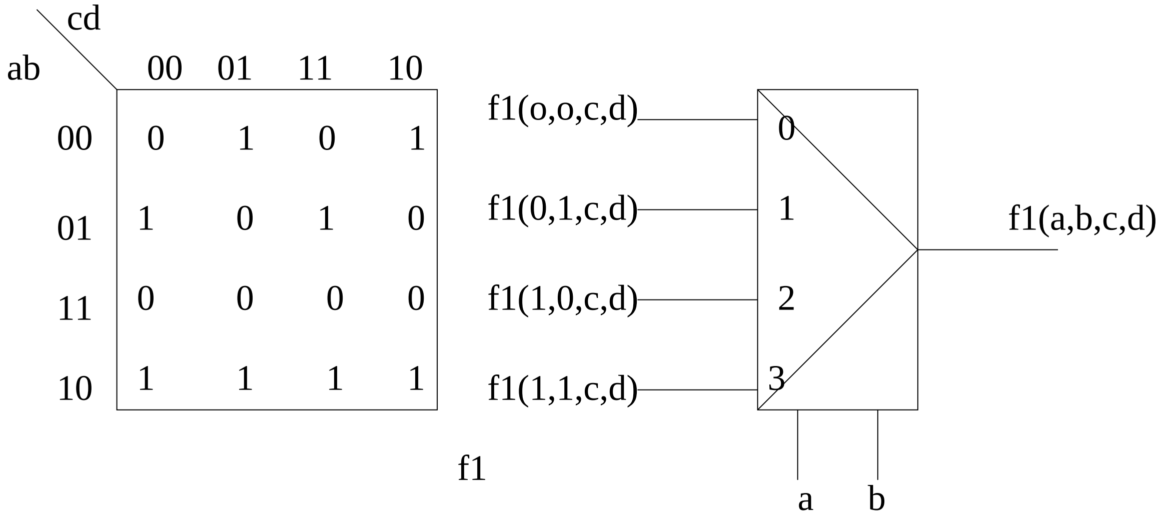


Figure 2:

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 used by notation 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 2b.

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 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. 3.

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. 4. After solving few examples, however, the reader f1(0,0,c,d) =

cd	00	01	11	10
	0	1	0	1

f1(0,1,c,d)=

cd	00	01	11	10
	1	0	1	0

f1(1,1,c,d) =

cd	00	01	11	10
	0	0	0	0

f1(1,0,c,d) =

cd	00	01	11	10
	1	1	1	1

Figure 3:

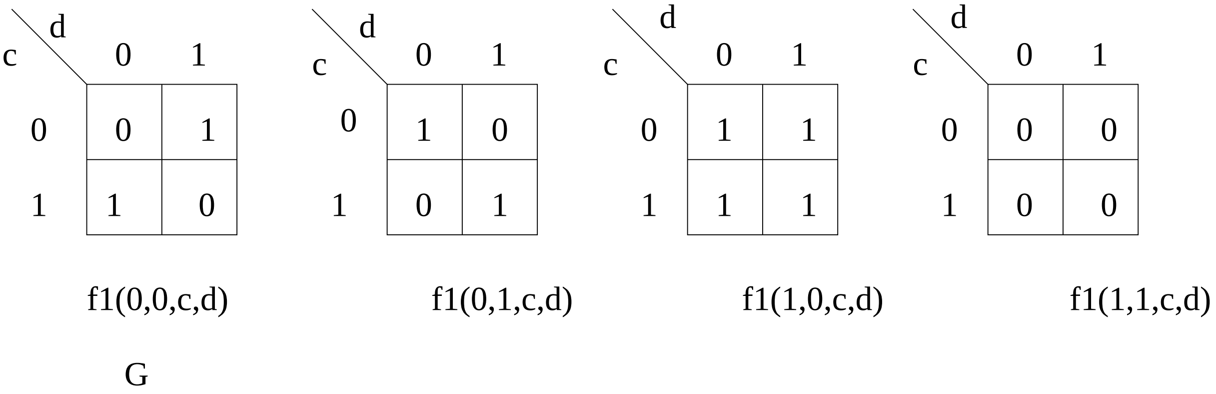


Figure 4:

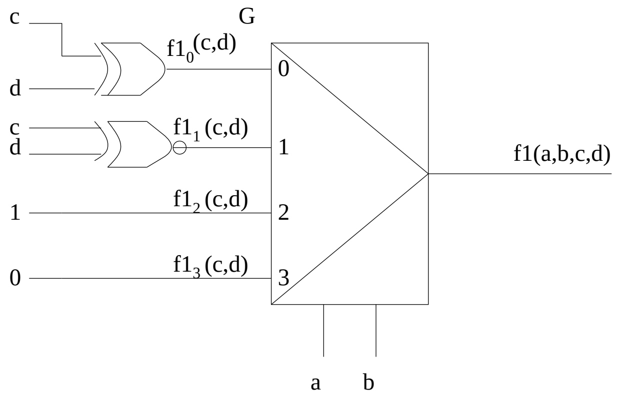


Figure 5:

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. 5.

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. 6.

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. 7a), the functions $f1_0(b,d)$, $f1_1(b,d)$, $f1_2(b,d)$, $f1_3(b,d)$ will be as shown in Fig. 7b and the realization as shown in Fig. 7c. Note the inverse order of variables in the headings of some of maps from Fig. 7b. If the designer has difficulties in using the "inverse" K-maps, he can rewrite some of them as shown in Fig. 7d. 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. 7c has the same cost as one from Fig. 5.

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. 8 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

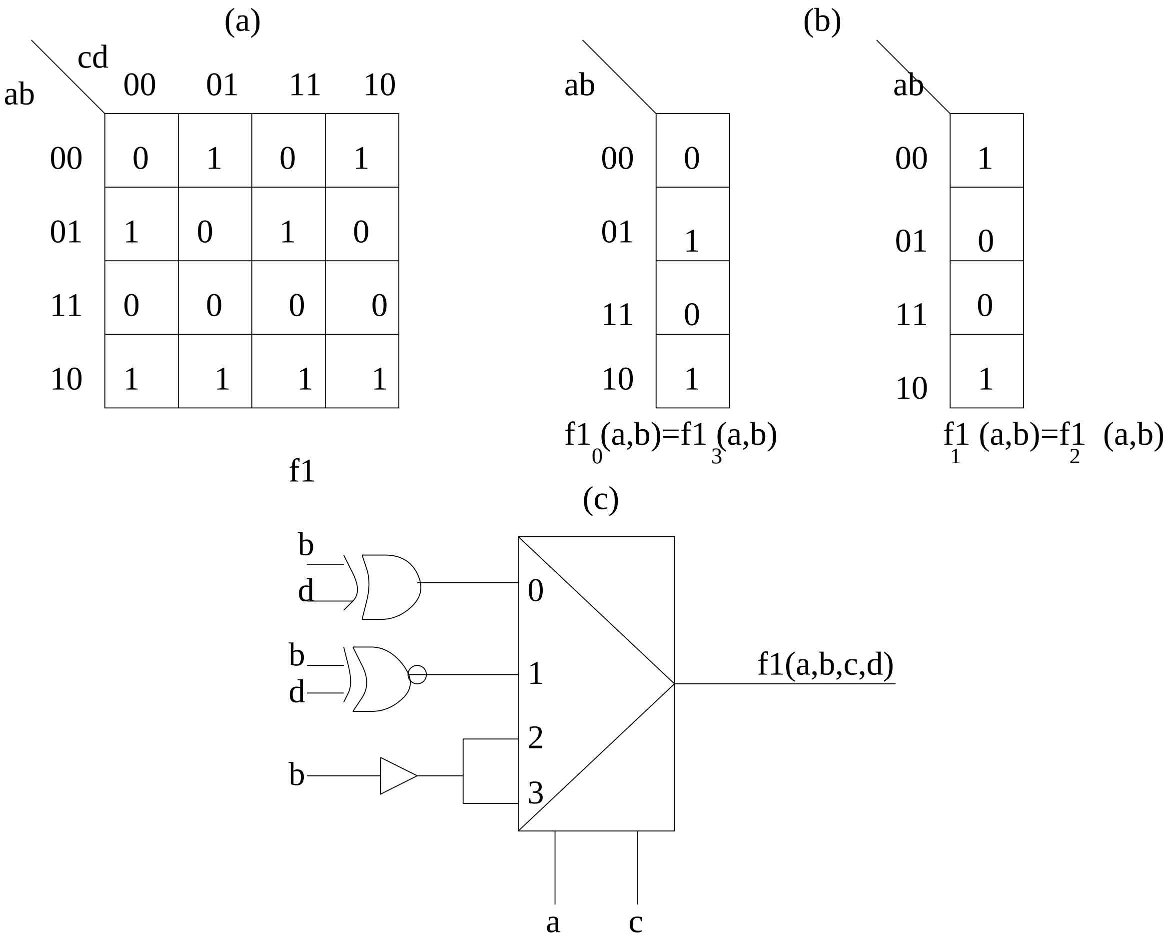

(a)
cd
ab
00 01 11 10
00 0 1 0 1
01 1 0 1 0
11 0 0 0 0
10 1 1 1 1
f1
(b)
ab
00 0
01 1
11 0
10 1
f1_0(a,b)=f1_3(a,b)
ab
00 1
01 0
11 0
10 1
f1_1 (a,b)=f1_2 (a,b)
(c)
b
d
b
d
b
0
1
2
3
f1(a,b,c,d)
a
c

Figure 6:

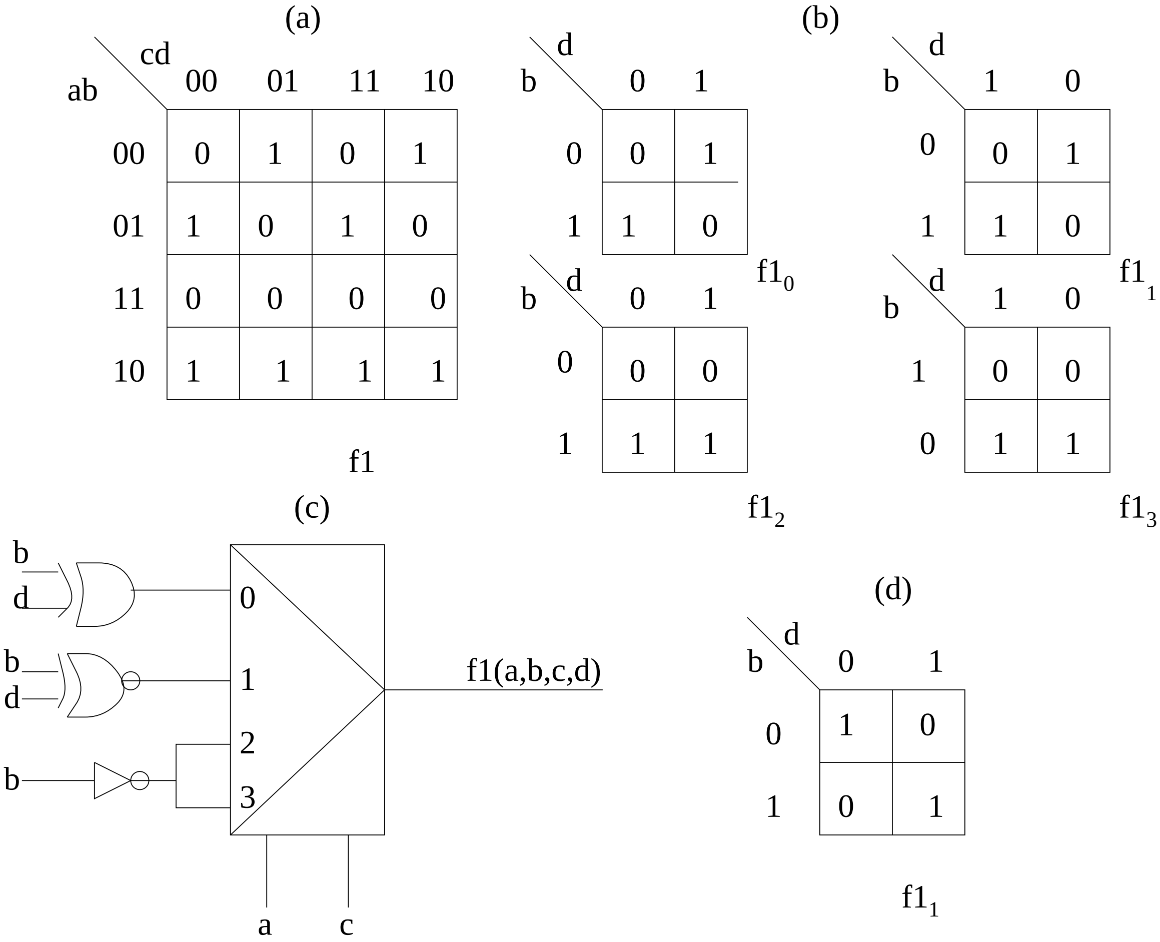


Figure 7:

variables (such products are shown in bold - for instance group 3 for ab in Fig. 8a). 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. 8a 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. 8.
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 paper [] we illustrate how the above method can be extended to design with generalized multiplexers in multiple-valued, fuzzy and continuous logics.

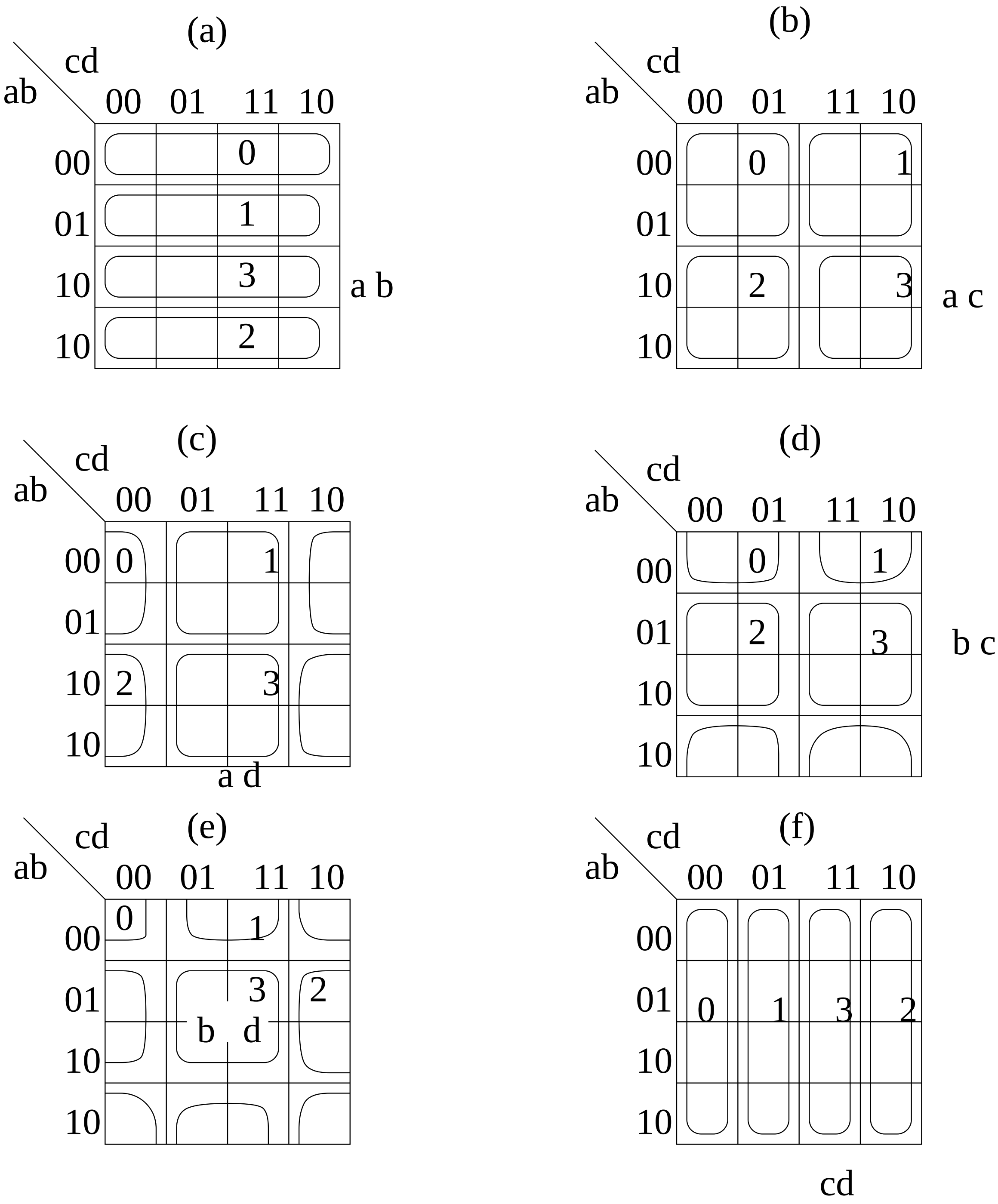

(a)
cd
ab
00 01 11 10
00
01
10
10
0
1
3
2
a b
(b)
cd
ab
00 01 11 10
00
01
10
10
0
1
2
3
a c
(c)
cd
ab
00 01 11 10
00
01
10
10
0
1
2
3
a d
(d)
cd
ab
00 01 11 10
00
01
10
10
0
1
2
3
b c
(e)
cd
ab
00 01 11 10
00
01
10
10
0
1
3
2
b d
(f)
cd
ab
00 01 11 10
00
01
10
10
0 1 3 2
cd

Figure 8:

4 Basic Patterns for Decompositions.

The authors are 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 *bond and free sets*: Curtis Decomposition [?], Steinbach et al (XBOOLE) Decomposition [?], and Perkowski et al (PUB) Decomposition [148, 147, 156]. The decompositions of Ashenhurst [?]; Luba et al [?]; Lai, Pedram et al [?], and many other are just special cases, or different realizations, of the Curtis decomposition [156]. While most authors differentiate between disjoint and non-disjoint decompositions, the concept of the Repeated Variable Maps (RVMs), introduced below, allows to explain them in a uniform way, and the CDBs allow to realize all these decompositions uniformly in software (including those from [?, 148, 147, 156]).

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 *bond set*, and set $A \cup C$ is a *free set*. If $C = \phi$ the decomposition is disjoint and the RVM becomes a standard Karnaugh Map. If $C \neq \phi$ 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 bond 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 bond 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 [?, ?, 148, 147, 156, ?] 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 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.

Similarly we can define other combined patterns of DC, ON, OFF, F, and $\overline{F}$. Let us observe that if a function has DC/ON/OFF pattern on columns then it is independent on the variables from the bond set. Analogically, if a function has DC/ON/OFF pattern on rows then it is independent on the variables from the free 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 for the lack of space, and all possible decomposition methods are based on finding these patterns in functions.

Definitions of Patterns. 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. *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. *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. *Column OR decomposition* exists with respect to the set of column variables RV if there exists at least one column that has the ON Pattern. Analogically one can define the *Column AND decomposition* and the *Column EXOR decomposition.* Row and Column decompositions are also called Weak Decompositions [?]. There exist then *Weak AND, Weak OR*, and *Weak EXOR* decompositions.

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, there exists a DC/ON/OFF Pattern on columns. Equivalently, *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, there exists a DC/ON/OFF Pattern on rows. *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, there exists a DC/ON/OFF Pattern on columns. Equivalently, *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, there exists a DC/ON/OFF Pattern on columns. *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. *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. Or, *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.* They are all called *Immediate Decompositions.*

An example of the RVM is shown in Figure ??. Fig. ??a presents a standard Kmap of 3-input function f. Assuming b to be a repeated variable, the Bond Set $\{b,c\}$ (the columns) and the Free Set as $\{a,b\}$, one creates a RVM from Fig. ??b. 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 Fig. ??c, the OFF Patterns $(a + b)$ and $(\overline{b}\ \overline{c})$ are found, which lead to the Strong AND Decomposition: $(a + b) \cdot (\overline{b}\ \overline{c})$. Finally, for the same RVM in Fig. ??d, 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 Fig. ??d. These patterns lead to Strong EXOR Decomposition: $(b\ c) \oplus (a\ +\ b)$ from Fig. ??e. Fig. 2e clearly shows the incomplete patterns from Fig. ??d after their completion with 0's and 1's. Bold symbols correspond to original cares from the RVM.

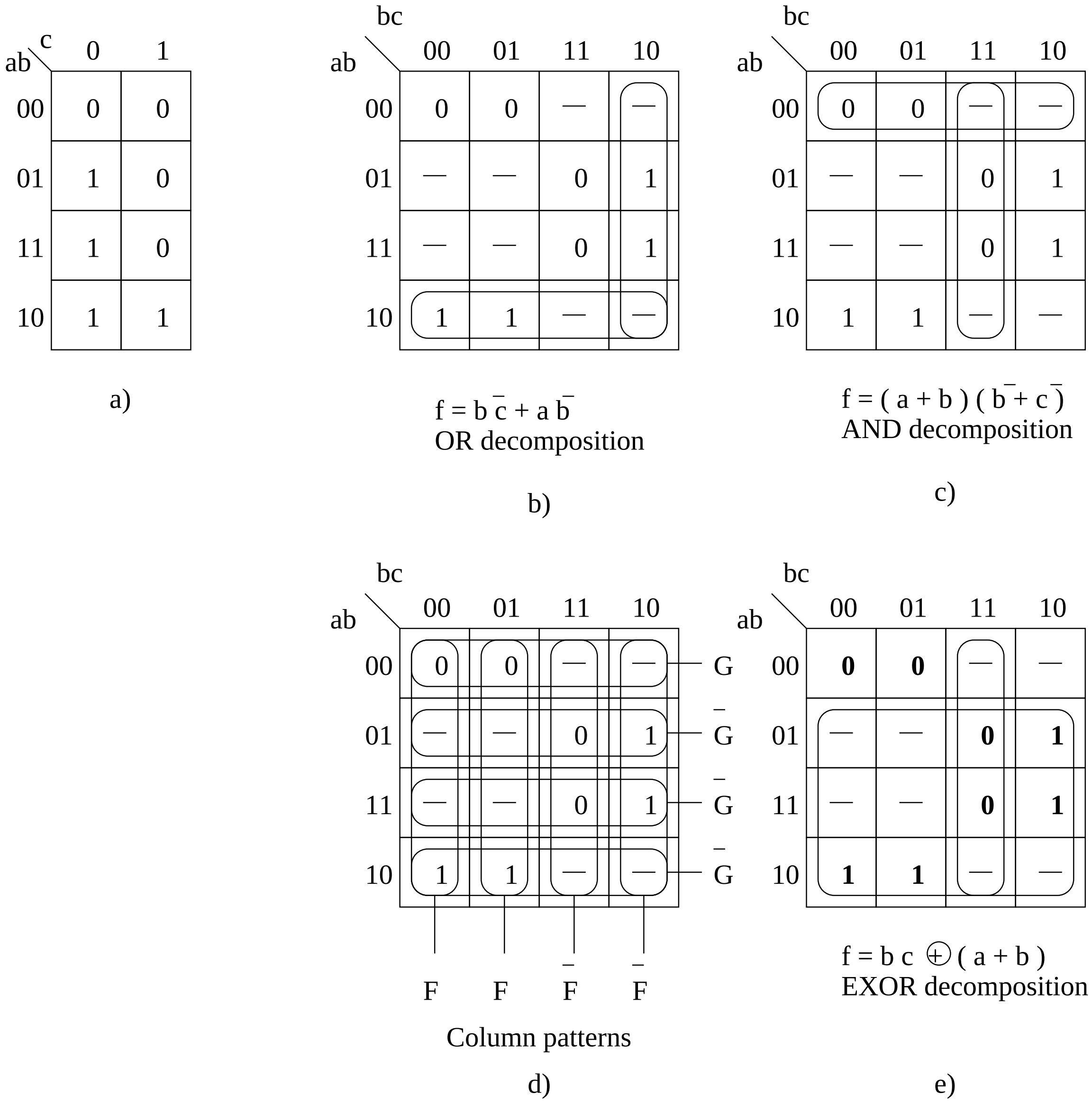


Figure 2. Basic patterns and decompositions.

5 Ashenhurst Decompositions for Completely Specified Functions.

5.1 Disjoint Ashenhurst Decomposition of Completely Specified Functions.

Example 5.1 Let us first observe that function $f1_0(c, d)$ in Fig. 5 is the negation of function $f1_1(c, d)$. Therefore, the realization from Fig. 5 can be re-drawn as shown in Fig. 9.

Next, the function from Fig. 9 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 bond variables* {c, d} and the *set of free variables* {a, b}. Such decompositions are discussed, among others, in [164].

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 Figs. 9 and ?? that if we want to have a single connection from predecessor to successor then all of the data inputs to the multiplexer have to be a constant 0, a constant 1, an output of predecessor or a negation of predecessor (the negation is created inside of the successor). This is basically a proof of the Ashenhurst theorem [8].

