A New Representation of Strongly Unspecified Switching Functions and its Application to Multi-Level AND/OR/EXOR Synthesis.

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

Abstract— The paper presents a new representation of Boolean and multiple-valued functions, called *Cube Diagram Bundles.* **This representation is especially good for very strongly unspecified functions. 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 we discuss a generalized, goal-oriented multi-level decomposition, that makes special use of EXOR-based decompositions. Our unified approach includes the decompositions of Ashenhurst, Curtis, Steinbach et al, Luba et al, and Perkowski et al as special cases.

I. Introduction.

Boolean and multiple-valued functions that include very many don't cares are becoming increasingly important in several areas of applications [14]. In this paper we introduce a new representation and decomposition method for such functions. The paper 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 [7], Decision Diagrams [3] and Rough Partitions [10], 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 6 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 [1] is also presented - it is shown that contrary to the more general case of Curtis Decomposition [6], 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 belongs to the *Basic Decompositions* of the system. Although in some respects similar to the approach from [16], 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 [16]. Any function can serve as a goal function, and three reduction types (EXOR, OR and AND) of reducing to a goal function are presented. Section 7 presents the combined search strategy that uses all the decompositions, and section 8 presents numerical results and conclusions.

II. Cube Diagram Bundles to Represent Discrete Mappings.

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 one uses two representation methods: Multiple-Valued Cube Calculus (Positional Notation) and Multiple-Valued Decision Diagrams. These methods have been also extended to incompletely specified functions.

All these representations are being 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 [2] 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 [10] 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 [11, 8]. While cubes seem superior in problems with a limited number of levels, such as SOP or ESOP synthesis, the DDs are superior for general-purpose Boolean function manipulation, tautology, technology mapping, and verification.

This paper 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 Cube Calculus [7], Decision Diagrams [3], Rough-Partitions [10], and Boolean Relations [4]. This new representation is general and can be applied to both binary and multiple-valued functions (state machines) **in the same way.** It allows to add and remove variables (functions) in the synthesis process, which would be difficult or inefficient using other representations. **All operations are reduced to set-theoretical operations on DDs.**

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 other. This author believes 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 gaining in importance: binary and multiple-valued, very strongly unspecified functions. We will call this class SUF - *Strongly Unspecified Functions.* Such functions occur in Machine Learning (ML) [14], 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 (BSUF). 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 (MVSUF) are important in ML and KDD areas.

TABLE I

	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

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, 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 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, - 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) [10]. 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, and the origination of a block - which value of input variables it comes from - is lost. This makes some operations in this representation not possible, and some other not efficient. Our other source of inspiration is the concept of Generalized Don't Cares [15]. 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 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, but the other three are new, and 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 cofactor functions, and not their equivalence classes. CDBs represent functions with generalized don't cares, while Luba represents only classical don't cares. 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, and we plan to test other DD packages in the future. 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

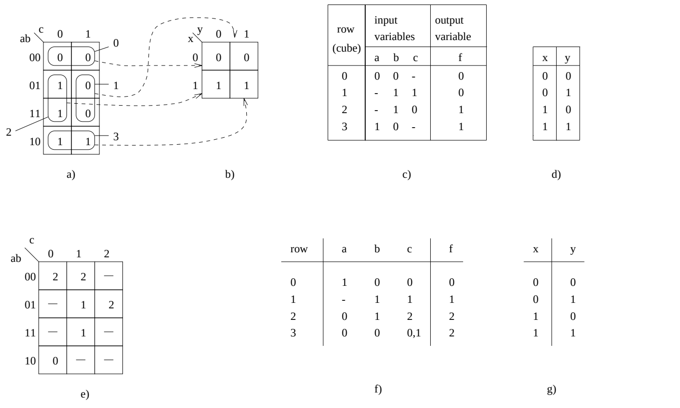

Figure 1. Mapping from primary to secondary variables: (a) original function with primary input (b) swcondary space with secondary input variables x and y (c) table of function f (d) encoding of primary cubes MV function: (e) K-map (f) table (g) encoding to secondary variables

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 [11, 8]. While the authors from [11, 8] 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. 1a. 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. 1b 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. 1a is shown in Fig. 1c, and the encodings of its rows to secondary input variables is shown in Fig. 1d. 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}], [pointer to BDD for {0,3}, pointer to BDD for {1,2}], [pointer to BDD for {0,2,3}, pointer to BDD for {0,1,3}]} ;;var. a,b,c
$Out(F)$ =
{[pointer to BDD for {0,1}, pointer to BDD for {2,3}]}
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. 1f, for an incompletely specified function with standard don't cares from Fig. 1e. It has two binary input variables, a and b; a ternary input variable c, and a 3-valued-output. The encoding of primary cubes with secondary input variables x and y is shown in Fig. 1g.

