Dissertations and Theses   Dissertations and Theses

1990

An effective cube comparison method for discrete spectral transformations of logic functions

Ingo Schafer
Portland State University

AN ABSTRACT OF THE THESIS OF Ingo Schäfer for the Master of Science in Electrical Engineering presented May 14, 1990.

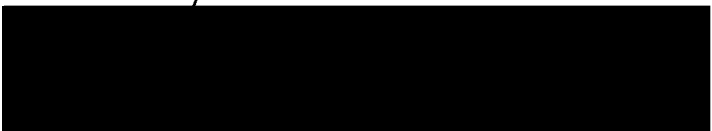Title: An Effective Cube Comparison Method for Discrete Spectral Transformations of Logic Functions

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:

Marek A. Perkowski, Chair

Małgorzata E. Chrzanowska-Jeske

Marek Elżanowski

Spectral methods have been used for many applications in digital logic design, digital signal processing and telecommunications. In digital logic design they are implemented for testing of logical networks, multiplexer-based logic synthesis, signal processing, image processing and pattern analysis. New developments of more efficient algorithms for spectral transformations (Rademacher-Walsh, Generalized Reed-Muller, Adding, Arithmetic, multiple-valued Walsh and multiple-valued Generalized Reed-Muller) their implementation and applications will be described.

Recently a new method to generate the Rademacher-Walsh spectrum has been introduced. In this thesis this method has been taken and generalized for all the above mentioned transformations. It will be further called the Cube Comparison method. The main advantage of this method is, that it generates the spectrum directly from the cube representation of Boolean and multiple-valued input, binary output functions and does not use the classical approach of matrix calculation or derived from it the Fast Transformations. Thus, it overcomes the limitation caused by the memory requirement of all the current implementations.

The programs for the binary transformation presented in the thesis allow the calculation of the complete spectrum for completely and incompletely specified Boolean functions having up to 32 literals. However, this is only a limitation of the implementation and can be changed by using different data structures. To be able to use functions represented in the classical way fast preprocessing algorithms like the generation of the disjoint cubes from the nondisjoint ones, or the generation of dense arrays have been developed.

The implementation of the multiple-valued Walsh transformation makes use of an algorithm to convert the multiple-valued representation of the input function to a binary one. This binary representation can be taken directly for the binary Rademacher-Walsh transformation.

For the multiple-valued Generalized Reed-Muller (GRM) transformation the Cube comparison method is used first to transform all literals to the chosen polarity and second to calculate using these literals the final multiple-valued Generalized Reed-Muller form.

As an application of the fast GRM implementation the minimization program CANNES (CANonical Nor Exor Synthesizer) is introduced. It makes use of the recently introduced Canonical Restricted Mixed Polarity (CRMP) Exclusive Sum of Product

forms. Such a form is preferred over the standard Sum of Product form (SOP) in designing for testability.

AN EFFECTIVE CUBE COMPARISON METHOD FOR DISCRETE

SPECTRAL TRANSFORMATIONS OF LOGIC FUNCTIONS

by

INGO SCHÄFER
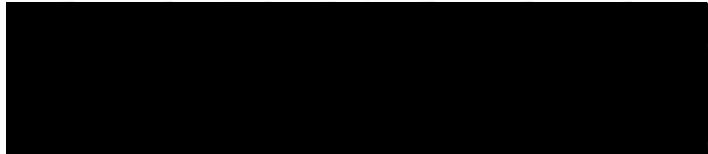
A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL ENGINEERING

Portland State University
1990

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Ingo Schäfer

presented May 14, 1990.

Marek A. Perkowski, Chair

Małgorzata E. Chrzanowska-Jeske

Marek Elżanowski

APPROVED:

Rolf Schaumann, Chair, Department of Electrical Engineering

C. William Savery, Vice Provost for Graduate Studies and Research

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

GENERAL INTRODUCTION

I.1 APPLICATIONS AND PROPERTIES OF SPECTRAL TRANSFORMATIONS

In the last two decades there has been a growing interest in digital theory, espe-
cially in two areas, logic design and analysis of switching circuits, and non-sinusoidal
communication (1-34).

The representation of a function f(t) in its sampled time domain contains only
local information of the function. The advantage of the Fourier Transformation F(ω) also
called spectrum of the function f(t) is, that it contains much more global information of
the signal (1,2). Similarly the representation of logic functions in their Boolean or
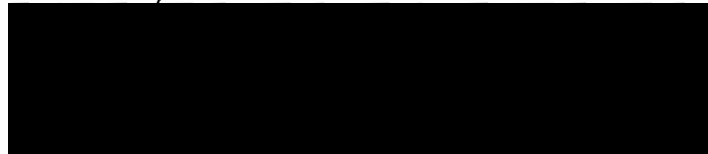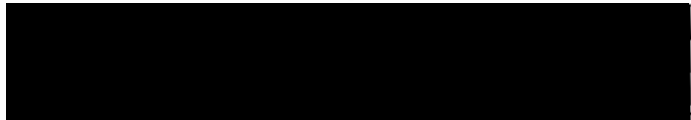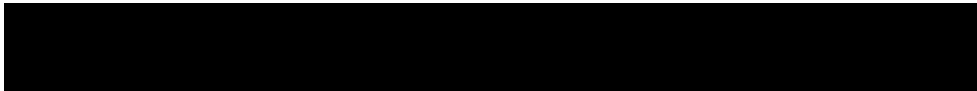Multiple-Valued domain is less global than their discrete counterparts to the Fourier
Transformation, the discrete global spectral transformations (such as Walsh-type
Transformations).

The advantage of the discrete spectral transformations compared to the conven-
tional Fourier methods in analysis and synthesis of logic circuits is that they can be easily
implemented on digital computers. Because of their applicability to high speed digital
computation in areas such as image processing and communication systems they are seen
as an important signal processing tool (1-11).

The Walsh (1-7,12-15) and Reed-Muller transformations have been found espe-
cially advantageous for fault detection and synthesis of logic circuits (1-7,16-18). A
given logic function is defined uniquely by its spectral coefficients. Therefore, any mal-
function of the hardware related to this function will be seen in its spectral characteris-

tics. The two above mentioned transformations are well suitable for the design of easily-testable circuits. This is because both transformations favor the design of circuits with high percentage of exclusive-OR (EXOR) gates, where every input combination to an EXOR gate is a test condition for it (1-7). Thus, for circuits containing EXOR gates a smaller number of tests has to be generated to be able to test the whole circuit. Therefore, spectral transformations are used to both test generation of arbitrary circuits and design of special circuits which are easy testable.

Since logic functions are described completely by their spectral coefficients then the latter can be used for functions classification (4,7). It is well known that the number of possible different functions of a given number of variables is enormous. Therefore, it is useful to find the characteristic of the functions that would allow identical hardware realizations for a group of them. The following important properties of the functions can be found by spectral classification methods: linear separability, monotonicity, summability, asummability, and dual-comparability.

The decomposition of Boolean functions (4-6) is used to reduce its complexity and improve its testability. Finding decompositions in the Boolean domain has been found very difficult. In (4-6,28,29) efficient methods to decompose functions into linear (implementation using only one EXOR gates) and non-linear (implementation using non-EXOR gates) parts was introduced for the spectral domain. The two basic types of decomposition are serial and parallel linearization (6,28,29). The serial linearization decomposition can only be done by spectral methods. The disadvantage to decompose Boolean functions in spectral domain in comparison to the Boolean domain is the time necessary for performing the transformation. Thus, fast efficient algorithms for spectral transformations would clearly give the advantage to the decomposition of Boolean functions in spectral domain.

The Walsh-type and Reed-Muller transformations are also used in multidimen-

multiple-valued input binary output functions. The basic theory underlying the method proposed here was recently introduced in (3,4,12-15) for the Walsh transformation. The new method further called Cube Comparison Method overcomes the problem of memory requirements completely. At least there is hardly any internal computer memory requirement, and the memory requirements to store the complete spectra on the hard disk are inherent to the problem. Another advantage of the Cube Comparison Method is, that it is very well suited for the development of data flow graphs and algorithms for parallel processing (every spectral coefficient can be calculated independent from the other ones). Even with serial processing of the complete spectrum the new introduced algorithms are faster then most of the other implementations.

I.2 GENERAL DESCRIPTION OF DISCRETE SPECTRAL METHODS

The existing implementations of Rademacher-Walsh and Generalized Reed-Muller transformations use algorithms for matrix multiplication or Fast Transformations (1,2,4). Some basic Definitions necessary for the introduction of the Cube Comparison Method can be found in Appendix A.

From the point of view of speed and computer memory utilization the worst method to generate the spectrum is to apply directly the matrix multiplication definition of an orthogonal transformation (1,2,10,11). It has large memory requirements and the calculation time is very long due to the large number of multiplications as well as addition and subtraction operations. For a function having N literals a $2^N \times 2^N$ matrix is required to perform the transformation. Hence, there are N(N-1) additions or subtractions necessary for the Walsh transformation by using a matrix of order N.

Example I.1:

The function $F = abc + a\bar{b}c + \bar{a}\bar{b}c$ is a function of three input variables. The function can be represented in the form of the vector V=< 0,1,0,0,0,1,0,1 >, where

1 stands for an on-minterm used in the function F. For all other not used terms the value in the vector position is 0. The matrix here has to be a 8×8 matrix. The matrix chosen below is the one for the Rademacher-Walsh transformation. In Figure 1 the matrix and the final spectrum of the matrix multiplication are shown.

$$
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & -1 & -1 & -1 & -1 & 1 & -1 & 1 \\
1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 \\
1 & 1 & 1 & 1 & -1 & 1 & 1 & -1
\end{bmatrix}
\begin{bmatrix}
0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1
\end{bmatrix}
=
\begin{bmatrix}
3 \\ -3 \\ 1 \\ -1 \\ -1 \\ 1 \\ -1 \\ 1
\end{bmatrix}
\begin{matrix}
1 \\ S_1 \\ S_2 \\ S_3 \\ S_{12} \\ S_{13} \\ S_{23} \\ S_{123}
\end{matrix}
$$

Figure 1. Matrix multiplication.

The ordering of spectral coefficient shown in Figure 1 is used for the Rademacher-Walsh, Arithmetic, and Adding spectra. In this presentation it will be called the *straight* order.

All essential radix-2 transformations like Walsh-type, Reed-Muller, Arithmetic and Adding have the property, that their transformation matrix T for order N higher than two can be obtained by a Kronecker multiplication:

$$T_N = T_2{}^{[k]}$$

where [k] in the exponent means the application of the Kronecker product for n times, with $k = \log_2 N$.

With this property it is particularly easy to obtain fast transformations by a process of matrix factorization. Hence, several Fast Transformations were developed and implemented for Walsh-type (1,2,4) and GRME Transformation (9,16,17). The advantage of the Fast Transformations is the reduction of the number of operations. For instance, for the above shown Walsh transformation the number of operations is reduced

to Nlog$_2$N. Substantial disadvantages however still exist. The data has to be rearranged prior or subsequent to a transformation which takes additional processing time. The large memory requirement is still the other disadvantage. Therefore all current implementations are limited to Boolean functions using up to 16-20 input variables, depending on the internal memory of the computer system.