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

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

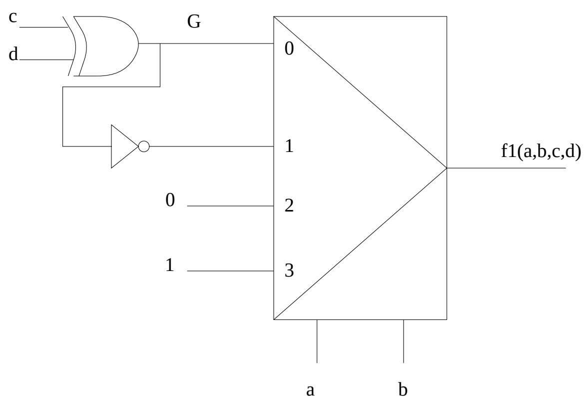


Figure 9:

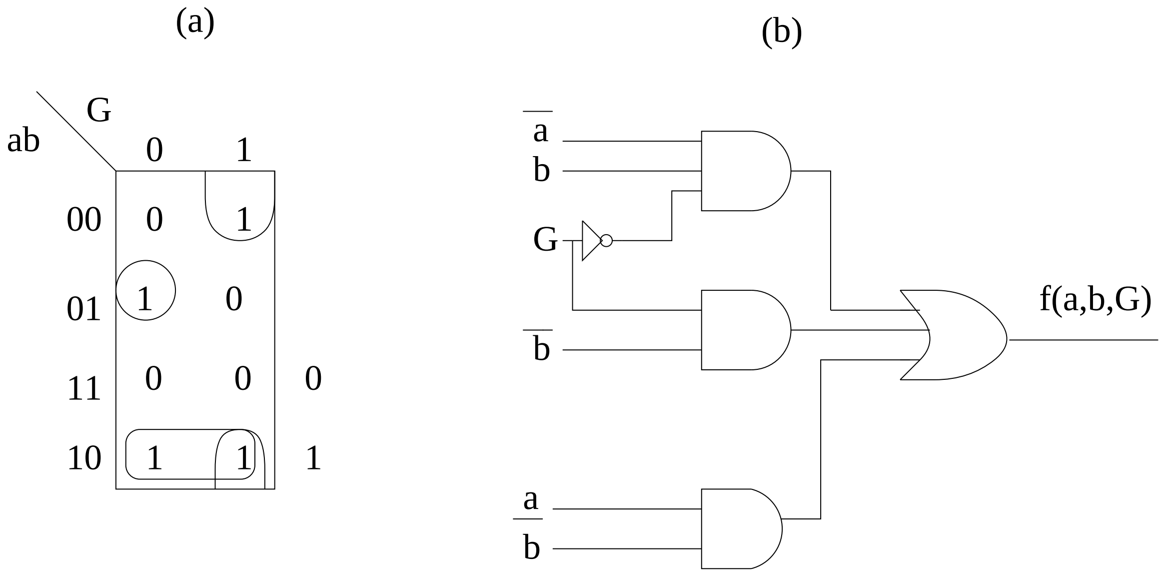


Figure 10:

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

If one draws a K-map with set of colums corresponding to bond set variables, the functions $K(B)$ and $L(B)$ will be called *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 a bond 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 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 nonexistence of a decomposition. He does this subsequently for all other sets of free variables. There is no need to redraw the sub-K-maps. To find the existence of the decomposition can be more time consuming and it may require, for K-maps with many variables, re-drawing of some submaps.

Example 5.2 Let us go back to function $f1(a, b, c, d)$ from example 5.1. We assume address variables a, b. Loop $a\,\overline{b}$ of ones and loop $a\,b$ of zeros are discarded so 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 decomposed realization of this function is shown in Fig. 10.

The K-map of successor is created from the K-map of Fig. 2, G, and $\overline{G}$, as in Fig. 10a. This function is minimized using two-level Boolean minimization to the circuit from Fig. 10b.

Example 5.3 For the same function let us now select variables c and d as address variables (Fig. 6a).

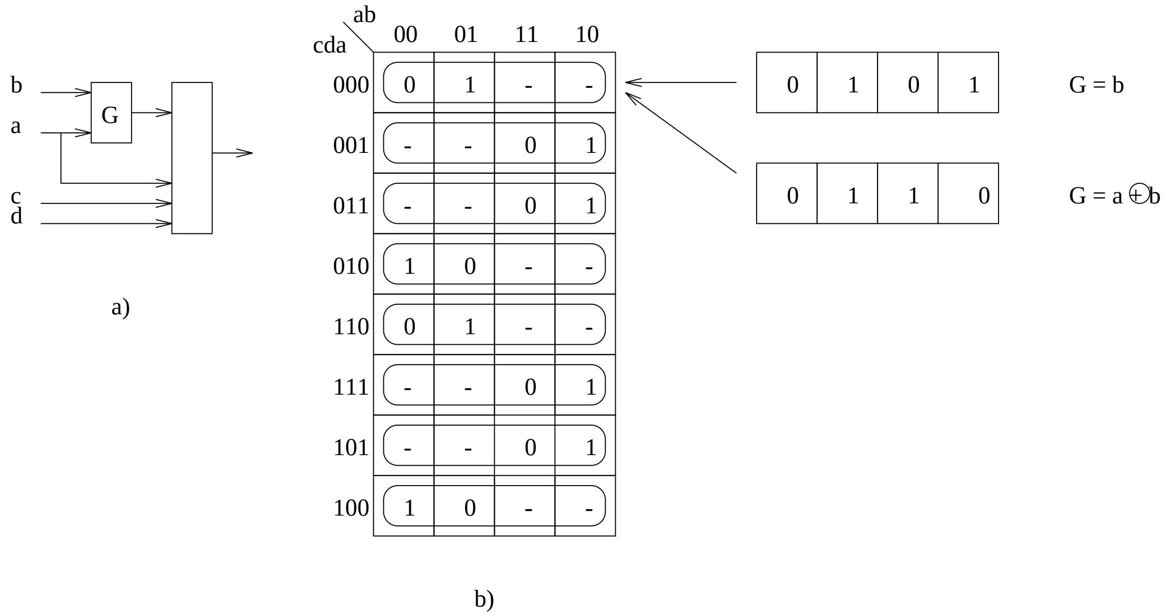


Figure 11:

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.

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 4.1). 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 find when the function is strongly unspecified.

5.2 Non-disjoint Ashenhurst Decomposition of Completely Specified Functions.

Example 5.4 Let us again consider the function from Fig. 2a 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 (see *Example 4.3*). However, function may be decomposed using non-disjoint Ashenhurst decomposition technique. Let us consider decomposition scheme showed in Fig. 11a where variable a belongs to both free and bound sets $a = A \cap B$ and $A = \{c, d, a\}$, $B = \{a, b\}$.

K-map for this scheme is shown in Fig. 11b. Don't cares in that map correspond to situations which never happen. For instance, in loop $\overline{c}\,\overline{d}\,\overline{a}$ situation where bound set $a = 1$ never happens for 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. 12 shows a different way of using K-map to make the same decomposition. Decomposition scheme from Fig. 12a is equivalent to the scheme from Fig. 11a. K-map is shown in Fig. 12b. The loops drawn in K-map correspond to free set variables $A = \{c, d, a\}$.

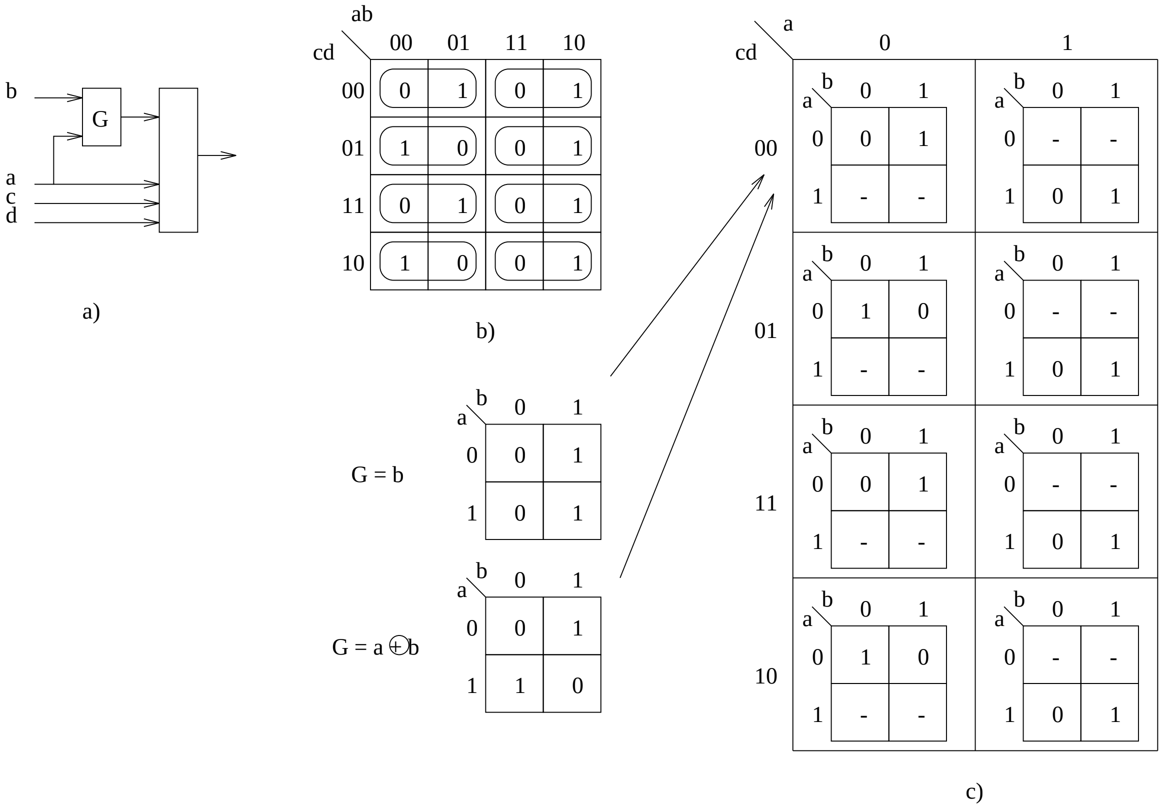


Figure 12:

To form K-maps for loop functions however, we procede in a way which is slightly different from the one used in former examples. K-map for the loop function $\overline{c}\,\overline{d}\,\overline{a}$ for instance, contain 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. 12c. If the DCs in these K-maps may be filled up in such a way that we end up with a function G (and possibly $\overline{G}$) then Ashenhurst decomposition exists. In our case we may fill up DCs to get $G = b$ ($A = \{c, d, a\}$, $B = \{b\}$) or $G = a \oplus b$ ($A = \{c, d, a\}$, $B = \{a, b\}$).

6 Ashenhurst Decomposition of Incompletely Specified Functions.

6.1 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 6.1 Given is function $f2(a,b,c,d,e,f)$ from Fig. 13.

Assuming address variables a and b, the loops are as shown in Fig. 13. We can immediately discard loop $\overline{a}\ b$ that can be completed to all zeros, as well as the loop $a\ b$ that can be completed to all ones. To help the reader the loops functions $\overline{a}\ \overline{b}$ and $a\ \overline{b}$ are rewritten to these of Fig. 14. It can be seen that both of them can be completed to the same function

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

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 (bond) sets, it is often not neccessary to generate loops systematically for all the possible free (bond) 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 (bond) set, the decomposition for the next large loop of the above types is investigated.

Example 6.2 We will solve the example from p. 171 in [164].

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

The K-map for this function is shown in Fig. 15.

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. 16. This suggests selecting the set $\{x,z\}$ as a free set. By examining the other loops for this set we find that loop $\overline{x}\ \overline{z}$ can be completed to all zeros. Now the remaining loops are found and compared (Fig. 17). As we see, assuming that loop $x\ \overline{z}$ corresponds to G, we notice that the loop $\overline{x}\ z$ will correspond to $\overline{G}$. The completion is shown in Fig. 18a.

Therefore the decomposition for free set $\{x,z\}$ is found, as shown in Fig. 18a. Fig. 18b, c presents two methods for realization of function $f(x,z,G)$. The final circuit is presented in Fig. 18e.

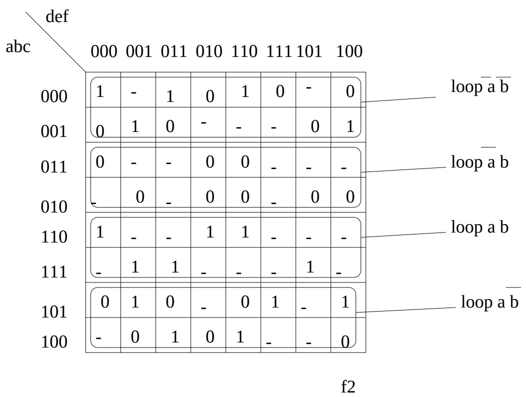


Figure 13:

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 14:

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 15:

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 16:

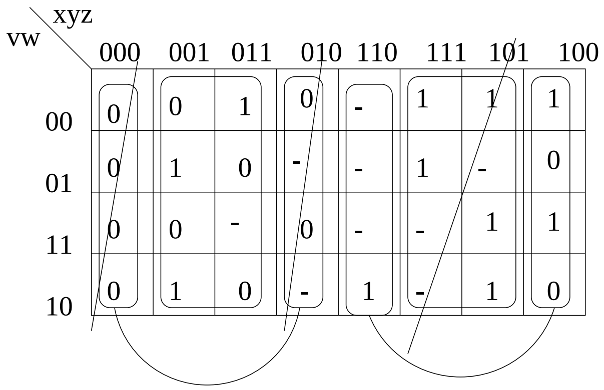


Figure 17:

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

See section 18, example 5.4, and section 6.1.

6.3 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 exension 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 $n = 2$).

Multiple-valued function (three possible values: 0, 1, 2) which is defined in Fig. 19 is Ashenhurst decomposable because the column multiplicity is equal to three (00,02,12 ; 10,01,21 ; 20,11,22).

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

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

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

As an example, let us again consider the function from Fig. 19 with DC of type $\{0, 1\}$ in the field 022 (a=0, b=2, c=2). Since this field can only take 0 or 1 value the function in not decomposable (column multiplicity is equal to four). However, if we put DC of any other type (containing 2) in the same field, the function will become decomposable.

Non-disjoint Ashenhurst decomposition is the one having non disjoint bound (B) and free (A) sets. If we choose $A = \{a, b\}$ and $B = \{b, c\}$ for the function from Fig. 19 we will come out with K-map in Fig. 20. The empty fields are DC fields of type $\{0, 1, 2\}$, which means that they may be replaced by 0, 1

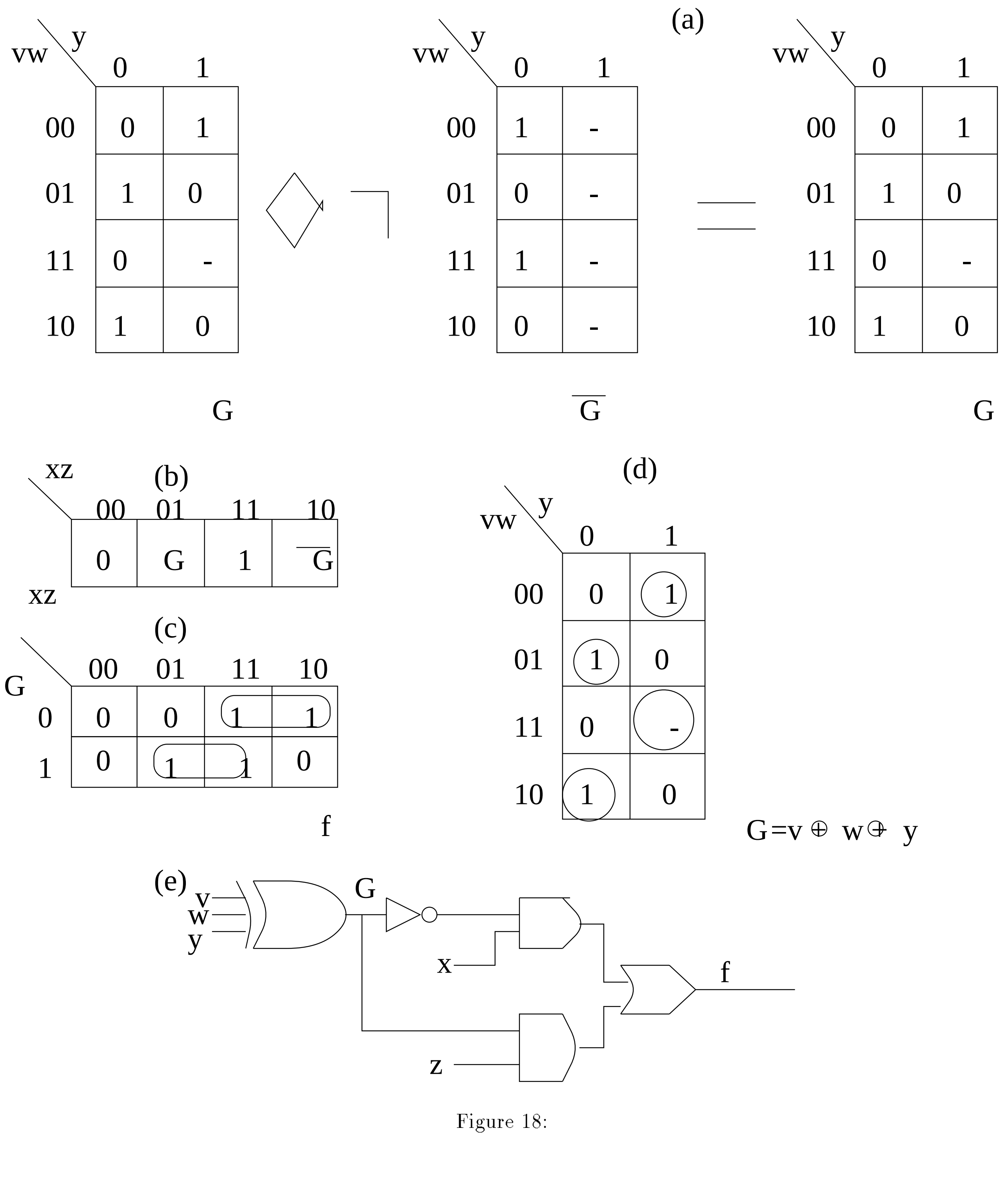

(a)
vw y
G
G
G
(b)
xz
00 01 11 10
0 G 1 G
(c)
G
00 01 11 10
0 0 0 1 1
1 0 1 1 0
f
(d)
G=v ⊕ w⊕ y
(e)
v
w
y
G
x
z
f

Figure 18:

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 19:

or 2 value. Since the disjoin decomposition exists non-disjoint decomposition exists either. Non-disjoint decomposition however, is usually used to decompose functions which are not disjoint decomposable.

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 20:

vw \ xyz	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 21:

7 Analytical Decomposition of Incompletely Specified Single-Output Boolean Functions.

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

Definition 7.1 *By a loop pair we will understand a pair:*

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

Definition 7.2 *Two functions are called the* compatible functions *when they can be completed to the same function.*

Definition 7.3 *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 7.4 *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 $1\Diamond 0 = e$) and we say that functions f_1 and f_2 are *not compatible.*

Definition 7.5 *Two loop pairs* $p_1 = [l_1, f(l_1)]$ *and* $p_2 = [l_2, f(l_2)]$ *are* merged *to a loop pair* $[\{l_1, l_2\}, f(l_1)\Diamond f(l_2)]$ *if* $f(l_1)$ *and* $f(l_2)$ *are compatible functions.*

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

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

Example 7.1 Let us consider a function given in a Karnaugh map of Fig. 21, assuming $B = \{c, d, e\}$.

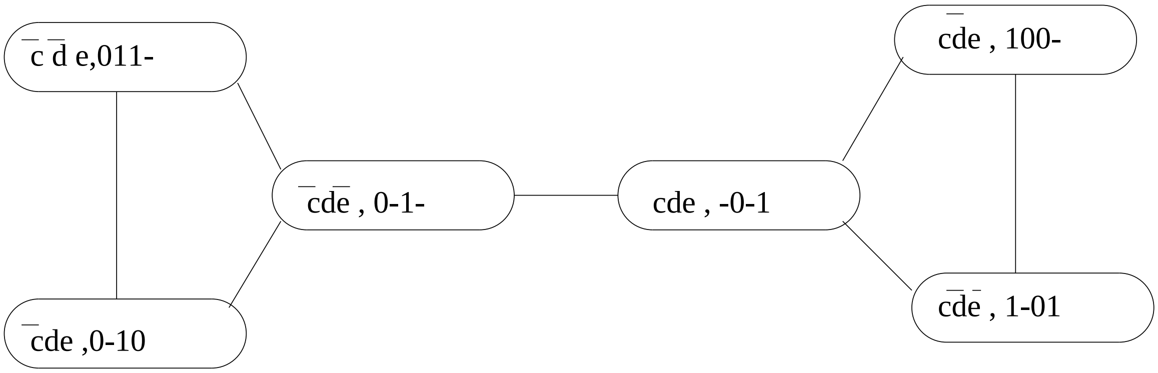


Figure 22:

The free set $B = \{c, d, e\}$, so the loop functions correspond to the columns of the map. After removing the loop $\overline{c}\,\overline{d}\,\overline{e}$ of 1's and the loop $c\ d\ \overline{e}$ of 0's, the *loop pair Compatibility Graph* of Fig. 22 is created with the remaining loop pairs as nodes.

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

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

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

Compatibility of loop pairs is denoted as an edge between nodes

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

In our notation, the nodes: [$\overline{c}\ d\ e$, 0-10] and [$\overline{c}\ d\ \overline{e}, 0-1-$] (see Fig. 22).

Similarly, loop pairs [$\overline{c}\ d\ \overline{e}$, 0-1-] and [$c\ d\ e$, -0-1] are compatible, since 0-1- -0-1 = 0011. However loop pairs [$\overline{c}\ d\ e$, 0-10] and [$c\ d\ e$, -0-1] are not compatible since 0-10 -0-1 = 001e (by e 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. 22. The next stage is to find *a partition of this graph to two maximum cliques.* A *clique* of a graph is a subgraph with any two nodes linked by an edge (a complete graph). *The maximum clique* is a clique with the maximum number of nodes (a clique that is not included in another clique). If after merging the loop functions from the first clique we would create function G then the merging from the another clique should be $\overline{G}$, otherwise there is no decomposition. The classical decomposition of the function does not exist in one of the following two cases:

- there is more than two cliques in the minimal partition.
- $\diamond_{c_i \in C_1} c_i \neq \overline{\diamond_{c_j \in C_2} c_j}$

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 for another order.

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

8 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 [156].

8.1 Ashenhurst Decomposition.

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

Property 1. Ashenhurst Decomposition with bond 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 bond 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 [?, 45, 67, ?, 147, 148, 156, ?]. 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, OFF and DC Patterns.*

2. Find two rows, r_i and r_j that are incompatible, and remove them.

3. From remaining rows create the set Remaining_Rows.

4. Pair_Counter := 1.

5. *Put row r_i to LEFT[Pair_Counter] and row r_j to RIGHT[Pair_Counter].*

6. *Take next row r_s in set Remaining_Rows and remove it from set Remaining_Rows.*

7. *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).

8. *If there are still rows in Remaining_Rows, go to 5.*

9. *Combine all sets LEFT[i] (i=1,...,Pair_Counter) to set LEFT,*
   Combine all sets RIGHT[i] (i=1,...,Pair_Counter) to set RIGHT.

10. *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:

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 combined
   row RIGHT3 to the end of array RIGHT, append 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 8.1 We continue the previous example, and solve it with algorithm **??**.

Let us assume, that first loop pair [$\overline{c}$ d $\overline{e}$, 0-1-] is selected. It is marked as G. Let us assume now, that as a second, we select the pair [c d e, -0-1]. The function from this pair has no common specified bits with function G, so the pair is added to set UNSPECIFIED. The next selected pair is [c $\overline{d}$ $\overline{e}$, 1-01]. Negation of function from this pair has a common complement with function G so $\overline{G}$:= 1-01. The pair [c $\overline{d}$ e, 100-] selected as the next one is compatible with $\overline{G}$, so now $\overline{G} = 1-01\diamond 100- = 1001$, so $G = 0110$.

Next pair [$\overline{c}$ $\overline{d}$ e, 011-] is selected and merged with G. Next [$\overline{c}$ d e, 0-10] is merged with G. Now the functions from the set UNSPECIFIED are taken into consideration again. The pair [c d e, -0-1] can be joined with $\overline{G}$ only. Decomposition exists with $G = 0110$ and $\overline{G} = 1001$.

9 PUB Decompositions.

9.1 Disjoint PUB Decomposition.

The method shown here was first introduced by Perkowski, Uong and Brown in 1987, [147, 148]. It can use arbitrary representation of switching functions. Although this method is similar to Curtis Decomposition, it differs from it. Curtis Decomposition uses multiplexer for bond set and encodes outputs of the bond set. PUB Decomposition uses free set variables for multiplexers and does not use encoding. Thus, it creates functions with higher value of DFC, but possible from simpler gates. It must be then still investigated, which of the decompositions creates less expensive realizations from gates of small fan-in. 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 *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 [$ON(f_24) \cap OFF(f_1) = \emptyset$]

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,*
- *$E = (c_1,\ c_2) \in RS \Longleftrightarrow$ loop functions $f(c_1)$ and $f(c_2)$ are not compatible (such functions are called incompatible).*

The nodes of the graph are loops (cofactors). The edge E is created in the incompatibility graph between two nodes when the corresponding loop functions of these loops 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 Loop-Pair-Compatibility-Graph clique partitioning 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 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.