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}], [pointer to BDD for {0,3}, pointer to BDD for {1,2}], [pointer to BDD for {0,3}, pointer to BDD for {1,3}, pointer to BDD for {2}] }

$Out(F)$ = { [pointer to BDD for {0}, pointer to BDD for {1}, pointer to BDD for {2,3}] }

The following points about CDBs are important:

1. Standard don't care positions are not stored in tables and CDBs, but generalized don't care positions are stored.

2. The definitions of a Binary Cube Diagram Bundles and Multiple-valued Cube Diagram Bundles (mvCDBs) are **exactly the same.** Therefore, the same operations are applied to them.

3. All Boolean operations on CDBs can be easily realized as set-theoretical operations on corresponding DDs.

4. The important concept of a cofactor is calculated using only the set-theoretical operations as well: cofactor CF of F with respect to cube C is calculated as follows: $CF :=$ DD(F)$\wedge$DD(C), $Var(CF) := Var(F)$-$Var(C)$.

5. The operations of derivative, differential, minimum, maximum, k-differential, k-minimum, and k-maximum of a function [2] are also realized. Since all these operations are based on set and cofactor operations, they can be easily realized because of points 2 and 3 above.

6. 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.

7. There is no difference in the representation of primary input variables, auxiliary variables and output variables. Therefore, CDBs are good to represent functions defined on combinational functions of input variables, one can just add new "columns", which means, new sets of "value DDs". 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 intermediate variables such as $A_2 = a \oplus b$, $A_3 = a \cdot b$, or $A_4 = a + b$ by calculationg corresponding functions on BDDs of a and b and storing new items in Var and Inp. This property allows also to: realize algorithms that operate on output functions as on variables; use auxiliary functions for synthesis; and re-use the existing functions in synthesis.

8. We created 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, but will be not discussed here [15].

9. During the incremental reading of the input data in the form of disjoint cubes of primary variables (rows of the "table"), the *encoding of these primary cubes as minterms* in the new space of secondary variables is executed. The goal of this encoding is to simplify all DDs of the CDB. This is done in such a way that the false minterms that are encodings of all primary OFF cubes are grouped near the cell 0...0 (the *minimum minterm* in the space of new variables). Similarly, true minterms of new space that are the encodings of all primary ON cubes are grouped near the 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. 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. This is done for all DDs in parallel.

III. Basic Patterns for Decompositions.

There are only three types of decomposition in the literature that are truly different and that make use of the concept of partitioning of input variables to *bound and free sets*: Curtis Decomposition [6], Steinbach et al (XBOOLE) Decomposition [2], and Perkowski et al (PUB) Decomposition [13, 12, 15]. The decompositions of Ashenhurst [1]; Luba et al [10]; Lai, Pedram et al [9], and many other are just special cases of Curtis decomposition or use only various representations [15]. While most authors differentiate between disjoint and non-disjoint decompositions, the introduced below concept of the *Repeated Variable Maps (RVMs)* allows to explain them in a uniform way, and the CDBs allow to realize all these decompositions uniformly in software.

In RVM, the rows of the map correspond to the Row Variables, and the columns correspond to the Column Variables. As we see, the Row Variables can be represented as $A \cup C$, and the Column Variables can be represented as $B \cup C$. Using Curtis terminology, set $B \cup C$ is a *bound set*, and set $A \cup C$ is a *free set*. If $C = \phi$ 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 bound set is $\{a, c, d\}$ and the variable a is a repeated variable, the new 4 * 8 map will have three variables for columns and two variables for rows (variable a stands in both rows and columns). Half of the RVM are don't cares. If variables a and c were repeated, and $\{a, c, d\}$ were a bound set, the new 8

* 8 map will have three variables for columns, and three variables for rows. It will have 75% of don't cares. As we see, even starting with a completely specified function, by repeating variables, very quickly one has to deal with very strongly unspecified functions. In addition, in ML applications, even the initial data can have more than 99.99% of don't cares. It is than absolutely crucial to be able to represent and manipulate such functions efficiently.