To overcome these problems, the method previously mentioned (3,4,12-15) was changed and generalized to all the above named transformations. It has the feature that each spectral coefficient can be directly generated from the cube representation of the function. The idea of this method described here is to make use of a kind of cube comparison among the indices of the spectral coefficients and the cubes. It will be further called Cube Comparison Method. In Table I an easy example for the Cube Comparison Method between the cube and the indices of the spectral coefficients is shown.

TABLE I

EXAMPLE FOR THE CUBE COMPARISON METHOD

cube	indices of the spectral coefficients							
	S_0	S_1	S_2	S_3	S_{12}	S_{13}	S_{23}	S_{123}
	000	1--	-1-	--1	11-	1-1	-11	111
001				1		1	1	1
-01				1			1	

Below the spectral coefficients (S_0, S_1,...,S_{123}) the Boolean representation of their indices (standard trivial functions, described later) is shown. In the first column the two cubes are given for which the spectra should be generated. The applied rules for the two cubes in our example for which the value of the spectral coefficient will be one, are:

- at least the positive literal (1s) of the cube have a corresponding "1" in the representation of the indices.

- the dc-literals of the cubes must not have a "1" in the corresponding position of the representation of the indices.

The example in Table I has illustrated only the first of all three parts of the Cube Comparison Method. The three parts are:

1. Comparison of a cube representation of a logic function with the cube representation of the indices of the spectral coefficients.

2. Calculation of the signs of the values of the spectral coefficients. They depend on the cube representation of the logic function and the order of the spectral coefficients.

3. The value of the spectral coefficients depends on the number of dc literals in the cubes of the logic function.

In the following Chapters the Cube Comparison Method is adapted for needs of different algorithms calculating spectra. Because those algorithms make use of the representation of a Boolean function as arrays of disjoint on- and dc- cubes, first a new efficient algorithm to generate disjoint cubes from the nondisjoint ones is discussed.

CHAPTER II

DISCRETE SPECTRAL TRANSFORMATIONS
FOR BOOLEAN FUNCTIONS

II.1 ALGORITHM TO GENERATE DISJOINT CUBES

II.1.1 Description

As mentioned in Chapter I, the algorithms described in the sequel make use of completely or incompletely specified Boolean functions in the form of arrays of disjoint on- and dc- (if any) cubes or an array of disjoint off-cubes. Such an algorithm was first introduced in (18-20). An improved version of it shown here has been designed, because the algorithm (18) is not well suited for the implementation on computer systems. The task of the algorithm is to generate the array of disjoint cubes from a Boolean function represented in the form of an array of nondisjoint on-cubes (in the case of completely specified Boolean function), or an array of on- and dc-cubes or on- and off-cubes (in the case of incompletely specified Boolean functions).

The same notation as in (18) is used in the description of the revised algorithm. Every time two cubes c_a and c_b are compared, the pointer a indicates the position of the cube which is the first in the pair of the cubes being compared, the pointer b indicates the position of the second cube in the pair. Note, that the pointer a changes its values from 1 to $n - 1$ (where n is the current number of cubes in the array) and the pointer b changes its values from n to 2 accordingly. In a given moment, the value of the pointer b is always greater by at least 1 than the value of the pointer a. During the execution of the algorithm the number of cubes in the array A can be different than the original number of

cubes. The symbols of cubical calculus that are used in the sequel follow the notations from (21-23). It is due to the fact that two cube operations used in the algorithm have the following properties : 1) disjoint sharp operator (denoted by $\#_j$) can generate more than one cube as its result, 2) absorption can remove some cubes and decrease by that the total number of cubes.

Symbols used:

a , b : pointers to two different cubes,

$cube_a$,

$cube_b$: determine the cubes pointed to by a , and b in the cube list,

d : number of solution cubes generated by the disjoint sharp operation,

m : number of cubes in the cube list before entering a new loop,

n : number of cubes during execution of the loop.

Algorithm: Generation of disjoint cubes

step 1. *set* a and b to the following positions in the cube list :

   $a := 1, b := n, m := n$

   where n is the initial number of cubes in the cube list.

step 2. Main loop :

   For each pair of cubes $cube_a$ and $cube_b$ from an array A *do* :

      if an intersection of cubes $cube_a$ and $cube_b$ is not an empty set

      then

            { if $cube_a$ absorbs $cube_b$

                then { substitute $cube_b$ by the last cube of the array A ;

                      $n := n - 1;$

if $b = a$ then go to step 3

else go to step 2}

else { calculate the d solution cubes from the disjoint sharp

operation $cube_b \ \#_j \ cube_a$;

replace $cube_b$ by one solution cube and add

the other ones to the end of the array A;

$n := n + d - 1$;

if $b \leq a$ then go to step 3

else go to step 2}}

else { $b := b - 1$;

if $b \leq a$ *then* go to step 3

else go to step 2}

step 3.

$b := n$;

$a := a + 1$;

if $a = m$ *then* stop

else $m := n$;

go to step 2.

The details of the implementation of the above algorithm and an example of its execution are shown in the sequel.

II.1.2 Implementation

For the implementation, the algorithm was changed and improved in order to take the advantage of some features of the C-language to become faster for computer process-

ing. Moreover, it has been implemented in three computer environments: SUN3/50-workstation, VAX 11/750, and Sequent SYMMETRY computer.

The implementation of the disjoint algorithm makes use of the the pointer structure for storing the array of cubes as follows:

```
struct cube_field
{
    unsigned short *value;    /* list of the literals of the cube            */
    short   p;                /* determines the number of "-"s in the cube   */
    short   type;             /* type determines if the cube is a DC- or ON-cube */
}cube;
```

This pointer structure determines a field in which one cube could be stored. The literals of each cube are stored in a list composed of short variables: *cube->value[i]*. To avoid wasting of the memory space, eight literals from a given cube are stored in a short variable (having 16 bits) where the internal representation of each cubical calculus symbols: $0/(10)$, $1/(01)$, $-/(11)$, and $\varepsilon/(00)$ takes two bits (where ε determines a contradictionary literal).

The element *cube->p* contains the number of "X" (dc-literals) in the cube. Finally, the element *cube->type = { ON , DC, OFF }* determines if the cube is an on-, dc- or off-cube. An additional pointer structure *unsigned int *cube_list* is used to store the addresses of the different cubes.

In order to have direct access to cubes and their values at any time, the list of terms is stored in an array. Because of the dynamical nature of the array, the *calloc()* and *realloc()* C functions (24) have been used. Such an approach speeds up an overall processing time.

The algorithm generates arrays of disjoint cubes for separate lists of on-, dc- or off-cubes. It uses mixed lists like those from Example II.1 as well.

Cube operations used in this version of the algorithm are disjoint sharp, intersection and equivalence of cubes (21,22,23). An absorb operator is implemented by two

operators in sequence: first an intersection of two cubes is performed, next, the matching operator takes the result of the intersection operator and the original cubes as arguments. When the result is equal to one of the original cubes then the larger cube is kept and the other (the one that is equal to the result of the intersection) is deleted from the list of cubes.

In order not to remove or change the positions of cubes during the execution of the algorithm only the addresses of cubes in a list are changed and removed. Such an approach is much easier to handle because the cubes in a given array can have an *arbitrary* number of literals and the length of an array storing the cubes dynamically changes. If a cube has to be removed (which happens for the disjoint sharp or absorb operations) then either the corresponding address in an address list is substituted by the address of the first cube generated by the disjoint sharp operation or in the case of the absorb operation the address corresponding to the last position in the address list is copied into the vacant position of this list.

In order to obtain an array for the addresses of cubes that always matches the size of the array of cubes, the realloc() function has been used. By doing so, one can use the address list like an array with direct access to each element of the list without the necessity of going through the elements of a linked list. These special two features: the usage of addresses in a list and the realloc() function (for the direct access of the array) have sped up our algorithm significantly.

The same basic notation from the Disjoint Algorithm in Chapter II.1.1 is used in the description of the program. In addition, the following symbols are used:

cube : a pointer to the cube address list.

sol[i] : solution cube of the i-th position in the solution cube address list.

sol-cubes : number of solution cubes from the disjoint sharp operation.

cube[a],

cube[b]   : determine the cubes from the positions a and b in the cube address list.

type   : indicates if the algorithm should be performed for on-, dc- or off-cubes.

Algorithm: Implementation of the Disjoint Algorithm

```
a = 1, b = n /* where n is the initial number of cubes in the cube address list */
do
{
    if ( cube->type != type )
            { a++;
            continue; }
    b = n;
    m = n;
            while ( b > a )
            {
                if ( cube->type != type )
                        { b--;
                        continue; }
                if ( intersect( cube[a], cube[b] ) == cube[b] )
                        cube[b] = cube[n];
                        n--;
                        cube = (cube *)realloc( cube , n * sizeof ( cube ) );
                else if ( intersect( cube[a], cube[b] ) != 0 )
                        disjoint-sharp( cube[b], cube[a] );
                        d = sol-cubes;
                        cube[b] = sol[1];
                        cube = (cube *)realloc( cube , (n+d-1) * sizeof ( cube ) );
                        for ( i = 2 ; i <= d ; i++ )
                                cube[n+i] = sol[i];
                        n = n + d - 1;
                b--;
            }
        a++
}while ( a < m-1 )
```

Example II.1:

An example of the execution of the algorithm is shown below. The order of the cubes is chosen in such a way that all different branches of the algorithm are passed through. If the cubes in the input array are sorted according to their size

then larger cubes are obtained as solution cubes but the number of performed sharp operations remains the same. Thus, the execution time of the Disjoint Algorithm will be the same, but for the sorting of the cubes additional time is necessary. performed on an sorted array the execution time will be the same. Figure 2 shows the states of the algorithm in the moments when the contents of the array of cubes has been modified. These pictures are referred to in the description below. The symbol "•" denotes the beginning of a new loop in the algorithm, the indentation shows the inner and outer loop.

step 1 : Initialization

       number of cubes $n = 5$;

       Figure 2a

step 2 : Loop

    • *$n = 5$; $a = 1$; $b = n = 5$;*

       intersection :cube[1] $\cap$ cube[4] = 1100;

       absorption:cube[4] $\neq$ 1100;

       disjoint sharp :cube[4] #; cube[1] :

           sol[1] = 01XX

           sol[2] = 111X

           sol[3] = 1101

   substitute cube[4] with sol[1];

   place the rest of solution cubes at the end of the array

   Figure 2b

   $n = n + d - 1 = 7$

   • *$n = 7$; $a = 1$; $b = 3$:*

       intersection : cube[1] $\cap$ cube[3] = $\varnothing$

- $n = 7 ; a = 1 ; b = 2 ;$

    intersection : cube[1] $\cap$ cube[2] = $\varnothing$

- $n = 7 ; a = 2 ; b = 7 ;$

    intersection :   cube[2] $\cap$ cube[7] = $\varnothing$

    - $n = 7 ; a = 2 ; b = 6 ;$

        intersection :   cube[2] $\cap$ cube[6] = 111X;

        absorption   :   cube[6] = 111X;

        substitute cube[6] with cube[n] and remove cube[n]