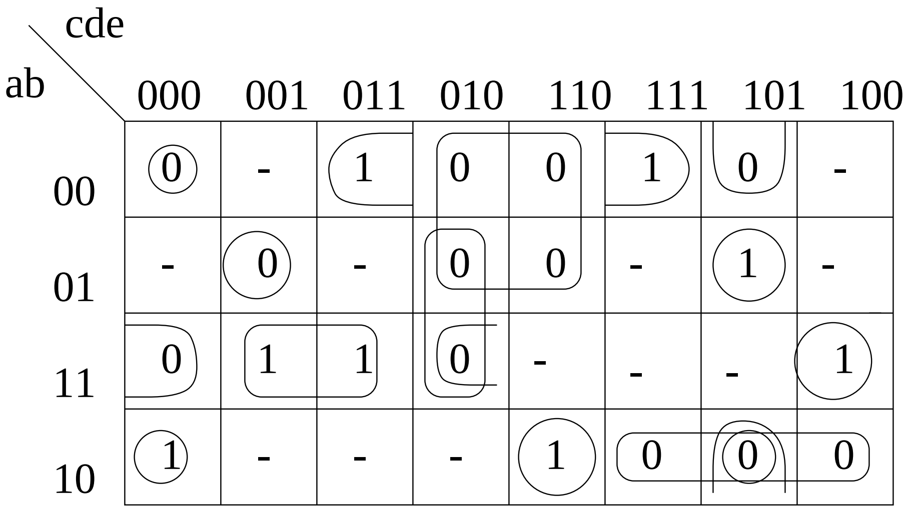


Figure 23:

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}

The function is presented in a Karnaugh map from Fig. 23.
We illustrate application of the optimal graph-coloring algorithm for $B = \{c, d, e\}$.

1. Find cofactors in the form of arrays of cubes.

ON(xx000) = {10xxx}	OFF(xx000) = {00xxx, 11xxx}
ON(xx001) = {11xxx}	OFF(xx000) = {01xxx}
ON(xx010) = {}	OFF(xx010) = {0xxxx, x1xxx, 11xxx}
ON(xx011) = {00xxx, 11xxx}	OFF(xx010) = {}
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.

3. Create the graph of loop pair incompatibility.

   For the pair of loops XX000, XX001 we have:

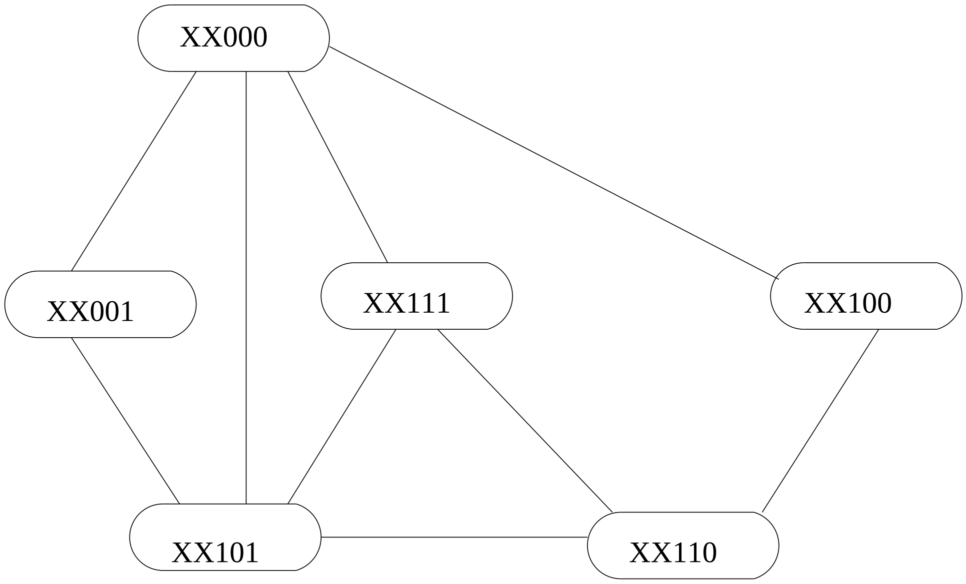


Figure 24:

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. 24 is created.

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

   After coloring with a minimum number of colors the graph is as in Fig. 25.

   The minimal coloring has then 3 colors: AA, BB, and CC. The 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:

   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,

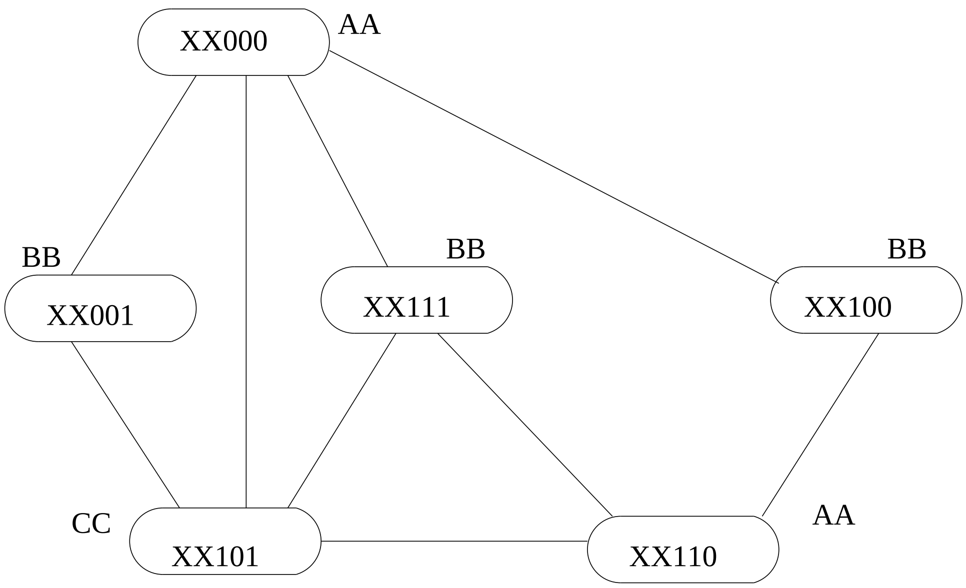


Figure 25:

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: functions being mergings of all loop functions of the same color replace the original loop functions. Now our K-map looks as in Fig. 26.

6. Depending on the number of colors and compatibility of F_1 and $\overline{F_2}$: either find classical decomposition or find a realization using a multiplexer that minimizes the number of various functions given on this multiplexer's data inputs.

The respective realization with multiplexers is shown in Fig. 27. 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 generalized decompositions introduced in [147].

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 data inputs to the multiplexer are 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.

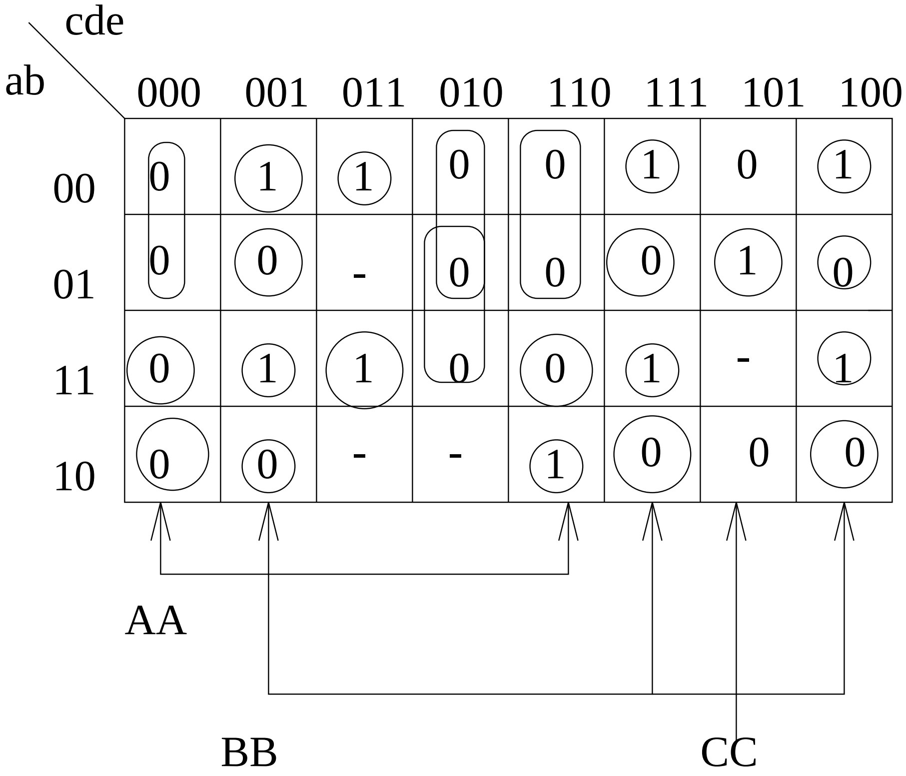

cde
ab
000 001 011 010 110 111 101 100
00 0 1 1 0 0 1 0 1
01 0 0 - 0 0 0 1 0
11 0 1 1 0 0 1 - 1
10 0 0 - - 1 0 0 0
AA
BB
CC

Figure 26:

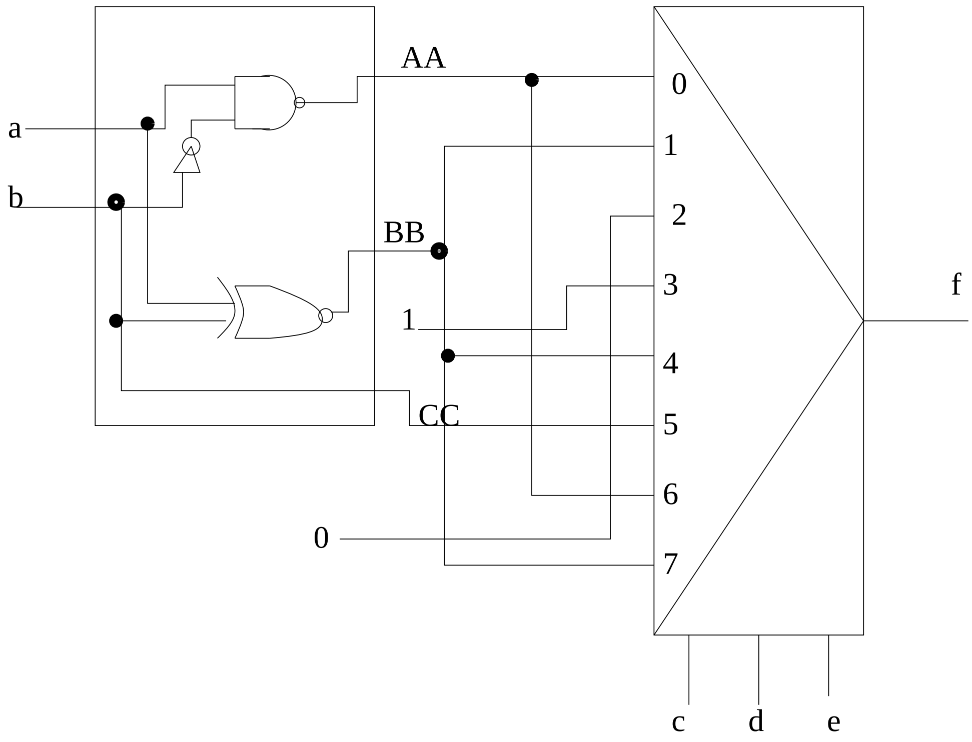


Figure 27:

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 [147]:

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.

1. In the first two cases it is very easy to find trivial functions. Costant 0 is assigned, when the ON-set of the cofactor function f_{COF_i} is empty. Costant 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}.

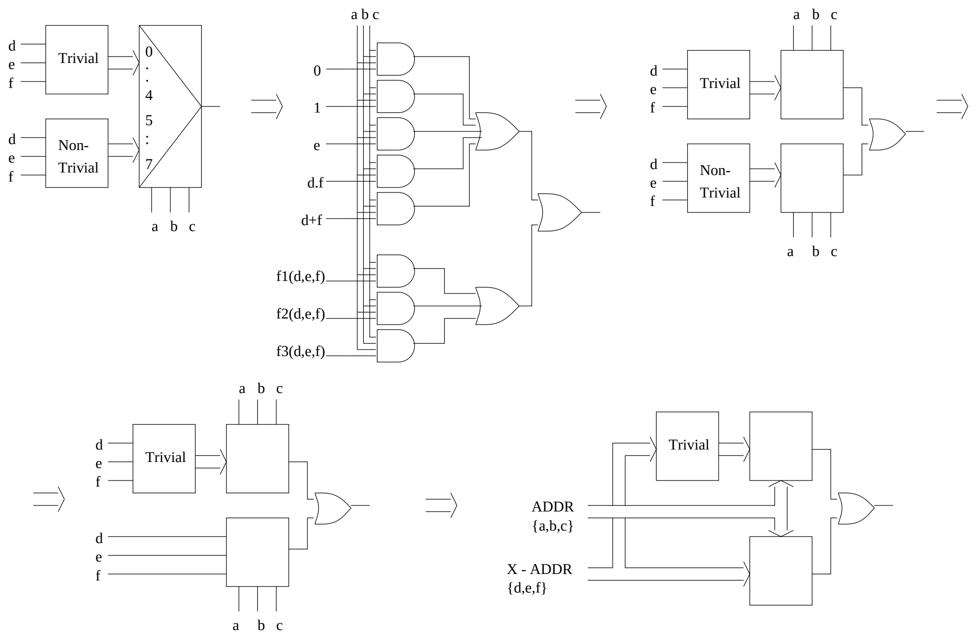


Figure 28:

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 extended to take into account not only the original input variables, but also any intermediate variables created in the decomposition process and the output variables (i.e. output functions).

Figure 28 presents different views at the Disjoint PUB Decomposition.

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 7.1* and define bound (B) and free (A) sets as being: $B = \{a, b, e\}, A = \{c, d, e\}$. Function's K-map is shown in Fig. 29 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 (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. 30. We need only two colors to color the graph which means that the function may be decomposable. It is decomposable if functions F_1 and F_2 coresponding to color 1 and color 2 meet the following condition: F_1 is compatible to $\overline{F}_2$. Let us now determine F_1 and F_2. Merging all the loop functions of the same color we obtain new functions (specified by ON and OFF sets):

AA:
ON(xxx000) ∪ ON(xxx110) ∪ ON(xxx101) = {100xxx, 011xxx}
OFF(xxx000) ∪ OFF(xxx110) ∪ OFF(xxx101) = {0x0xxx, 110xxx, 001xxx, 101xxx}

BB:
ON(xxx001) ∪ ON(xxx111) ∪ ON(xxx100) = {111xxx, 001xxx, 110xxx}
OFF(xxx001) ∪ OFF(xxx111) ∪ OFF(xxx100) = {011xxx, 101xxx, 100xxx}

K-map coresponding to the new functions is shown in Fig. 31. Functions F_1 and F_2 coresponding to colors AA and BB are: $F_1 = a,\overline{b},\overline{e} + \overline{a},b,e = (a \oplus b)(a \oplus c)(\overline{b \oplus e})$, $F_2 = a,b + \overline{a},\overline{b} = \overline{a \oplus b}$. The condition F_1 compatible to $\overline{F_2}$ is not met in our case so the function is not decomposable. Realization of our function using multiplexer is shown in Fig. 32.

Fig. 33 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		-	1			1	0	
011		0	-			-	1	
010	-			0	0			-
110	0			0	-			1
111		1	1			-	-	
101		-	-			0	0	
100	1			-	1			0

Figure 29:

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:

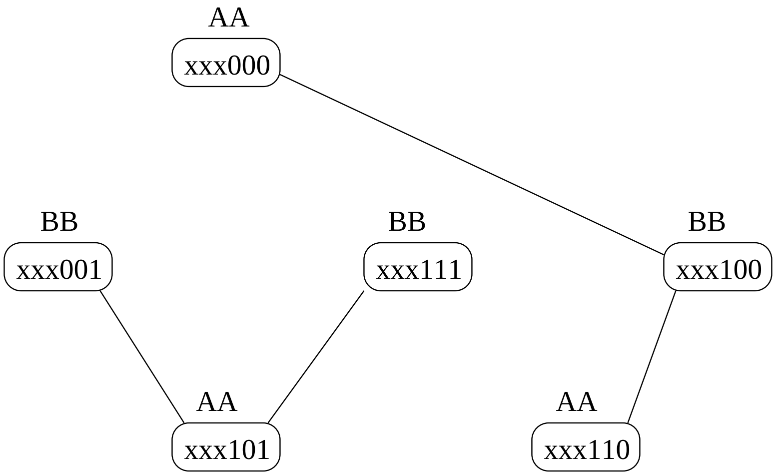


Figure 30: Incompatibility graph

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

AA

BB

Figure 31:

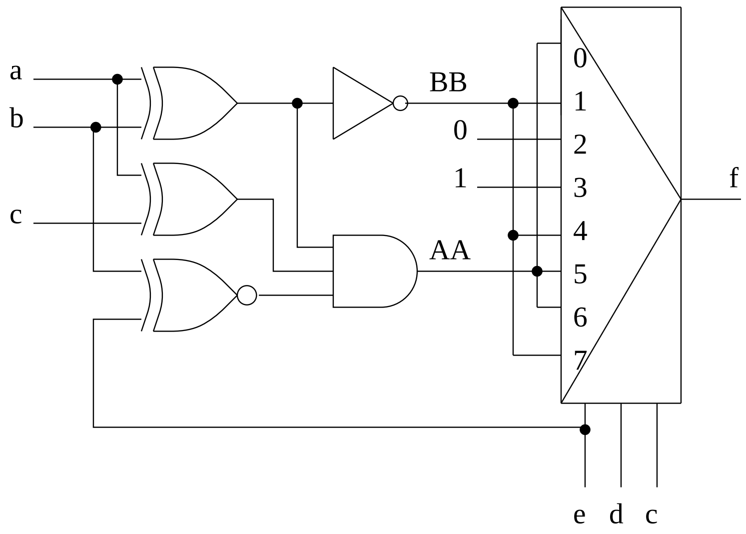


Figure 32:

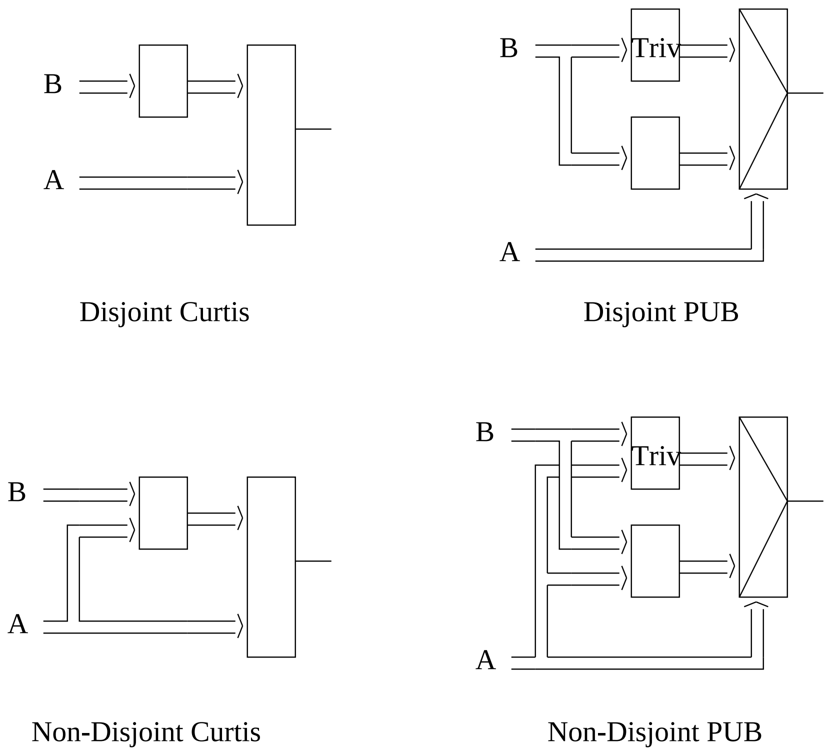


Figure 33:

10 Curtis Decomposition

10.1 Disjoint Curtis Decomposition.

A decomposition approach based on Graph Coloring, very similar to the one from section 7, can be applied to Curtis Decomposition. It will not be repeated it here, since it was presented in detail in [?, ?]. Instead of finding cofactors for free variables, one finds cofactors for bond 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.

Fig. ?? 8.1 shows the stages of decomposing function from example 9.1, solved using the Curtis Decomposition.

Fig. 8.1.

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

For free sets in PUB Decomposition we had:

- 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.

For bond sets in Curtis Decomposition we have:

- Bond Set Compatibility Graph - optimized by clique partitioning, or clique partitioning of care columns with similarity constraints.
- Bond 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 (bond variables in case of Curtis) is shown in Figure 34. Indices μ denote values of column multiplicity indices for bond sets for Curtis Decomposition. Indices ν denote values of row multiplicity indices for free (address) sets for PUB Decompositions.

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

10.2 Non-disjoint Curtis Decomposition.

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

Example 10.1 Let us consider the function from *Example 7.2* where bound (B) and free set (A) are defined as follows:

$A = \{a, b, e\}$, $B = \{c, d, e\}$

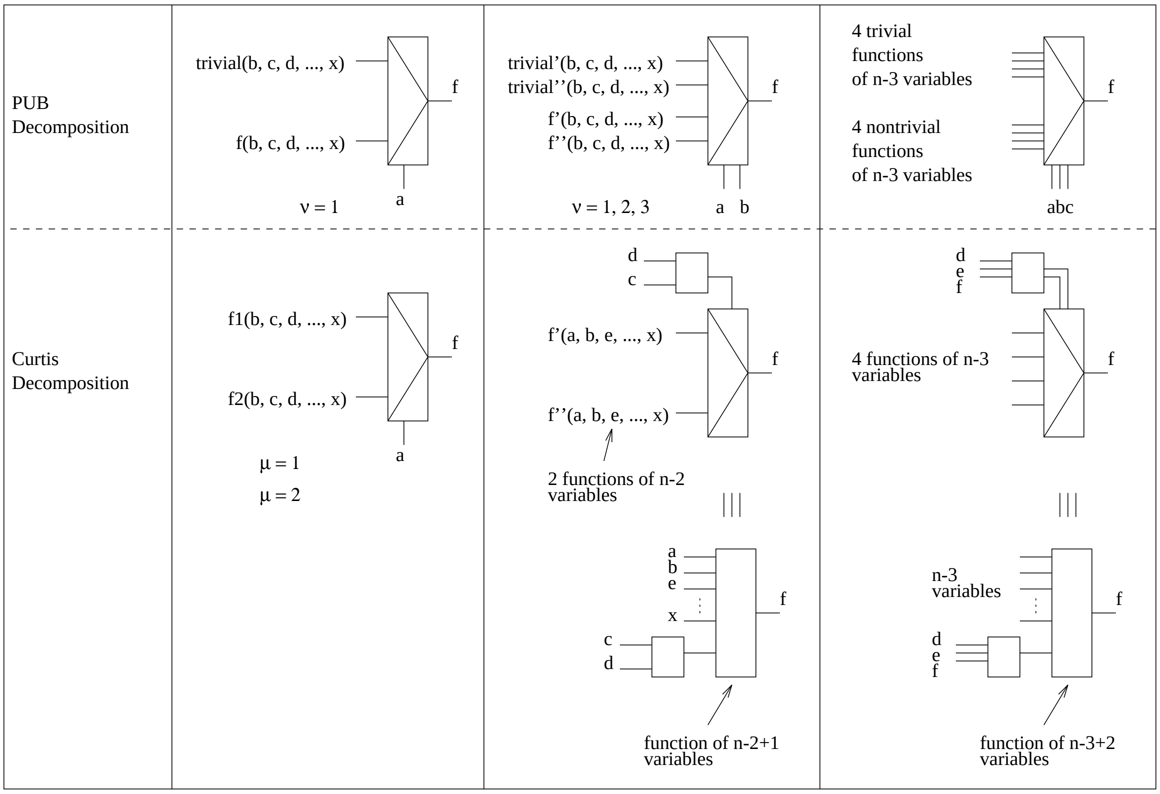


Figure 34: Comparison of PUB and Curtis decompositions for small k

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

a)

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

b)

Figure 35:

K-map of this function is repeated in Fig. 35a. Column multiplicity (CM) for this K-map is equal to 2 ({000,001,011,010,110,111 ; 101,100}), so the function is Ashenhurst decomposable (special case of Curtis decomposition). K-map obtained by filling DCs with 0s and 1s to get CM = 2 is shown in Fig. 35b. Loops 000, 010 and 101 contain only 0s and loop 111 contain only 1s. If we define the loop function corresponding to $a = 0$, $b = 0$, $e = 1$ as G we will see from Fig. 35b that any other loop function (different from 1s or 0s) is either equal to G or to $\overline{G}$.