The main observation of our unified and generalized approach is the observation that all decompositions [6, 2, 13, 12, 15, 10] use certain fundamental patterns in cofactors. These patterns can be easily observed in rows and columns of the RVM. Recall please, 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 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 any non-empty subset of zeros, ones, and don't cares). We will call it the F Pattern.

5. Pattern of function $\overline{F}$. 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 other combined patterns of DC, ON, OFF, F, and $\overline{F}$ can be defined. We will say that function has pattern X/Y/Z on columns (rows) if every column (row) cofactor has one of patterns X, Y or Z. Let us observe that if a function has DC/ON/OFF pattern on columns then it is independent on the variables from the free set. Analogically, if a function has DC/ON/OFF pattern on rows then it is independent on the variables from the bound set. Obviously some columns (rows) can be characterized as corresponding to several patterns. For instance, a column may 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, decompositions OR and AND can be found immediately just by analysing one row at a time, and without comparing rows to other rows. *Row EXOR decomposition* exists with respect to the set of row variables RV if for every row its pattern is either F Pattern or $\overline{F} Pattern$. (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). This case is then more difficult than the first two. Analogically one can define the *Column OR*, *Column AND*, and *Column EXOR* decompositions. Row and Column decompositions are also called Weak Decompositions [2]. There exist then *Weak AND, Weak OR*, and *Weak EXOR* decompositions (our understanding of weak and strong follows that from [2], and not the one from U.C. Berkeley). *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. *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. 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 decompositions of this type will be called the *Complex Gate Decompositions*.

An example of the RVM is shown in Figure 2. Fig. 2a 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. 2b. 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. 2c, 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. 2d, 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. 2d. These patterns lead to Strong EXOR Decomposition: $(b\ c) \oplus (a\ +\ b)$ from Fig. 2e. Fig. 2e clearly shows the incomplete patterns from Fig. 2d after their completion with 0's and 1's. Let us observe that from Fig. 2d one can find also $a\ \overline{b} \oplus b\ \overline{c}$, assuming the first three columns from left to have a pattern of $\overline{F}$.

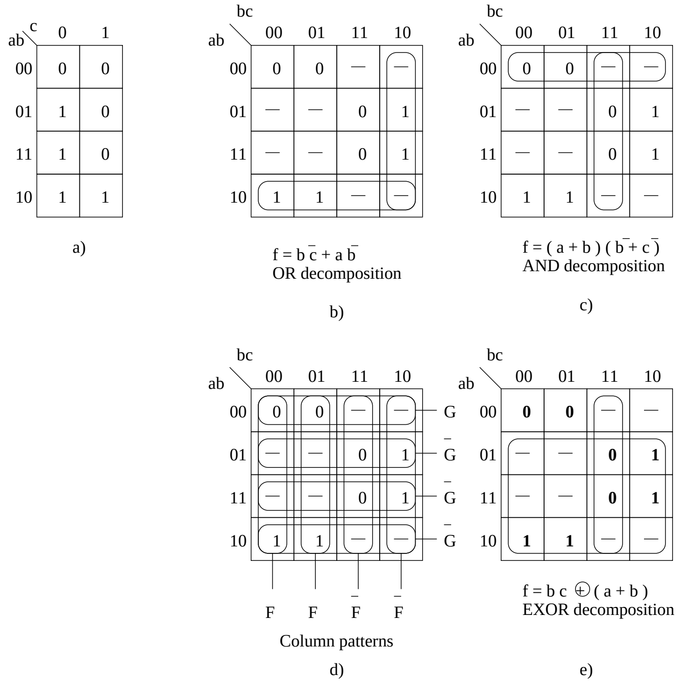

Figure 2. Basic patterns and decompositions.