        Figure 2c

- $n = 6 ; a = 2 ; b = 5 ;$

    cube[5] is DC-cube

- $n = 6 ; a = 2 ; b = 4 ;$

    intersection : cube[2] $\cap$ cube[4] = $\varnothing$

- $n = 6 ; a = 2 ; b = 3 ;$

    intersection : cube[2] $\cap$ cube[3] = 1X11;

    absorption   : cube[3] $\neq$ 1X11;

    disjoint sharp: cube[3] $\#_j$ cube[2] :

            sol[1] = 0X11;

    substitute cube[3] with sol[1]

    Figure 2d

- $n = 6 ; a = 3 ; b = 6 ;$

    intersection : cube[3] $\cap$ cube[6] = $\varnothing$

    - $n = 6 ; a = 3 ; b = 5 ;$

cube[5] is DC-cube

- $n = 6$; $a = 3$; $b = 4$;

    intersection : cube[3] $\cap$ cube[4] = 0111;

    absorption : cube[4] $\neq$ 0111

    disjoint sharp : cube[4] $\#_j$ cube[3] :

    $sol[1] = 010X$

    $sol[2] = 0110;$

    substitute cube[4] with sol[1];

    place sol[2] at the end of the array

    Figure 2e

- $n = 7$; $a = 4$; $b = 7$;

    intersection : cube[4] $\cap$ cube[7] = $\varnothing$

    - $n = 7$; $a = 4$; $b = 6$;

        intersection : cube[4] $\cap$ cube[6] = $\varnothing$

    - $n = 7$; $a = 4$; $b = 5$;

        cube[5] is DC-cube

- $n = 7$; $a = 5$; $b = 7$;

    cube[7] is DC-cube

- $n = 7$; $a = 6$; $b = 7$;

    intersection : cube[6] $\cap$ cube[7] = $\varnothing$

solution array :

Figure 2e

x_3, x_4

x_1, x_2

	00	01	11	10
00	-	0	1	0
01	1	1	1	1
11	1	1	1	1
10	1	0	1	1

X1,X2,X3,X4

$$\begin{bmatrix} \text{1X00 ON} \\ \text{1X1X ON} \\ \text{XX11 ON} \\ \text{X1XX ON} \\ \text{0000 DC} \end{bmatrix}$$

a. input array

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

$$\begin{bmatrix} \text{1X00 ON} \\ \text{1X1X ON} \\ \text{XX11 ON} \\ \text{01XX ON} \\ \text{0000 DC} \\ \text{111X ON} \\ \text{1101 ON} \end{bmatrix}$$

b. cube[4] $\#_j$ cube[1]

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

$$\begin{bmatrix} \text{1X00 ON} \\ \text{1X1X ON} \\ \text{XX11 ON} \\ \text{01XX ON} \\ \text{0000 DC} \\ \text{1101 ON} \end{bmatrix}$$

c. absorption of cube[6]

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

$$\begin{bmatrix} \text{1X00 ON} \\ \text{1X1X ON} \\ \text{0X11 ON} \\ \text{01XX ON} \\ \text{0000 DC} \\ \text{1101 ON} \end{bmatrix}$$

d. cube[3] $\#_j$ cube[2]

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

$$\begin{bmatrix} \text{1X00 ON} \\ \text{1X1X ON} \\ \text{0X11 ON} \\ \text{010X ON} \\ \text{0000 DC} \\ \text{1101 ON} \\ \text{0110 ON} \end{bmatrix}$$

e. cube[4] $\#_j$ cube[3]

Figure 2. The stages of an execution of the algorithm to generate a disjoint cube representation of a Boolean function.

In order to generate disjoint dc-cubes the *type* condition in the beginning of the algorithm has to be changed to DC and then the same algorithm will perform the generation of disjoint dc-cubes. In the above example, it is obvious, that the single dc-cube is a disjoint one so there is no need to use such an algorithm.

II.2 GENERAL REED-MULLER TRANSFORMATION

II.2.1 Description

There is a growing interest in the design of logic circuits using EXOR gates. The advantages of functions realized with such circuits are the reduced number of gates and even more importantly the easy testability. A particular field of interest is the minimization of logic functions with the Generalized Reed-Muller Expression (GRME) (25,26).

The Reed-Muller Expression (RME) (25,26) of a function $F(X_0, X_1, ..., X_{m-1})$ over the Galois Field 2 (GF(2)) is given by

$$F(X_0, X_1, \ldots, X_{m-1}) = a_0 \oplus a_1 X_0 \oplus a_2 X_1 \oplus a_3 X_0 X_1 \oplus \cdots \oplus a_{2^m - 1} X_0 \cdots X_{m-1} \text{,(1)}$$

where $(a_o, a_1, ..., a_{2^m - 1})$ are coefficients of the RME and $(1, X_0, X_1, ..., X_{m-1})$ are the basic vectors. The elements of the GF(2) are the operations of addition and multiplication for modulo 2, and the numbers (0,1).

One method to generate the RME from Sum Of Products Expression (SOPE) is to apply the following rules:

$$\bar{a} = 1 \oplus a$$
$$a(b \oplus c) = ab \oplus ac$$
$$a \oplus a = 0$$
$$a \oplus 0 = a$$

In Example II.2 it is shown how these rules are applied to an example function.

Example II.2

$$F=\bar{a}\,cd\oplus\bar{b}\,cd\oplus\bar{a}\,\bar{b}\,cd\oplus ab\bar{c}\oplus ab\bar{d}\oplus ab\bar{c}\,\bar{d}$$

$$=(1\oplus a)cd\oplus(1\oplus b)cd\oplus(1\oplus a)(1\oplus b)cd\oplus ab(1\oplus c)\oplus ab(1\oplus d)\oplus ab(1\oplus c)(1\oplus d)$$

$$=cd\oplus acd\oplus cd\oplus bcd\oplus cd\oplus acd\oplus bcd\oplus abcd\oplus ab\oplus abc\oplus ab\oplus abd\oplus ab\oplus abc\oplus abd\oplus abcd$$

$$=ab\oplus cd$$

This method is not very useful for computer implementation. Other methods like Fast Transformations and matrix multiplication were mentioned in Chapter I, but they have large memory requirements. Hence, it was investigated if the method of Cube Comparison (Chapter I) could be applied for the RME Transformation.

Let us first make the following Definitions that are necessary to describe the GRME Transformation with the Cube Comparison method.

Definition II.1

The equivalence operation ($\equiv$) is applied to two cubes. It is the bit-wise application of a bit operation that has the following operation-table.

a \ b	0	1	-
0	1	0	-
1	0	1	-
-	-	-	-

Figure 3. Equivalence operation.

Where the literals in the first column represent a certain literal *a* of the first cube and the literals in the first row represents a certain literal *b* of the second cube between which the equivalence operation should be performed.

Example II.3

$$\equiv \quad \begin{array}{r} 1010 \\ \underline{-011} \\ -110 \end{array}$$

Definition II.2

Let us denote the coefficients $(a_0, ..., a_{2^m-1})$ of the Reed-Muller expression according to Equation (1) as the spectral coefficients of the spectrum M. The *index i* of each spectral coefficient M_i is the cube representation of products of the basic vectors $(1, X_0, .., X_{m-1})$, where the products of basic vectors represent the set of standard trivial functions for the RME, shown in Figure 4.

Example II.4

The coefficient a_3 of the RME and its corresponding product of basic vectors $X_a X_b$ are now represented by the spectral coefficient M_{ab}.

Property II.1

The Reed-Muller transformation has the property, that it contains *literals* present in a given term only.

Example II.5

$$a\,b\,\bar{d} = a\,b\,(\,1 \oplus d\,) = a\,b \oplus a\,b\,d\,.$$

With these definitions and properties of the RME transformation we are able to prove the theorems necessary to develop the algorithm which makes use of the Cube Comparison method.

21

M_0 (a,b × c,d)

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

M_a

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

M_b

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

M_c

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

M_d

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

M_{ab}

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

M_{ac}

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

M_{ad}

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

M_{bc}

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

M_{bd}

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

M_{cd}

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

M_{abc}

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

M_{abd}

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

M_{acd}

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

M_{bcd}

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

M_{abcd}

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

Figure 4. Spectrum M for a binary function having four literals

Theorem II.1

The values of the spectral coefficients, for which the indices contain at least all the positive literals and not the dc-literals of a term t, are equal to the value "1".

Proof:

Any term t can be represented by its spectrum M. According to Property II.1 it is not possible that in the RME a literal can occur that is not used in the term t.

Theorem II.2

If the number of occurances of a spectral coefficient for a function F consisting of several terms is odd, the index of this coefficient, being represented as a cube, is a term of the RME for the function F.

Proof:

With the Definition II.2 and the property that a $\oplus$ a = 0 the proof for Theorem II.2 is trivial.

In Table II the application of these Theorems is shown for the function F used in Example II.2. The spectral coefficients M_1, M_{23} ... are named using the same characters for the index a=1, b=2, c=3 and d=4 as for the terms, in order to stress the connection between the indices and the term.

TABLE II

SPECTRUM M OF A FUNCTION F

term	M_0	M_a	M_b	M_c	M_d
$\bar{a}$cd					
$\bar{b}$cd					
$\bar{a}\bar{b}$cd					
ab$\bar{c}$					
ab$\bar{d}$					
ab$\bar{c}\bar{d}$					
result					

TABLE II

SPECTRUM M OF A FUNCTION F
(continued)

term	M_{ab}	M_{ac}	M_{ad}	M_{bc}	M_{bd}	M_{cd}
$\bar{a}$cd						1
$\bar{b}$cd						1
$\bar{a}\bar{b}$cd						1
ab$\bar{c}$	1					
ab$\bar{d}$	1					
ab$\bar{c}\bar{d}$	1					
result	1					1

	M_{abc}	M_{abd}	M_{acd}	M_{bcd}	M_{abcd}
$\bar{a}$cd			1		
$\bar{b}$cd				1	
$\bar{a}\bar{b}$cd			1	1	1
ab$\bar{c}$	1				
ab$\bar{d}$		1			
ab$\bar{c}\bar{d}$	1	1			1

As one can observe from Table II, the result ab $\oplus$ cd is the same as one obtained in Example II.2.

Definition II.3

The polarity cube is the vector representation of the negative and positive literals of a term t. For a positive literal the respective position of the polarity has the value 1, for a negative literal it has the value 0.

Example II.6

The polarity of the term ab$\bar{c}$d = 1101.

Let as observe, that the polarity for the RME has the value 1 for each literal. Now we want to expand the method for RME to a more general expression in which all polarities of variables are possible.

Definition II.4

A Generalized Reed-Muller expression (GRME) is an Exclusive SOPE (ESOPE) in which every term has the same polarity. It can be represented by a spectrum similar to one from the Definition II.2, where the standard trivial functions (respectively, the indices of the spectral coefficient of the new spectrum) have a different polarity for each chosen polarity. The connection between indices of spectral coefficients of a certain polarity and the coefficients of the RME spectrum is given by the equivalence operation.

Example II.7

Using the equivalence relation the spectral coefficient M_{abd} of the RME transformation is changed to the respective spectral coefficient of the GRME transformation with the polarity $\bar{a}b\bar{c}d$ (0101):

$$abd \equiv \bar{a}b\bar{c}d = \bar{a}bd$$

The spectral coefficient for the chosen GRME is now $M_{\bar{a}bd}$. Product $\bar{a}bd$ is the standard trivial function corresponding to $M_{\bar{a}bd}$.