11 Basic Decompositions.

The Basic Decompositions of the GUD program are:

- *Curtis Decomposition* (above section),
- *PUB Decomposition* [156].

They happen more often than the Immediate Decompositions presented in section 4. Hower, when executed with large value of multiplicity index μ they lead to difficult encoding problems and not necessarily minimum circuits (especially that we are never able to perform exhaustive search of sets A, B and C). We execute these decompositions only with small values of the multiplicity index μ = 3,...8.

11.1 Curtis Decompositions.

Only Curtis Decomposition will be discussed in this section. We have three variants of the algorithm, called: $Curtis_{color}$, $Curtis_{clique}$, and $Curtis_{concurrent}$, respectively.

Following the approach from [?], given the bond set $B \cup C$ and the free set $A \cup C$, the stages of the $Curtis_{color}$ decomposition are the following:

1. Find a fast approximate *graph coloring* of the Incompatibility Graph.
2. Use the encoding method that is similar to the input encoding algorithm for function H from [?].

   This algorithm is designed to minimize only the logic of function H.
3. Find functions G and H.

The reason to use here graph coloring instead of Clique Partitioning (as in literature) is to dramatically decrease the size of the memory. Let us assume that the N columns of the covering table correspond to the columns of the RVM, and the rows of the covering table correspond to the maximal cliques. Thus, for strongly unspecified functions, the number of rows is exponential, while the graph has only N * (N-1) / 2 edges.

Well known is a set covering algorithm that makes use of *essential rows, secondary essential rows* and *dominations of rows and columns* [149]. In case of non-cyclic covering tables, this algorithm finds the exact solution without backtracking. We created a similar algorithm for graph coloring.

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

Example 3. We will illustrate how to convert the covering problem to the coloring problem using a Kmap of a non-cyclic function f, Fig. 11.1a. Numbers denote the true minterms. All other cells are false minterms. Obviously in this case, after sharping 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 Fig. 11.1b. The stages of coloring the graph are shown in Fig. 11.1b - 11.1e. In Fig. 11.1b 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

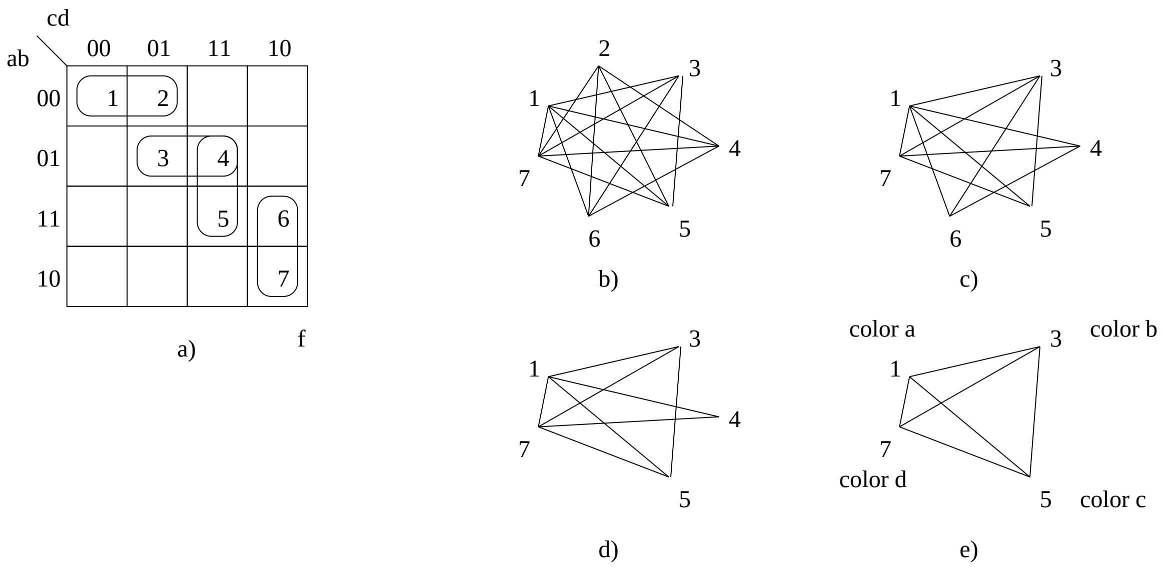


Figure 3. Stages of exact graph coloring

the graph from Fig. 11.1c. 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 Fig. 11.1d. 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, Fig. 11.1e, 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 [173]. 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 9.2.

In the second variant of Curtis Decomposition, $Curtis_{clique}$, given the bond 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 [153]. 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 bond 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-.

The third variant, $Curtis_{concurrent}$, assumes the (overlapping) bond and free sets as well. It uses the method that is similar to the concurrent state minimization and state assignment for state machines from [175]. It is based on the analogy that states of the machine can be compared to columns in the decomposition RVM. This way, concurrent state minimization and state assignment for a state machine is translated to a 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.

12 Goal-Oriented Reduction Decompositions.

These decompositions are used in two cases:

1. When some already realized function F_G is close to the function F to be decomposed (we mean by this a high correlation of variables F_G and F in CDB). This function is then treated as a goal function F_G.
2. When some of the previously attempted (Immediate or Basic) decompositions of F was "nearly possible", which means, many columns had patterns corresponding to the given type of decomposition. In such case, the method from [?] is used to create the goal function F_G and select the Reduction_Type_Operator.

In both above cases, the procedure Reduction is next called, to execute the reduction, possibly also the decomposition, and put all newly created subfunctions to stack $OPEN$ of functions to realize.

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 ON(F_C) := ON(F) $\oplus$ OFF(F_G) ; OFF(F_C) := OFF(F).

   If Reduction_Type_Operator = 'AND'

   then ON(F_C) := ON(F) ; OFF(F_C) := ON(F_G) $\oplus$ OFF(F).

   If Reduction_Type_Operator = 'EXOR'

   then ON(F_C) := ON(F_G) $\wedge$ OFF(F) $\vee$ OFF(F_G) $\wedge$ ON(F) ;

   OFF(F_C) := ON(F_G) $\wedge$ ON(F) $\vee$ OFF(F_G) $\wedge$ 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.

13 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. Other weak decompositions include Weak PUB Decompositions from [147, 148, 156].

Although these decompositions occur quite often, there exist still functions, for which none of these weak decompositions exists. (The weak EXOR decomposition from [?] exists for all functions, but we do not use it).

The few-level-circuits synthesis methods include, besides the $EXOR^2\_SOP$ circuit synthesis method discussed in [?], 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 [161, 147, 148, 156]. 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.

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

Decompositions of Multiple-Valued Functions have been discussed by:

- Thelliez [?] (only for ternary logic).
- Walliuzzaman and Vranesic [27].
- Fang and Wojcik [?].
- Luba, Mochocki and Rybnik [125, 123, 122]
- Abugharrbieh and Lee [?] [?]

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

One approach to decomposition of multiple-valued logic functions is to extend the AC binary approach. This approach is represented by Walliuzzaman and Vranesic, [27], and Thelliez, [?], for ternary logic. Especially the partition-based approach of Luba allows him to create an algorithm that is a straightforward adaptation of his former binary algorithms. Another approach is represented by Fang and Wojcik, [?]. Their paper is a very good reference for some non-FD approach to Boolean Decomposition, but it is not of an interest to this paper, because it is so different in principle.

Papers by Luba et al on decomposing multiple-valued functions are connected to "machine learning" [?, ?]. On one hand, their approach to decomposition is similar to both classical Curtis, and Walliuzzaman's or Abugharrbieh's and Lee's approaches to mv decomposition. The difference is that they apply the partition theory. The sufficient condition in this approach leads to the calculation of minimal cover of the maximum compatible classes. Some new results are obtained by using the method described by Selvaraj in his Ph.D. Thesis [162]. 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, [47, 82, 98]. These problems have been investigated from a number points of view: one of them is whether the whole set of attributes is always necessary to define a given partition of an universe and the other concerns the simplification of decision tables, namely the reduction of condition attributes in a decision table.

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

From the technical point of view, the decomposition phase 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 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.

15 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 15.1 *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 (0 or 1). The value - (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 15.2 *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 it's output variable j can take and

Definition 15.3 *A Generalized Multiple-Valued input, Multiple-Valued output function with n input, m output variables is defined as a mapping:*

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

For the sake of clarity truth tables may be viewed as 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* [100].

Definition 15.4 *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 [?, 144]. 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 15.5 *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.

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 [19], 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 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 9.1.

Consider a multiple-valued function F given in Table 10.1a.

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 :

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

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

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 10.1b, 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 15.6 *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)*

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

Table 3: Another Truth Table

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.

Let us do one more example (table 3).

Some of the indiscernibility relations for input variables are:

The partition of input variable X_0 is the following:

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

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:

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

Figure 36:

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.

16 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.

For example:

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

Use the following encoding method:

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

we obtain the Karnaugh maps as follows:
The corresponding function is: $P_1(X_0) = \bar{b} + \bar{a}\bar{c}$.

The corresponding function is: $P_2(X_0) = a + b$

Their respective BDDs are as follows:

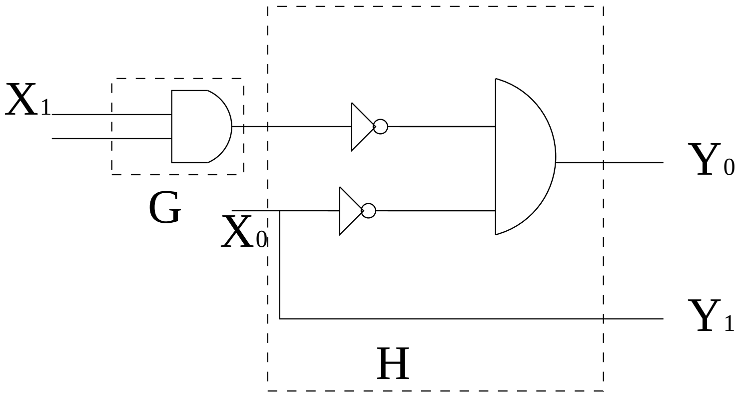


Figure 37: Decomposition of f

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

Figure 38:

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

Figure 39: Karnaugh map for $P_1(X_0)$

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

Figure 40: Karnaugh map for $P_2(X_0)$

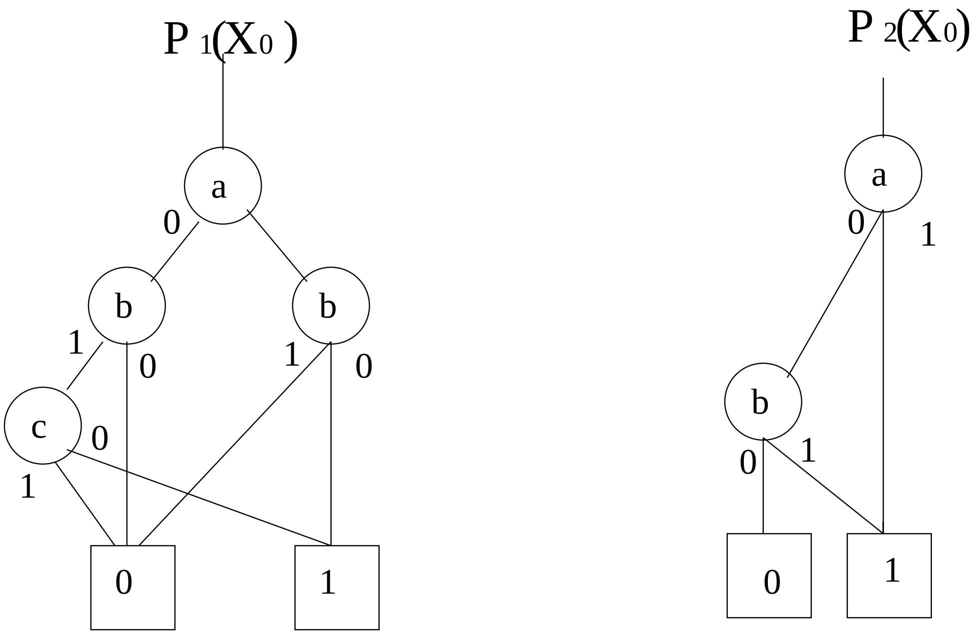


Figure 41:

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.

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.
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.

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 $B_1 \sharp B_2 = \oslash <=> B_1 \bigcap \bar{B}_2 = \oslash$. With this method, it is easy to check if one set is less than or equal to another because we know BDD package can do AND operation, and we can

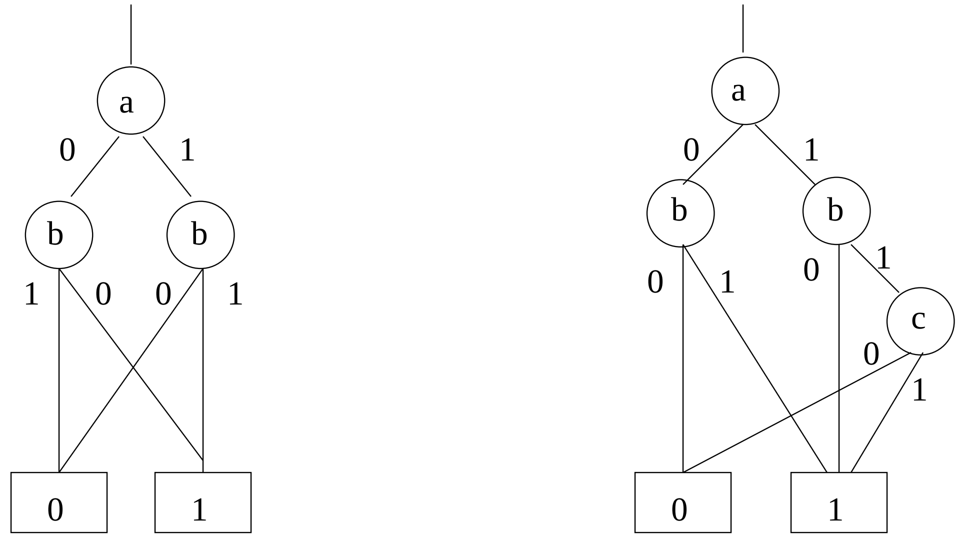


Figure 42: $\Pi_1 = \bar{a}\bar{b} + ab$, $\Pi_2 = a\bar{b} + bc + \bar{a}b$

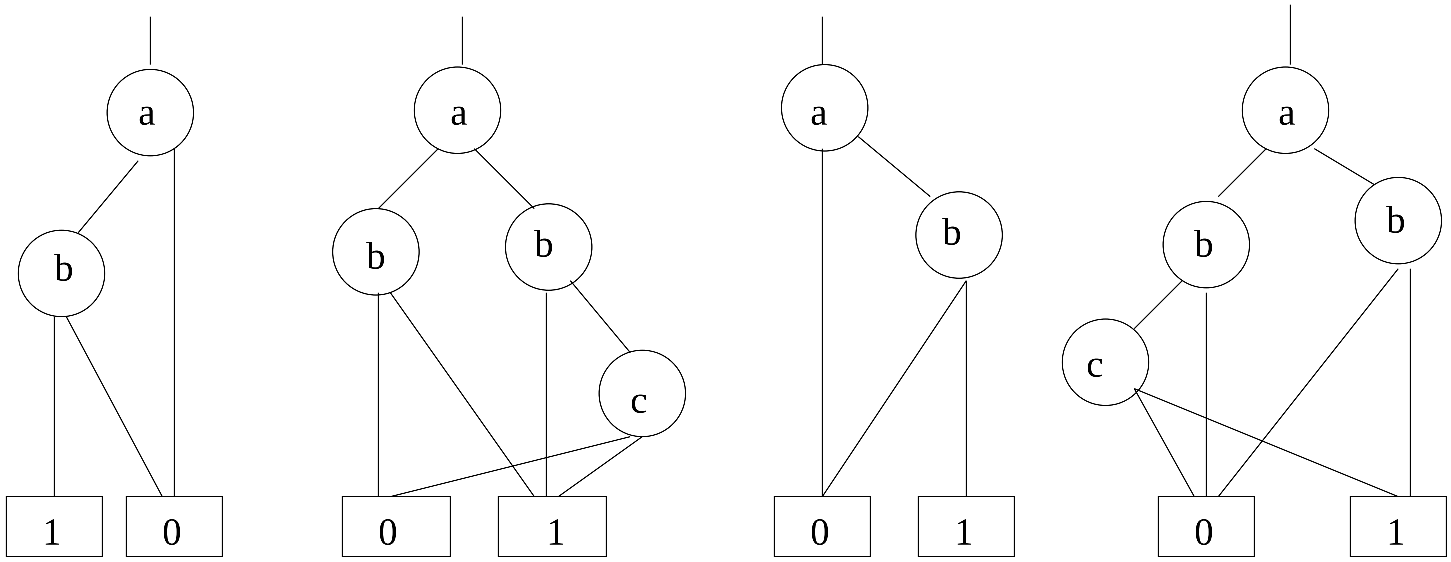


Figure 43: $P_1\Pi_1 = \bar{a}\bar{b}$, $P_2\Pi_2 = a\bar{b} + bc + \bar{a}b$, $P_2\Pi_1 = ab$, $P_1\Pi_2 = a\bar{b} + \bar{a}b\bar{c}$

realize inverter just by exchanging the position of leaf node 0 and 1.

17 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 $C \subseteq A$.

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 contrary to the method proposed by Ciesielski [96], circuits implementing components G and H are, in general, 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 [48], 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 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 [169] states the sufficient condition for the existence of a serial decomposition.

Theorem 17.1 *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 $[\log_2 L(\Pi_G)]$, here $L(\Pi)$ denotes the number of blocks of partition Π, and $[x]$ denotes the smallest integer equal to or large than x.

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

It is evident that partition Π_G represents component G, and the product of partitions $P(A)$ and Π_G corresponds to H. The truth tables of the resulting components can thus be obtained from these partitions.

Example 17.1 Let us decompose the function F of Table 10.1. For $A = \{x_1, x_6\}$, $B = \{x_2, x_3, x_4, x_5\}$, $C = \emptyset$, we have

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

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

Consider

$$Pi_G = (1, 2, 3, 5, 6, 8, 10; 4, 7, 9).$$

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

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 10.2. 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 10.3.

Example 17.2 For another example, in Table 10.1, 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 bondset $B = \{X_1, X_2\}$ and free set $A = \{X_0\}$. The number of output signal from G is $\lceil \log_2 2 \rceil = 1$.

The questions are:

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

Definition 17.1 *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 17.2 *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 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.

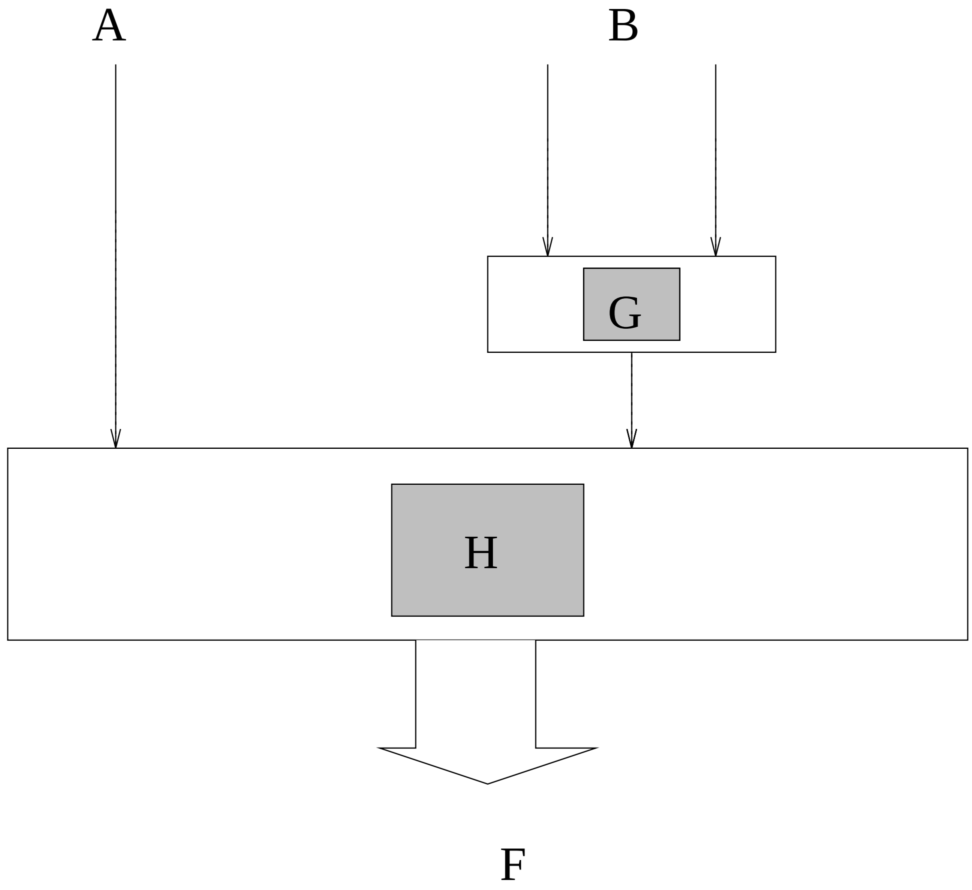


Figure 44: Decomposition Block Diagram of Function f

	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 4:

Definition 17.3 *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 $P(A) \cdot P_{ij}(B \cup C) \leq P_F$.*

For example, in Table 10.1, we have:

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

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 $P(B)$, 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 function G and H from Table 10.2. $g = X_1, Y_0 = X_1 X_0, Y_1 = X_0$.The circuit of this function is shown in Fig 11.3.

18 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 [83, 142].

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 1 for a case of decision tables can be stated as follows:

Theorem 18.1 *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 Fig. 12.1. The procedure of making a final decision is as follows: an intermediate decision is made on the basis of the attributes' subset B and then taking into consideration both the intermediate decision and the attributes subset A, the goal decision is made, which is equal to the corresponding value of the function F.

In the theorem the partition Π_G represents mapping function G, and the product of partitions $P(A)$ and Π_G corresponds to H. The component decision tables of the resulting mappings can be obtained from these partitions.

The decomposition gain comes from the fact that in some cases the table representation size of the function generated using decomposition will be much smaller than the representation of a single unified function.

The same example can be used to explain how 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 12.1.).

Example 18.1 Let us decompose the function F of Table 5 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 12.2.. The table of function H can be derived by reencoding input vectors of F using an intermediate variable g. The table obtained in this way is shown in Table 12.3.

Further processing of such tables is presented in [169].

19 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 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 mutual complements. It means, the sets of *edges* of these graphs are disjoint and the union of the sets of edges creates a full graph.

In Figure 13 , the number in each node (denoted by a circle) is the column number. The letter beside the circle is the color assigned to the node (column) after graph coloring. 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. 17.2.

19.1 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. 17.2.
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).

20 The Set Covering Approach to the Column Compatibility Problem.

20.1 The compatibility relation.

As it is seen from Theorem 10.1 (former 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 final function F , i.e. to find $P(B)$, such that there exists $\Pi_G \geq P(B)$ that satisfies condition (????4) in Theorem 10.1 (former 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.

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

Definition 20.1 *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_j satisfies condition (??4) in Theorem* **??**, *i.e., if and only if:*

$$P(A) \cdot \Pi_{G'} \leq P_D \tag{13}$$

Definition 20.2 *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 20.3 *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 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 an one element set B_j.

Example 20.1 For the function of Example 1????, and the same sets A, B as in Example 2????, let us denote the blocks of $P(B)$ as B_1,...,B_8 respectively, i.e. $P(B) = \{B_1, ..., B_8\}$, where $B_1 = \{1\}$, $B_2 = \{2,8\}$, $B_3 = \{3\}$, $B_4 = \{4\}$, $B_5 = \{5\}$, $B_6 = \{6,10\}$, $B_7 = \{7\}$, $B_8 = \{9\}$.

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

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

As

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

does not satisfy

$$P(A) \cdot P_{24}(B) \le P_F \; (for\ P_F,\ see\ Example\ 10.1),$$

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

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

and

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

Thus, B_2 and B_5 are compatible. In a similar way we check the compatibility of each pair of blocks in $P(B)$ 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 14.1. 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 corresponding MCCs (for the sake of simplicity we write S_j sets and CC-lists by indexes of B_j):

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

As the last C-list represents 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 $P(A) \cdot \Pi \leq P_D$, then the partition Π_G satisfying the same inequality i.e. $P(A) \cdot \Pi_G \leq P_D$ 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 14.2.

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 [?, ?], 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 20.2 For the MCCs from Example 14.1. (former4), one of the minimal disjoint covers (clique partitions) is MCC_7, MCC_2 - $\{B_6\}$ = $\{\{B_1, B_2, B_3, B_5, B_6\}, \{B_4, B_7, B_8\}\}$, and the corresponding Π_G is Π_G = {1,2,3,5,6,8,10 ; 4,7,9}, as it was assumed in Example 1?????.

21 Graph Coloring Approach to the Column Compatibility Problem.

The graph coloring approach 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. *Graph coloring*, [?, ?, ?, ?] is one in which every two nodes linked by an edge are assigned different colors. *Minimum graph coloring* is one with the minimum number of colors. The number of colors in the minimum coloring is called the chromatic number of the graph.