IV. 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 (section 3); Strong PUB Decompositions; and the Ashenhurst Decomposition. (The PUB decompositions [15] will be not discussed because of lack of space). All these decompositions can be efficiently found in CDBs using cofactors and set-theoretical operations [15]. Let us make a point that the more strongly is the function unspecified and the larger is set C, the more probable is that an Immediate Decomposition will exist for this function.

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

Property 1. Ashenhurst Decomposition $F = H(G(B \cup C), A \cup C)$ with bound set $B \cup C$, free set $A \cup C$, and single-output binary function G exists if all row patterns are: ON Pattern, OFF Pattern, F Pattern and $\overline{F}$ Pattern.

Property 2. Ashenhurst Decomposition with bound set $B \cup C$ and free set $A \cup C$ exists if all column patterns are $F1$ Pattern and $F2$ Pattern, $F2 \neq F1$. In other words, column multiplicity index $\mu = 2$.

Both these properties can be used to verify the existence of Ashenhurst Decomposition. 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 [10, 9, 12, 13, 15, 16]. All these problems are in general NP-hard. However, in case of Ashenhurst decomposition, the problem can be solved by a polynomial algorithm. The following algorithm is based on Property 1.

Algorithm 1.

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. From remaining rows create the set *Remaining_Rows*.
3. Pair_Counter := 1.
4. Put row r_i to LEFT[Pair_Counter] and row r_j to RIGHT[Pair_Counter].
5. Take next row r_s in set *Remaining_Rows* and remove it from set *Remaining_Rows*.
6. Compare r_s with arrays LEFT and RIGHT.

a) If there exists a pair (LEFT[k], RIGHT[k]) such that r_s is incompatible with both LEFT[k] and RIGHT[k], then exit "No Ashenhurst Decomposition".

b) Else if for all v from 1 to Pair_Counter r_s is compatible with LEFT[v] and r_s is compatible with RIGHT[v] then

  if RIGHT[Pair_Counter] $\neq \phi$ then
    Pair_Counter := Pair_Counter + 1;
    put r_s to LEFT[Pair_Counter];
else
    LEFT[Pair_Counter] :=
        Combine_Rows(r_s, LEFT[Pair_Counter]);

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

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

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

Procedure Combine(LEFT,RIGHT,r_s).

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.

Algorithm 1A, based on Property 2, can be applied to mv-output functions and is very similar; Algorithm 1 is usually more efficient, but can be applied only to binary-output functions.

V. Basic Decompositions.

Basic Decompositions are *Curtis Decomposition* and *PUB Decomposition* [15]. They happen more often than Immediate Decompositions. Hower, when executed with large value of *Multiplicity Index* μ they lead to difficult encoding problems and the not necessarily minimum circuits (especially that we are never able to perform exhaustive search of sets A, B and C). We execute, therefore, these decompositions only with small values of $\mu =$ 3,...8. Because of lack of space, we present only the *Curtis Decomposition*: $F = H(G(B \cup C), A \cup C)$ where G is a $\lceil log_2(\mu) \rceil$-output function. Following the approach from [16], given the bound set $B \cup C$ and the free set $A \cup C$, first a fast approximate *graph coloring* of the *Column Incompatibility Graph (CIG)* is found with the nodes corresponding to columns, and the edges to incompatible pairs of columns. Compatible are columns that can be combined (can be completed to the same pattern). Next the encoding method that is similar to the input encoding algorithm for function H from [16] is applied, but which attempts to minimize both H and G. Finally, functions G and H are found.

The reason to use here graph coloring instead of Clique Partitioning 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 *maximum cliques*. Thus, for strongly unspecified functions, the number of rows is exponential, while the graph has only N * (N-1) / 2 edges.