To be able to use the RME transformation characterized by the Theorems II.1 and II.2 in order to create a GRME transformation according to the Definition II.4 the counterpart of these theorems have to be found for GRME.

Theorem II.3

To make use of the Theorem II.1 for the GRME transformation, the equivalence operation between every cube of the SOPE and the polarity cube has to be performed to create argument cubes to be applied in Theorem II.1.

Proof:

The standard trivial functions, respectively the indices of the spectral coefficients of the GRME spectrum according to Definition II.4 have the same polarity as the

polarity of the GRME itself. Hence, the analogous theorem to Theorem II.1 is that all values of spectral coefficients for which the indices contain all the literals of term t that have the same polarity as the polarity cube and not the dc-literals of term t, are equal to value "1".

With the application of Theorem II.3 we are now able to find the spectral coefficients of a GRME. This is possible by first performing the equivalence operation between the terms of the SOPE and the polarity cube and next applying Theorem II.1. The indices of the spectral coefficients have still the polarity of the RME. To obtain proper indices the following theorem is used.

Theorem II.4

To obtain the GRME from the spectrum calculated according to the Theorem II.3 the equivalence operation has to be performed between the standard trivial functions for this GRME and its polarity.

Proof:

According to Definition II.4 every set of standard trivial functions, necessary for the different GRME polarities, can be obtained by performing the equivalence operation between the cubes of the standard trivial functions and the polarity. Hence, this operation has to be performed between the spectrum being in RME and the polarity cube in order to obtain the correct polarity of the solution terms in GRME.

With the four Theorems II.1- II.4 we are now able to formulate an algorithm to generate the GRME from a SOPE.

Notation:

term : cube of the function in SOPE.

polarity :

chosen polarity for the GRME transformation.

newterm :

cube after performing equivalence operation between *polarity* and *term* .

covalue :

variable to count the occurances of a spectral coefficient.

solterm :one final term of the GRME.

Algorithm : Generalized Reed-Muller transformation

do for each term

   newterm = term ≡ polarity

do for each possible index

   do for each newterm

      if (newterm is covered by the index)

         covalue = covalue + 1;

   if (covalue is odd)

      solterm = index ≡ polarity;

II.2.2 Implementation

The above shown algorithm has been developed to be well suited for computer implementation. Therefore, it can be directly used as a subprogram.

To store one term, the structure *term_field* is used. This structure has been created in order to be able to compare directly the term representation with indices of spectral coefficients.

The variables value1 and valuex used in the structure are needed to compare the term representation with an index of a spectral coefficient. As mentioned in Chapter II.1.2, two bits are necessary to store one binary literal. To be still able to compare the

cube representation of the term and the index, a cube is stored in two distinct fields. The literals of the *term->value1* are 1 for those literals of the term that are either 1 or -. The dc literals of the term are stored in *term->valuex*. With this approach, using a long variable having 32 bits, the implementation is limited to 32 literals per term.

```
struct term_field
{
        unsigned long value1;    /* field to store one's of term        */
        unsigned long valuex;    /* field to store dc's of term         */
        unsigned short order;    /* number of positive literals         */
}*term;
```

The pointer *unsigned int *term_list* is used to store the addresses of the term in a dynamical array.

Because the order of solution terms is irrelevant it is possible to generate the indices for the whole spectrum with a single counting loop. With this approach one obtains the fastest possible way to generate the indices.

With these properties the implemented algorithm for the GRME transformation has the following structure.

Notation:

term : current processed cube of the term_list.

index : index of the currently processed spectral coefficient.

total :   number of spectral coefficients.

terms : number of terms.

coeff : a variable to determine if the number of occurances of a coefficient is odd or even.

Algorithm : GRME transformation

```
for ( index = 0 ; index < total ; index++ )
        { coeff = 0;

        /* calculate coefficient for all terms */

        for ( k = 0 ; k < terms ; k++ )
                { term = term_list[k];
                if ( ( index & term->valuex ) != 0 )
                        continue;
                if( ( index & term->value1) == term->value1 )
                        { if ( coeff == 0 ) coeff = 1;
                        else coeff = 0; }}
        if ( coeff == 1 )
        /* index of the spectral coefficient is a solution term */
                output(index); }
```

II.2.3 A complete example

The list of terms shown in Table III taken as an example to illustrate the algorithm of the GRME transformation, is the same as in Example II.2. The cube of the term is shown together with the corresponding value1 and valuex, to help the reader to analyze the algorithm.

TABLE III

COMPUTER REPRESENTATION OF AN ARRAY OF CUBES

term	value1	valuex
0-11	0111	0100
-011	1011	1000
0011	0011	0000
110-	1101	0001
11-0	1110	0010
1100	1100	0000

The chosen polarity of the GRME is 0101. First the terms are matched with the polarity according to the equivalence operation (Table IV, Theorem II.3). Because the dc literals (valuex) do not change in this operation, they are not shown in Table IV.

TABLE IV

EQUIVALENCE OPERATION ON CUBES

before equivalence		after equivalence	
term	value1	newterm	value1
0-11	0111	1-01	1101
-011	1011	-001	1001
0011	0011	1001	1001
110-	1101	011-	0111
11-0	1110	01-0	0110
1100	1100	0110	0110

Next the RME transformation is applied for the new terms (variable *newterm*) of Table IV. The Table V shows in each row the RME transformation for one term. In the row before the last the value (*covalue*) of the complete spectrum is shown. According to Theorem II.2 this value is "1" when the number of occurances of a spectral coefficient in the respective columns of the table is odd. The RME of the *newterms* is represented by the indices of those spectral coefficients having value "1".

TABLE V

SPECTRUM M FOR THE FUNCTION OF TABLE IV

newterm	M_0	M_1	M_2	M_3	M_4
1-01					
-001					1
1001					
011-					
01-0			1		
0110					
covalue			1		1
index			-1--		---1

TABLE V

SPECTRUM M FOR THE FUNCTION OF TABLE IV
(continued)

newterm	M_{12}	M_{13}	M_{14}	M_{23}	M_{24}	M_{34}
1-01			1			
-001					1	1
1001			1			
011-				1		
01-0	1				1	
0110				1		
covalue	1					1
index	11--					--11

newterm	M_{123}	M_{124}	M_{134}	M_{234}	M_{1234}
1-01			1		
-001				1	
1001		1	1		1
011-	1				
01-0		1			
0110	1			1	1
covalue					
index					

The set of terms in RME from Table V (indices) has to be changed to the correct polarity. This is done by applying the equivalence operation between the term cubes and the polarity cubes. The final GRME of the initial SOPE of the terms before and after performing the equivalence operation (after equivalence) is shown in Table VI.

TABLE VI

EQUIVALENCE OPERATION FOR THE FUNCTION OF TABLE V

before equivalence	after equivalence
-1--	-1--
---1	---1
11--	01--
--11	--01

The GRME in Table VI can be written as the GRME $b \oplus d \oplus \bar{a}b \oplus \bar{c}d$.

II.2.4 Execution times

The execution times shown here and those in the following Chapters refer to a SUN 3/50 workstation. The meaning of the abbreviations in the time table is as follows (all values are in seconds):

- *u*   elapsed user time.

- *s*   elapsed system time.

- *no*   cube has no dc literals.

- *some*   cube has some dc literals.

- *many*   cube has many dc literals.

- *cubes*   number of cubes in the array.

- *literals*   number of literals per cube.

- *RM*   times for common Reed-Muller transformation.

- *GRM*   times for heuristic polarity for less changes.

In Table VII the execution times for different input functions and for two different polarities are shown. The polarity of the GRME transformation is chosen by some heuristic methods to obtain a GRME with less changes in the input function. Hence, for the GRME transformation the execution time is significantly shorter.

TABLE VII

EXECUTION TIME FOR THE REED-MULLER TRANSFORMATION

cubes	literals	type	RM		GRM	
			u	s	u	s
1	10	no	0.0	0.1		
1	10	some	0.0	0.1		
1	10	many	0.0	0.1		
10	10	no	0.7	0.1	0.1	0.1
10	10	some	0.1	0.1		
10	10	many	0.1	0.1		
1	14	no	0.5	0.1		
1	14	some	0.3	0.1		
1	14	many	0.2	0.1		
10	14	no	13.9	0.2	2.0	0.1
10	14	some	1.5	0.1		
10	14	many	1.3	0.1		
1	18	no	5.0	0.1		
1	18	some	3.6	0.1		
1	18	many	3.5	0.1		
10	18	no	124.1	1.3	32.0	0.4
10	18	some	20.5	0.1		
10	18	many	20.2	0.1		
1	22	no	177.6	0.1		
1	22	some	57.1	0.1		
1	22	many	56.3	0.1		
10	22	no	-	-	511.9	2.7
10	22	some	324.2	0.2		
10	22	many	323.6	0.1		

II.3 GENERALIZED ESOPE TO SOPE TRANSFORMATION

II.3.1 Description

The algorithm introduced here is a general case of a transformation from an ESOPE to a SOPE. Hence, the inverse RM transformation is just one special case of this transformation.

The general idea of the method to generate any ESOPE to a SOPE is that for each possible pair of ESOPE terms their overlapping parts have to be removed, which means applying the identity: term $\oplus$ term $= 0$.

Example II.9

In Figure 5 the method of removing overlapping parts is shown on a simple example function.

a. $\bar{a}\mathrm{b}\,\hat{}\,\mathrm{cd}$ b. $\bar{a}\mathrm{b}\bar{c} + \mathrm{acd} + \bar{a}\bar{b}\mathrm{cd} + \bar{a}\mathrm{bc}\bar{d}$
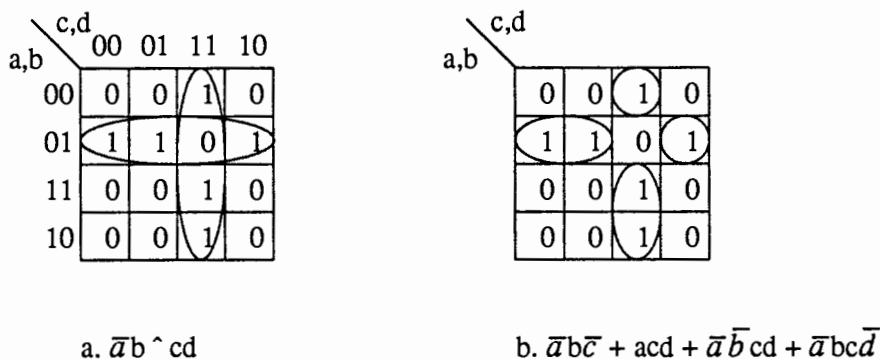
Figure 5. Karnaugh maps for ESOPE and SOPE.

In Figure 5a. the ESOPE of the function $\bar{a}\mathrm{b} \oplus \mathrm{cd}$ is shown. By removing the overlapping part in the Karnaugh map one obtains a SOPE of this function.

To remove all overlapping parts for several terms each pair of terms has to be compared similarly to the algorithm to generate disjoint cubes introduced in Chapter II.1. There is only one difference between the method to generate disjoint cubes from nondisjoint ones and the method to generate a SOPE from any ESOPE. While in the Disjoint Algorithm the sharp operation (22) is performed between cubes that are nondisjoint, for the ESOPE to SOPE transformation algorithm the overlapping part of the two cubes has to be removed. The removal of this part can be done by performing the sharp operation between each of the two terms and their overlapping part.