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 presents here is a fast graph coloring method. This method has been programmed and tested on many examples, it resulted in excellent colorings. The method found exact colorings for graphs in [?] and even found better coloring than one claimed to be the minimal in the 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 the color assignment to a node will produce a minimum increase of color-in-bar's. The *color-in-bar's* (restrictions) are the colors that the node cannot be assigned with, which are denoted by $\overline{A}$, $\overline{B}$..., $\overline{A}\ \overline{B}$, $\overline{A}\ \overline{C}$, ... as in Figure 24. After each color assignment to a node, the complexity of the graph is decreased. This is a greedy method with global evaluation. The next example is used to illustrate this method.

Figure 24a shows a graph that needs to be colored. Start from the node with the most number of edges, that is node 2, assign color A to it. This color assignment results in that nodes 1, 3, 5 and 6 cannot be assigned with color A, denote this restriction on those nodes by color-in-bar $\overline{A}$'s, and remove all corresponding edges as shown in Figure 24b. Then, color the node with the most number of color-in-bar's. If there are more than one node with the same number of color-in-bar's, chose the node with the most number of edges. If there are still more than one node, evaluate the influence of each color assignment, and assign the node with a color which results in a minimum influence. If a node can be assigned with more than one color, the evaluation of the influence of each color assignment is also required. According to the rules stated above, nodes 5 and 6 are selected because they have the same number of color-in-bar's and the same number of edges. Assigning color B to node 6 will result in a restriction $\overline{A}\ \overline{B}$ on node 1 and a restriction $\overline{A}\ \overline{B}$ on node 5. While assigning color B to node 5 will result in a restriction $\overline{A}\ \overline{B}$ on node 6 and a restriction $\overline{B}$ on node 4. Because one $\overline{A}\ \overline{B}$ restriction and one $\overline{B}$ restriction result in less influence than two $\overline{A}\ \overline{B}$ restrictions, assigning a color to node 5 produces less influence than assigning a color to node 6. Node 5 is selected, color B is assigned to it, as shown in Figure 21(c). The same way, color C is assigned to node 6, as shown in Figure 21(d), and color B to node 1, as shown in Figure 24e. 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 24f.

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

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

N is the number of nodes in the graph. The edge percentage (P) is simply the percentage of 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 $T \;\; N^{2.5}$.

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 23.

22 AND,OR,EXOR Decompositions.

In this section we will present our implementation of the "gate-type" decomposition approach of Steinbach, Bochmann, Posthoff and Dressing. This is a "gate-type" decomposition - 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 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.

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 re-formulating these operators for other types of decompositions: multiple-valued, fuzzy, and other logic. The general decomposition method expressed by operators should remain the same.

The symbolic operators are: complementation, 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 that we call a *function bundle*.

Function bundle is a structure of:

- a *cube_list* of input variables, denoted *cli*.
- a *cube_list* of output variables, denoted *clo*.
- a list of ordered partitions for input variables, denoted BI,
- a list of ordered partitions for output variables, denoted BO. If BO has just one element, it is denoted F = (M0, M1).

For a multiple-valued variable X with p values, the *ordered partition* is a list of pointers to BDDs, starting from pointer to $BDD_{value_0}, ...BDD_{value_{p-1}}$.

22.1 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 22.1 For a given function:

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

Derivative:

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

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 are shown in Fig. 45.

Graphical illustration of operations derivative, minimum and maximum for ON and OFF cubes are shown in Fig. 2. The figure illustrates these operations with both Karnaugh maps and partitions.

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.

22.2 Grouping.

By a *grouping*, [?], we understand the dismantling 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 o, which uses a disjunctive, conjunctive or linear grouping, the decomposition uses gates OR, AND or EXOR, 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 x_i and for $h(X) \;=\; h(X \setminus x_i)$, the grouping degenerates into a simple splitting off of a variable. The weakest form of the grouping (the splitting off discussed earlier) exists if A and B contain only one variable each. A prerequisite for a grouping is that the function possesses the characteristic of groupability in at least ???-a type and after at least one variable partitioning-???. If this characteristic is present, we can usually form several different groupings.

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 disjunctive, conjunctive or linear groupabilities are formulated.

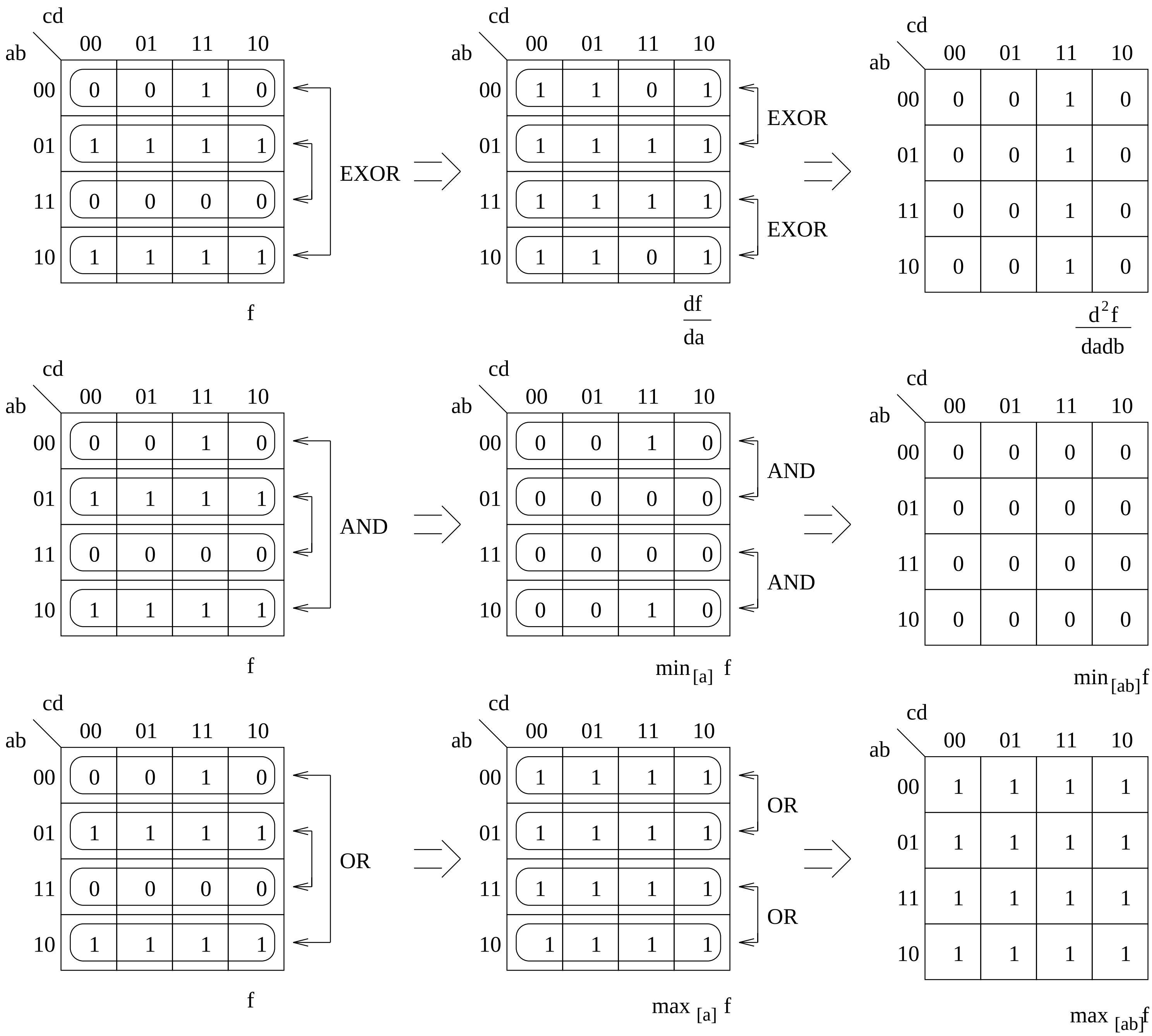

cd
ab
00 01 11 10
EXOR
AND
OR
f
df/da
d²f/dadb
min[a] f
min[ab]f
max [a] f
max [ab]f

Figure 45:

Definition 22.1 *A Boolean function $f(A,B,C)$ is said to be (disjunctively, conjunctively of linearly) groupable 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 groupable, it has been proven in [?] that all Boolean functions, which are not groupable by the EXOR gate decomposition, are always disjunctively or conjunctively weakly groupable.

Definition 22.2 *A Boolean function $f(A,B)$ is said to be* (disjunctively, conjunctively, or linearly) weakly groupable *with operation $\bigcirc \in$ { OR, AND, EXOR } and the variables set A if*

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

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 better balanced circuits.

A maximum size of A and B results in a smaller vector C, to make functions $g(A,C)$ and $h(B,C)$ being dependent on less variables.

As a criterion for the best grouping the smaller one of A and B should be as large as possible. If more than one grouping has the same number of variables in the smaller vector, the grouping with the largest vector is the best. In contrast to the area-optimizing algorithm where one vector is filled first, now the number of variables in vectors A and B is increased alternately.

Example 22.2 Let us look at the following example in Fig. 46, 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.

Graphical ilustration of min, max and derivative operations for this function is shown in Fig. 47b, c, d.

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 (24)

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

From all the compact groupings that exist for a binary function $f(X)$ we select the grouping that is the largest with respect to the half-order relation (24).

A weakly groupable function can be implemented by several weak groupings. For a disjunctively or conjunctively weakly groupable 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 altered.

The question "is an incompletely specified function with the functions $f(X)_{ON}$ and $f(X)_{OFF}$ for A and B groupable?", can be answered using the following existence theorems.

Theorem 22.1 *An incompletely specified function with functions $f(X)_{ON}$ and $f(X)_{OFF}$ for A and B is* disjunctively groupable, *if and only if*

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

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

f(a,b,c,d)

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

q(a,b,c,d)

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

r(a,b,c,d)

Example for Logic Decomposition

Figure 46: The Example of Care Functions

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 in the B direction.

Dual situations exist in conjunctively groupable functions.

Theorem 22.2 *An incompletely specified function with functions $f(X)_{ON}$ and $f(X)_{OFF}$ for A and B is* conjunctively groupable, *if and only if*

$$f(X)_{OFF} \;\cdot\; \max_{A}{}^{k} f(X)_{ON} \;\cdot\; \max_{B}{}^{k} f(X)_{ON} \;=\; 0 \tag{26}$$

The function $find\_A\_B$ determines strong nonlinear (disjunctive or conjunctive) groupability, if such a groupability exists. The conditions for the groupability can be described using care functions $f(X)_{ON}$ = M1 and $f(X)_{OFF}$ = M0, from bundle $FB(X, f)$.

```
Subroutine find_A_B

GIVEN:

Bundle of function $f$ FB(x, f).

Type:  ' d ' for disjunctive
       ' c ' for conjunctive grouping.

Function find_A_B(FB, type, A, B), returns the
value "true", if FB is groupable according to the
required strong "type" of grouping ( 'd' or 'c').
In this case the grouping sets A and B are returned.
```

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

f

a)

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

$\frac{df}{da}$

b)

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

$\min_a f$

c)

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

$\max_a f$

d)

Figure 47: Wrong figure !!!

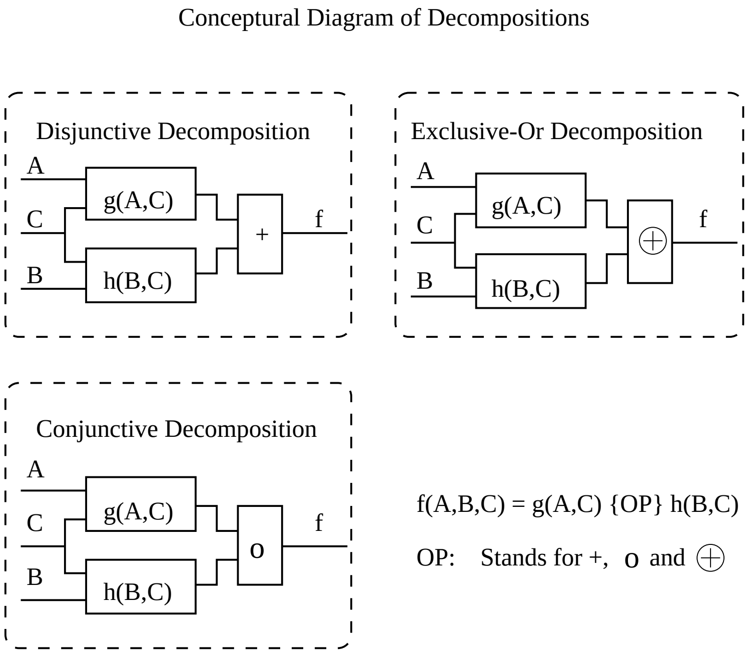


Figure 48: Strong Disjunctive, Conjunctive and Linear Decompositions.

```
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;
```

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

f_{ON}

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

f_{OFF}

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

$\max_{[ab]} f_{ON}$

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

$\max_{[cd]} f_{OFF}$

Figure 49:

```
    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 created here *)
  end;
end;
```

If groupability was determined, then the bundles of functions $g(A,C)$ and $h(B,C)$ are calculated. Different possibilities exist for grouping. For each of the three types of grouping we would like to find large functions.

The grouping bundles $FB(A,C,\{g\})$ with $g(A,C)_{ON}$ = M1g and $g(A,C)_{OFF}$ = M0g as well as $FB(B,C,\{h\})$ with $h(B,C)_{ON}$ = M1h and $h(B,C)_{OFF}$ = M0h can be calculated for the disjunctive grouping with the formulas (27) to (30), if $FB(x,\{f\})$ is disjunctively groupable 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 conjunctive grouping we obtain 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 disjunctive decomposition, next for conjunctive decomposition. It looks for compact groupability, and if the answer is "yes", it forms grouping bundles using (27) to (30) or (31) to (34).

```
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 grouping of the bundle with bdds M1 and
M0 is possible.  In this case the grouping bundles for
g(A, C) and h(B, C)
for the selected strong groupability 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' grouping *)
   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;
```

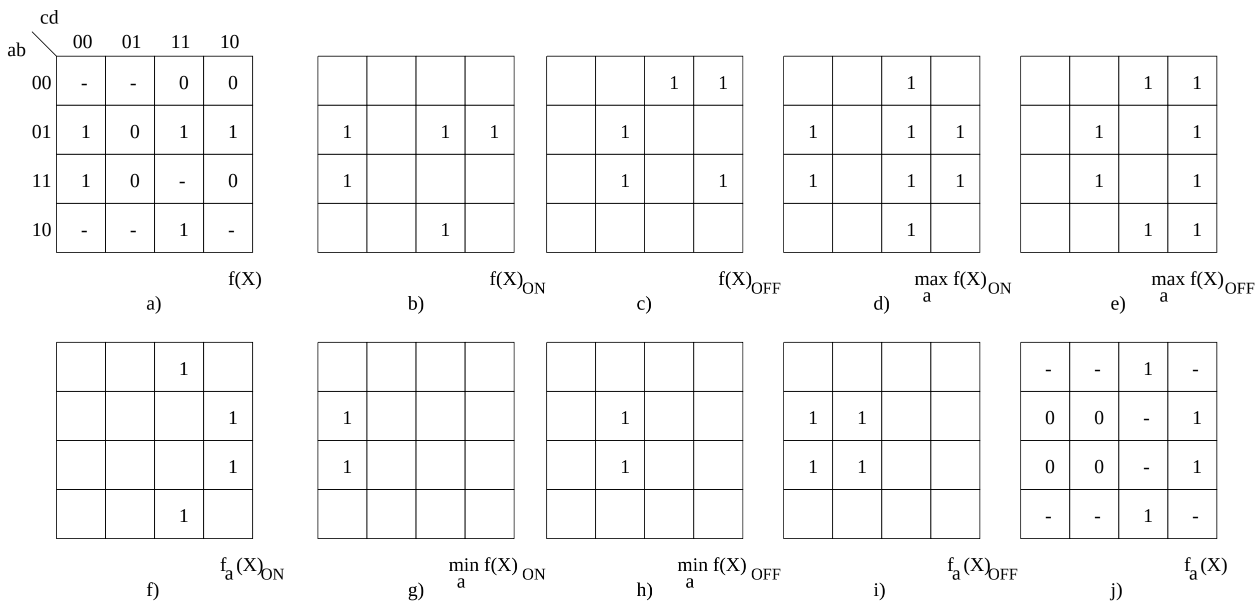


Figure 50:

22.3 Linear Groupability.

The linear groupability for sets A and B is determined step by step. First, the derivative of the incompletely specifed function for a single variable x_a is calculated using Lemma 15.1.

Lemma 22.1 *The derivative of an incompletely specified function $f(X)$ with respect to variable x_a is an incompletely specified function specified 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 22.3 Figure 50 illustrates calculation of the above formulas using Kmaps.

Figure 50f shows $f_a(X)_{ON} = \max_a f(X)_{ON} \cdot \max_a f(X)_{OFF}$, and Fig. 50i illustrates $f_a(X)_{OFF} = \min_a f(X)_{ON} + \min_a f(X)_{OFF}$. Finally, Fig. 50j shows the composite incompletely specified derivative which can be obtained 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 (35) assumes for all argument change combinations relative to variable x_a the value one. Values zero are created in the derivative with respect to variable x_a (36), if relative 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 described in the subroutine *derivative* presented below.

```
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;
```

Function $f(X)$ is linearly groupable 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.

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.

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

f(X)

a)

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

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

c)

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

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

c)

Figure 51:

Example 22.4

$$\frac{\partial^2 f(X)}{\partial a \partial c} = \frac{\partial^2 f(X)}{\partial a \partial d} = 0$$
$$\frac{\partial^2 f(X)}{\partial b \partial c} = \frac{\partial^2 f(X)}{\partial b \partial d} = 0$$

Function $f(X)$ is linearly goupable with respect to variable a and set of variables $\{c, d\} = B$. Therefore $f(X)$ is linearly groupable with respect to sets $A = \{a, b\}$ and $B = \{c, d\}$.

When such sets A and B are found, the linear groupability 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 22.5 Given is function from Fig. 52. $B = 00$ was selected (Fig. 52a) and respective $g(A) = f(a = 0, b = 0, c, d) = c \oplus \overline{d}$ is found, Fig. 52b. Next $h(B, c)$ is found (Fig. 52e) by EXOR-ing $f(X)$ (Fig. 52c) and $g(A)$ (Fig. 52d). Finally max with respect to set A is applied to remove vacuous variables (Fig. 52f).

However, below more efficient methods to test linear groupability and to find functions g and h will be given.

Theorem 22.3 *(Dresig 1988). An incompletely specified function with functions $f(X)_{ON}$ and $f(X)_{OFF}$ is linearly groupable 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 exclusive-or groupability 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 22.6 Figure 53 illustrates calculation of linear groupability according to Theorem 22.3.

Fig. 53 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 22.4.

Theorem 22.4 *The reduced incompletely specified function with the care functions:*

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

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

contains all Boolean functions of the incompletely specified function that are linearly groupable for (x_a, B).

Example 22.7 Figure 54 illustrates the usage of Theorem 22.4.

The question, whether a binary function is linearly groupable 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 linear groupability $f(A, B, C)$ as in Figure 14.1 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 linearly groupable with respect to variable x_a functions are contained in an 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.

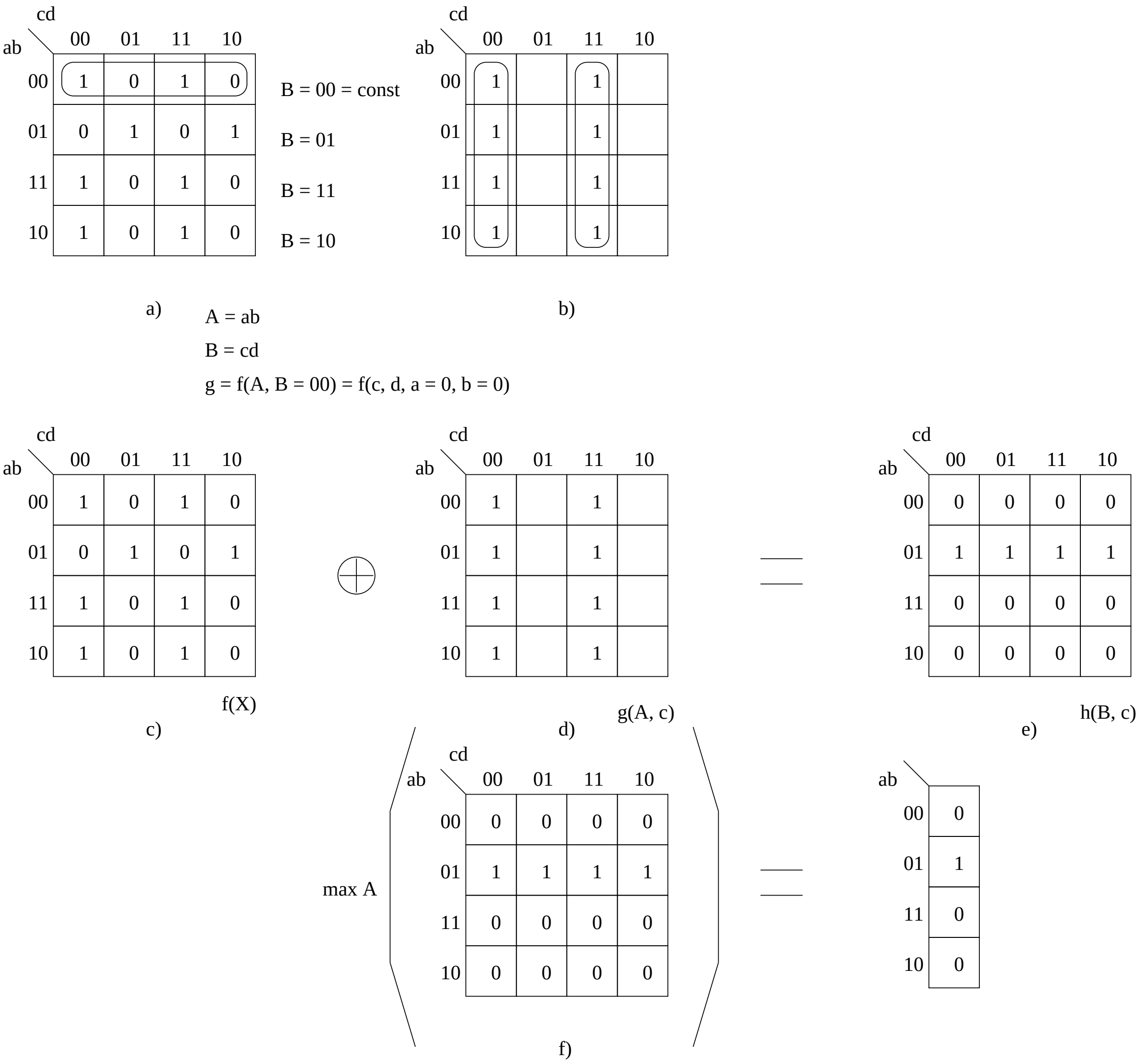

cd
ab
00 01 11 10
B = 00 = const
B = 01
B = 11
B = 10
a)
b)
A = ab
B = cd
g = f(A, B = 00) = f(c, d, a = 0, b = 0)
f(X)
c)
d)
g(A, c)
e)
h(B, c)
max A
f)

Figure 52:

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

$f(X)$

a)

0	-	1	1
-	-	1	1
-	-	1	1
0	-	1	1

$f_a(X)$

b)

		1	1
		1	1
		1	1
		1	1

$f_a(X)_{ON}$

c)

$f_a(X)_{OFF}$

d)

		1	1
		1	1
		1	1
		1	1

$\max_{\{c,\,d\}} f_a(X)_{ON}$

e)

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

$\max_{\{c,\,d\}} f_a(X)_{OFF}$

f)

(e) . (f) = 0

-	-	1	1
-	-	1	1
-	1	1	1
-	1	1	1

$f_b(X)$

j)

Figure 53:

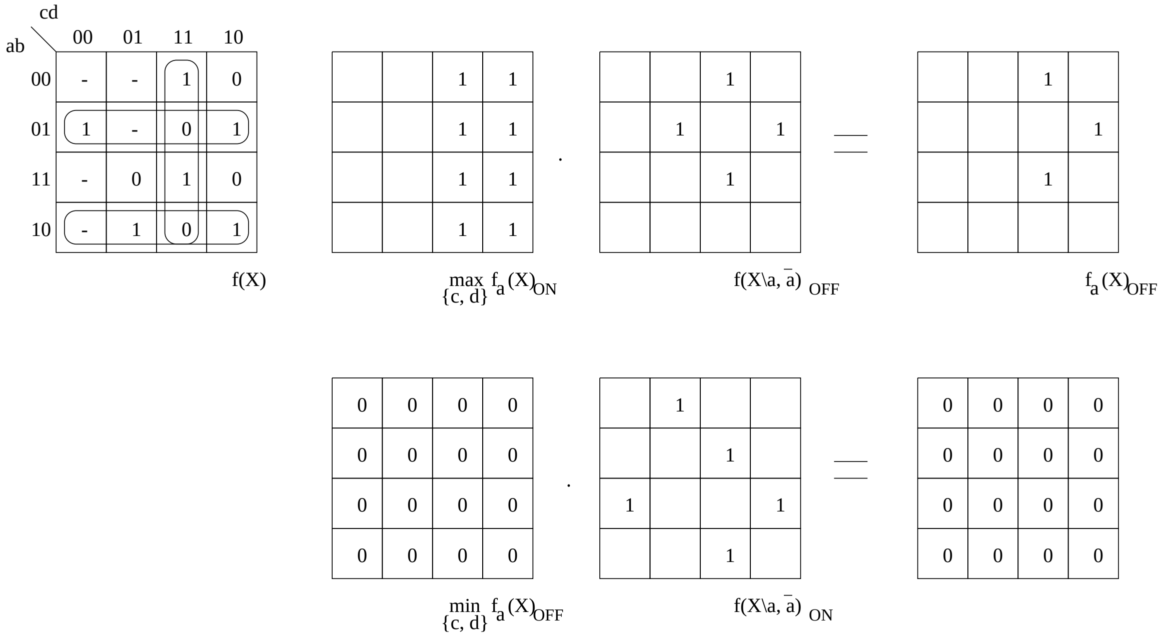


Figure 54:

If in the derivative with respect to variable x_a there exists a constant function in every cofactor of set B, then after derivative for B operation 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 (41) helps only to find a reduced groupability, we will use it now to create the solution method for the linear groupability for arbitrary variable sets A and B. Using (41) the linear groupability 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 linear groupability for sets B and A. That is because the individual linear groupabilities 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 (41) to additionally satisfy formulas (42) and (43) in cofactors, then linearly groupable functions are preserved relative to these A and B. This will be for most, but not for all such functions, so some linearly groupable functions with respect to 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 after linear groupable (42) and (43) to a cofactor function. In this cofactor, if the groupability exists for $x_j \in A$ and B, then the probability is substantially higher than for the original function that at least one linearly groupable function with respect to $(x_i, x_j \in A)$ and B.

```
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).

In analogy to the analysis of the nonlinear groupability with the subroutine *linear_grouping* a strong linear decomposition is found, if function f is linearly groupable. 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 linear groupability as efficiently as possible.

