7 Modern Explanation of Ashenhurst-Curtis Decomposition.

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

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

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

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

7.1 Basic Data Formats and Definitions

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

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

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

```
.i 25
```

specifies the number of input variables (25).

```
.o 20
```

specifies the number of function outputs (20).

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

specifies the names of input variables.

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

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

specifies the names of function outputs.

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

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

```
.type  fr
```

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

```
.end
```

marks the end of the input logic.

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

Fundamental Definitions

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

The ON set is the collection of all ON cubes. The OFF set is the collection of all OFF cubes. The DC set is the collection of all DC cubes. Cube, ON cube, OFF cube, DC cube, ON set, OFF set and DC set are showed in Figure 1.

The *Cube Calculus* is a set of operations applied to cubes and arrays (lists) of cubes. In our description, we will use the Intersection operation, which can be illustrated using maps, arrays of cubes, or BDDs.

Figure 2 shows the rules of Intersection operation. The ϵ is the result of the Intersection of 0 with 1 or 1 with 0. x or X have the same meaning as the "-".

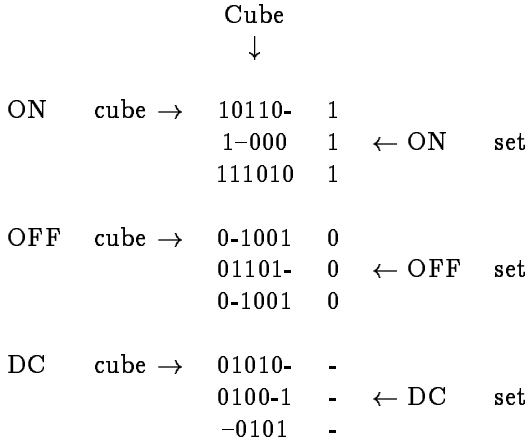


Figure 1: Cubes and cube sets

From Figure 2, the rules are:
$0 \cap 0 = 0$,
$0 \cap 1 = \epsilon$,
$0 \cap x = 0$,
$1 \cap 0 = \epsilon$,
$1 \cap 1 = 1$,
$1 \cap x = 1$,
$x \cap 0 = 0$,
$x \cap 1 = 1$,
and $x \cap x = x$.

Figure 3 shows the general decomposition scheme. Boolean decomposition uses Boolean representation.

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

```
1010xx
100x1x
------
10ϵ01x
```

The *Decomposition Chart* [617, 27, 153] is a chart that is similar to the Karnaugh map with the only difference being that the column and row indexes of the decomposition chart are in the straight binary order, while that of the Karnaugh map are in the Gray code order. Figure 4b shows an example of a decomposition chart.

The corresponding Karnaugh map is shown in Figure 4a. The column of the chart is denoted as a vector of its successive minterms. For example, column 1 in Figure 4b is denoted as a vector [1, 1, 0, 1]. Because there is no essential difference between the Karnaugh map and the decomposition chart, Karnaugh maps will be used instead of decomposition charts for illustration.

The *Bound Set* is a set of variables forming the columns of the decomposition chart. In Figure 4b, { c, d, e } is a bound set.

The *Free Set* is a set of variables forming the rows of the decomposition chart. In Figure 4b, { a, b } is a free set.

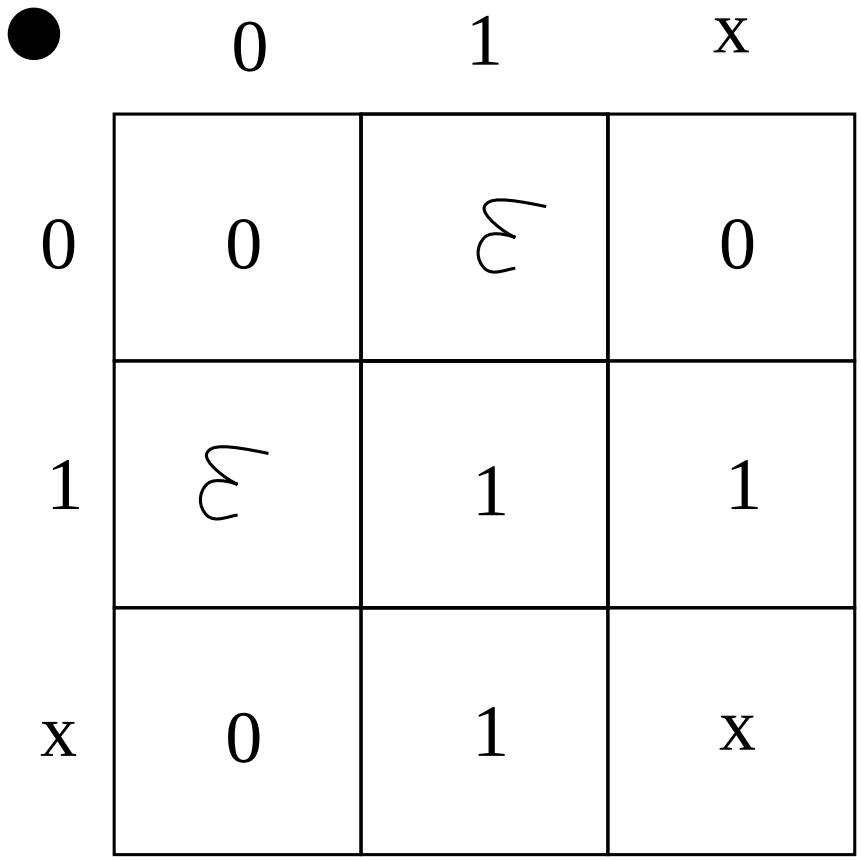


Figure 2: Intersection operation

The *Column Multiplicity* denoted by $\mu(A \mid B)$, is the number of different columns in a decomposition chart. In $\mu(A \mid B)$, A stands for the free set, B stands for the bound set. For example, in Figure 4b, $A = \{a, b\}, B = \{c, d, e\}$ and $\mu(A \mid B) = \mu(ab \mid cde) = 3$.

If two horizontally corresponding cells in two columns of the decomposition chart are (0,0), (1,1), (0,x), (1,x), (x,0), (x,1) or (x,x), these two cells are called *compatible*. If all the corresponding cells in two columns are compatible, these two columns are called *compatible*. Otherwise, they are called *incompatible*. In Figure 4b columns 1 ([1, 1, 0, 1]) and 6 ([1, x, 0, x]) are compatible, while columns 5 ([0, 1, 1, x]) and 6 ([1, x, 0, x]) are incompatible. In this explanation, we will use maps, Cube Calculus, or BDDs, to test whether two columns (called also *column functions*, *column cofactors* or *K-map loops*) are compatible or not. The formula [488] to test the compatibility of two columns (columns i and j) is:

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

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

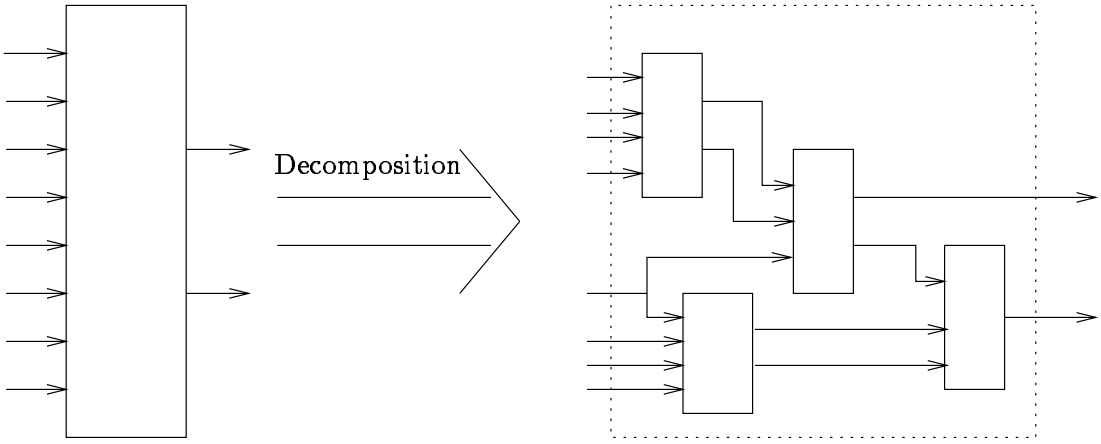


Figure 3: Decomposition

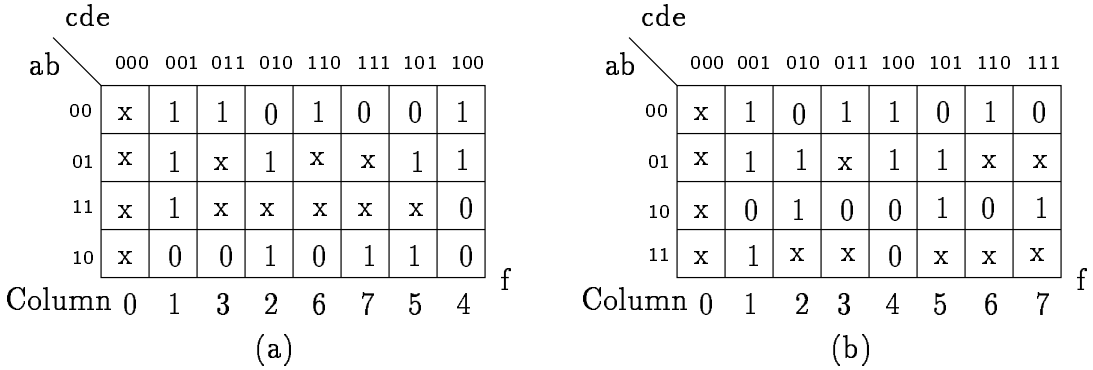


Figure 4: Karnaugh map and decomposition chart

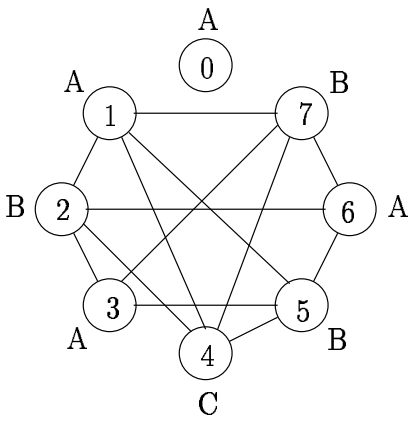


Figure 5: Example of an incompatibility graph

how functions ON and OFF are realized. For instance, ON and OFF are represented as cube arrays in TRADE.