In the case of a description without Karnaugh maps the overlapping part is determined by the intersection (22) of the cubes.

With the above described change of the disjoint algorithm one can directly obtain the algorithm shown below to generate an SOPE for any ESOPE.

Algorithm : ESOPE to SOPE Transformation

step 1. *set* a and b to the following positions in the term list :

$a := 1, b := n, m := n$

where n is the initial number of terms in the term list.

step 2. Main loop :

For each pair of terms $term_a$ and $term_b$ from an array A

if an intersection of terms $term_a$ and $term_b$ is non-empty set

then

do for $term_a$ and for $term_b$

{ if $term_b$ equals the intersection

then do { if $term_b$ is the last of the array then remove $term_b$

else { substitute $term_b$ by the last term of the array A ;

$n := n - 1; $ }}

else if $term_a$ equals the intersection

then do { substitute $term_a$ by the last term of the array A ;

$n := n - 1;$

$b := n$ }

else do for { calculate the d solution terms from the disjoint sharp

operation $term_{a/b}$ $\#_j$ *intersection* ;

replace $term_{a/b}$ by one solution term and add

the other ones to the end of the array A ;

$n := n + d - 1;$

if $b \leq a$ *then* go to step 3 }

}

go to step 2.

else do { $b := b - 1;$ if $b \leq a$ *then* go to step 3

else go to step 2}

step 3.

$b := n;$

$a := a + 1;$

if $a = m$ *then* stop

else $m := n;$

go to step 2.

II.3.2 Implementation

From the discussed difference between the Disjoint Algorithm (Chapter II.1) and the algorithm for ESOPE to SOPE Transformation, one can easily change the implementation of the Disjoint Algorithm for the ESOPE to SOPE Transformation.

According to this the program makes use of the same structures described for the Disjoint Algorithm.

II.3.3 A complete example

In Figure 6a a step by step transformation of the ESOPE of Figure 6a to the final SOPE in Figure 6e is shown.

First the terms 1X00 and X1XX are intersected. Because the intersection is not empty, the intersection cube is sharped from both terms. The initial two terms in the array are substituted by the two of the solution terms of the sharp operation. The other solution terms are appended to the end of the array (Figure 6b). Next the following pair of terms is taken. This procedure is repeated until every pair determined by the shown algorithm is compared.

II.3.4 Execution times

The here used notation is the same as shown in Chapter II.2.4.

TABLE VIII

EXECUTION TIMES FOR ESOPE TO SOPE TRANSFORMATION

cubes	literals	type u	time s	
1	8	no	0.0	0.1
10	8	no	0.1	0.1
10	8	some	0.0	0.1
10	8	many	0.0	0.1
1	16	no	0.4	0.1
10	16	no	8.9	0.1
10	16	some	6.1	0.1
10	16	many	5.1	0.1
1	22	no	27.3	0.1
10	22	no	545.9	3.2
10	22	some	341.9	0.3
10	22	many	340.0	0.1

$x_1 x_2$ \ $x_3 x_4$

X1,X2,X3,X4

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

1X00 ON
1X1X ON
XX11 ON
X1XX ON
0000 DC

a. input array

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

1000 ON
1X1X ON
XX11 ON
01XX ON
0000 DC
111X ON
1101 ON

b. first loop

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

1000 ON
101X ON
XX11 ON
01XX ON
0000 DC
1101 ON

c. second loop

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

1000 ON
1010 ON
0X11 ON
01XX ON
0000 DC
1101 ON
1111 ON

d. third loop

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

1000 ON
1010 ON
0X11 ON
010X ON
0000 DC
1101 ON
1111 ON
0110 ON
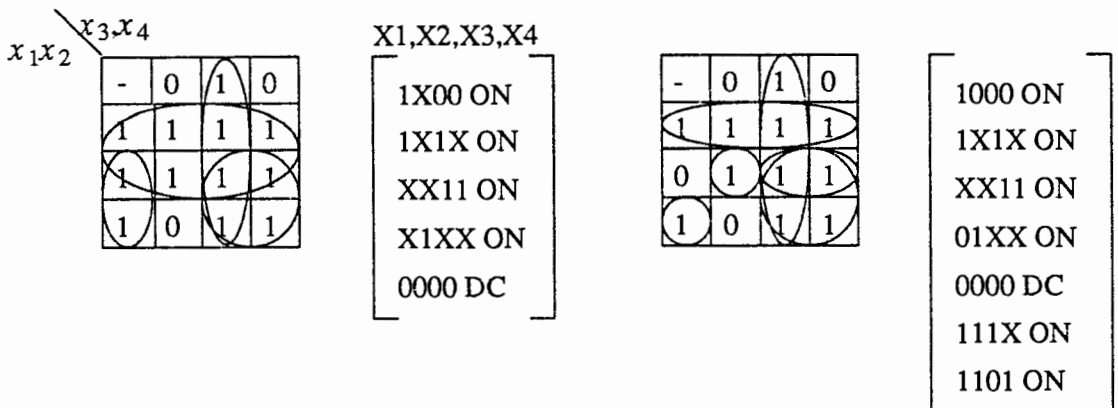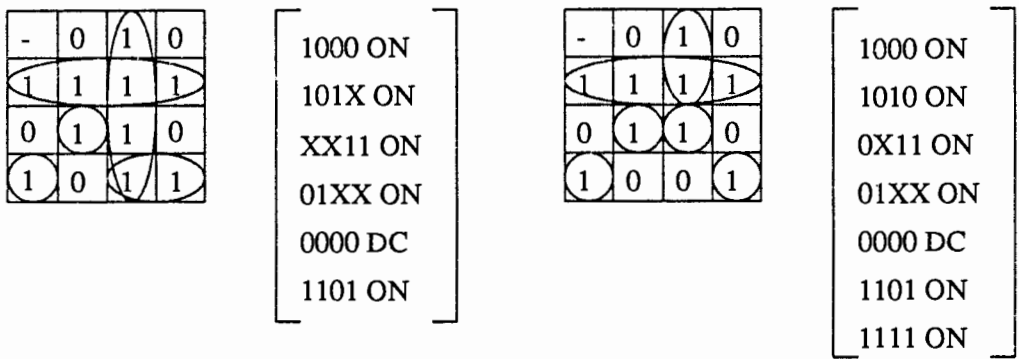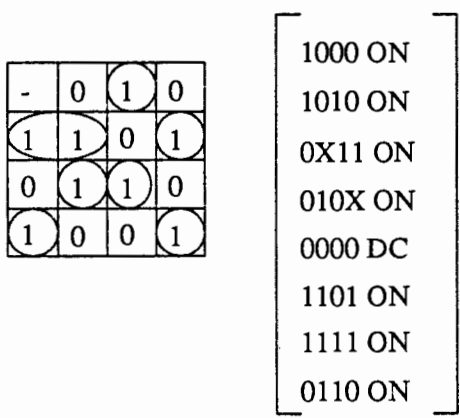
e. output array

Figure 6. Step by step execution of ESOPE to SOPE transformation

II.4 GENERALIZED ADDING AND ARITHMETIC TRANSFORMATIONS

II.4.1 Description

In (27) the Generalized Adding and Arithmetic Transformations (GAD, GAR) were introduced. We expect, that these transformations can be useful for applications in multidimensional signal processing and image processing as well as for designing and testing of digital circuits.

As mentioned in Chapter I. the GAD and GAR Transformations (GARD Transformation, for short) are generated by some essential order-2 matrices. These are shown in (Figure 7).

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

a. Adding   b. Arithmetic

Figure 7. Essential order-2 matrices for GAD and GAR Transformations.

As an application of the Cube comparison method for this transform the R spectrum is used. The R spectrum is a conventional coding scheme where the value of the spectral coefficient <0, 1, 0.5> corresponds to <off 0, on 1, dc -> representing the possible minterm type.

Because the GARD and GRME Transformations have similar order-2 matrices, the same Cube Comparison method as for the GRME can be applied. The difference between the GARD Transformation with respect to the GRME Transformation is, that for the first transformation the value of the spectral coefficient is an important magnitude.

The Properties to generate the spectrum for the GARD Transformation according to the Cube Comparison method make use of the same notation as in the previous Chapters. The only additional variable used here is defined below.

pol   is used as a prefix to indicate that the term/index is matched with the polarity

Properties for the GARD Transformation:

Property II.2

All terms are matched with the polarity cube according to the equivalence opera-

tion (Definition II.1), see Theorem II.3.

$$polterm = term \equiv polarity$$

Property II.3

The value of the spectral coefficient is according to the R vector, <on-1.0, dc-0.5>

when the polterm is covered by the polindex = index ≡ polarity:

$$polterm \subseteq polindex$$

Property II.4

The complete spectrum for the array of minterms is calculated by adding the

spectra for each minterm.

For the GAR transformation also negative values of spectral coefficients can also

occur. The sign for the value of the spectral coefficients depends on the order (number of

subindices of the coefficients) of the coefficient. Thus, the straight order defined in

Chapter 1 is a convenient order to calculate the sign. The following Properties for the

GAR transformation define the calculation of the signs.

Property II.5

The value of a spectral coefficient calculated according to Property II.4 has to be

negated if the number of ones in polterm is odd.

Property II.6

Additionally to Property II.5 the value of the spectral coefficient has to be negated

if the order of the coefficient is even.

From these Properties the algorithm to generate the spectrum for the GARD

Transformation has been developed. The notation listed below is used.

value the value of one spectral coefficient is stored in this variable according to Property II.3.

sign determines the sign of the spectral coefficient calculated for the GAR Transformation according to Property II.4, II.5.

Algorithm: GARD Transformation for multiple polarities

do for each minterm

{

    polterm = term ≡ polarity;

    if number of ones in polterm is odd and GAR Transformation is chosen

        sign = -1;

    else sign = 1;

    do for each spectral coefficient

    {

        polindex = index ≡ polarity;

        if order of spectral coefficient is even and GAR Transformation is chosen

            sign = sign * -1;

        if polterm = polterm ∩ polindex

        {

            if dc minterm

                value = value + sign * 0.5;

            else

                value = value + sign * 1.0;

        }

    }

}

An example which shows the different steps of these algorithm can be found in Chapter II.4.3.

II.4.2 Implementation

The algorithm especially designed for computer implementation is suitable for the generation of the C-code. As its advantage all spectral coefficients are generated separately for the whole set of minterms, so no memory to store the spectral coefficient is needed. Therefore, the spectral coefficients can directly be stored on the hard disk. The structure *struct minterm* is used to store the minterms.

```
struct minterm
{
        unsigned long value;      /* bit representation of the minterm literals */
        float type;    /* is according to minterm type 1.0, 0.5 or 0.0  */
}term;
```

In the field *minterm->value* the ones of the input minterm are stored. Thus the maximum number of literals in the input function can be 32. The field *minterm->type* determines the value of the spectral coefficients according to the minterm type (on - 1; dc - 0.5). This way of representing the type is used because it directly determines the value of the spectral coefficient.