```
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
```

linearly groupable function in the bundle with M1 and M0.
In this case, sets A and B for strong
grouping 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 linear groupability there are very many different grouping functions h and g. Finding good functions h and g is difficult, since EXOR operation is involved here. During the analysis of the linear groupability we limited already the function, in order to obtain linear groupability for the individual A[i] in connection with B.

However, the reduced function can be, with a very low probability, not linearly groupable.

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 less ON terms.

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 grouping function $g(A,C)$ is finally found, when in (47) $u(A,C) = 0$ results. Otherwise the iterative process is continued again with $xb0$ selected according to (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 (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 (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.

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

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

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

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

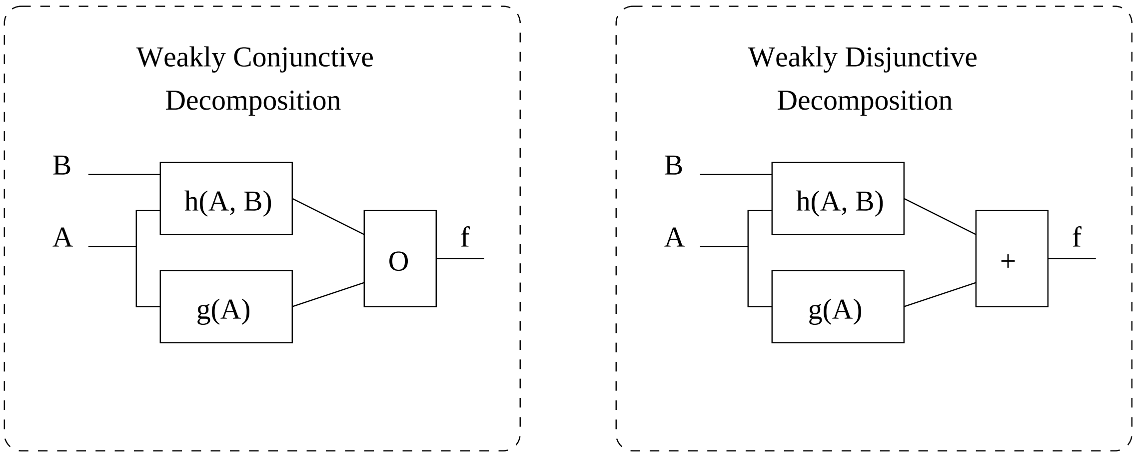
The subroutine *strong_linear* analyzes a given function $f(X)$ with respect to linear strong groupability. It forms a pair of grouping functions (g,h) using (44) to (53), if $f(X)$ is a linearly groupable function. The two functions $g(A,C)$ and $h(B,C)$ are generated with the identical lower procedure selection. A verification is recommended of functions g and h calculated by *strong_linear*, since it is possible that two groupings (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.

```
subroutine strong_linear

M1: bdd of the ON-set of 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 linear grouping of the bundle with M1 and M0 is
possible.  In this case the grouping functions for g(A, C) and h(B, C)
of a strong linear grouping 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;
```

23 Weak Decompositions.

A strongly groupable function is decomposed to two simpler subfunctions. Otherwise (e.g. with symmetrical functions) we must reiterate to other synthesis procedures. One of such procedures, called *weak decomposition*, is based on *weak grouping*, and is presented in Fig.55.
Figure 55a presents the *Conjunctive Weak Decomposition*, and Figure 55b presents the *Disjunctive Weak Decomposition.*



f(A, B) = h(A, B) {OP} g(A)
OP = O or +

Figure 55: Conjunctive and Disjunctive Weak Decompositions.

The weak grouping can be made disjunctively, conjunctively or linearly, with OR, AND or EXOR gates, respectively. A binary function is nonlinearly weakly groupable, if the respective criteria have been satisfied.

Every function has linear groupability, and can be decomposed respectively. Using the linear weak grouping we transform function $f(A, B)$ to an EXOR of function $g(A)$, selected by us, and function $h(A, B)$, found by EXORing $f(A, B)$ and $g(A)$.

Although every function is linearly weakly decomposable, this should be our last resort in decomposition strategies. The nonlinear disjunctive or conjunctive weak grouping always simplifies 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, weak decomposability is repeated.

In a function bundle $FB(X, \{f\})$ with the above care functions, at least one weak grouping relative to set B exists, disjunctive or conjunctive, if the following Theorem 16.1 or Theorem 16.2 are satisfied, respectively.

Theorem 23.1 *An incompletely specified function with care functions $f(X)_{ON}$ and $f(X)_{OFF}$ is disjunctively weakly groupable with set B if and only if:*

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

Example 23.1 A completely specified function $f(A, B)$ is presented in Fig. 56a. Let us assume that set A corresponds to rows and set B to columns. It is easy to find function $g(A)$ for disjunctive 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. 56b. Condition in Theorem 23.1 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. 56c, 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 disjunctive decomposition is strong, otherwise it is weak.

The final disjunctive decomposition of function from Fig. 56a is shown in Fig. 56d.

Concluding, finding disjunctive 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 23.2 The enumeration of minterms for partitions is shown in Fig. 57.

Using partitions, the above presented steps from Example 23.1 are executed as follows.

$$Pi_f = (0,1,3,8,9,10,11;2,4,5,6,7,12,13,14,15)$$

$$Pi(A) = (0,1,2,3;4,5,6,7;8,9,10,11;12,13,14,15)$$

$$Pi(A) \cdot Pi_f = ((0,1,3)(2);(0)(4,5,6,7);(8,9,10,11)(0);(0)(12,13,14,15))$$

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

$$h(A,B) = (0,1,3,8,9,10,11;(2,4,5,6,7,12,13,14,15)-(4,5,6,7)-(12,13,14,15)) = (0,1,3,8,9,10,11;2)$$

Conjunctive weak decomposition is dual to disjunctive weak decomposition.

Theorem 23.2 *An incompletely specified function with functions $f(X)_{ON}$ and $f(X)_{OFF}$ is* conjunctively weakly groupable *with set B, if and only if:*

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

Example 23.3 Let us assume that set A corresponds to rows and set B to columns. It is easy to find function $g(A)$ for conjunctive 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. 58. 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. 58. If it is a function of only variables from B the conjunctive decomposition is strong, otherwise it is weak.

The final disjunctive decomposition of function from Fig. 58a is shown in Fig. 58d.

Concluding, finding conjunctive 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.

Example 23.4 Using partitions, the above decomposition can be found as follows:

Fig. 16.5.

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 23.3 *Every Boolean function that is not disjunctively weakly groupable and not conjunctively weakly groupable has the property of exclusive-or groupability.*

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

f(A, B)

a)

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

g(A)

b)

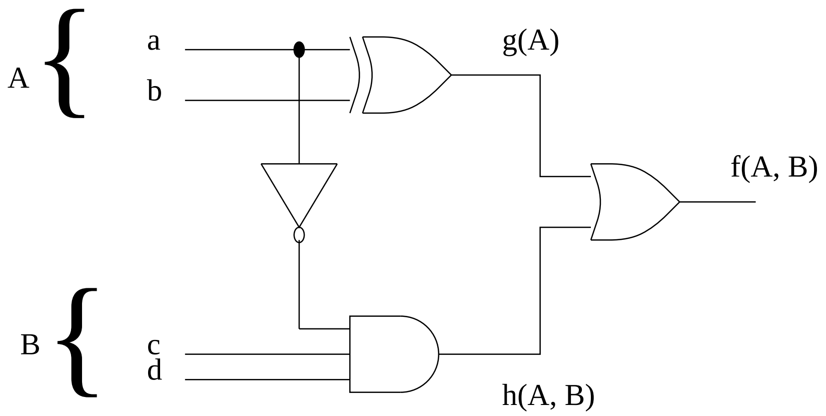
0	0	1	0
-	-	-	-
0	0	0	0
-	-	-	-

h(A, B)

c)

A = ab

B = cd



d)

Figure 56:

ab \ cd	00	01	11	10
00	0 (0)	0 (1)	1 (2)	0 (3)
01	1 (4)	1 (5)	1 (6)	1 (7)
11	0 (8)	0 (9)	0 (10)	0 (11)
10	1 (12)	1 (13)	1 (14)	1 (15)

f(A, B)

A = ab

B = cd

Figure 57:

If the variable set B cannot be further increased in (23.7) or (23.7), without these inequalities changing to equalities, then weak groupability exists. For each weak groupability there are several weak groupings. In the formulas (56) to (63) the indicated weak groupings were selected in such a way that as simple as possible function $h(A, B)$ is created.

Under the prerequisite that (23.7) is satisfied, the *Disjunctive Weak Decomposition*, see Fig. 55b, can be made into bundles $BF(A, \{g\})$ with $g(A)_{ON}$ = M1g and $g(A)_{OFF}$ = M0g as well as $BF(A, B, \{h\})$ with $h(A, B)_{ON}$ = M1h and $h(A, B)_{OFF}$ = M0h after the formulas (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 (23.7) is satisfied, then a *Conjunctive Weak Decomposition*, Fig. 55b, 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 bdd's of a bundle for disjunctive and conjunctive weak groupability, and forms (if possible) the weak grouping bundles using (56) to (59), or (60) to (63), respectively.

Application of these formulas to calculate disjunctive and conjunctive decompositions is shown in the next example.

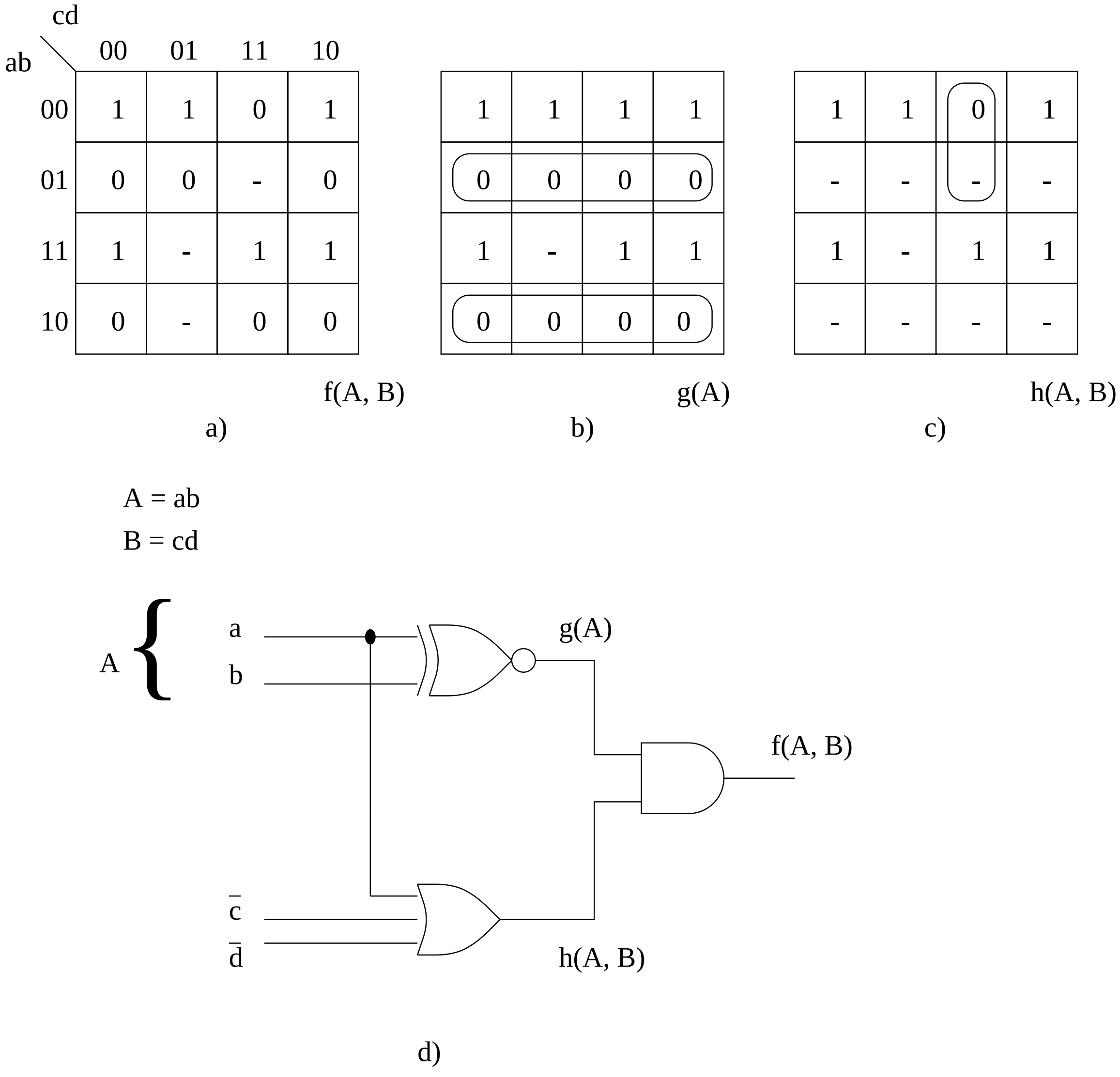


Figure 58:

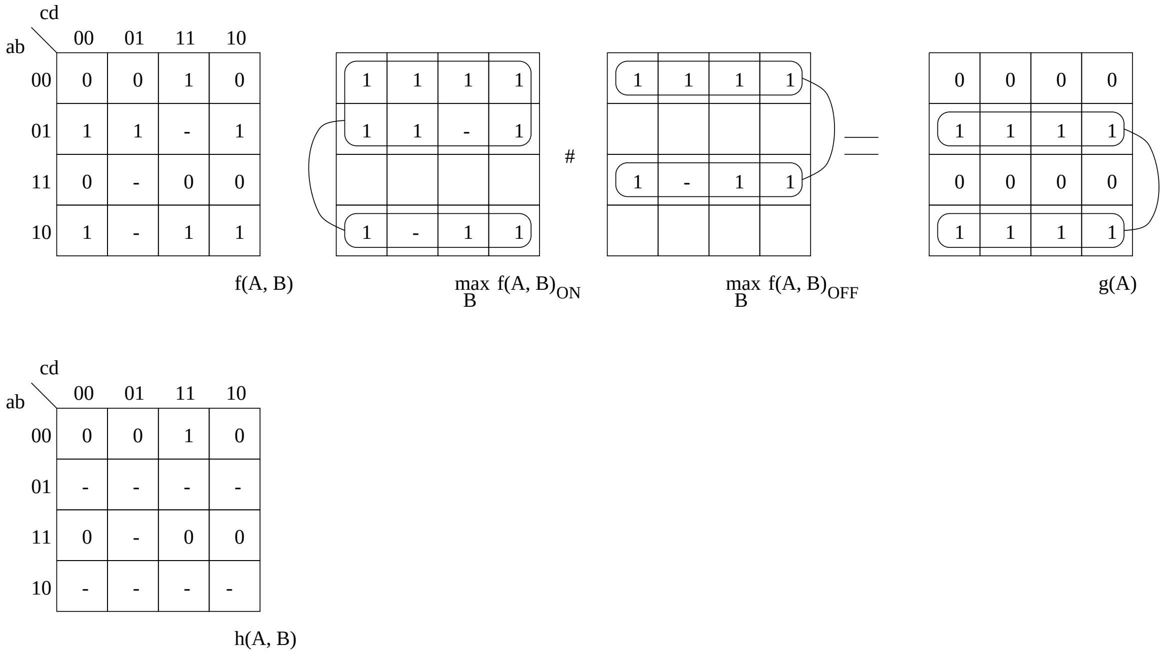


Figure 59:

Example 23.5 Figure 59 illustrates graphically using of the above formulas in weak disjunctive decompositions.

Example 23.6 Figure 60 illustrates graphically using of the above formulas in weak conjunctive decompositions.

Example 23.7 Figure 61 illustrates graphically the weak linear decomposition. Formulas and 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. 61a, and EXOR-ed from $F(A, B)$ as $g(A) = a\ b$, thus creating $h(A, B)$ from Fig. 61b. The final circuit is shown in Fig. 61c.

`Subroutine nonlinear_weak`

`GIVEN:`

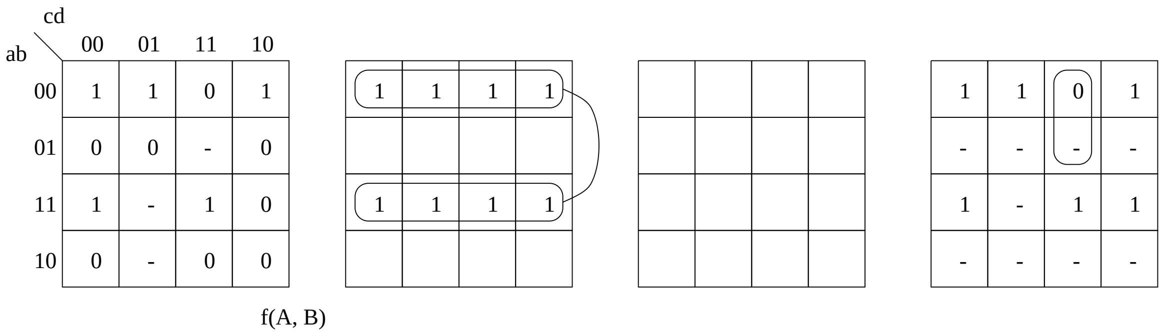


Figure 60:

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

f(A, B)

a)

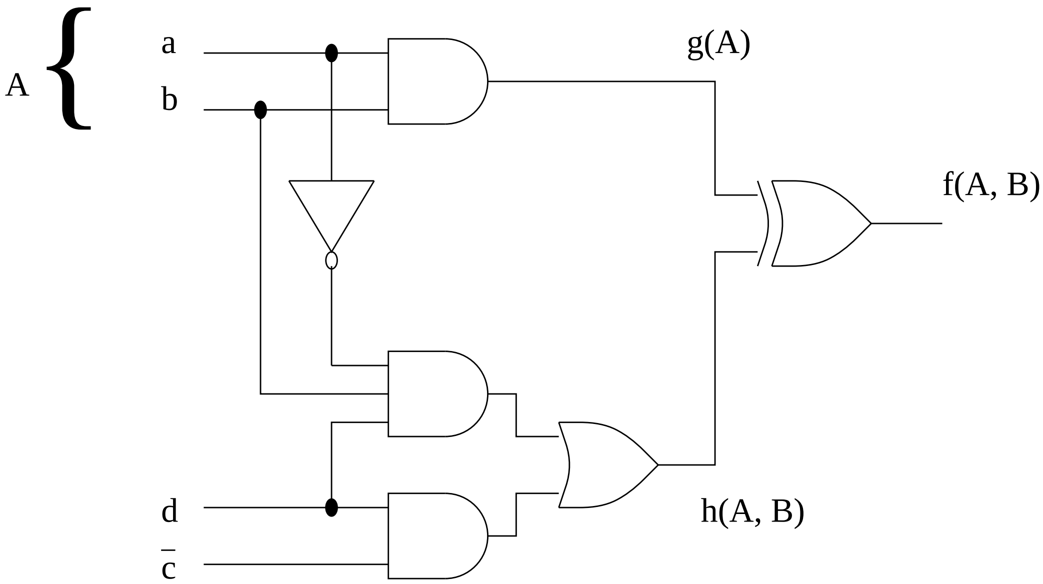
0	1	0	0
0	1	1	0
0	1	0	0
0	1	0	0

g(A)

b)

A = ab

B = cd



c)

Figure 61:

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 nonlinear
weak grouping of the bundle with M1 and M0
is possible. In this case the weak grouping
bundles for g(A) and h(A, B) are calculated
for the selected compact
groupability 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.

```
FUNCTIONS:

grp_test: tests the groupability.