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

Figure 5 shows an Incompatibility Graph corresponding to the decomposition chart in Figure 4b.

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

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

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

In Figure 5, 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.

7.2 Decomposition of the Incompletely Specified Functions.

In this section, the functional Boolean decomposition of incompletely specified functions will be presented. The basic ideas follow [27, 153, 154, 549] and the general approach based on graph coloring is patterned after [488, 678, 464].

Curtis has described the decomposition of completely specified functions in [154]. He proved the *fundamental theorem*:

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

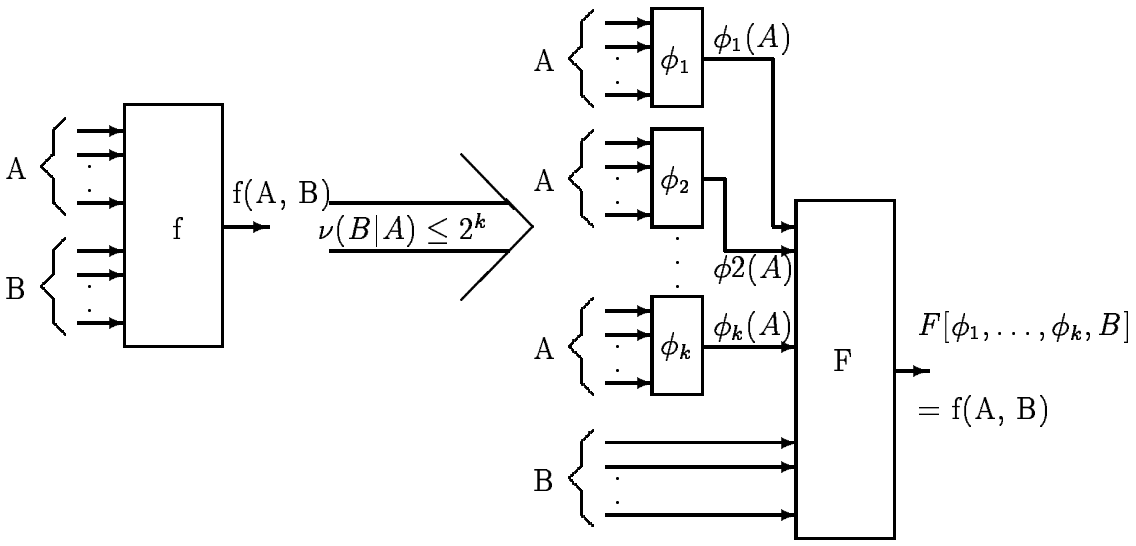


Figure 6: Curtis decomposition

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

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

The graphical representation of this theorem is shown in Figure 6.

From Figure 6 we observe that, after decomposition, the big block f is broken into several smaller sub-blocks ϕ_1, ϕ_2, ..., ϕ_k and F.

The essential problem of the decomposition of incompletely specified function is how to assign DC outputs as 0 or 1 to minimize the column multiplicity. Because the number of colors in a properly colored incompatibility graph is the same as the number of different columns (column multiplicity) in a decomposition chart [488], the problem of finding the smallest column multiplicity can be transformed into that of performing the proper graph coloring to find the smallest number of colors. We use the following criterion:

Assume n to be the expected number of output variables from the blocks with bound set as inputs, and n be less than the number of variables in the bound set. If the column multiplicity is equal to or less than 2^k, and k is less than or equal to n, the decomposition is *successful* (or the function is *decomposable*) for this bound set under the expected value of n. Otherwise, the function is *non-decomposable* for this bound set under the expected value of n.

After a successful decomposition, the number of input variables of each sub-function (decomposed blocks, like ϕ_1, ϕ_2, ..., ϕ_k *and* F in Figure 6) is decreased, and the complexity of each sub-function is decreased as well. This will be illustrated with an example.

Figure 7a is the Karnaugh map of function f with don't care outputs. For instance, one may intend to decompose the function f into several sub-functions (denoted by L, M and N in Figure 7c) with the input variables of each sub-function less than or equal to four.

According to the rules presented above, the incompatibility graph is created as shown in Figure 8.

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

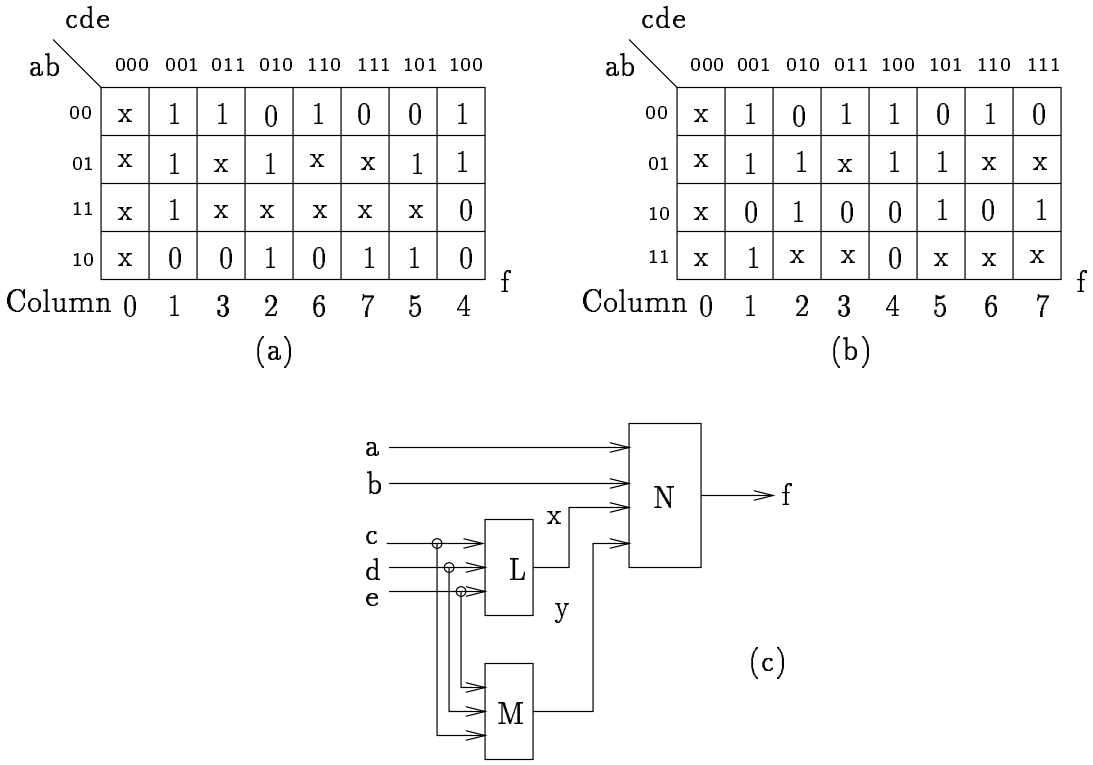


Figure 7: Karnaugh map, decomposition chart and the expected decomposition

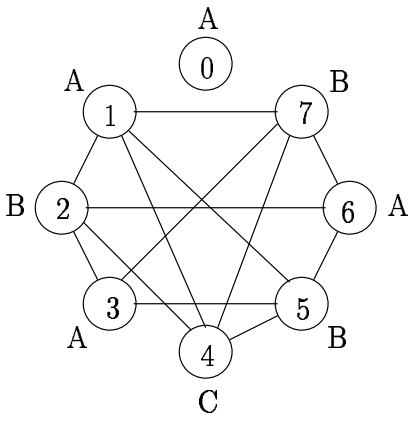


Figure 8: Incompatibility graph

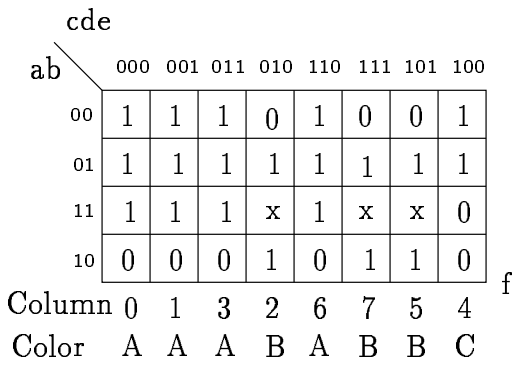


Figure 9: Final don't care assignment

For example, columns 0, 1, 3 and 6 in Figure 7a are combined and replaced by a new vector [1, 1, 1, 0] as shown in the final don't care assignment in Figure 9. In the above example, we have chosen the variables a and b as the free set and variables c, d and e as the bound set. This partition results in a successful decomposition in the sense of the column multiplicity less than or equal to three. In Figure 7c, x and y are the encoded outputs of the bound set, two variables are enough for three different columns ($3 \leq 2^2 = 4$). The encoding of Bound set will be discussed in the next section.

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

7.3 Bound Set Encoding.

There are many methods [722, 724] to implement the decomposed blocks (blocks L, M and N in Figure 7c). Here we introduce an algorithm to encode the bound set that aims at simplifying the block N. The encoding algorithm assigns adjacent codes (Gray code) to the similar columns. This increases the number of large cubes in the block N. The similarity (or difference) between two columns is measured by the so-called *Difference Factor*. The more similar the two columns, the lower the value of the Difference Factor. The *Difference Factor* is the number of minterms in which the two columns are not identical. The Difference Factor between the i-th and j-th columns is:

Difference Factor = minterm_size(ON(i) $\cap$ OFF(j)) + minterm_size(OFF(i) $\cap$ ON(j))

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

In this example, bound set {*c, d, e*} forms eight columns as shown in Figure 7a. After graph coloring, three different columns were found. These three columns are: [1, 1, 1, 0] corresponding to color A, [0, 1, x, 1] corresponding to color B and [1, 1, 0, 0] corresponding to color C as shown in Figure 9. We

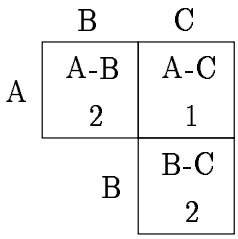


Figure 10: Similarity Factor Table

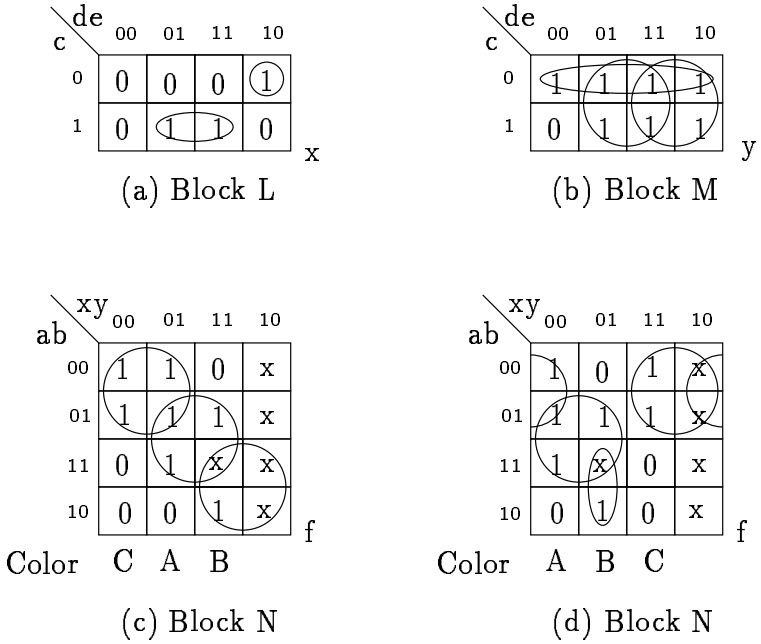


Figure 11: Decomposed Karnaugh maps

introduce two new variables x and y to encode the bound set $\{c, d, e\}$. First let us calculate the Difference Factors. The Difference Factor between columns corresponding to color A and B has a value of 2. The value of the Difference Factor between columns corresponding to color A and C is 1. And the value of the Difference Factor between columns corresponding to color B and C is 2.