To use an array for the minterms, which can match its size according to the number of input minterms, a dynamic memory allocation is used. The pointer *unsigned int *minterm-list* is therefore taken to store the addresses of the minterms. Because of the use of the function realloc() for the dynamic memory allocation, it can be treated like an array of addresses.

As shown in Chapter II.4.1, the straight order is necessary to calculate the value of the spectral coefficients for the GAR transformation. Hence, an algorithm has been developed to generate the indices in this order. In this algorithm the complete set of indices is calculated respective to the given order. The algorithm is illustrated in Example II.10.

Example II.10:

The indices of the spectral coefficients of the first order have only one literal. Respectively, the cube representations of the indices have one 1 each. The first order indices of a five-valued function are shown in Table IX.

TABLE IX

FIRST ORDER INDICES OF SPECTRAL COEFFICIENTS

coefficient	S_1	S_2	S_3	S_4	S_5
index	10000	01000	00100	00010	00001

The third order coefficients of the same function are shown in Table X.

TABLE X

THIRD ORDER INDICES OF SPECTRAL COEFFICIENTS

coefficient	index
S_{123}	11100
S_{124}	11010
S_{125}	11001
S_{134}	10110
S_{135}	10101
S_{145}	10011
S_{234}	01110
...	...
S_{345}	00111

Let us denote by k the number of literals, necessary for the current order. As it can be observed, to obtain all indices for one order of coefficients, one has to generate all possible arrangements of k symbols "1" in a field of the length determined by the number of input variables. For this purpose a recursive algorithm has been designed. The recursive algorithm behaves like a certain set of loops, where, their number is determined by the order that has to be generated.

For the third order coefficient generation shown in Table X, three loops are neces-

sary. The outer loop is responsible for changing the position of the leftmost "1" from the first to the third position. In the next inner loop the "1" on the position right to the leftmost "1" has to go to the position right to the third position. Again for the next inner loop the "1" right to the "1" from the next outer loop has to go right to the last position of the "1" in the next outer loop.

The procedure consisting of these three loops looks as follows.

Notation:

a,b,c   : determine the positions of the three ones.

index   : is the final index such as one shown in Table X

first[i] : is the index *i* of the spectral coefficient S_i

```
for ( a = 0 ; a <= 3 ; a++ )
{
        for ( b = a+1 ; b <= 4 ; b++ )
        {
                        for ( c = b+1 ; c <= 5 ; c++ )
                                index = first[a] + first[b] + first[c];
        }
}
```

As seen above, for each different order another number of loops is necessary. To overcome this problem, a recursive algorithm has been designed that behaves such as a given number of loops.

Notation:

from : determines the start number of the loop.

to   : determines the end number of the loop.

order : determines the order that should be generated.

times : determines the current depth of the recursion.

44

Algorithm: One order of coefficients for the GARD Transformation.

```
short order; /* calculated order    */

gen_coeff(from,to,times)
short from;   /* loop beginning    */
short to;    /* loop end        */
short times; /* depth of recursion  */
{
 int i,k;
 float coeff;  /* final value of the spectral coefficient */

 times++;
 for ( i = from ; i <= to ; i++ )
 {
    index = index + first[i+1];
    if ( times < order )
       gen_coeff(i+1,to+1,times);
    else
    {
      coeff = 0;
      for ( k = 0 ; k < minterms ; k++ )
      {
         minterm = minterm_list[k];
         if ( (minterm->value & ((index)^(!polarity))) ) == minterm->value )
            coeff = coeff + (float)sign * minterm->type;
      }
      output(coeff);
    }
    index = index-first[i+1];
 }
}
```

II.4.3 A complete example

An example for both, the Arithmetic and the Adding transformation illustrates the generation of the spectral coefficients. In TABLE XI the array of minterms for the chosen function is shown. The last column of the minterms determines: "-" the dc-minterm, "1" the on-minterm.

TABLE XI

INCOMPLETELY SPECIFIED BOOLEAN FUNCTION

minterm	type
0001	1
0101	1
1001	1
1010	1
1110	1
0111	-
1111	-

In the Tables XII and XIII the minterms of the function shown in Table XI are applied to generate the spectra for the GAR and GAD Transformations. The final spectra, calculated by adding the values of all partial coefficients are shown in the last rows of Tables XII and Table XIII.

TABLE XII

SPECTRUM S FOR THE GAD TRANSFORMATION
FROM THE FUNCTION OF TABLE XI

minterm	S_0	S_1	S_2	S_3	S_4	S_{12}	S_{13}	S_{14}	S_{23}	S_{24}	S_{34}
0001 on	0	0	0	0	1	0	0	1	0	1	1
0101 on	0	0	0	0	0	0	0	0	0	1	0
1001 on	0	0	0	0	0	0	0	1	0	0	0
1010 on	0	0	0	0	0	0	1	0	0	0	0
1110 on	0	0	0	0	0	0	0	0	0	0	0
0111 dc	0	0	0	0	0	0	0	0	0	0	0
1111 dc	0	0	0	0	0	0	0	0	0	0	0
result	0	0	0	0	1	0	1	2	0	2	1

minterm	S_{123}	S_{124}	S_{134}	S_{234}	S_{1234}
0001 on	0	1	1	1	1
0101 on	0	1	0	1	1
1001 on	0	1	1	0	1
1010 on	1	0	1	0	1
1110 on	1	0	0	0	1
0111 dc	0	0	0	.5	.5
1111 dc	0	0	0	0	.5
result	2	3	3	2.5	6

46

TABLE XIII

SPECTRUM S FOR THE GAR TRANSFORMATION
FROM THE FUNCTION OF TABLE XI

minterm	S_0	S_1	S_2	S_3	S_4	S_{12}	S_{13}	S_{14}	S_{23}	S_{24}	S_{34}
0001 on	0	0	0	0	1	0	0	-1	0	-1	-1
0101 on	0	0	0	0	0	0	0	0	0	1	0
1001 on	0	0	0	0	0	0	0	1	0	0	0
1010 on	0	0	0	0	0	0	1	0	0	0	0
1110 on	0	0	0	0	0	0	0	0	0	0	0
0111 dc	0	0	0	0	0	0	0	0	0	0	0
1111 dc	0	0	0	0	0	0	0	0	0	0	0
result	0	0	0	0	1	0	1	0	0	0	-1

minterm	S_{123}	S_{124}	S_{134}	S_{234}	S_{1234}
0001 on	0	1	1	1	-1
0101 on	0	-1	0	-1	1
1001 on	0	-1	-1	0	1
1010 on	-1	0	-1	0	1
1110 on	1	0	0	0	-1
0111 dc	0	0	0	.5	-.5
1111 dc	0	0	0	0	.5
result	0	-1	-1	.5	1

As mentioned above, each spectral coefficient in the program is directly generated by adding the values for each minterm. Therefore, it is generated for the whole set of minterms before calculating the next coefficient. Within each coefficient (denoted as coeff in the algorithm) the value for each minterm is added to the initially zero value of coeff. For instance, for the GAD transformation the intermediate values are shown in Table XIV.

TABLE XIV

INTERMEDIATE VALUES FOR THE GAD TRANSFORMATION

minterm	S_0	S_1	S_2	S_3	S_4	S_{12}	S_{13}	S_{14}	S_{23}	S_{24}	S_{34}
0001 on	0	0	0	0	1	0	0	1	0	1	1
0101 on	0	0	0	0	1	0	0	1	0	2	1
1001 on	0	0	0	0	1	0	0	2	0	2	1
1010 on	0	0	0	0	1	0	1	2	0	2	1
1110 on	0	0	0	0	1	0	1	2	0	2	1
0111 dc	0	0	0	0	1	0	1	2	0	2	1
1111 dc	0	0	0	0	1	0	1	2	0	2	1
result	0	0	0	0	1	0	1	2	0	2	1

minterm	S_{123}	S_{124}	S_{134}	S_{234}	S_{1234}
0001 on	0	1	1	1	1
0101 on	0	2	1	2	2
1001 on	0	3	2	2	3
1010 on	1	3	3	2	4
1110 on	2	3	3	2	5
0111 dc	2	3	3	2.5	5.5
1111 dc	2	3	3	2.5	6
result	2	3	3	2.5	6

Execution times for the GAD transformation will be presented in Chapter II.5.4.

II.5 INVERSE ARITHMETIC AND ADDING TRANSFORM

II.5.1 Description

The INverse Generalized ARithmetic and aDding transformations here called INGARD, is one single algorithm for both the inverse Adding and the inverse Arithmetic Transformation. Let us first observe the main Property of the forward GARD Transformation, that led to the INGARD algorithm.

Property II.7

In the smallest order where not all spectral coefficients are equal to zero, every spectral coefficient can have only the values (+/-)1 or (+/-)0.5.

Because of the importance of this Property it will be illustrated in the following Example.

Example II.11

The complete set of minterms for a four variable input function and their complete spectra are shown in Table XV. The minterms are ordered in a way, to illustrate the above Property. Hence, one can observe that the leftmost 1 of the spectrum in the row below is always right of the leftmost 1 of the spectrum in a certain row.

TABLE XV

ILLUSTRATION OF THE PROPERTY OF
THE SMALLEST ORDER

minterm	S_0	S_1	S_2	S_3	S_4	S_{12}	S_{13}	S_{14}	S_{23}	S_{24}	S_{34}
0000	1										
1000		1				1	1	1			
0100			1			1			1	1	
0010				1			1		1		1
0001					1			1		1	1
1100						1					
1010							1				
1001								1			
0110									1		
0101										1	
0011											1
1110											
1101											
1011											
0111											
1111											

TABLE XV

ILLUSTRATION OF THE PROPERTY OF
THE SMALLEST ORDER
(continued)

minterm	S_{123}	S_{124}	S_{134}	S_{234}	S_{1234}
0000					
1000	1	1	1		1
0100	1	1		1	1
0010	1		1	1	1
0001		1	1	1	1
1100	1	1			1
1010	1		1		1
1001		1	1		1
0110	1			1	1
0101		1		1	1
0011			1	1	1
1110	1				1
1101		1			1
1011			1		1
0111				1	1
1111					1

The following Properties can be derived directly from Property II.7.

Property II.8

The set of minterms can be calculated by generating the minterm always for the leftmost (for the straight order) non-zero spectral coefficient, where the spectrum of the minterm has to be subtracted from the complete spectrum.

Property II.9

The next leftmost spectral coefficient not equal to zero (as defined in Property II.7) is always right (for the straight order) of the previous one.

Property II.10

The value of the leftmost spectral coefficient must always be (+/-)1 or (+/-)0.5 depending on the type of the minterm, either on- or dc-minterm, and the transformation, either the inverse Adding or the inverse Arithmetic one.

Property II.

> The correct solution minterm is calculated by performing the equivalence opera-
> tion (Definition II.1) between the minterm generated according to Properties II.7 -
> II.10 and the polarity cube.

> The algorithm derived from these Properties is shown below:

Algorithm : INGARD

    do

        take leftmost spectral coefficient with value $\neq 0$;

        minterm = index of this spectral coefficient;

        generate spectrum for this minterm;

        subtract spectrum of the minterm from the total spectrum;

        solution minterm = minterm $\equiv$ polarity

    while (spectrum $\neq$ zero)

For the Arithmetic Transformation the negative values for the value of the spec-
tral coefficients can occur. Therefore, it is possible that the values for some spectral
coefficients can reduce themselves to zero. However, as shown for the inverse Adding
Transformation, this can not occur in the smallest order (Property II.7). Hence, the same
algorithm as for the inverse Adding Transformation can be taken for the inverse Arith-
metic Transformation as well.