A set covering algorithm is well known that makes use of *essential rows, secondary essential rows* and *dominations of rows and columns*. In case of non-cyclic covering tables, this algorithm finds the exact solution without backtracking. We will present a similar algorithm for graph coloring of the CIG. 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 (this is a *cyclic graph*, a counterpart of a *cyclic set covering table*), a coloring

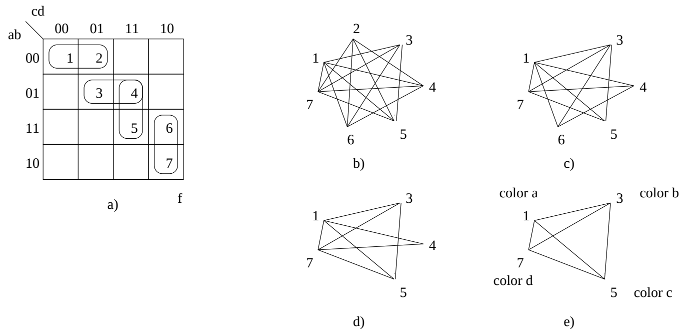

Figure 3. Stages of exact graph coloring

choice is done as in [16]. 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 the popular 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 CIG graphs 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. 3a. 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 (IG)* corresponding to this map is shown in Fig. 3b. The stages of coloring the graph are shown in Fig. 3b - 3e. In Fig. 3b node 2 has neighbors 4,5,6,7, and node 1 has neighbors 3,4,5,6,7. Therefore, node 1 dominates node 2, and 2 is removed from the graph, leading to the graph from Fig. 3c. Now node 6 has neighbors 1,3,4 and node 7 has neighbors 1,3,4,5. Thus node 7 dominates node 6 and node 6 is removed. This leads to the graph from Fig. 3d. Now node 3 has neighbors 1,5,7 and node 4 has neighbors 1,7. So, node 3 dominates node 4 and node 4 is removed. Now, Fig. 3e, the graph is a complete graph. Its nodes are colored with different colors, as shown. With respect to dominations, node 2 is colored with the same color as node 1, which is *color a*, node 4 is colored with *color b*, and node 6 is colored with *color d*.

In CIG, columns of nodes colored with the same color are combined as compatible groups of columns which determines column partitioning and the value of μ. In some covering problems, like in SOP minimization, a group of CIG nodes colored with the same color are not necessarily all compatible, and group compatibility must be checked [5]. It can be proven, however, that in the *Column Compatibility Problem* in Functional Decomposition (for both binary and mv cases), if columns in a set are pairwise compatible the set of all these columns is compatible as well [15]. Then the solution from CIG coloring is always correct. This is, however, no longer true for mv functions with the generalized don't cares [15].

VI. Goal-Oriented Reduction Decompositions.

We will use two stacks. *The OPEN stack* stores the subfunctions to be realized, from which the one evaluated as the easiest to realize is selected for the realization first. *The DONE stack* stores the completely realized subfunctions. These functions can be re-used during reduction decompositions. Also, DONE is used to reconstruct the entire circuit when the OPEN stack becomes empty.

The Goal-Oriented Reduction 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). Function F_G becomes then a goal function.

2. When one of the previously attempted (Immediate or Basic) decompositions of F was "nearly possible". We mean by this that many cofactors in this decomposition had patterns corresponding to a given type of decomposition. In such case, the method from [16] is used to create the goal function F_G and select the *Reduct_Type_Oper*, which defines the type of the decomposition.

In both above cases, the procedure Reduction is next called, to execute the reduction, possibly also to execute the decomposition, and put new subfunctions to $OPEN$.

Procedure Reduction$(F, F_G, Reduct\_Type\_Oper)$.

Function F_G is a goal function, F is the decomposed function. F_C is the correcting function.

1. If Reduct_Type_Oper = 'OR' then
$ON(F_C) := ON(F) \oplus OFF(F_G)$; $OFF(F_C) := OFF(F)$.
If Reduct_Type_Oper = 'AND' then
$ON(F_C) := ON(F)$; $OFF(F_C) := ON(F_G) \oplus OFF(F)$.
If Reduct_Type_Oper = '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 ($F := F_G$ Reduct_Type_Oper F_C) to $DONE$ stack.

VII. The Entire 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. Using various fast and greedy decompositions is better for a very strongly unspecified function than a single method of high complexity. Try simple decompositions first. If various previous attempts failed then try more complex circuit structures and decomposition types.