A Table of Difference Factors is created as shown in Figure 10.

Because the Difference Factor between columns corresponding to color A and C is smaller (with a value of 1), these two columns are put in adjacent positions, as shown in Figure 11c. Let us code the column corresponding to color C as 00, the column corresponding to color A as 01 and the column corresponding to color B as 11 as shown in Figure 11c, which is the Karnaugh map of the block N. Color A has the code 01, which means that x is equal to 0 and y is equal to 1 for all columns with the color A. These columns are 000, 001, 011 and 110 in Figure 9, therefore the cells 000, 001, 011 and 110 of the Karnaugh map in Figure 11a, which is the Karnaugh map of the block L, are filled with 0 because x is equal to 0. The same cells in Figure 11b, which is the Karnaugh map of the block M, are filled with 1 because y is equal to 1. The same way, color B has the code 11, which means that both x and y are equal to 1. Columns 010, 111, and 101 correspond to color B, therefore the cells 010, 111 and 101 of the Karnaugh maps in both Figure 11a and (b) are filled with 1. Color C has the code 00, both x and y are equal to 0, column 100 correspond to color C, the cell 100 of the Karnaugh maps in

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

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

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

Figure 12: Pseudo-code of bound set encoding

both Figure 11a and (b) are filled with 0.

Two variables can encode up to four columns ($2^2 = 4$). There are only three columns, corresponding to color A, B and C, that need to be encoded in our example. We fill the remaining column (column 10 in Figure 11c) with don't cares (*DC column*). The existence of this newly introduced DC column will further simplify the block N. This example shows that even if the input function is completely specified, the algorithm may introduce DCs in the middle of the process, which is very useful for the simplification of the later stages.

Figure 11d shows a Karnaugh map of an alternative implementation for the function f, which uses the natural order of the colors. Clearly, the Karnaugh map in Figure 11c is simpler than that in Figure 11d.

The pseudo-code for bound set encoding is shown in Figure 12. The BLIFF format of the result is as follows:

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

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

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

```
.model example 16
```

specifies the name of the model (example).

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

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

```
.outputs f
```

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

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

with the following ON set describes the logic of a node (sub-block L in Figure 7c). The input variables to this node are c, d, e, and the output variable is x.

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

with the following ON set describes the logic of a node (sub-block M in Figure 7c). The input variables to this node are c, d, e, and the output variable is y.

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

with the following ON set describes the logic of a node (sub-block N in Figure 7c). The input variables to this node are a, b, x, y, and the output variable is f.

```
.end
```

marks the end of the model.

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

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

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

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

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

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

There are basically three approaches to the Variable Partitioning problem:

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

8 Roth-Karp Decomposition

8.1 Introduction

Roth's serial decomposition technique, [549], can be illustrated as follows. Given $F(a, b, c, d, e)$ a selection is made of free and bound variable sets (these sets are not necessarily disjoint) and tests are made for the existance of single-output or multiple-output predecessor functions. Let us assume a single-output predecessor $g_1(a, b)$, as illustrated in Figure 13.

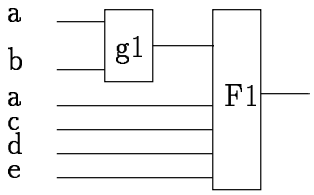


Figure 13: Serial decomposition with single output predecessor, one stage

Function $F_1(g_1(a, b), a, c, d, e)$ is the image of F and its ON-OFF sets can be computed from the ON-OFF sets of F in terms of the outputs $g_1(a, b), a, c, d$, and e (as shown in a later example). Let us assume the simple disjoint case, i.e. $g_1(a, b)$ is a single-output function. The image of F will exist if the following conditions hold :

Given functions $F(B, A)$ and $g_1(B)$, there exists a function f such that:

$$F(B, A) \leq f(g_1(B), A) \tag{1}$$

(i.e. for each set of inputs B, A where the value of F is defined, f has the same value) if and only if:

1. input cubes b_i and b_j, are incompatible with respect to F $(F(b_i, a_k) \neq F(b_j, a_k))$, where same a_k implies $g_1(b_i) \neq g_1(b_j)$

   or equivalently,

2. $g_1(b_i) = g_1(b_j)$ implies compatible input cubes b_i and b_j, with respect to F, $(F(b_i, a_k) = F(b_j, a_k))$

Function f is then called the image of $F(B, A)$, such that for all cubes b_i, a_j where the value of F is defined, $f(g_1(b_i), a_j)$ has the same value. Compatible cubes intersect and $\neq$ denotes different values of a scalar Boolean function.

If F_1 is still not realizable with a library component, its image $F_2(g_2(a, c), g_1, e, d)$ is formed if a single-output predecessor $g_2(a, c)$ exists.

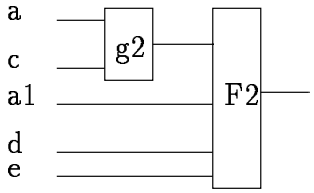


Figure 14: Stage two decomposition F_2

The ON-OFF sets of F_2 can be also derived from the sets for F_1 in terms of g_2, g_1, e, and d; if F_2 is a realizable component then this is the last step of the decomposition process, otherwise the image of F_3 has to be obtained, etc.

	ON		OFF
	a b c d		a b c d
$u_1 =$	1 0 1 x	$v_1 =$	1 1 x 1
$u_2 =$	1 x 1 0	$v_2 =$	1 x 0 x
$u_3 =$	0 1 x x	$v_3 =$	x 0 0 x

Table 1: ON OFF cubes for bound set $B = \{ab\}$ and free set $A = \{cd\}$

The test for existance of a single-output predecessor is as follows: We want to test if $F(a, b, c, d)$ has an image $F_1(g_1(B), A)$ where the bound set $B = \{a, b\}$ and the free set $A = \{c, d\}$. Let us arrange the ON and OFF array of F as in Table 1.

Let's denote the set of ON cubes by $U = \{u_1, u_2, u_3\}$ and the set of OFF cubes by $V = \{v_1, v_2, v_3\}$. Any two cubes u and v, one in the ON set and the other in the OFF set, must be mapped to distinct cubes in the image of F; if u and v have common free-set parts $u - B, v - B$, then the bound-set parts $u - A, v - A$, are the only distinguishing features of u and v and $g_1(u - A) \neq g_1(v - A)$ must hold. Otherwise some cubes of the image F_1 would appear in both the ON and OFF sets of F_1.

In the above example:
$u_1 - A = 10$, $u_2 - A = 1\text{x}$, $u_3 - A = 01$;
$u_1 - B = 1\text{x}$, $u_2 - B = 10$, $u_3 - B = \text{xx}$;
$v_1 - A = 11$, $v_2 - A = 1\text{x}$, $v_3 - A = \text{x}0$;
$v_1 - B = \text{x}1$, $v_2 - B = 0\text{x}$, $v_3 - B = 0\text{x}$;
The above requirement can be also stated as follows: the bound-parts of u and v are incompatible if their free-parts intersect.

For example $u_1 - B$ and $v_1 - B$ intersect at 11; $F(1011) = 1$, $F(1111) = 0$. Thus we must have:
$F_1(g_1(10), 11) \neq F_1(g_1(11), 11)$ and this requires $g_1(10) \neq g_1(11)$.

All pairs of cubes u, v must be considered to determine if a decomposition with g_1 exists. A systematic way to do this is to list all pairs which intersect and the corresponding incompatible bound-part cubes. Figure 16 shows a graph of compatible values of predecessor $g_1(a, b)$.

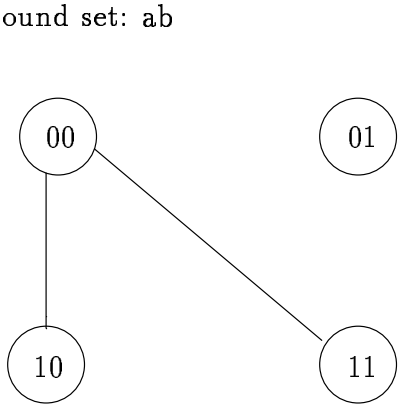


Figure 15: Compatibility Graph of $g_1(a, b)$ values

This graph can be interpreted as the following set of implications :

- $u_1 - B$ intersects $v_1 - B$ implies $g_1(10) \neq g_1(11)$

- $u_3 - B$ intersects $v_1 - B$ implies $g_1(01) \neq g_1(11)$
- $u_3 - B$ intersects $v_2 - B$ implies $g_1(01) \neq g_1(1x)$
- $u_3 - B$ intersects $v_3 - B$ implies $g_1(01) \neq g_1(x0)$

The above Compatibility Graph from Fig. 16 cannot be partitioned into two cliques of compatible elements, therefore $g_1(a, b)$ has 3 distinct values:

$g_1(10) \neq g_1(11) \neq g_1(01)$.