II.5.2 Implementation

For the implementation of the algorithm it is necessary to read from the input file
the values of the spectral coefficients according to their indices. The routine which per-
forms this task is similar to the *gen_coeff* procedure shown in Chapter II.4.2 for the for-
ward transformation. In the *gen_coeff* procedure the part where the coefficient is

calculated has to substituted by the input of the value of the spectral coefficient from the hard disk file. Below the structure in which the value has to be read is shown.

```
struct spectral_coef
{
        unsigned long index;/* subscript index  of the spectral coefficient */
        double value;  /* value of spectral coeff      */
        unsigned short order;   /* order of the spectral coefficient        */
}*spectrum;
```

Here the realloc() function can be used for the whole structure. It is then possible to access directly each spectral coefficient like an element of an array. The main procedure to find the leftmost non-zero spectral coefficient and generating the minterm, is shown below. It makes use of the previously defined variables and the additional variables *order_no* and *TRANSFORM* :

order_no

> determines the current processed order.

TRANSFORM

> determines if Arithmetic or Adding Transformation.

Algorithm : INGARD implementation

```
for ( i = 0 ; i < coeffs ; i++ )
{
     order_no = spectrum[i].order;
     if ( spectrum[i].value != 0 )
     {
          /* minterm is identical to the index of the spectral coefficient */
          if ( abs(spectrum[i].value) == 1.0 )
               /* minterm is on minterm */
               type = ON;
          else
               /* minterm is dc minterm */
               type = DC;
          minterm->type = (float)sign * spectrum[i].value;
          minterm->value = (spectrum[i].index  );
          output(minterm);
          /* subtract the spectrum of the minterm from total spectrum */
          subtract_transform(TRANSFORM);
     }
}
```

II.5.3 A complete example

For the illustration of the inverse transformation the spectrum generated by the forward Arithmetic Transformation according to Table XI has been chosen.

TABLE XVI

INVERSE GAR TRANSFORMATION

	S_0	S_1	S_2	S_3	S_4	S_{12}	S_{13}	S_{14}	S_{23}	S_{24}	S_{34}
spectrum	0	0	0	0	1	0	1	0	0	0	-1
0001 on	0	0	0	0	0	0	1	1	0	1	0
1010 on	0	0	0	0	0	0	0	1	0	1	0
1001 on	0	0	0	0	0	0	0	0	0	1	0
0101 on	0	0	0	0	0	0	0	0	0	0	0
1110 on	0	0	0	0	0	0	0	0	0	0	0
0111 dc	0	0	0	0	0	0	0	0	0	0	0
1111 dc	0	0	0	0	0	0	0	0	0	0	0

	S_{123}	S_{124}	S_{134}	S_{234}	S_{1234}
spectrum	0	-1	-1	.5	1
0001 on	0	-2	-2	-.5	2
1010 on	1	-2	-1	-.5	1
1001 on	1	-1	0	-.5	0
0101 on	1	0	0	.5	-1
1110 on	0	0	0	.5	0
0111 dc	0	0	0	0	.5
1111 dc	0	0	0	0	0

The spectrum shown in the first row of Table XVI is the initial spectrum. Now the minterm for the leftmost spectral coefficient not equal to zero (S_4) has to be generated. The spectrum for this minterm is immediately subtracted from the previous spectrum. The new spectrum is shown in the second row of the Table, where the minterm is shown in the first column. The loop to generate all minterms is repeated until the generated spectrum becomes zero for all coefficients.

II.5.4 Execution times for GARD and INGARD Transformations

Because of the nature of the forward GAD Transformation, the execution time

does not depend on the ratio of on- to off-literals in a term. The Table XVII presents first the time for the forward GAD Transformation, followed by the time for the inverse GAD Transformation for the same example. The first column includes the number of minterms and the second column gives the number of literals of those minterms. The same example as used to determine the execution time was assumed. The symbols u and s for the elapsed time have the same meaning as in Chapter II.2.4.

TABLE XVII

EXECUTION TIMES FOR THE GAD AND
INGAD TRANSFORMATIONS

minterms	literals	forward transform		inverse transform	
		u	s	u	s
1	8	0.1	0.1	0.1	0.2
10	8	0.5	0.1	0.5	0.2
1	16	22.6	0.3	23.3	0.9
10	16	31.2	0.3	55.1	1.0

II.6 RADEMACHER-WALSH TRANSFORMATION

II.6.1 Description

An algorithm to calculate the spectral coefficients of the Rademacher-Walsh Transformation for completely specified Boolean functions directly from the SOPE was shown in (4,3). It has the disadvantage that for several cases an additional correction of the Boolean function was required. This algorithm was improved to make use of completely and incompletely specified Boolean functions represented in an array of disjoint cubes such as generated with the algorithm shown in Chapter II.1. This method (12,13) has been applied in the development of the algorithm for the computer implementation (14,15).

Let us first recall the basic properties for the generation of the Rademacher-Walsh

spectrum directly from the representation of disjoint cubes (12,15).

Example II.12

The array of disjoint cubes generated in Example II.1 is used to generate the spectrum S shown in Table XVIII. Each row contains the values of the spectral coefficients for the given cube. The final spectrum shown in the last row is calculated by the sum of the respective values for the cubes, the only exception is the dc-coefficient S_0.

TABLE XVIII

SPECTRUM S OF THE RADEMACHER-WALSH TRANSFORMATION

cube	S_0	S_1	S_2	S_3	S_4
1-00 on	12	4	0	-4	-4
1-1- on	8	8	0	8	0
0-11 on	12	-4	0	4	4
010- on	12	-4	4	-4	0
0000 dc	7	-1	-1	-1	-1
1101 on	14	2	2	-2	2
0110 on	14	-2	2	2	-2
spectrum	-9	3	7	3	-1

cube	S_{12}	S_{13}	S_{14}	S_{23}	S_{24}	S_{34}
1-00 on	0	4	4	0	0	-4
1-1- on	0	-8	0	0	0	0
0-11 on	0	4	4	0	0	-4
010- on	4	-4	0	4	0	0
0000 dc	-1	-1	-1	-1	-1	-1
1101 on	-2	2	-2	2	-2	2
0110 on	2	2	-2	-2	2	2
spectrum	3	-1	3	3	-1	-5

TABLE XVIII

SPECTRUM S OF THE RADEMACHER-WALSH TRANSFORMATION
(continued)

cube	S_{123}	S_{124}	S_{134}	S_{234}	S_{1234}
1-00 on	0	0	4	0	0
1-1- on	0	0	0	0	0
0-11 on	0	0	-4	0	0
010- on	4	0	0	0	0
0000 dc	-1	-1	-1	-1	-1
1101 on	-2	2	-2	-2	2
0110 on	-2	2	2	-2	-2
spectrum	-1	3	-1	-5	-1

In Table XVIII one can observe the following Properties (taken from (14,15)) of the Rademacher-Walsh spectrum:

1   The contribution of the on-cube of degree m to full n-space spectrum of function F (where n is a number of variables, and p is a number of dc-variables in the function F) is related as follows:

s_0 in full n-space $= 2^n - 2 \times 2^p$

and

s_I in full n-space $= s_I$ in m-space $\times 2^p$, where $I \neq 0$.

2   The contribution of the dc-cube of degree m to full n-space spectrum of function F is related as follows:

s_0 in full n-space $= 2^{n-1} - 2^p$

and

s_I in full n-space $= s_I$ in m-space $\times 2^{p-2}$, where $I \neq 0$.

The following properties of the signs of each spectral coefficient s_I, where $I \neq 0$, are valid for on- and dc-cubes of any degree:

3   If in a given cube the x_i variable of a Boolean function is in affirmation, then the sign of the corresponding first-order coefficient is positive, otherwise for a variable that is in negation, the sign of the corresponding first-order coefficient is negative.

4   The signs of all even-order coefficients are given by the negation of the multiplication of the signs of the related first-order coefficients.

5   The signs of all odd-order coefficients are given by the multiplication of the signs of the related first-order coefficients.

The following properties have to be applied additionally for the dc spectral coefficient:

6   The value of a dc spectral coefficient s_0 is equal for a completely specified Boolean function to the sum of all the corresponding contributions from all on-disjoint cubes, but it requires a correction factor $-(k-1) \times 2^n$, where k is a number of disjoint cubes in the on-array of cubes.

7   The value of a dc spectral coefficient s_0 is equal for an incompletely specified Boolean function to the sum of all the corresponding contributions from all on- and dc-disjoint cubes, but it requires a correction factor $-(k-1) \times 2^n - l \times 2^{n-1}$, where k is the number of disjoint on- cubes, and l is the number of disjoint dc-cubes.

II.6.2 Implementation

It has been investigated to make use of the Cube Comparison method for the implementation of the Rademacher-Walsh Transformation. As one can observe from Table XVIII and from the properties given in Chapter II.6.1, the value of the spectral coefficients depends on the positive as well as on the negative literals of the cubes. The Cube Comparison method to obtain the spectral coefficients with non-zero value from the spectrum is the same as for the RME Transformation (Theorem II.1.). Similarly to the

Arithmetic Transformation, the signs of the value of the spectral coefficients depend on the order of the coefficient. Thus, again the straight order (Rademacher-Walsh order) has to be applied. Because of these similarities, the algorithm uses the same structure as for the GRM Transformation (Chapter II.2.2). The procedure to generate the indices in the straight order is similar to the *gen_coeff* procedure (Chapter II.4.2) for the GARD Transformation.

The meaning of the different fields in the structure has changed To adopt to the Cube comparison method necessary for the Rademacher-Walsh Transformation. Therefore the structure *term_field* is shown below.

```
struct term_field
{
    unsigned long value1;    /* is "1" for dc- and on- literals of the term    */
    unsigned long valuex;    /* is "1" for the dc-literals of the term    */
    unsigned short order;    /* value according number of dc's    */
    unsigned short type; /* determines on-, dc- or off-type of cube */
}*term;
```

The procedure to generate the indices according to the straight order is shown below. It generates one complete order of spectral coefficients for the given order in the variable *order*. The variable *sign* determines if the order is an odd or even one. In the subroutine value_coeff() the final sign *s* of the spectral coefficient has to be determined. This procedure will be described in the sequel. The value of a spectral coefficients depends on the number of dc literals in the cube (Properties 1 and 2). This value is calculated once per cube and stored in the field *term->order*. All other variables and the array (first[i]) have the same meaning as in Chapter II.2.2 and II.4.2.

Algorithm : Generation of one order of spectral coefficients

```
short order; /* order that has to be generated    */
short sign;  /* determines sign according to the generated order */
unsigned long index;    /* global variable to store the index  */

gen_coeff(from,to,times)
short from;
```

```
short to;
short times;
{
      int i;
      times ++;
      for ( i = from ; i <= to ; i ++)
            { index = index + first[i+1];
            if ( times < order )
                  gen_coeff(i+1,to+1,times);
            else
                  /* calculation of the value for the whole array of cubes */
                  value_coeff();
            index = index - first[i+1]; }
}
```

The main difference to the GAD, GAR and GRME Transformations is that the sign of the value of the spectral coefficients depends on the sign of the respective coefficients of the first order. Therefore, after the generation of the value for a spectral coefficient, the sign has to be determined. The procedure *value_coeff* has been designed to perform the calculation of the sign according to Properties 3 and 4.