2. Use DFC (see [14] for definition) to measure the costs of partial solutions and be thus able to compare them.

3. The user can control the algorithm using several parameters.

Besides OPEN and DONE, Algorithm 3 uses the EVAL stack, which stores the decomposition attempts, in order to compare them one with another and select the best decomposition. In making choice decisions, the following parameters of F_i are taken into account: number of true minterms, number of false minterms, number of true cubes, number of false cubes, sets A, B, C together with best patterns for them. number of input variables, % of ON Pattern columns, % of OFF Pattern columns, % of DC Pattern columns, % of $F/\overline{F_i}$ Pattern columns, % of Approximate ON Pattern columns, % of Approximate OFF Pattern columns, % of Approximate DC Pattern columns, % of Approximate $F_i/\overline{F_i}$ Pattern columns, cost parameters, distance, number_of_bound_sets, and other.

Algorithm 3. Total Decomposition Strategy.

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, Reduct_Type_Oper),
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, Reduct_Type_Oper)
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 OPEN = ϕ then return the SOLUTION
else go to 2.

VIII. Numerical Results and Conclusion

Table 2 shows the number of 5/3 CLBs for various strategies. Second line is number of levels. Number in parentheses is the number of EXOR decompositions. For comparison, CLB count and not DFC values are given. Table 3 shows the DFC-optimized solutions with their times (SPARCstation 5).

	s1	s2	s3	s4	s5	s6
bw	27	27	27	27	27	27
i=5,o=28	1	1	1	1	1	1
bench	16	16	16	18	18	16
i=6,o=8	2	2	2	2	2	2
fout	26	26	26	26	26	26
i=6,o=10	2	2	2	2	2	2
f51m	9	9	15	12	13	13
i=8,o=8	3	3	3	3	3	3
rd84	11	11	12	12	12	11
i=8,o=4	3	3	3	3	3	3
test1	70	65	74	70	73	73
i=8,o=10	4(5)	4(4)	4(7)	4(4)	4(7)	4(6)
clip	38	29	48	32	40	40
i=9,o=5	4(2)	4(1)	4(3)	3	4(2)	4(2)
alu2	23	21	25	44	38	24
i=10,o=8	3	3	3	5(2)	5(1)	3
b9	29	46	46	59	50	35
i=16,o=5	4	6(3)	5(2)	7(5)	6(5)	4
duke2	235	259	280	360	292	264
i=22,o=29	9(8)	9(11)	8(14)	8(24)	8(13)	9(9)
misex2	31	32	38	42	45	38
i=25,o=18	3	3	3	4	4	3

We introduced a new representation and a new general decomposition approach for strongly unspecified multi-output functions. Similarly, extensions for mv logic decomposition have been done. This approach opens several new research areas: input variable re-encoding problem to simplify DDs; using new decompositions in machine learning; efficient solving of combinatorial problems (such as graph coloring); bound set encoding and variable partitioning. Although the preliminary results are very good, we believe we will be able to further improve them with more sophisticated bound set partitioning and encoding algorithms. The method should be also compared with other DFC-based approaches to ML that use SOPs, trees, Curtis decompositions, and ESOPs [14].

References

[1] R.L. Ashenhurst, "The Decomposition of Switching Functions", *Proc. Int. Symp. of Th. of Switching,* 1957.

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

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

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

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

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

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

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

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

[10] T. Luba, J. Rybnik, "Algorithmic Approach to Discernibility Function with Respect to Attributes and Object Reduction," *Int. Workshop on Rough Sets,* Poznan 1992.

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

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

[13] 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.

[14] M. Perkowski, T. Ross, D. Gadd, J. A. Goldman, and N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," *This Workshop,* August 1995.

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

[16] W. Wan, and M. 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. Euro-DAC,* pp. 230 - 235, 1992.

name	in	out	cubes	DFC	time
9sym	9	1	158	288	1.2
rd84	8	4	515	384	3.0
rd73	7	3	274	192	1.2
sao2	10	4	133	992	579.8
clip	9	5	271	1344	1972.8
5xp1	7	10	143	576	4.1
b12	15	9	72	544	7691.4
bw	5	28	97	864	0.5
duke2	22	29	406	4892	179.4
cc	21	22	96	768	3.6
cu	14	8	1150	448	2.7
misex2	25	18	101	992	10.2
c8	28	18	166	1120	10.0
alu2	10	6	315	2528	180.7