This implies that $g_1(a, b)$ can not be a single-output binary function. The Roth-Karp algorithm [549] considers a multiple-output predecessor at this point, such that the number of mutually compatible bound-part sets $\leq 2^t$ where t is the number of outputs of the predecessor: $F_1(g_1(B), g_2(B), \ldots, g_t(B), A)$.

In the above example there were three compatible bound-part sets 11, 10, and 01. Hence a decomposition can be found if a two-output predecessor block is used $3 < 2^2, t = 2$. Two-output predecessor block, as required by Roth's test for single-output/Ashenhurst predecessor above is shown in Figure 16.

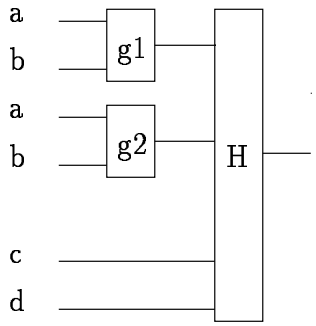


Figure 16: Two-output predecessor block in serial decomposition

Another way to state the above requirement is that the ON set of F must be identical to the ON set of H (image of F). The ON terms of F above can be mapped to the distinct values of $g_1(ab)g_2(ab)cd$ (minterms in the ON set of the image H). The truth table for F is as in Table 2.

a	b	c	d	F
1	0	1	0	1
1	0	1	1	1
1	1	1	0	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1

Table 2: Truth table for F

With a, b as the bound set variables, the decomposition chart shows that one possible mapping is $ab = 10$ to $g_1g_2 = 00$, $ab = 11$ to $g_1g_2 = 01$, and $ab = 01$ to $g_1g_2 = 10$. The truth table for H can then be filled in as in Table 3.

Each row corresponds to a row in the table for F, with variables ab mapping to g_1g_2 as specified above. This represents an arbitrary encoding of the columns in the decomposition chart of F given by Kmap from Table 4.

The relation between the input cubes of the bound set variables ab and g_1g_2 is shown in Table 5.

Columns C_1, C_3 form compatible $class_1$, encoded with $g_1g_2 = 01$, C_2 is $class_2$ with $g_1g_2 = 10$, C_4 is $class_3$ with $g_1g_2 = 00$. H depends on the predecessor functions g_1 and g_2 according to:

$$H = g_1\overline{g_2} + \overline{g_1}c\overline{d} + \overline{g_2}c$$

Also $g_1(a,b) = \overline{a}b$ and $g_2(a,b) = \overline{ab} + ab$.

The final image:

$H = \overline{a}b + (a + \overline{b})c\overline{d} + (a\overline{b} + b\overline{a})c$

The above observations have been stated formally in two theorems by Roth and Karp [549]:

Theorem 8.1 *Given $F(B, A)$ and $g_1(B)$, there exists $F_1(g_1(B), A) = F$ iff for all cubes b_i and b_j included in B, b_i incompatible (non-intersecting) with b_j implies $g_1(b_i) \neq g_1(b_j)$.*

Theorem 8.2 *If the cubes of bound set B can be partitioned into k mutually compatible sets then a decomposition with a t-output predecessor exists, where $k \leq 2^t$ and*

$$F = F_1(g_1(B), \ldots, g_t(B), A)$$

A circuit synthesis procedure was presented in Karp [318], based on a-priori knowledge of all the predecessor (library) functions $g_1(), g_2(), ..., g_k()$. This requirement can not be maintained in a general decomposition framework. Instead, any form of the predecessors $g_i()$ can be assumed and the DFC minimization strategy should rely on bound and free set evaluations for decomposition forms such as simple (column multiplicity = 2) disjoint, iterative disjoint, and complex disjoint and non-disjoint. The image of F should be constructed at each level as in the example above.

The iterative disjoint form has been defined in Karp [318], pp.300-303 as shown in Figure 17, with single output predecessor at each stage.

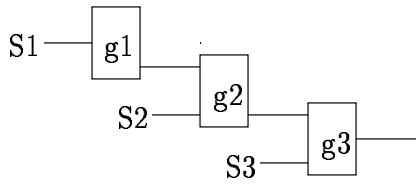


Figure 17: Iterative serial disjoint decomposition

The $S1, S2$, and $S3$ are disjoint sets and cover all variables of F, and all $g_i()$ functions are single output.

9 Bibilo-Yenin Decomposition.

Ideas of Ashenhurst and Curtis, and particularly Roth and Karp have been re-implemented by Bibilo and Yenin before 1987 and published in a book [70]. However, the comparison of the quality of their method with the quality of similar approaches in the West is difficult, since they did not use benchmark functions. In the book there is no critical evaluation of the proposed improvements, no references to recent Western literature, and no comparisons to Western programs. They use more efficient algorithms for partitioning and column compaction which is performed together with the encoding. This is perhaps a very good idea worth further study. It is somehow similar to the concurrent state minimization and state assignment, as proposed in [364, 124].

Bibilo and Yenin report to solve functions of 40 input variables, and they tested their programs on randomly generated functions. Since randomly generated functions are very hard, the performance of their approach seems to be good, but it is hard to assess. Some of their algorithms should be implemented and compared.

10 He and Tolkersen Decomposition.

In 1993 He and Tolkersen proposed a new Boolean extraction algorithm based on K-maps. Because of K-map size restrictions, their algorithm is not very practical as presented. However, there are certain new ideas in their paper which can be perhaps adapted to one of the modern representations of Boolean functions, such as BDDs or partitioned representations. This will be discussed in the sequel.

11 Spectral Approach of Shen and McKellar.

The main drawback of using the basic method of Disjoint Decomposition is to check $2^m - m - 2$ "maps" for an m-input function. However, it is possible to devise algorithms that use necessary conditions for the existence of a decomposition in order to prune out certain combinations of input variables which do not belong to bound sets of any decompositions. This effectively speeds up the procedure in all but pathological cases.

The search for a fast algorithm for the disjunctive decompositions of switching binary functions was studied by Shen at el [607, 608, 609]. They presented a fast algorithm based on testing a necessary condition for decomposability of switching functions. During this process, candidate bound (CB) sets were generated. The CB sets are a smaller subset of all possible bound sets. By performing an additional minor test on CB sets, Shen et al were able to determine the decomposability of a switching function in disjunctive form. Shen and McKellar [607, 608, 609] devised an algorithm that detected candidate partitions for disjoint decompositions of logic functions but required further testing of candidates, even though the set of candidates contained far fewer partitions that the original set of all possible partitions. They also showed that Reed-Muller (RM) canonical form was easier to test for decompositions than the disjunctive normal form (DNF). Thus, the idea of a speedup resulting from applying a new function representation has been further reinforced.

g_1	g_2	c	d	H
0	0	1	0	1
0	0	1	1	1
0	1	1	0	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1

Table 3: Truth table of H, for $g_1(a,b)$, $g_2(a,b)$

cd-ab	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	-	1	0	1
10	-	1	1	1
	C_1	C_2	C_3	C_4

Table 4: Kmap

a	b	g_1	g_2
0	0	0	1
0	1	1	0
1	0	0	0
1	1	0	1

Table 5: Bound set encoding table

Tests	Rule No. 1	Rule No. 2
Urine analysis		x
Blood test	y	
Glucose tolerance test	$\overline{z}$	$\overline{z}$
Positive Diabetes	[1 - 0.6]	

Table 7: Fuzzy decision table

- variable z be represented by the tolerance level of sugar by the body based on the results of a glucose tolerance test.

The rules that determine if a person has a diabetes are:

1. Rule 1. If a person has a high level of sugar in the blood and a low level of tolerance, this person may have diabetes.

2. Rule 2. If a person has a large amount of sugar in the urine and a low level of tolerance, this person may have diabetes.

The value in the table represents the grades of membership of the function which determines when someone has diabetes.
The fuzzy valued switching function which represents this decision table is

$$f(x, y, z) = y\overline{z} + x\overline{z}$$

If we let $G(x, y) = x + y$ then this function can be decomposed as follows:

$$f(x, y, z) = (x + y)\overline{z} = G(x, y)\overline{z} = F[G(x, y), \overline{z}]$$

Since the limit of f for the diabetes condition is a grade membership of 0.6, if $G(x, y)$ has a grade membership less than 0.6 then the glucose tolerance test need not be administered to deduce that the person does not have diabetes.[221]

Concluding, Example 19.9 shows that decomposition of a fuzzy function works correctly, but there is more considerations to be made in order to have a function decomposable. This makes decomposition even more rare to occur than a binary decomposition. When decomposition does work the amount of operations is reduced. We are investigating improvements and generalizations to the presented methods of fuzzy function decomposition.

20 The Decomposition of Continuous Functions.

The only paper about generalizing the AC decomposition to continuous functions is one by Ross et al. [546]. However, the method of orthogonal decision diagrams for continuous variables, developed by Pierzchala, Perkowski and Grygiel, should be also easily adaptable to AC decomposition, because of the close link of orthogonal and AC decompositions [510].

Perkowski, Pierzchala and Grygiel developed methods to realize Galois Field operations for small fields in analog/mixed *Field Programmable Analog Array - FPAA*. Paper [510] gives an example of realization of GF(4) operations. In this way, Galois Field type of expansions can be realized in hardware in an analogous way to Galois(2) expansions, i.e. AND/EXOR canonical forms. The regular array structure of the FPAA allows also to realize structures based on Orthogonal Expansions which are not Galois Expansions. The structure of the corresponding circuit resembles a PLA as well, and has columns corresponding to orthogonal functions over $GF(2^n)$, where the functions are realized by cascades of Galois Addition and Galois Multiplication gates, and pass gates. Each column is Galois-multiplied by a constant, and next these values are Galois-added in rows. Thus, the column is a generalization of an AND-term of a PLA, and a row is a generalization of an OR-term (interestingly, a very similar array structure allows also to realize fuzzy logic circuits). Further, the trellis expansions of functions can be also generalized to continuous logic, where obviously variable repetition is required in a general case. Several other generalizations to other algebras, including fuzzy logic, as well as their realizations in regular array structures have been also described in another paper (not published yet). This is a fruitful area of our current research.

In another, yet unpublished paper, they observe that general decision diagrams, universal cells, regular structures and decompositions are so general, that they apply not only to binary logic, but also to multiple-valued logic, Galois Fields and other algebras (fields and some rings). They are also applicable to continuous, i.e. analog logic, for instance fuzzy logic. This helped to create the concept of universal analog-digital "tissue" which would allow to realize a very wide category of logics and other formal systems in hardware. A practical realization of these efforts is the concept of Field Programmable Analog (Mixed) Array which is for analog (mixed) circuits what FPGA is for binary circuits [510].