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:

\begin{theorem}
%{\em Monotony Theorem.}
If an incompletely specified function includes no disjunctively
groupable function with respect to sets A and B then
there is no disjunctively groupable function for (A,xi) and B.
there is no disjunctively groupable function for A and (B,xi).
\end{theorem}

This is used in the above algorithm.

\begin{verbatim}

 Calculate initial groupability 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;
```

24 Towards a Unified Approach to Decomposition.

For the purposes of this research, we have created the following hierarchical classification of decompositions:

1. Decompositions that assume special types of blocks.
   (a) Dietmeyer's decomposition.
   (b) Wojcik's decomposition.
   (c) Davidson decomposition.
2. Decompositions that do not assume special types of blocks,
   (a) serial decompositions.
   (b) parallel decomposition.
   (c) generalized decomposition of Jozwiak.

Serial Decompositions are the following:

1. Non-Encoded Decompositions.
2. Encoded Decompositions - Curtis. (Until now only one type of encoded decompositions is known - Curtis).

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.

Strong Disjoint Ashenhurst Decompositions are the following:

1. Strong AND Disjoint,
2. Strong OR Disjoint,
3. Strong EXOR Disjoint,

Non-Disjoint Ashenhurst Decompositions are the following:

1. Strong Non-Disjoint Ashenhurst Decompositions,
2. General Non-Disjoint Ashenhurst Decompositions.

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.

PUB Decompositions are the following:

1. Disjoint PUB Decomposition.
2. Non-Disjoint PUB Decomposition.

Weak Decompositions are the following:

1. Weak AND Decomposition,
2. Weak OR Decomposition,
3. Weak EXOR Decomposition.

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.

Encoded Decompositions are the following:

1. Disjoint Curtis Decompositions.
2. Non-disjoint Curtis Decompositions.

Fig. 62 presents the above decompositions, Fig. 63 presents Curtis Decompositions, and Fig. 64 presents the Column-Multiplicity-Based Decompositions.
Three approaches will be discussed in more detail:

- 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 uses the well-known calculus of arrays of cubes (such calculus is presented for instance in [63, ?, 56]). Learning of this material will be useful for people who want to understand the uses of Cube Calculus in CAD algorithms. The use of Cube Calculus here illustrates also the indendence of the decomposition concepts from the representation of the data.
- The partition method will provide an approach to large functions, especially strongly unspecified functions, and respective efficient computer algorithms, [169]. The blocks of partitions in this paper are encoded as BDDs for additional improvement of efficiency.

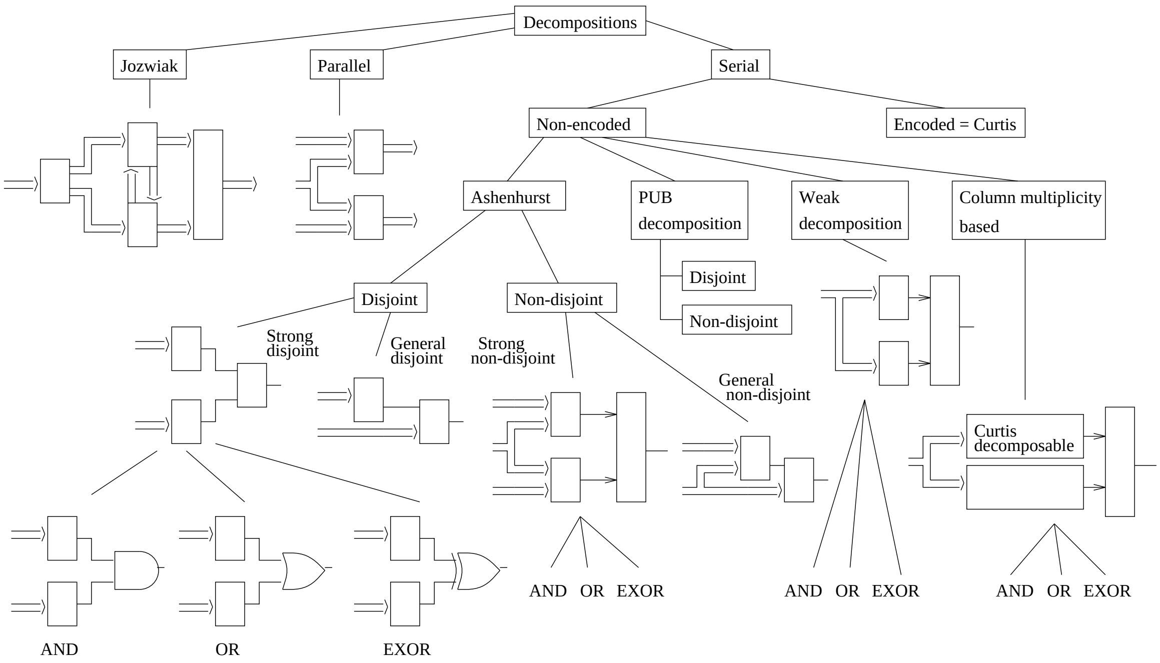


Figure 62:

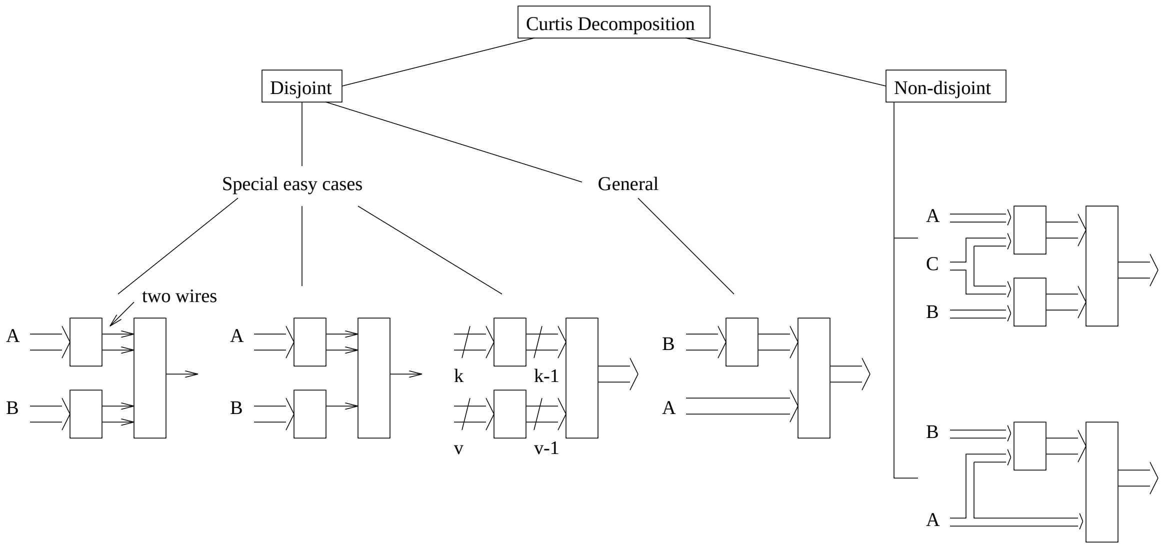


Figure 63:

Based on the analysis of the literature, the following observations were crucial to the development of the theory presented below.

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 [119], lists of disjoint ON and OFF cubes [?], and variable-partitions on ON and OFF minterms or cubes [143]. 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 comes 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 DDPC representation is totally new in the literature, and has all advantages of the previous ones 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 of minterms most efficiently.

2. The DDPC 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 [?, 119, 143, ?, 55, ?, ?, ?, ?, 21, 29, ?]. 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 bond 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, that is lost and repeatedly found in classic decomposition algorithms, is retained efficiently here for subsequent use.

Search for Good Bond and Free Sets.
Several search strategies have been used in the literature to find the best partitions for both the disjoint and non-disjoint cases. A parameterized program can be created that would allow to easily create and compare several of them: depth-first search, depth-first-search-with-one-successor, breadth-first search (a variant in Flash by Mark Axtell), best-bound (Haomin Wu/Perkowski [?]), limited best-bound search (Haomin Wu, Craig Files [140]), A* algorithm of Artificial Intelligence (Haomin Wu, Perkowski); searching subsets of a subset of variables (Shen/Kellar [25, 26, 31], Varma/Trachtenberg [?]), using predetermined order based on a heuristic of evaluating pairs of variables (cube based approach is presented by Wei Wan and Perkowski [?]), search by "add and subtract" variables from and to bond and free sets - Steinbach [?, ?], search by selecting a single variable based on complex block separation criteria of partition calculus heuristics (Luba [45, 67, 121, 143]), other BDD variable selection and ordering methods [?, ?, ?], and other decision tree like variable selection methods (such as in the C4.5 program by Quinlan). The search methods based on correlations of input and output variables should be investigated, since their results can be used for AC decomposition. Such ideas appeared in papers of Oliviera and Zaki [?], Mike Noviskey [?], and Grygiel and Perkowski [?].

Column Multiplicity Minimization.
In the past, the Column Multiplicity Minimization Problem was reduced either to the set covering with maximum cliques problem, [67, 169], 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, 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:

- 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 [74, 143, ?] 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 paper.

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 bond sets would also be useful for decomposition of fuzzy functions, where variables x and $\overline{x}$ should always come together.

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.

The following types of decompositions will be presented in sections:

- Ashenhurst disjoint decomposition for completely specified functions.
- Ashenhurst disjoint decomposition for incompletely specified functions.
- Ashenhurst non-disjoint decomposition for completely specified functions.
- Ashenhurst non-disjoint decomposition for incompletely specified functions.
- PUB disjoint decomposition for incompletely specified functions.
- PUB non-disjoint decomposition for incompletely specified functions.
- Curtis disjoint decomposition for incompletely specified functions.
- Curtis non-disjoint decomposition for incompletely specified functions.
- XBOOLE disjoint and non-disjoint decompositions for incompletely specified functions.
- Fuzzy decompositions for completely specified functions.

Figure 64 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 tes 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 application algorithms. They are of several types:

- decompositions (such as Curtis, PUB, strong OR, EXOR, weak AND, etc),

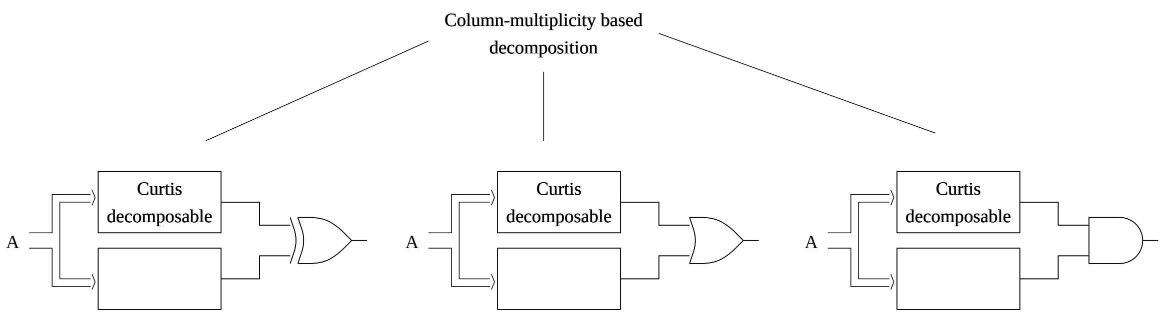


Figure 64:

- realizations of combinatorial algorithms in Curtis decomposition (such as graph-coloring and set-covering).
- bond set encoding algorithms.
- variable partitioning algorithms (r-admissability, entropy, search,etc).

In the new version of this document it will be shown that all these decomposition can be uniformly derived from a single model.

Marcel, this is our role. I have some ideas, but you can help me to formalize them.

25 Decomposition Strategy.

The comprehensive Decomposition strategy, Algorithm 3, 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 3 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:

1. number of true minterms,
2. number of false minterms,
3. number of true cubes,
4. number of false cubes,
5. sets A, B, C together with best patterns for them.
6. number of input variables,
7. % of ON Pattern columns,
8. % of OFF Pattern columns,
9. % of DC Pattern columns,
10. % of $F/\overline{F}$ Pattern columns,
11. % of Approximate ON Pattern columns,
12. % of Approximate OFF Pattern columns,
13. % of Approximate DC Pattern columns,
14. % of Approximate $F/\overline{F}$ Pattern columns,
15. cost parameters,
16. distance parameters,

17. number_of_bond_sets parameters,

18. and other.

Algorithm 3.

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 .

26 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.

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

The method for Multiple-Valued-Output functions with standard don't cares [156] 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.

26.2 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 [156]. When two columns (cofactors), f and g, are compared for compatibility in binary logic, the condition is: "for every two corresponding cells of these columns (in the same row of RVM), the cells must be the same, or one cell is a don't care." In case of generalized don't cares, this condition is generalized to: "for **every** row i of the RVM the set-intersection of values of the cells $C_i(f)$ and $C_i(g)$ of columns f and g of RVM must be non-empty, i.e.: $\forall\ i,\ C_i(f) \cap C_i(g) \neq \phi$." In such case, columns f and g are combined, and every combined cell $C_i(fg) := C_i(f) \cap C_i(g)$. Next, the Compatibility Graph is created with nodes for columns, and edges for pairs of compatible nodes. However, standard maximum cliques cannot be used, since, contrary to a 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 [173]. 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 5. Given is a function with 4 binary variables and a 5-valued output variable from the map in Fig. reffig:fig5ma (illustrated earlier in Table 1). Bond 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 Fig. ??b. The nodes represent columns from the map. A column near the edge between nodes C_i and C_j represents the combined column C_{ij}. As we see, nodes $C_0 = 00$ and $C_2 = 10$ are not compatible, since for instance $\{0{,}1\} \cap \{2{,}3\} = \phi$. 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 Figs. 5c,d, respectively, and the decomposed circuit in Fig. ??e. See [?] for applications to Attributes Reduction problem.

ab \ cd	C0 00	C1 01	C3 11	C2 10
00	0,1	0,3	1,3	2,3
01	1,2	—	0,1	2,3
11	0	0,3	—	—
10	0,3	0,4	—	1,4

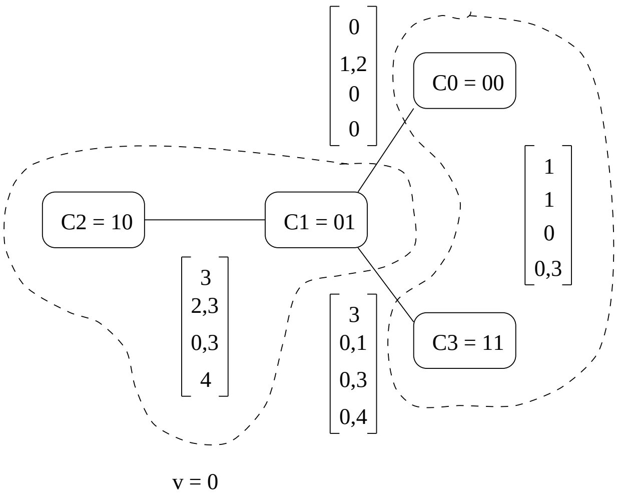
a)

c \ d	0	1
0	1	0
1	0	1

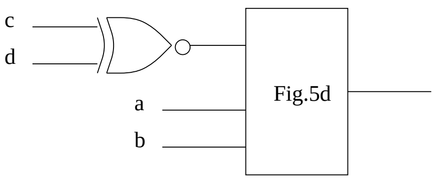
c)

ab \ v	0	1
00	3	1
01	2,3	1
11	0,3	0
10	4	0,3

d)



b)



e)

Figure 5. Decomposition of MV-output function with generalized don’t cares.

27 Applications to Machine Learning.

Recently, there has been an increasing interest in applying methods developed in design automation to other fields (see DAC'94 and Euro-DAC'94 panel discussions). Amazingly, the techniques developed in the last 15 years by the design automation community have been so universal and powerful, that they are also increasingly used in areas outside circuit design. For instance, they are now used in automatic theorem proving, robotics, industrial operations scheduling, stock market prediction, genetics research, and many others. It is then quite probable, that the design automation methods will lead to breakthroughs in these other fields.

One of the sub-areas of design automation that can find numerous external applications is logic synthesis. Unfortunately, the potential of logic synthesis for external applications is still less appreciated (by both the CAD and Machine Learning communities, and the general research and industrial circles) than the potentials of placement, routing, scheduling, placement, simulation, and database techniques of design automation.

Until very recently, the logic synthesis discipline formulated efficient methods and algorithms to minimize and optimize only the hardware of digital computers and other digital circuits. This goal was achieved very successfully, and the logic synthesis techniques developed in universities and industry in the last 15 years became one of the sources of the success of Design Automation in creation of such products as Intel's Pentium microprocessor. In the last few years, however, one can observe some increased trend to apply these methods also in image processing, machine learning, knowledge discovery, knowledge acquisition, data-base optimization, AI, image coding, automatic theorem proving and verification of software and hardware.

For instance, the ideas from logic synthesis and machine learning (ML) are starting to converge [195] [188]. The goals are similar. In Knowledge Discovery and Databases (KDD) and ML, at the core of every problem is how to effectively generalize concepts from data [198]. In the circuit design community, the end product is the realization of a minimum complexity circuit with respect to the number of gates, depth, inputs, literals, or product terms. The problems are analogous. In KDD, we have a database with fields and records. A set of records defines the concept. In logic synthesis, the fields of the records are the binary variables (inputs) of the circuit and each record is a specification of a minterm (true or false). The entire set of these records defines the circuit.

This part of the report is related to the use of the decomposer in ML and KDD applications. We will examine the performance of the decomposer on typical machine learning problems and compare the results with a standard decision tree learner, C4.5, with a multi-variant functional decomposer AFD, with Espresso, and with Exorcism. We will look at their performance on several small, noise free, binary functions without any "missing values" (i.e. completely specified functions, using circuit design terminology). We will also explore their performance on the heart disease database from the University of California-Irvine[1] (UCI). The comparison will serve to highlight the strengths and the shortcomings of all approaches, on a well-known and realistic problem. In this part, we will not present algorithms, but rather to demonstrate the applicability of logic synthesis, and specifically EXOR-based synthesis, to KDD and ML. All four algorithms share the common goal of a consistent, minimal complexity solution. All four differ in their method to find it.

28 The Basic Research Ideas of the PTG.

The Pattern Theory Group (PTG) in the Avionics Directorate at Wright Laboratory develops new system-level concepts for military applications, mostly based on image processing and machine learning technologies. Since 1988, the PTG developed a radically new approach to machine learning, where by "machine learning" we understand any method of teaching a computer to recognize any kind of patterns. Specifically, we are examining Supervised Classification Learning paradigms. The machine

[1] At present, the point of contact for this database is David W. Aha (aha@ics.uci.edu) and it is important to the authors that he is recognized here for his efforts to maintain it.

learning technologies most influential in military applications until now have been the neural nets and the bayesian methods. If successfully completed, the new approach of PTG will allow both automatic learning of any kind of images, and automatic creation of algorithms. Interestingly, in contrast to most of the well-known approaches, the approach of the PTG is based on logic synthesis methods: the so-called Curtis Decomposition of Boolean functions is applied here as the main component[200]. Many decomposition ideas have been implemented in the programming system FLASH (the Function Learning and Acquition Synthesis Hotbed) developed by this group[202]. This system is a Testbed for machine learning based on the logic synthesis approach. The group also compares FLASH to other logic optimizers and machine learning programs, (such as Espresso and C4.5, respectively) from the point of view of the robustness of learning[184, 183, 182, 185].

Simplifying, the main difference of the logic approach and the previous approaches to machine learning is that in these previous methods, the recognizing network had some assumed structure which was next only "tuned" by the learning process (for instance, by decreasing and increasing the numerical values of some coefficients). Thus, creating a new structure is acomplished by setting some coefficients to zero. All "new" structures are in a sense "hidden" in the assumed mother-structure. Also the type of an element, such as a formal neuron in Neural Nets, or a polynomial in Abductory Inference Mechanism (AIM[2]). [190] is decided before the learning takes place.

In contrast, in the Curtis Decomposition approach, there is no apriori assumption of structure of the learning network, nor on the type of the elements. The elements are arbitrary discrete mappings (functions). Both the structure and the elements are **calculated** in the learning process, and this process is entirely based on finding patterns in data.

The central concept of Pattern Theory is a "pattern." In Pattern Recognition and Knowledge Discovery the problems with nice representations based on "features" belong to a more general class of problems with strong "patterns." Pattern finding is, therefore, a generalization and formalization of feature extraction. The goal of the Pattern Theory is to support "automating" the pattern finding process, and to construct a representation of a function based on examples; therefore, this is a method for constructive induction. Furthermore, it constructs this representation by minimizing complexity as in Occam-based learning. The Ashenhurst Function Decomposition (AFD) method based on Curtis Decomposition implemented in FLASH is unusual in that it learns both the architecture of the combinational representation and the component functions **from** the examples.

Induction is often modeled as the process of extrapolating samples of a function. This extrapolation requires both the samples and the "inductive bias." The bias towards low complexity, as in Occam's Razor, is particularly important. There is a strong theoretical basis for Occam-based learning, see for example [178, 179]. Kolmogorov complexity was developed specifically for induction (reference [191]); however, it has been proven that its exact computation is not tractable. There have been some tractable measures of complexity used in actual implementations of Occam-based learning, such as the Abductory Inference Mechanism [190] which uses polynomial networks and C4.5 [199] which uses decision trees. However, the measures of complexity used in these applications are relatively narrow, which implies some compromise in the robustness of the learning; for example, neither of these methods would find the parity function to have low complexity even though it is highly patterned and can be easily computed. The challenge is to develop robust *and* tractable measures of complexity.

Pattern Theory [200] treats robust complexity determination as the problem of finding a pattern. Pattern theory uses Decomposed Function Cardinality (DFC), proposed by Y. S. Abu-Mostafa as a general measure of complexity [177, p.128]. DFC is based on the cardinality of a function. After all, a function is a *set* of ordered pairs and, as with any set, has a definite property in its number of elements or cardinality. DFC is defined as the sum of the cardinalities of the components of a combinational representation of a function, when that sum has been minimized. DFC is especially robust in the sense that it reflects patterns of many different kinds. Its robustness is supported by its relationship to more conventional measures of complexity, including circuit size complexity, time complexity, decision tree

[2] Trademark of AbTech Corp.

or diagram size and Kolmogorov complexity. If a problem has low complexity by any of these measures then it will also have a low DFC [200, Chapter 4]. The PTG work has concentrated on functions with binary inputs, but the concept is easily extended to continuous and multiple-valued functions [203].

The decompositions are evaluated based on the sum of the function cardinality of the decomposed partial blocks. Of course, the real DFC of f is less than or equal to this sum in any particular (usually approximate) realization. The objective of the decomposition is to search through partitions of input variables to find one that produces the smallest total DFC. In other words, the "cost" of the feature is measured in terms of DFC and that cost is minimized. Therefore, features are constructed to minimize their computational complexity. The use of DFC as the measure of complexity allows for robustness in the type of feature that can be generated.

Let us observe that all presented in previous sections methods: and in particular, the Curtis decomposition, look for certain patterns in data: in Curtis decomposition these patterns are in columns of the map corresponding to the cofactors of the bond set of variables [12]. Let us also observe that in all cases we minimize a certain cost of the circuit. Traditionally in circuit minimization one calculates the number of terms and the number of literals. In our case, we calculated additionally the total DFC as a sum of function cardinality of all non-decomposable blocks (this is an upper bound approximation of the minimum DFC). For an arbitrary non-decomposable block in Curtis Decomposition, the DFC of the block is calculated as 2^k where k is the number of inputs to the block. In "gate-based" minimizers such as Espresso and EXORCISM it is then fair to assume that a DFC of a decomposable gate (such as AND, OR or EXOR) is equal to the total DFC of a circuit equivalent to this gate, that is constructed from two-input gates. The DFC of a four input AND gate, OR gate, or EXOR gate is then $2^2 + 2^2 + 2^2 = 12$, since such gates can be decomposed to balanced trees of three two-input gates.

The AFD algorithms that we discuss all have the above in common. They differ in the number of partitions explored. The simplest method chooses one partition at random. It then tries to merge columns, encode new states, and then pick another partition at random from this new state until the one it picks does not decompose to a smaller number of unique columns. We will call this version of AFD "lotto." The reader will note this is not even a hill-climbing algorithm because only one partition at a given level is explored.

A more complex algorithm looks at all partitions at a given level and 40 random partitions one level deeper. Essentially, this is a 2-ply look ahead search where we only explore 40 grand children. It chooses a particular partition over another based on the function's DFC.

29 Summary of DFC Measurements and Applications

A number of experiments have been conducted to assess the generality of DFC across different problem domains (see [200, 202, 201]. The DFC of over 800 non-randomly generated functions was measured, including many *classes* of functions (numeric, symbolic, chaotic, string-based, graph-based, images and files). Roughly 98 percent of the non-randomly generated functions had low DFC (versus less than 1 percent for random functions). The 2 percent that did not decompose were the more complex of the non-randomly generated functions rather than some class of low complexity that AFD could not deal with. It is important to note that when AFD says the DFC is low, which it did some 800 times, it also provides an algorithm (or a description of the pattern or features found). AFD found **the** classical algorithms for a number of functions. Each of these algorithms are represented in a combinational form that includes intermediate states. These intermediate states are features in the sense of concentrating information.

There is also high correlation between DFC and a person's judgment of the strength of a pattern in an image, the degree of compression by a data compression program, and the Lyapunov exponents of logistic functions. This shows DFC's ability to reflect patterns within each domain, despite their different nature.

In learning experiments, AFD has been compared to back-propagation trained Neural Networks (NN), AIM, C4.5, and standard logic minimization tools. These comparisons used a broad range of function types (from the 800 mentioned above). For each of the test functions, AFD performed near the best of any method, while the other methods generalized well on some functions but not on others.

In the context of ML, when one talks about "noise," it is assumed that he is refering to the situation where you have some training data classified correctly as a "1" or a "0" but then "noise" flips that bit to the incorrect entry. Another situation is that the value of some minterm is or becomes unknown - a "don't care". This would be refered to as "unknown" in the ML community. "Noise" and "unknowns" are related - ML techniques can be used to restore noisy images [180]. Of course, we do not know which pixels (or feature values) are "noisy" - so we treat those that are most suspicious as don't cares.

The more robust generalization performance of AFD, including dealing with noise and unknown data, is a reflection of its robust measure of complexity, DFC. It is also an indication that the useful inductive bias of these various standard methods results from their parsimonious tendencies rather than their particular representation schemes (be they thresholded weighted sums, polynomials, decision trees or Boolean operators).

30 Towards Improved Approaches to Logic Minimizers for Machine Learning

Although the AFD approach of FLASH gives very robust results, it is slow, which essentially restricts its practicality to 20 or even less, variables. It is, therefore, important, to be able to compare the FLASH decompositional logic approach to other logic approaches that use the same DFC measure, but introduce some restricted bias resulting from the assumed network's structure. Such approaches are then faster and can be used for larger variable functions.

In this respect, the well-known circuit minimizer Espresso, the standard machine-learning program C4.5, and ESOP minimizer EXORCISM will tested together with our decomposer. These programs have the following structure/gate-type biases: Espresso assumes a two-level AND-OR network, EXORCISM assumes a two-level AND-EXOR network, C4.5 assumes an ordered tree. (The input variables can be multiple-valued).

The questions arise:

- How much of the network's simplicity is lost by assuming these structures?
- How much is gained in the speed of the program with respect to a bias-free decomposer? Is this speedup worth an increased DFC and thus a more limited extrapolation capability?
- Is the method with a biased structure still robust enough for practical applications?

Other important question that must be faced with while developing improved minimizers for machine learning applications is the following:

- What are the reasons that machine learning using logic synthesis is not exactly the same as circuit design using logic synthesis?

This question is very important practically. Improving the performance of the FLASH system orders of magnitude without sacrificing much of its robustness (DFC) is required for making it useful for such important military applications as High Resolution Radar, for example.

The data (switching functions) used in learning and algorithm design applications by the PT group are *arbitrary switching functions*. Thus, the standard and generally applicable minimization procedures of "logic synthesis" can be applied. An extremely important observation is that these functions have *quite different properties* than the data taken from industrial companies on which the programs are tested in the "logic synthesis" community (MCNC benchmarks).

In theory, the algorithm should work well on any type of data. However, since all practical network minimization problems are NP-hard, all practical algorithms, by neccessity, are heuristic in nature. Thus, they are very dependent on the type of data. Taking into account the data characteristics (such as closeness to unate Boolean functions) was, in principle, the main reason of the commercial success of two-level logic minimizers in circuit-design applications.

What is it that distinguishes the machine learning data from the circuit design data? Our preliminary answer is the following:

1. ML problems have an extremely high percent of don't cares (Don't cares are combination of argument value for which the function value is not specified.) The missing data can be represented as don't cares.

2. Arguments (variables) in ML problems are naturally multiple-valued, which means that they take any discrete number of values and not only true (1) or false (0). For instance, the size can be large (2), medium (1) or small (0).

3. The ML problems have some kind of "counting properties" (one counts the numbers of repetitions of some patterns, like the well-known problem of learning even/odd properties in perceptrons).

4. ML problems have various kinds of symmetries and generalized symmetries.

Decomposer is able to find a pattern of EXOR (parity) or similar functions, even when it is corrupted by "unknowns". This is a difficult problem in machine learning. To explain this case on an example, let us assume that we recognize the even/odd parity function. For a completely specified function and a relatively small (less than 16) number of variables, the AFD minimizer finds the exact minimal result (EXOR of input variables) quite fast. When we add some "unknowingness" to this function by replacing some ones and zeros with "don't cares", we should still be able to find the EXOR of inputs solution, since the underlying principle function did not change, only its pattern has been corrupted, "hidden" by the unknown values. This seemed to work, but when the percentage of don't cares increases and the number of variables inverases, the method yields poor results. First it ceases to recognize the "EXOR of variables" pattern, and second, on 96-variable functions, it finds no EXOR's at all and looses track of any patterns (so does a human on this case).

Another positive property of the decomposer is simultaneous classification of patterns to more than two categories (you want not only to distinguish a "friend from foe" airplane, but you want to learn its orientation, speed, etc.). In terms of logic synthesis, this property corresponds to concurrent minimization of switching functions with many outputs. Currently FLASH operates on single-output functions, but Decomposer works with multi-output functions. There are many decomposers that decompose multi-output functions, but all of them have been designed for circuit design. One needs a minimizer for **strongly unspecified, multi-valued input, multi-output functions.**