Perkowski and Hong [486] and Luba and Lasocki [397] presented a case of decomposing a function with various numbers of values in each variable. The recent research of the author is on the expansion of these methods to include also continuous and fuzzy variables. It is done using generalized decision diagrams that have nodes corresponding to binary, multiple-valued, continuous and fuzzy variables.

21 The Generalized Decomposition.

Although Pattern Theory (PT) group interests are only in combinational logic decomposition, from the mathematical point of view the decomposition of Finite State Machines is a close issue. Historically, the decomposition methods have been applied to state machine design for many years, starting with works of Hartmanis and Stearns in 1960's and 1970's. Hartmanis and Stearns created a very general and computationally convenient mathematical apparatus, called *"Partition Algebra"*, and proved powerful theorems about parallel, cascade and other decompositions of finite state machines. It was, however, observed that these methods are inefficient for practical state machines, and rarely a good decomposition exists for a random machine. In general, the criticism and the reception of these methods in industry were initially similar to that of the reception of AC decomposition in case of combinational circuits. However, there has been recently an increased interest in state machine decomposition methods in companies such as Philips, Siemens, and Synopsys and in top research universities. Moreover, there exists a feeling that new decompositions and techniques can be developed that will be useful for decomposing both switching functions and state machines [312, 313].

The partition algebra has been used by Luba et al for combinational decomposition, leading to a totally new approach and practical results of a very good quality. Recently, a new generalized theory of decomposition of both state machines and combinational logic has been developed by Jozwiak et al. [312, 313]. It is yet uncertain to us, whether the theory of Jozwiak does indeed bring new practical decompositions in case of combinational circuits (other than a common presentation framework with sequential circuits). However, it definitely allows to create much more general decompositions of sequential machines by considering more decomposition types, including some quite complex new types that are able to perform decomposition ("splitting") of internal states.

It is possible to extend the current model of Pattern Theory Group to state machines. This would be the simplest and most natural extension of the current model, because the state machines include the combinational logic, and there is no popular model of computability model that would be located in the hierarchy between the two. If ever the Pattern Theory group would decide to make a state machine generation their most basic model of algorithm Machine Learning then the classical model of state machine decomposition would be is a good candidate. Especially the "Generalized Decompositions" of Jozwiak are good candidates because of their similarity to the current decomposition model investigated by the Pattern Theory group, and with respect to the use of the generalization of Luba's partition-based decomposition model at PSU.

22 Representation of Functions.

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

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

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

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

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

1. Karnaugh maps: Ashenhurst, Curtis, Ross et al, Torkelson, Luba.
2. Cubes: Hwang and Owen, (IEEE Transactions on CAD, May 1992), Karp [549], Perkowski/Brown [486], Wan/Perkowski [678], Bibilo [70], Steinbach [83] (a variant called TVL - ternary vector lists that has a different coding of symbols 0, 1, and X and uses disjoint cubes).
3. Canonical Positive Polarity Reed-Muller Forms: Shen and Kellar, Trachtenberg and Varma.
4. Walsh Transforms: Karpovsky, Trachtenberg and Varma, Stankovic.
5. OBDDs: Chang and Marek-Sadowska [132], Lai, Pedram, and Vrudhula [351], and Sasao [580].
6. Edge-Valued Binary Decision Diagrams (EVBDDs) by Lai, Pedram, and Vrudhula [351].
7. Partitions of minterms: Luba, Selvaraj.
8. Partitions of cubes: Luba [398], Selvaraj [601, 602].

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

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

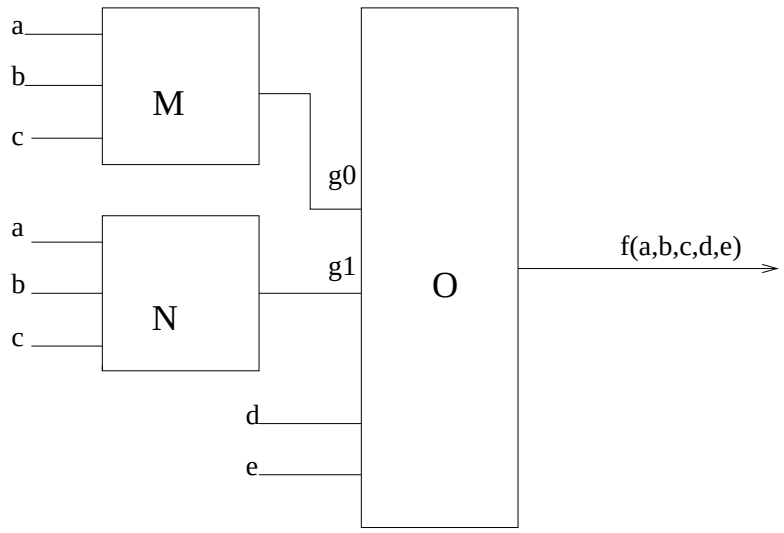


Figure 45: A schematic diagram of the decomposition

4. Array of Cubes representing an ESOP.

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

1. Maps, expressions Walliuzzaman and Vranesic, [677].

2. Expressions, Wojcik and Fang, [211].

3. Maps, Abugharbieh, [7].

4. partitioned minterms: Luba, [398].

5. multiple-valued cubes and minterms: Sasao 1989, [579].

6. Multiple-Valued diagrams [579].

Single output functions have been discussed in: Ashenhurst, Curtis, Ross, Steinbach. Multi-output functions have been presented in: Luba [398], Pedram [351], He and Torkelson [285], and Karp [318].

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

22.1 Various Types of Decision Diagrams.

Reduced, ordered Binary Decision Diagrams (BDDs) provide a compact and canonical representation of Boolean functions. These data structures are used in the decomposition program by Lai, Pedram and Vrudhula [351]. When the function is incompletely specified, there are two ways to represent it using BDDs. The first is to generalize the BDD to 'Ternary DDs' that have three terminal nodes, standard nodes 0 and 1, and a '*' node corresponding to the don't care value of the function. This method was applied by Lai, Pedram and Vrudhula [351].

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

Pedram et al use also their new data structure, called Edge-Valued Binary Decision Diagrams (EVBDDs) [350]. EVBDDs provide a representation of Boolean functions over the integer domain and have been shown to be useful for verification of arithmetic functions. Another similar and even more powerful decision diagrams have been recently created by Brayant and Chen, 1994, and called Binary Moment Diagrams. These topics are the areas of recent research.

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

Lai, Pedram and Vrudhula proposed in DAC '93 one of the most successful approaches to AC decomposition [351]. Their approach is applicable to both disjunctive and non disjunctive decompositions, both completely and incompletely specified Boolean functions, single-output and multi-output.

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

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

Consider a simple example of a function $f(a, b, c, d, e)$. Let $bound\_set = \{a, b, c\}$ and $free\_set = \{d, e\}$. Here $k = 0$ hence a disjunctive decomposition is applied. Figure 45 shows a schematic diagram of this decomposition.

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

Notations

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

22.2.1 Disjunctive Decomposition on BDDs

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

Example 22.1

Let $f = x_0x_1x_2x_3 + x_0x_1x_2\overline{x}_3\overline{x}_4 + x_0x_1\overline{x}_2\overline{x}_4 + x_0\overline{x}_1x_2\overline{x}_4 + x_0\overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_0x_1x_2\overline{x}_4 + \overline{x}_0x_1\overline{x}_2\overline{x}_3 + \overline{x}_0\overline{x}_1x_2x_3 + \overline{x}_0\overline{x}_1x_2\overline{x}_3\overline{x}_4 + \overline{x}_0\overline{x}_1\overline{x}_2\overline{x}_3$. The corresponding decomposition chart is as shown in Fig 46.

Looking at Figure 46 we can see that there are 3 distinct columns. Hence we know that there is a simple disjunctive decomposition with the chosen bound and free sets. But how can this be found by looking at the BDD?

The Pedram's method is this:
Draw the BDD with such a variable ordering that the variables from the Bound Set are on the top of the BDD, and then they are followed by the variables from the Free Set. Figure 47 shows the BDD of our example.

Figure 48 shows how the columns of the decomposition chart correspond to paths along the BDD. If one looks at path 101 along the BDD it results in $\overline{x}4$ which is the same as column 101. This illustrates the correspondence of the decomposition chart and the BDD. Figure 49 explains what a cut set is.

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

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

Figure 46: A Decomposition Chart to Example 21.2

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

Example 22.2 Let us assume a Boolean function of 5 variables f(a,b,c,d,e). Let {a,b} be the bound set and {c,d,e} be the free set. Assuming a simple disjunctive decomposition with two g functions, the decomposition would look like shown in Figure 50.

M and N are called the predecessor blocks and O is called the successor block. The entire process is described in Figure 51, 52 and 53.

22.2.2 Finding predecessor and successor blocks.

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

STEP 1: Construct a BDD.

STEP 2: Find the best cut of the BDD.

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

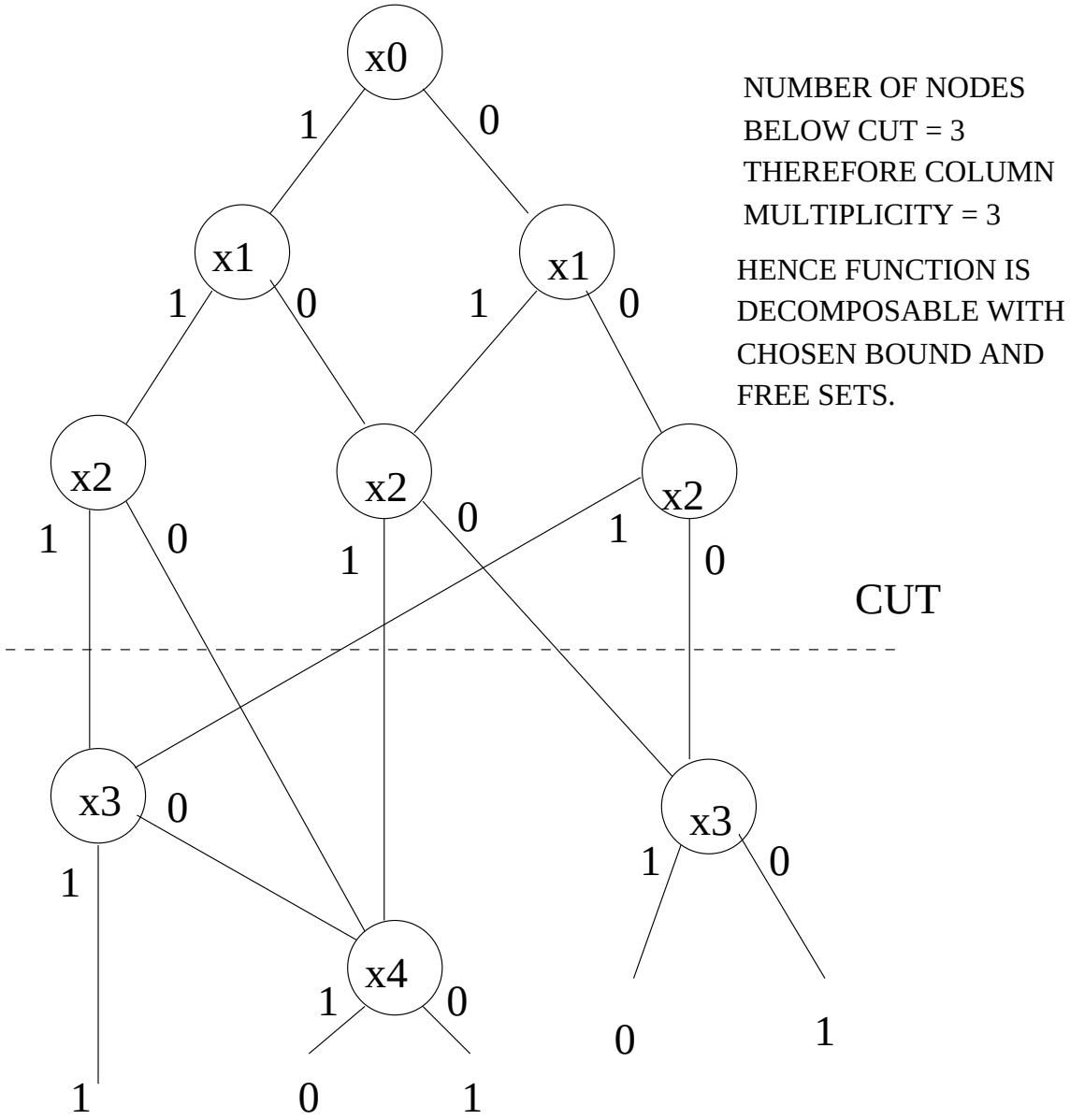
Let g=10, h=01, i=00. Since we have used two bits for encoding we have two g functions $g0$ and $g1$ as shown in the Figures 52, 53 and 54.

At this point we know that the successor function has inputs of $g0, g1, x3, x4$. Hence the original cut set of the BDD will remain the same, and variables g0 and g1 will stand on the top of the BDD. This is shown in Figure 54.

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

Pedram's method was also applied to incompletely specified functions. It is interesting how well is this method suited for decomposition of multi-input, multi-output functions with a large number of don't cares? Figure 55 shows an example of an incompletely specified function, and Figure 56 shows the corresponding 'ternary' BDD with 3 terminal nodes - 1,0 and X.

It can be seen from this BDD that what is needed is some kind of technique to combine compatible nodes to the smallest number of nodes. For instance, familiar graph coloring or set covering methods can be used, so that the nodes of the BDD will be combined, just like combining columns in the Karnaugh



NUMBER OF NODES BELOW CUT = 3
THEREFORE COLUMN MULTIPLICITY = 3

HENCE FUNCTION IS DECOMPOSABLE WITH CHOSEN BOUND AND FREE SETS.