What is also missing in both "industrial circuit" and "machine learning" decomposing systems, is the **decomposition of multiple-valued input, multiple-valued-output functions.**

Why is this important? In theory, which is also the approach of the PT group, any multiple-valued variable can be encoded by a vector of binary variables. What happens, however, in learning situations is, that the learning system inferences rules that depend on the encoding of multiple-valued variables with binary variables. To give an example, if the system would infer from a large set of data that people who live close to power lines develop cancer, we would perhaps treat such "invention" with due care. If the system would, however, infer that people whose third bit of encoded distance from the line is 1 develop cancer, we would treat such inference as a "coding-related" artifact. Therefore, the best approach to the learning system would be not to use coding at all, but perform the inference on the variables that are natural for any given feature; e.g. either binary (man, woman), or multiple-valued (distance in yards).

Original Function	Actual DFC	Average Error		
		C4.5	Decomposition	Espresso
$kdd4.efr$	2	0	0	0
$kdd1.efr$	8	0.32	0	0.96
$kdd8.efr$	8	6.35	0	5.64
$kdd6.efr$	12	2.48	3.72	2.64
$kdd3.efr$	12	1.28	2.72	3.52
$kdd2.efr$	16	2.76	2.4	12.86
$kdd10.efr$	20	17.52	8.18	17.16
$kdd9.efr$	20	13.79	6.55	16.54
$kdd7.efr$	28	20.69	10.53	5.69
$kdd5.efr$	36	10.52	11.11	8.44
Average		7.57	4.52	7.345

Table 5: DFC and Average Error

Original Function	Actual DFC	# of Samples		
		C4.5	Decomposition	Espresso
$kdd4.efr$	2	8	7	9
$kdd1.efr$	8	31	25	40
$kdd8.efr$	8	83[3]	25	51
$kdd6.efr$	12	74	67	76
$kdd3.efr$	12	61	76	54
$kdd2.efr$	16	97	126	113
$kdd10.efr$	20	200	60	181
$kdd9.efr$	20	224	104	205
$kdd7.efr$	28	256	126	51
$kdd5.efr$	36	249	251	229
Average		128.3[4]	86.7	100.9

Table 6: DFC and # of Samples Needed to Learn

31 Small Problems

31.1 Overview

This section describes the first set of test functions. Each of these functions consist of 8 binary variables with a single binary output. All of the functions are void of noise and missing values. Each function was chosen as to in some way represent a "database-like" concept. It is noted to the reader that the logic synthesis approaches generalize to discrete and continuous data fields [203] as well as multi-valued outputs.

31.2 Experimental Design

Here, we show an abridged version of our test results with brute-force decomposition (AFD), Espresso, C4.5, and EXORCISM. We used the well-known KDD functions: kdd1.efr - kdd10.efr. The comparison of results of Decomposer, Espresso and EXORCISM on these functions is in Table 6, part 1. The functions are explained in section 10.

[3] C4.5 with pruning learned this function with 46 samples.
[4] The average with the better score for F_8 is 124.6.

The tests on the individual functions were as follows. First, each method was given a random set of data to train on ranging from 25 to 250 out of a total of 256 possible cases. Once the method was trained, the entire 256 cases were tested and the number of differences were recorded as errors. This procedure was repeated 10 times for a given sample training size in intervals of 25. Thus, the total number of runs for each function was 100 of varying sample size. None of the learning was incremental. All of the runs were independent.

For brevity, the comparison graphs were omitted in lieu of a condensed summary of each of the learning curves. For each function, the average number of errors for the entire run was recorded in Table 1. It was then possible to compare brute-force decomposition, Espresso, C4.5, and EXORCISM with these averages for a given function. Table 2 shows the number of samples necessary before the learning method obtains a concept such that in all ten separate runs, the number of errors was 0. The value at the bottom of each table is the average over all of the functions. The smaller the number here, the better the performance.

32 Characterization of Benchmark Functions

We thought it may be interesting to describe some examples of benchmarks that we used in addition to MCNC benchmarks. More "machine learning benchmarks" are available from U.C. Irvine. (http://www.ics.uci.edu/AI/ML/MLDBRepository.html) The WL benchmarks are available from Pattern Theory Program: Tim Ross, ross@speedy.aar.wpafb.af.mil. Some of the benchmark names from the tables are separated into several groups listed and briefly explained below.

LEARNING8_SET. This set of functions is intended to be representative of a wide variety of functions for testing machine learning systems.

RANDOM. There are 3 randomly generated functions, generated with FLASH with seeds 1,2, and 3: *rnd1, rnd2,* and *rnd3.*

RANDOM MINORITY ELEMENTS. There are 5 functions generated with a fixed number of minority elements placed at random. The seed for all was 1: *rnd_m1, rnd_m5. rnd_m10, rnd_m25, rnd_m50.*

RANDOM WITH VACUOUS VARIABLES. *rndvv36.* copied from Taylor set, was randomly generated except for x3 and x6 being vacuous.

BOOLEAN EXPRESSIONS. These are functions intended to represent database concepts for knowledge discovery[184]. *kdd1*: (x1 x3) + x2'; *kdd2*: (x1 x2' x3)(x4 + x6'); *kdd3*: NOT (x1 OR x2) + (x1' x4 x6); *kdd4*: x4'; *kdd5*: (x1 x2 x4')+(x3 x5' x7 x8)+(x1 x2 x5 x6 x8)+(x3' x5'); *kdd6*: x2 + x4 + x6 + x8; *kdd7*: (x1 x2) + (x3 x4) + (x5 x6) + (x7 x8); *kdd7b*: ((x1 x2) .XOR. (x3 x4)) .NAND. ((x5 x6) + (x7 x8)); *kdd8*: (x1 x2') XOR (x1 x5); *kdd9*: (x2 XOR x4)(x1' XOR (x5 x7 x8)); *kdd10*: (x1 $\rightarrow$ x4) XOR (NOT (x7 x8))(x2 + x3).

Note: *kdd7* and *kdd7b* differ in only 9 places and have the same decomposition architecture (and DFC). *multiplexer, mux6,* used in [189], this is a 2-address bit, 4-data bit multiplexer with two vacuous variables (x0 and x1) to make 8 inputs.

DEEP FUNCTIONS.

and_or_chain8 = AND(OR(AND(OR(AND(OR(AND(x1,x2),x3),x4),x5),x6),x7),x8), *or_and_chain8.*

MONKISH PROBLEMS. These are 8 binary variable "approximations" to the Monk's problems of Thrun'91 et. al. [207]. x1: head shape (rnd, octagonal), x2: body shape (rnd, octagonal), x3: smiling (yes, no), x4,x5: holding (sword, balloon, flag, whistle), x6,x7: jacket color (red, yellow, green,blue), x8: has tie (yes, no). *monkish1*: head shape equals body shape or jacket is red. *monkish2*: exactly 2 of 6 attributes have 1st value. *monkish3*: (jacket green & has sword) or (jacket not blue and body not oct.)

STRING FUNCTIONS. Palindrome acceptor, *pal,* palindrome output, *pal_output,* randomly generated 128 bits then mirror imaged them to create the outputs of an 8 variable function. Doubly palindromed output, *pal_dbl_output,* as above but generated 64 bits and flipped them twice. 2 interval acceptors from FLASH, *interval1* accepts strings with 3 or fewer intervals (i.e. substrings of all zeros or all ones). *interval2* accepts strings with 4 or fewer intervals 2 sub-string detectors. *substr1* accepts strings with the sub-string "101". *substr2* accepts strings with the sub-string "1100".

IMAGES. *chXfY* means character X from font Y of the Borland font set. All were generated with the Pascal program charfn.exe. *ch8f0* - kind of a flat plus sign, *ch15f0* - an Aztex looking design, *ch22f0* - horizontal bar, *ch30f0* - solid isosceles triangle, *ch47f0* - slash, *ch176f0* - every other column of a checker board, *ch177f0* - checker board, *ch74f1* - triplex J, *ch83f2* - small S (thin strokes), *ch70f3* - sans serif F, *ch52f4* - gothic 4.

SYMMETRIC FUNCTIONS. *parity. contains_4_ones,* (f(x)=1 iff the str x has 4 ones). *majority_gate,* f(x)=1 iff x has more 1's than 0's.

OTHER ACCEPTORS. *primes8,* output is 1 iff the input string, as a decimal number, is prime. 0 and 1 are not considered prime.

NUMERICAL FUNCTIONS. *addition, add0, add2, add4* - outputs bits of a 4 bit adder, 0 is the most significant bit. *greater_than:* f(x1,x2)=1 iff x1 > x2. *subtraction: subtraction1, subtraction3* - output bits 1 and 3 of the absolute value of a 4-bit difference. 0 is most significant bit. *modulus2,* output bit 2 of 4-bit modulus 0 is the most significant bit. *remainder2,* output bit 2 of 4-bit remainder 0 is the most significant bit.

GEOMETRIC FUNCTIONS. Generated using nearest neighbor (Hamming distance): - note that majority gate is effectively a nearest neighbor type function with samples:

00000000 0
11111111 1

nnr1: kind of a four corners partitioning of the space. based on samples:

00000000 0
00001111 1
11110000 1
11111111 0

nnr2: has two randomly selected points as templates. based on samples:

00111010 0
11011110 1

nnr3: has eight randomly selected points as templates. based on samples:

00010011 1
00111010 1
01000101 0
01011001 0
01110010 1
10001101 0
11000111 1
11011110 0

33 Heart Disease Database

The Cleveland heart disease database[5] used in our experiments was the "processed" data where only 14 of the original 76 attributes were used in classification. For our purposes, we will further modify the data (convert to binary) as shown in Table 4 for testing with all learning algorithms. The intervals (for continuous type data) were basically chosen to be evenly distributed along the appropriate line segment. There were no special rules to delineate the data in order to be advantageous to a particular learning algorithm. Some of the variables were discrete and had a natural encoding. E.g., variable 9 for chest pain had 4 values, thus it was encoded with 2 binary variables.

One additional alteration was made to the database. Six of the 303 patient records contained some missing values. They were all removed because at present, the logic synthesis tools have no reasonable way of handling this data. It was decided rather than encoding the missing value as a separate value or using some other method, we would simply omit those records. Also, because of the binary encoding, it happened that 2 different records in the original database were encoded to the same values including their output. This duplicate entry was discarded. The end result is a transformation of the 14 variable discrete/continuous data to a 26 binary variable problem. There are a total of 296 records.

34 Analysis of the Heart Disease Problem

This section will be added to the final version of the report.

[5] The authors wish to express their gratitude to Dr. Robert Detrano, M.D., Ph.D. and the V.A. Medical Center, Long Beach and Cleveland Clinic Foundation for the use of their database.

[6] T wave inversions and/or ST elevation or depression of > 0.05 mV

[7] by Estes' criteria

	Heart Variable	Value	Binary Encoding
x_1 x_2	variable 3 Age		00 $\leq$ 30 01 31-50 10 51-70 10 ¿70
x_3	variable 4	Sex	0=female 1=male
x_4 x_5	variable 9 chest pain	typical angina atypical angina non-anginal pain asymptomatic	00 =1 01 =2 10 =3 10 =4
x_6 x_7 x_8	variable 10 resting blood pressure (mm Hg)		000 $\leq$ 100 001 101-115 010 116-130 011 131-145 100 146-160 101 161-175 110 176-190 111 $\geq$ 190
x_9 x_{10} x_{11}	variable 12 serum cholesterol (mg/dl)		000 $\leq$ 160 001 161-200 010 201-240 011 241-280 100 281-320 101 321-360 110 361-380 111 $\geq$ 380
x_{12}	variable 16 fasting blood sugar		0= $\leq$ 120mg/dl 1 $\geq$ 120mg/dl
x_{13} x_{14}	variable 19 resting electro- cardiographic results	normal having ST-T wave abnormality[6] showing probable or definite left ventricular hypertrophy[7]	00 =0 01 =1 10=2
x_{15} x_{16} x_{17}	variable 32 Maximum Heart Rate Achieved		000 $\leq$ 100 001 101-115 010 116-130 011 131-145 100 146-160 101 161-175 110 176-190 111 $\geq$ 190
x_{18}	variable 38 exercise induced angina		0=no 1=yes
x_{19} x_{20}	variable 40 ST depression induced by exercise relative to rest		00 $\leq$ 1.5 01 $\geq$ 1.5 - 3.0 10 $\geq$ 3.0 - 4.6 10 $\geq$ 4.6
x_{21} x_{22}	variable 41 slope of the peak exercise ST segment	unsloping flat downsloping	00 =1 01 =2 10 =3
x_{23} x_{24}	variable 44 number of major vessels colored by flourosopy	0-3	00 =0 01 =1 10 =2 11 =3
x_{25} x_{26}	variable 51 Thal	normal fixed defect reversible defect	00 =3 01 =6 10 =7
	Output	Healthy Disease(1-4)	0 1

Table 7: Heart Disease Database Binary Conversion

35 Conclusions and Future Research.

In this report we introduced a new representation and a new general decomposition/synthesis approach for strongly unspecified multiple-valued multi-output functions. Our main immediate application was Machine Learning, but we forsee also future applications in circuit design.

The approach presented here can be specialized and expanded to many future variants which are a subject of many works of our gruop. It seem to open 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;
- $EXOR^2\_SOPs$ circuits synthesis;
- 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 are currently in the process of implementing the decomposition algorithm from section 8, and we must be done with it before the end of September.

The new representation introduced by us allows also to utilize Sum-of-CDEC, TANT, ESOP, $EXOR^2\_SOP$, and other similar circuits for Machine Learning applications. 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 further investigated (see [155] for the first results). Since Fuzzy logic circuit decomposition can be reduced to 4-valued logic decomposition, the approach can be also used to decompose Fuzzy Logic circuits [?].

The goal of the research outlined here is to prove, step-by-step, the following:

- the new representation is superior to all previous representations for function decomposition.
- a decomposition program can be created that will be much superior to all the current systems as a machine learning program on functions with a small and medium number of variables.
- a decomposition program can be created that will be superior to all the current systems as a machine learning program on functions with more than 40 variables.
- a system for functions with a very high percentage of don't cares, superior to previous ones can be created.

Soon more testing results will be available, as well as comparison with other machine learning programs on the Heart Disease Problem Database. Our ultimate 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.

The program must be robust across various classes of data from the learning benchmarks. Combining SOP and ESOP minimizers, like Espresso and EXORCISM, into a single program with a functional decomposer will create a program that would be superior to any of them. We have to find the best way to do this.

We believe that the analyses of various approaches to machine learning, including SOP, ESOP, trees, decision diagrams, and functional decomposers performed at Wight Labs and in our previous papers pinpointed strengths and weaknesses of all analyzed approaches: AFD is clearly superior on small functions but it is not yet tractable on larger ones; Espresso has trouble with "counting" type of dependencies such as parity and arithmetic circuits but handles don't cares relatively well; EXORCISM is superior to Espresso and C4.5 on many larger functions, but has trouble with very strongly unspecified functions, and C4.5 - the defacto standard machine learning tool, can handle more special cases of data and user requirements, but its quality is often worse than other approaches. We believe, all these comparisons will be helpful to create a superior program.

The observation that functions in Machine Learning are very strongly unspecified and thus none of the known approaches work well, makes the requirements on the minimization programs in circuit design and machine learning very different, a point that has not yet been sufficiently observed and appreciated. **This fact calls for the development of totally new approaches to synthesis,** and is a very positive opportunity for people working in the area of development of new algorithms for logic synthesis. Instead of adapting the algorithms created for circuit design, new algorithms should be created **from scratch**, and from the very beginning they should take into account the problem specifics. Moreover, since these algorithms run only in software, and only DFC minimization is important, EXOR gates or any other gates or universal modules are as good as any other commonly used gates and cells, and there is no problem of its realization or speed. The space of investigations is then here much wider than in classical logic synthesis for circuit design applications.

Missing values, and especially noise, are still not adequately part of the circuit design world but are a reality in KDD and ML. It will be necessary to find solutions to these issues if we are to use logic synthesis tools in these fields.

We believe that machine learning will become a new and fruitful area for logic synthesis research, and an application territory for logic minimizers. There exist big challenges, but also great wins for successful programs. The first research results that appreciate this synergy and try to link the two worlds of the "machine learning community" and the "design automation community" already start to appear: new decision-diagram approaches were presented in 1994 by Ron Kohavi [187, 188], and Arlindo Oliveira [195].

It is quite possible, that problems with an unusually high percent of don't cares, first observed in ML, will be also better identified in the area of circuit design; for instance, when more sophisticated compilers of VHDL and other high-level languages will become to proliferate.

References

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

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

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

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

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

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

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

[8] 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.

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

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

[11] R.M. Karp, F.E. McFarlin, J.P. Roth, J.R. Wilts, "A Computer Program for the Synthesis of Combinational Switching Circuits," *In Proc. AIEE Annual Symposium on Switching Circuits Theory,* pp. 182-194, 1961.

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

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

[14] 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.

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

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

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

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

[19]

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

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

[22] 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.

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

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

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

[26] 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.

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

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

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

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

[31] 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.

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

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

[34] 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.

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

[36] C.C. Cheung, B.W. Ehrich, "Minimization of Tree-Type Universal Logic Circuits," *IEEE Trans. on Comput,*, Vol. 24, No. 11, pp. 1110-1113, 1975.

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

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

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

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

[41] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

[55] P.N. Bibilo, S.V. Yenin, "Synthesis of Combinational Networks Using the Method of Functional Decomposition," ("Sintez Kombinacjionnych Schem Metodami Funkcijonalnoj Dekompozicji,") *Nauka i Tekhnika,* Minsk, 1987, (in Russian).

[56] R. Rudell, 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.

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

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

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

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

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

[62] 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.

[63] .

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

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

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

[67] 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).

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

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

[70] 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.

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

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

[73] 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.

[74] M.A. Perkowski, J. Liu, J.E. Brown, "Rapid Software Prototyping: CAD Design of Digital CAD Algorithms," In G. W. Zobrist (ed), *Progress in Computer-Aided VLSI Design,* Vol. 1, pp. 353-401, 1989.

[75] 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.

[76] S. Yang, M. Ciesielski, "A Generalized PLA Decomposition With Programmable Encoders," In the *Proc. of the Intern. Workshop on Logic Synthesis,* May 1989, pp. 1-13.

[77] S. Yang, M. Ciesielski, "PLA Decomposition with Generalized Decoders," *Proc. of the ICCAD,* pp. 312-315, 1989.

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

[79]

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

[81] R. J. Francis, J. Rose, K. Chung, "Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Array," *Proc. 27th ACM/IEEE Design Automation Conf.*, 1990, pp. 613-619.

[82]

[83] 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.

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

[85] D. Filo, J.C. Yang, F. Mailhot, 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.

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

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

[88]

[89] R. Murgai, N. Shenoy, R.K. Brayton, A. Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithm for Table Look Up Architectures," *Proc. IEEE Intern. Conf. on Computer Aided Design,* pp. 564-567, 1991.

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

[90]

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

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

[93] 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.

[94] P. Sicard, M. Crastes, K. Sakouti, G. Saucier, "Automatic Synthesis of Boolean Functions on Xilinx and Actel Programmable Devices," *Proc. Euro ASIC '91,* pp. 142-145, 1991.

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

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

[97]

[98] 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.

[99] L. Jozwiak, F. Volf, "An Efficient Method for Decomposition of Multiple-Output Boolean Functions and Assigned Sequential Machines", *Proc. European Conference on Design Automation,* pp. 114-122, 1992.

[100] A. Skowron, 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.

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

[102]

[103] T. Luba, 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.

[104]

[105] T. Luba, 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.

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

[107] T. Luba, J. Rybnik, "Algorithmic Approach to Discernibility Function with Respect to Attributes and Object Reduction", *Int. Workshop on Rough Sets: State of the Art and Perspectives,* Poznan 1992.

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

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

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

[111] 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.

[112] W. Wan, 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.

[113] 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.

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

[115] M.L. Axtell, T.D. Ross, M.J. Noviskey, "Performance Comparison Between Occam-Based, Back-Propagation, and Abduction Learning Networks," In *IEEE Int. Conf. on Neural Network Applications in Signal Processing,* 1993.

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

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

[118] 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.

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

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

[121] T. Luba, H. Selvaraj, A. Krasniewski, "A New Approach to FPGA-based Logic Synthesis," *Workshop on Design Methodologies for Microelectronics and Signal Processing,* pp. 135-142, Gliwice-Cracow 1993.

[122] T. Luba, R. Lasocki, J. Rybnik, "An Implementation of Decomposition Algorithm and its Application in Information Systems Analysis and Logic Synthesis," *International Workshop on Rough Sets and Knowledge Discovery,* pp. 487-498, Banff 1993,

[123] T. Luba, J. Rybnik, "Algorithmic Approach to Discernibility Function with Respect to Attributes and Objects Reduction," *Foundation of Computing and Decision Sciences,* Vol. 18, No. 3-4, pp. 241-258, 1993.

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

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

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

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

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

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

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

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

[132]

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

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

[135] 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.

[136] H. Selvaraj, A. Czerczak, A. Krasniewski, T. Luba, "A Generalized Decomposition of Boolean Functions and its Application in FPGA-Based Synthesis," *IFIP Workshop on Logic and Architecture Synthesis,* pp. 147-166, Grenoble 1993.

[137] 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.

[138] 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.

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

[140] 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.

[141] 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.

[142] 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.

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

[144] T. Luba, 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.

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

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

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

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

[149] M. A. Perkowski, M. Chrzanowska-Jeske, T. Shah, "Minimization of Multioutput TANT Networks for Unlimited Fan-In Network Model," *Proc. of ICCD'90,* pp. 360 - 363, Boston, MA, September 1990.

[150] M. A. Perkowski, 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.

[151] 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.

[152] 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.

[153] 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.

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

[155] M. A. Perkowski, T. Ross, D. Gadd, J. A. Goldman, and N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," *submitted to this Workshop,* April 1995.

[156] M. A. Perkowski, T. Luba, S. Grygiel, R. Lisanke, N. Iliev, P. Burkey, M. Burns, R. Malvi, C. Stanley, Z. Wang, H. Wu, F. Yang, S. Zhou, and J. S. Zhang, "Unified Approach to Functional Decompositions of Switching Functions," *PSU Report*, March 1995.

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

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

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

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

[161] 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.

[162] Selvaraj Ph.D.

[163] R. Kohavi, 1994. Several recent papers.

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

[165] A. Oliveira, 1994.

[166] Voigt

[167] Murgai, Brayton,

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

[169] Luba, ISMVL Invited Paper, 1995.

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

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

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

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

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

[175] E. B. Lee, 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.

[176] Perkowski, cdb applications

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

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

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

[180] 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.

[181] 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.

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

[183] 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.

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

[185] 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.

[186] 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.

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

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

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

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

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

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

[193] 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.

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

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

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

[197] 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.

[198] 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.

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

[200] 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.

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

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

[203] 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.

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

[205] 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.

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

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

[208] 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.