Figure 47: BDD of the decomposition chart

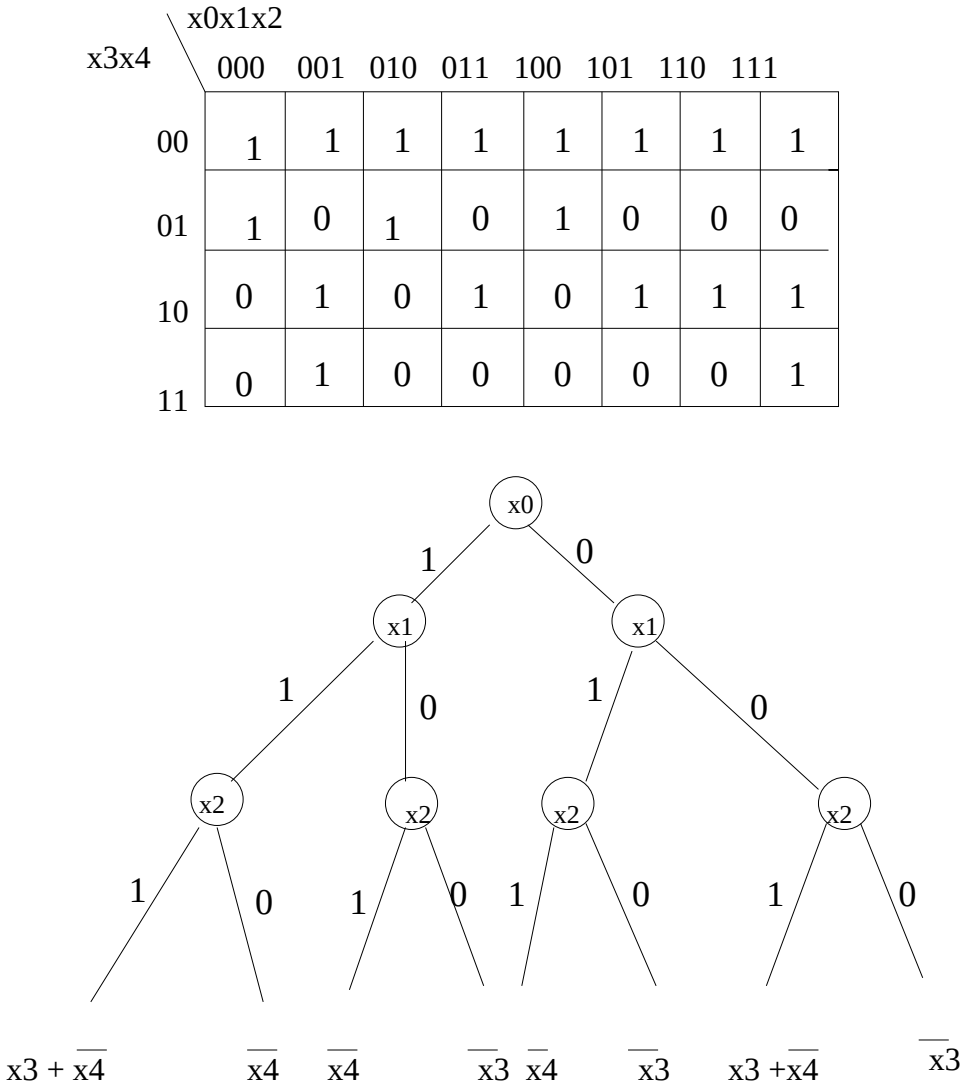


Figure 48: Correspondence between columns of the decomposition chart and BDD

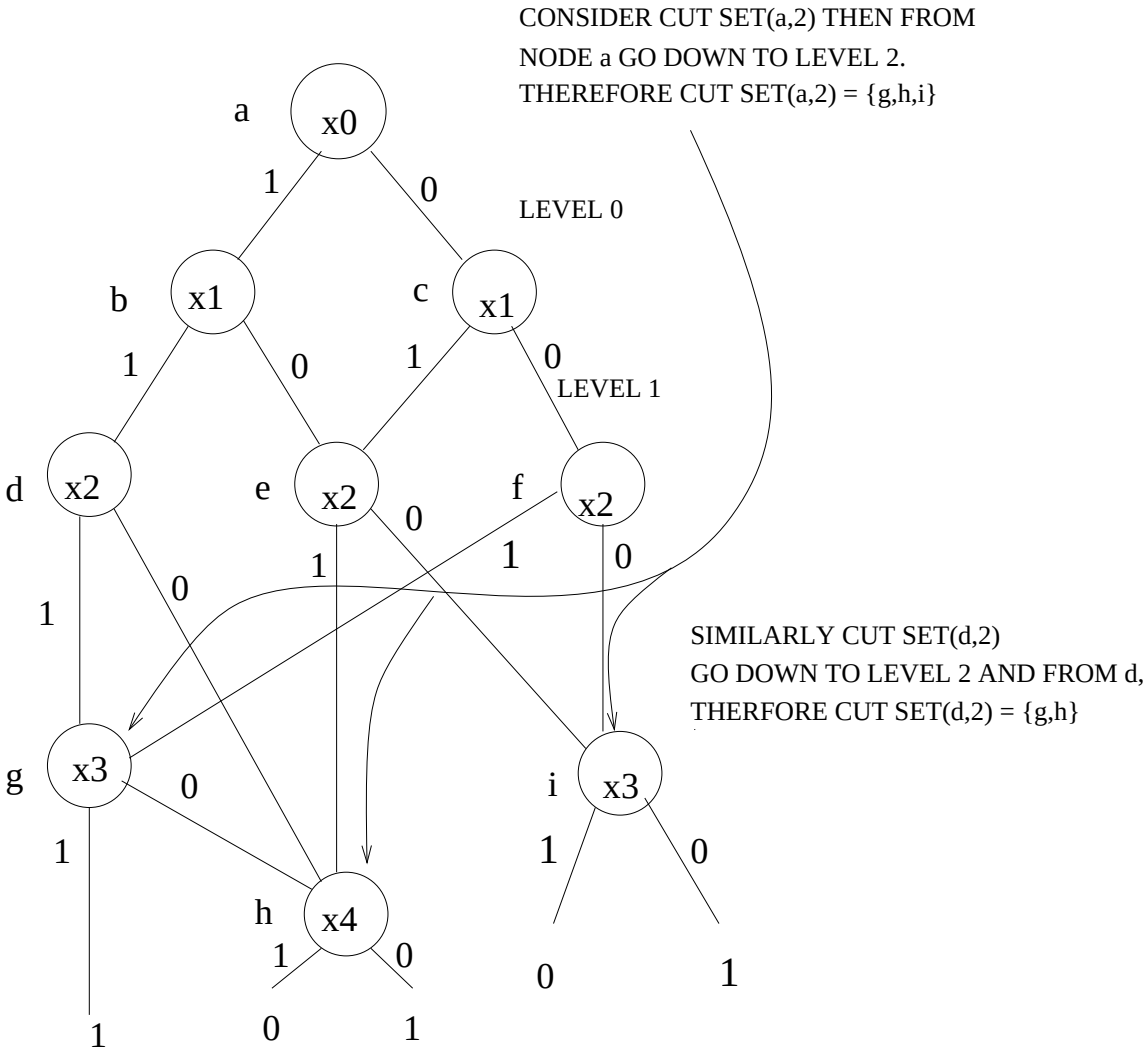


Figure 49: Explanation of a cut set

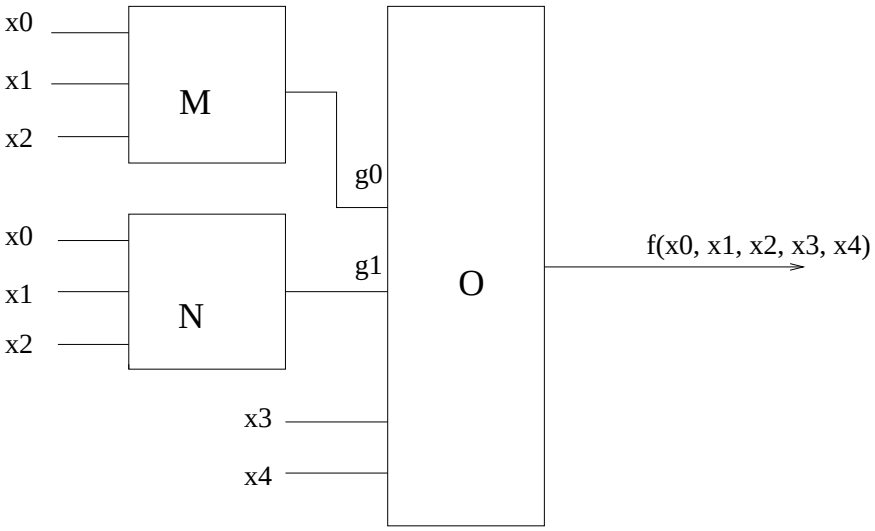


Figure 50: Structure of the decomposition with two g functions

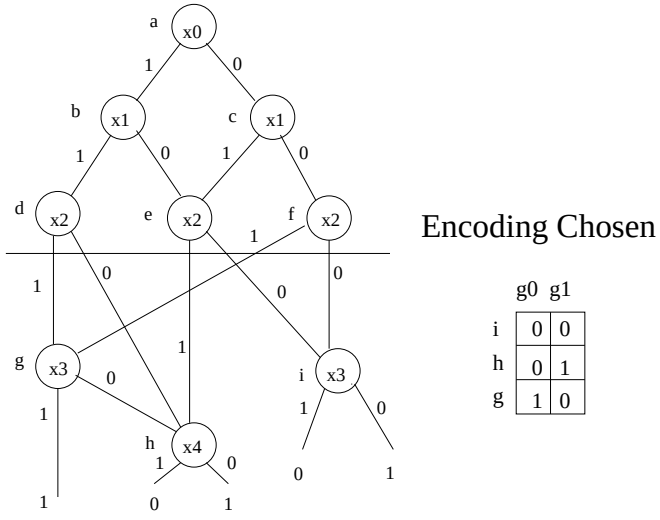


Figure 51: A BDD Example

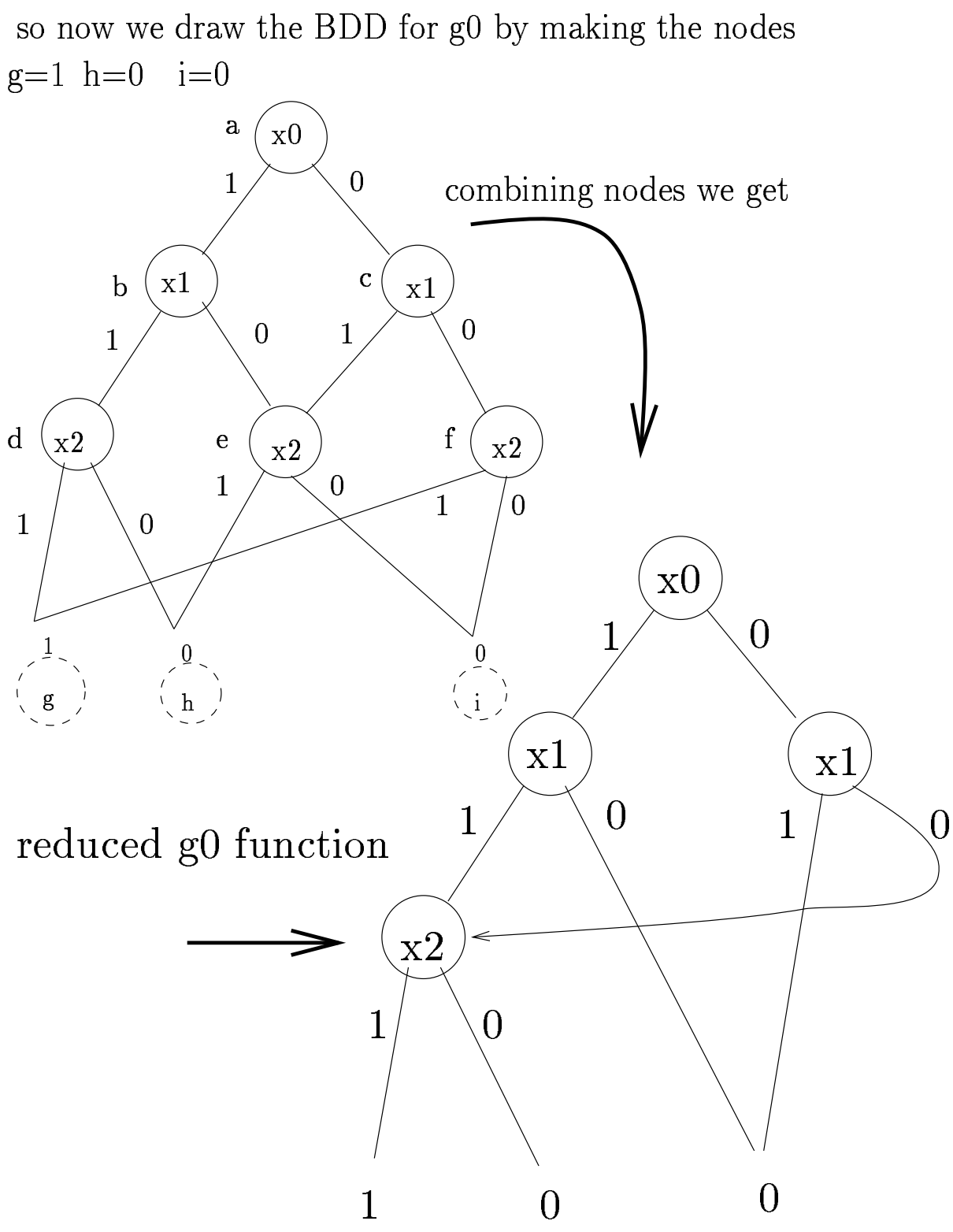


Figure 52: Finding the predecessor block

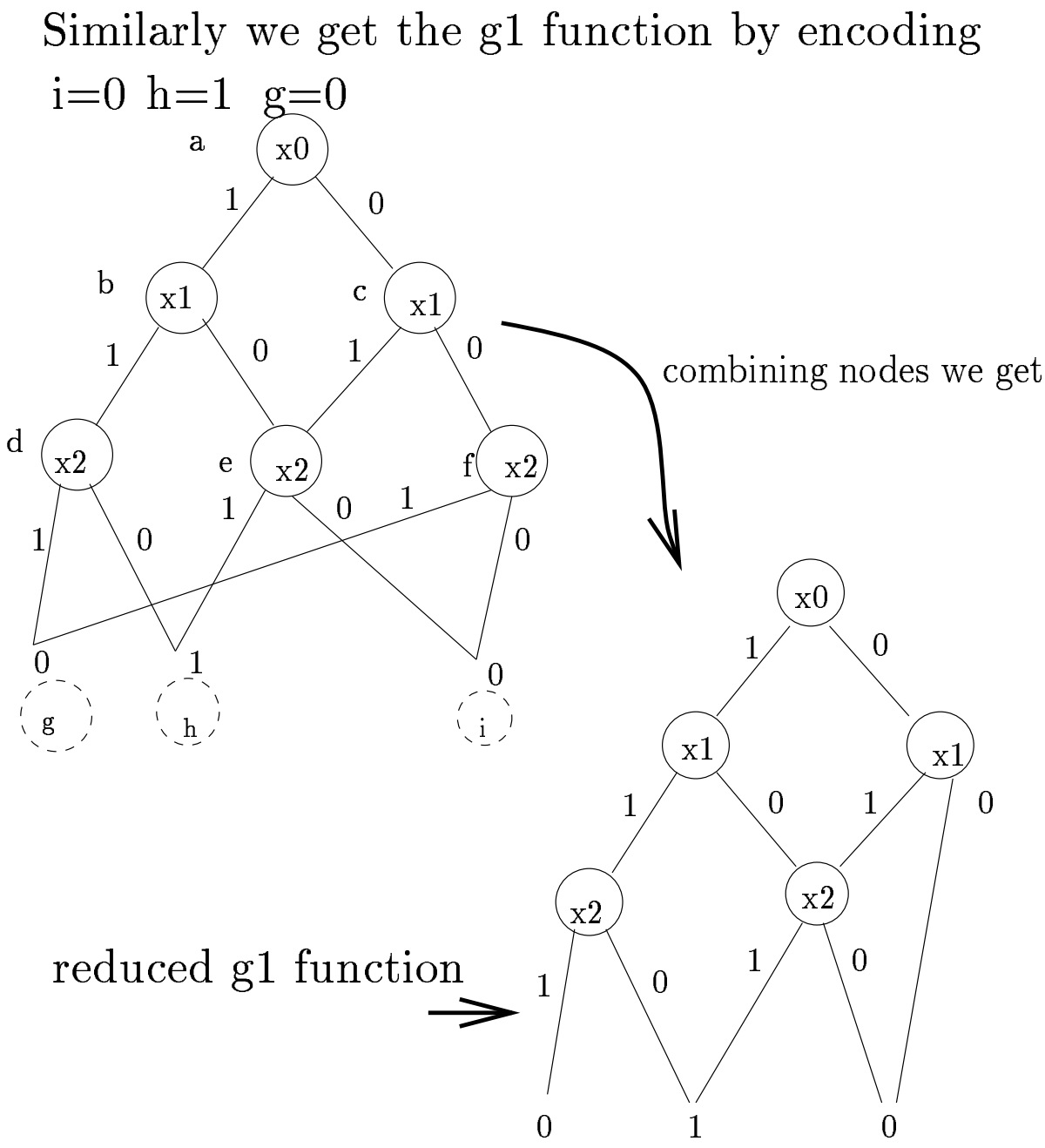
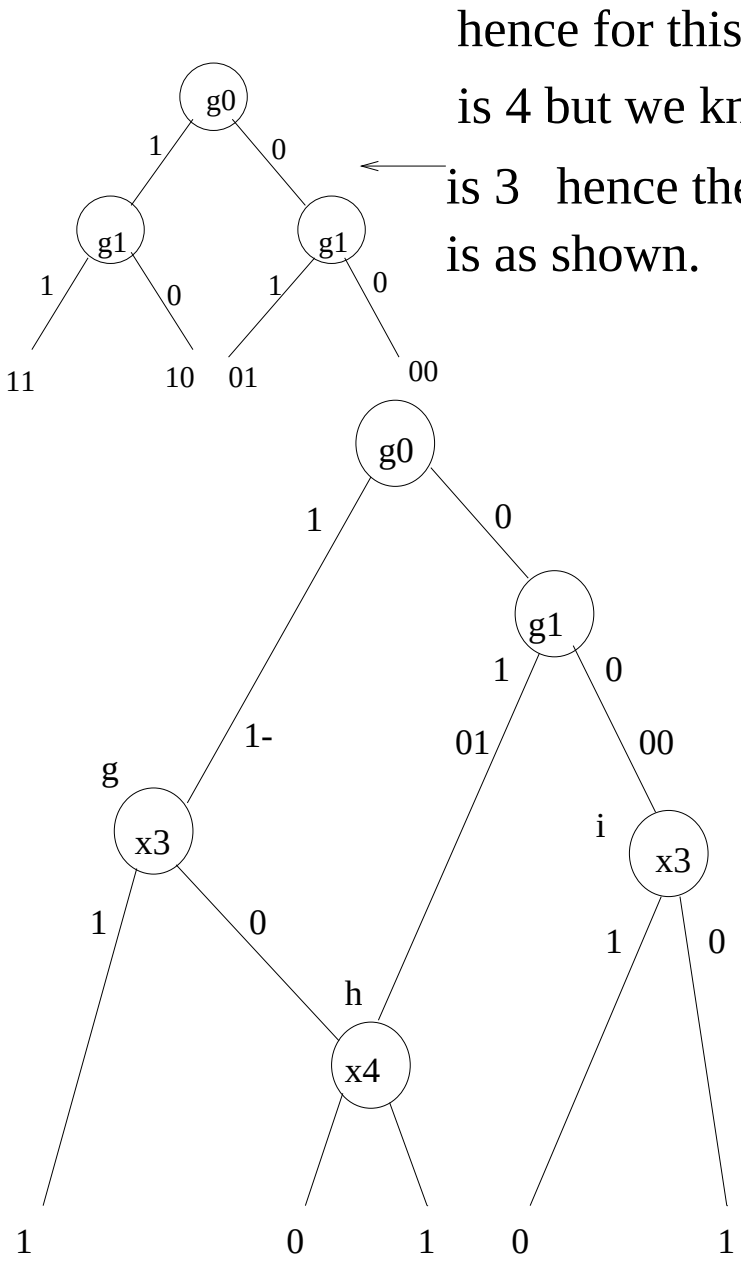


Figure 53: Finding the predecessor block



hence we get the BDD of our sucessor function

Figure 54: Finding the successor block

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

Figure 55: Decomposition chart of an incompletely specified function

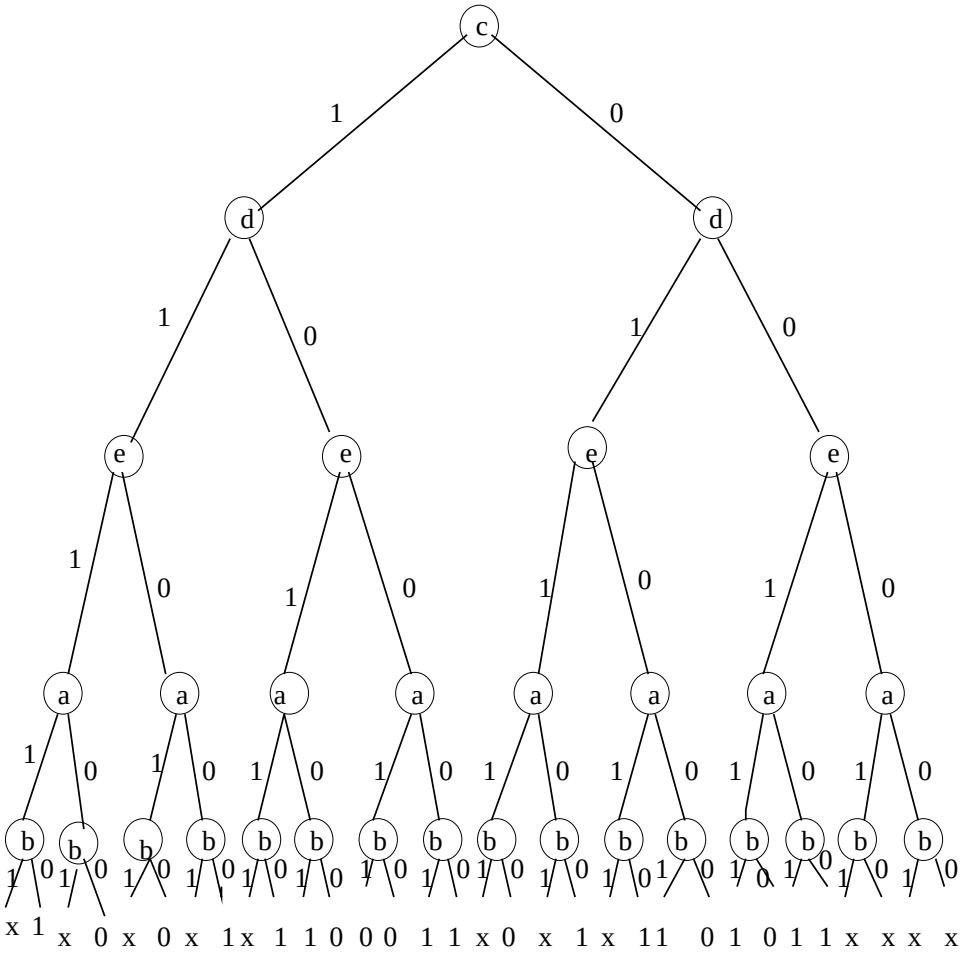


Figure 56: BDD with 3 nodes for an incompletely specified function

map so as to reduce the column multiplicity. One possibility would be to create an Incompatibility Graph and apply graph coloring, another possibility is to create Compatibility Graph, maximum cliques, and solve the coloring table. One more approach would be to create a new method performing 'column minimization' in a smart heuristic way directly on the BDD.

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

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

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

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

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

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

22.3 New Classes of AND/EXOR Decision Diagrams.

The Direct Acyclic Graph (DAG) based representation and decision diagrams which are more general than BDDs have been recently created and applied. While Hurst in 1985 [302] only mentioned a similar approach, recently both a theory and several synthesis tools have been developed in our group, [695, 591, 590, 696], [697, 567, 568, 569, 698, 179, 498, 499], and also by Sasao [579] Steinbach [55], Rosenstiel [326, 598], and Marek-Sadowska [132, 561].

Perkowski et al [590, 592, 498] formulated the new class of Decision Diagrams, called *Kronecker Decision Diagrams (KDD)*. These Decision Diagrams are the generalization of the popular Binary Decision Diagrams (Bryant, 1986) and Functional Decision Diagrams, Kebschull, Schubert, and Rosenstiel, [326], and are more compact than both of the former Decision Diagrams. The method is much more efficient than FDDs, and can be applied to very large functions given in multi-level net-lists.

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

For any Decision Diagram to prove to be useful, the compactness of the representation has to be compared with the ease of construction and manipulation. A package for representation and manipulation of functions has been developed [179, 55], which shows the compactness of the KDDs together

with ease of manipulation and construction. It has been shown that for the standard, hard, benchmark examples, KDDs are on the average 35% more compact than Binary Decision Diagrams, with some reductions of up to 75% being observed. The minimization scheme is based on the state of the art minimization schemes for BDDs, namely, dynamic variable ordering with sifting algorithm (Rudell, IWLS 1993, [558, 559]). Here the sifting is performed for both the order of variables as well as the type of decompositions.

Furthermore, a class of functions was presented for which both BDD and FDD representations are exponential in size but KDD is of polynomial size [179]. This property together with the canonicity and the ease of construction and manipulation distinguishes the major significance of the KDDs. Although, thanks to their canonicity, the KDD seem to be the most prospective of the introduced by our group diagrams as a general purpose representation of Boolean Functions, we found that other diagrams are best for mapping to fine-grain FPGAs. Recently Ho and Perkowski developed a concept of Free Kronecker Decision Diagrams [499]. They showed experimentally that it gives very good results on mapping to Atmel 6000 series FPGAs. The difference between the Kronecker Functional Decision Diagrams and the Free Kronecker Decision Diagrams (FKDD) is that in the former diagrams all nodes at certain level have the same variable and the same (one of three) expansion type. We generalized this concept to *'Pseudo-Kronecker Decision Diagrams'* where in each level we have still one variable, but all three types of expansions are possible in nodes. This has been generalized even further, to FKDDs, where there can be many different variables in a level. The tree or a graph is thus no longer 'ordered'. Although it is more difficult to create a program for generating this type of diagrams, they reduce significantly the number of nodes, and result in better mappings to FPGAs. It is possible to make these free diagrams canonical, implement operations on them, and treat them as the general-purpose function representations.

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

Finally, new classes of diagrams, called canonical *'Boolean Ternary Kronecker Decision Diagrams (BKTDD)'* have been created, that can theoretically be better than all other diagrams [498]. For selecting the expansion type, one has 1 choice in BDDs and original FDDs, 2 choices in modified FDDs, 3 choices in KDDs, and 12 choices in BKTDDs. There is no danger of loosing good choices in BKTDDs, since the three standard expansions of the KDD are still available in them. Therefore, the exact BKTDD is always not worse than the exact KDD.

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

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