Procedure : Calculation of the sign of the value for a spectral coefficient

```
value_coeff()
{
      short i,k;
      short s; /* local variable for the sign */
      long value;

      value = 0;
for ( k = 0 ; k < cubes ; k++ )
{
      cube = cube_list[k];
      if ( (cube->valuex & index) == 0 )
      {
            /* determine sign of the value */
            s = sign;
            for ( i = 1 ; i <= values ; i++ )
            {
                  if ( (first[i-1] & index) != 0 )
                        /* check if value1 has zero on this position */
                        if ( (first[i-1] & cube->value1) == 0 )
                              /* sign has to be negated */
                              s = s * -1;
            }
            value = value + s*cube->order;
```

}
}
output(value);

With the above algorithm all spectral coefficients except the dc-coefficient S_0 can be generated. This has to be generated separately according to Properties 1,2,6 and 7.

II.6.3 A complete example

The same array of cubes representing the function f as in Example II.1 and II.12 is taken to illustrate the different steps of the Rademacher-Walsh algorithm.

First the dc-coefficient is generated according to the formulas of Properties 1,2,6 and 7. Similarly to the GARD Transformation, the values are immediately added, what is shown in Table XIX. In the last row the final value is given after the required correction according to Properties 6 and 7.

TABLE XIX

GENERATION OF THE DC-COEFFICIENT S_0

cube	S_0
1-00 on	12
1-1- on	20
0-11 on	32
010- on	44
0000 dc	51
1101 on	65
0110 on	79
spectrum	-9

Next all spectral coefficients are generated according to the shown algorithm *gen_coeff()* (Chapter II.6.2), where again each spectral coefficient is calculated by adding immediately the contribution of each cube, from the array of cubes, to the value of the spectral coefficient. The intermediate values of the spectral coefficients are given in Table XX.

TABLE XX

INTERMEDIATE VALUES FOR THE RADEMACHER-WALSH SPECTRUM

cube	S_1	S_2	S_3	S_4
1-00 on	4	0	-4	-4
1-1- on	12	0	4	-4
0-11 on	8	0	8	0
010- on	4	4	4	0
0000 dc	3	3	3	-1
1101 on	5	5	1	1
0110 on	3	7	3	-1
spectrum	3	7	3	-1

cube	S_{12}	S_{13}	S_{14}	S_{23}	S_{24}	S_{34}
1-00 on	0	4	4	0	0	-4
1-1- on	0	-4	4	0	0	-4
0-11 on	0	0	8	0	0	-8
010- on	4	-4	8	4	0	-8
0000 dc	3	-5	7	3	-1	-9
1101 on	1	-3	5	5	-3	-7
0110 on	3	-1	3	3	-1	-5
spectrum	3	-1	3	3	-1	-5

cube	S_{123}	S_{124}	S_{134}	S_{234}	S_{1234}
1-00 on	0	0	4	0	0
1-1- on	0	0	4	0	0
0-11 on	0	0	0	0	0
010- on	4	0	0	0	0
0000 dc	3	-1	-1	-1	-1
1101 on	1	1	-3	-3	1
0110 on	-1	3	-1	-5	-1
spectrum	-1	3	-1	-5	-1

II.6.4 Execution times

In Table XXI execution times for different input functions are shown. The corresponding notation can be found in Chapter II.2.4. The execution times for the new algorithm introduced here are compared to the times of the Spectral Synthesis System of the Drexel University (32,33). The following notation is used:

newalgorithm

times for the algorithm introduced here.

S^3 times of the Spectral Synthesis System.

<div align="center">TABLE XXI</div>

<div align="center">EXECUTION TIMES FOR THE RADEMACHER-WALSH TRANSFORMATION</div>

cube	literal	type	new algorithm		S^3
			u	s	
	10				1.8
1	10	no	0.3	0.1	
1	10	some	0.2	0.2	
1	10	many	0.2	0.1	
10	10	no	1.2	0.1	
10	10	some	0.3	0.2	
10	10	many	0.2	0.2	
	14				32
1	14	no	5.1	0.2	
1	14	some	2.9	0.1	
1	14	many	2.9	0.2	
10	14	no	24.1	0.2	
10	14	some	4.3	0.2	
10	14	many	3.9	0.2	
10	16	no	106.7	0.9	
10	16	some	16.6	0.5	
10	16	many	16.0	0.2	
	18				382
1	18	no	88.6	0.9	
1	18	some	41.4	1.0	
1	18	many	46.7	1.0	
10	18	no	458.0	1.4	
10	18	some	63.2	0.4	
10	18	many	63.2	0.3	

Because the Spectral Synthesis System makes use of a Fast Transformation, the complete set of minterms has to be specified for the transformation. The Cube Com-

parison method can perform the transformation directly on the representation of cubes. Thus, the execution times depend on the number of cubes. Therefore, the execution time for 1 and for 10 cubes of different types is shown in the above table.

CHAPTER III

MULTIPLE-VALUED TRANSFORMATIONS

III.1 MULTIPLE-VALUED WALSH TRANSFORMATION

III.1.1 Description

The spectral representation of multiple-valued input binary output functions for the Walsh Transformation was introduced by B.Falkowski (30). In such a representation the spectrum for each mv-function is composed of a vector of Walsh Transformations, each of them defined for one product of two input variables of the function. For functions having only two input variables the spectrum is composed out of one Walsh Transformation.

Let us first review some basic definitions for multiple-valued input, binary output functions.

Definition III.1

A multiple-valued input, binary output incompletely specified function f (*mv function*, for short) is a mapping $f(X_1, X_2, \ldots, X_n) : P_1 \times P_2 \times \cdots P_n \rightarrow B$, where X_i is a multiple-valued variable (*mv-literal*), and $P_i = \{0,1,\ldots,p_i-1\}$ is a set of *true values* that this variable may assume, p_i is the number of values of variable X_i. B = $\{0,1,-\}$ denotes the off-, on- and don't care value. This is a generalization of an ordinary n-input switching function f: $B^n \rightarrow B$.

Definition III.2

For any subset $S_i \subseteq P_i$, $X_i^{S_i}$ is a literal X_i representing the function such that

$$X_i^{S_i} = \begin{cases} 1 & \text{if } X_i \in S_i \\ 0 & \text{if } X_i \notin S_i \end{cases}$$

Definition III.3

A product of literals, $X_1{}^{S_1}X_2{}^{S_2}....X_n{}^{S_n}$, is referred as a *product term* (also called *term*, *product* or *cube*). A product term that includes literals for all function variables $X_1, X_2,...,X_n$ is called the *full term*.

The Multiple-valued Walsh Transformation (MWT) introduced in (30) is based on an algorithm to convert the mv-function to a set of two-dimensional maps. This description of maps was taken to illustrate the conversion. For the computer implementation, the mv-function and its conversion to a Boolean function are represented as arrays of cubes. The array of cubes representing the multiple-valued function is mapped to its logical equivalent in the form of a Boolean function represented as an array of cubes. From this point on, one can calculate the spectrum of each such Boolean function according to the algorithm described in Chapter II.6. This method of the calculation of Walsh spectrum of Boolean functions requires to represent such functions in the form of arrays of disjoint on- and dc- cubes. After the two-dimensional multiple-valued arrays (every array represents one out of all possible products of two distinct multiple-valued literals) have been converted to Boolean arrays of cubes the algorithm to generate disjoint cubes from nondisjoint ones (Chapter II.1) is applied to all the cubes describing the function. The last step in the generation of a partial spectrum for the mv-function is the calculation of the Walsh spectrum from the arrays of disjoint cubes. The detailed description of the algorithm to perform this task can be found in Chapter II.6. The complete structure of the implementation of the MWT is shown in Figure 8.

Since the binary Walsh Transformation and the disjoint algorithm have already been described in previous Chapters, only the conversion algorithm to change multiple-valued input functions to binary ones will be described in the next section.
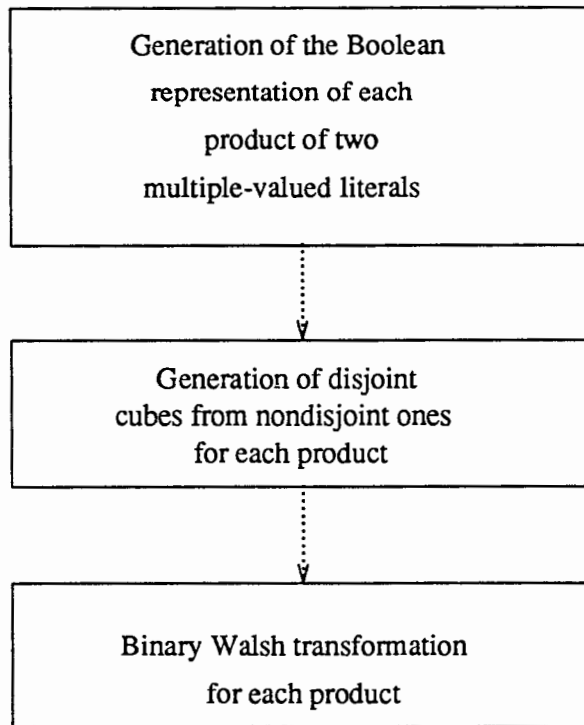
Figure 8. Structure of the multiple-valued Walsh implementation.

III.1.2 Implementation of the conversion algorithm

The general principle is to convert each possible combination of two multiple-valued literals of a binary function to its logical equivalent of Boolean arrays of cubes.

The different steps of the conversion algorithm are illustrated with the mv-function $X^{0,1,2,4}Y^{0,2}$ where X is a five valued literal and Y a three-valued literal. Since the mv-function has only two literals then only one Boolean array of cubes describes fully this function. The conversion is performed by changing the multiple-valued notation (in decimal code) to the binary one. The result of this conversion for the considered mv-function is shown in Figure 9.

The implemented algorithm first generates two list of cubes for each literal of the product of literals. The first list contains the corresponding binary on-representation (true cube) for each value of a singular multiple-valued literal. If a multiple-valued literal does not have a number of 2^n values (where $n=2$ for the literal Y and $n=3$ for the literal

X in our example), then a second list containing *not used values* as the representation of the dc-cubes is used. For example, the literal X is described by the two list in Table XXII and XXIII.

	Y	0	1	-	2
X		00	01	11	10
0	000	1	0	-	1
1	001	1	0	-	1
3	011	0	0	-	0
2	010	1	0	-	1
-	110	-	-	-	-
-	111	-	-	-	-
-	101	-	-	-	-
4	100	1	0	-	1

multiple-valued product of two literals :

$$X^{0,1,2,4} \, Y^{0,2}$$

this product converted to the binary array :

ON - cubes :

$$\begin{bmatrix} 00000 \\ 00010 \\ 00100 \\ 00110 \\ 01000 \\ 01010 \\ 10000 \\ 10010 \end{bmatrix}$$
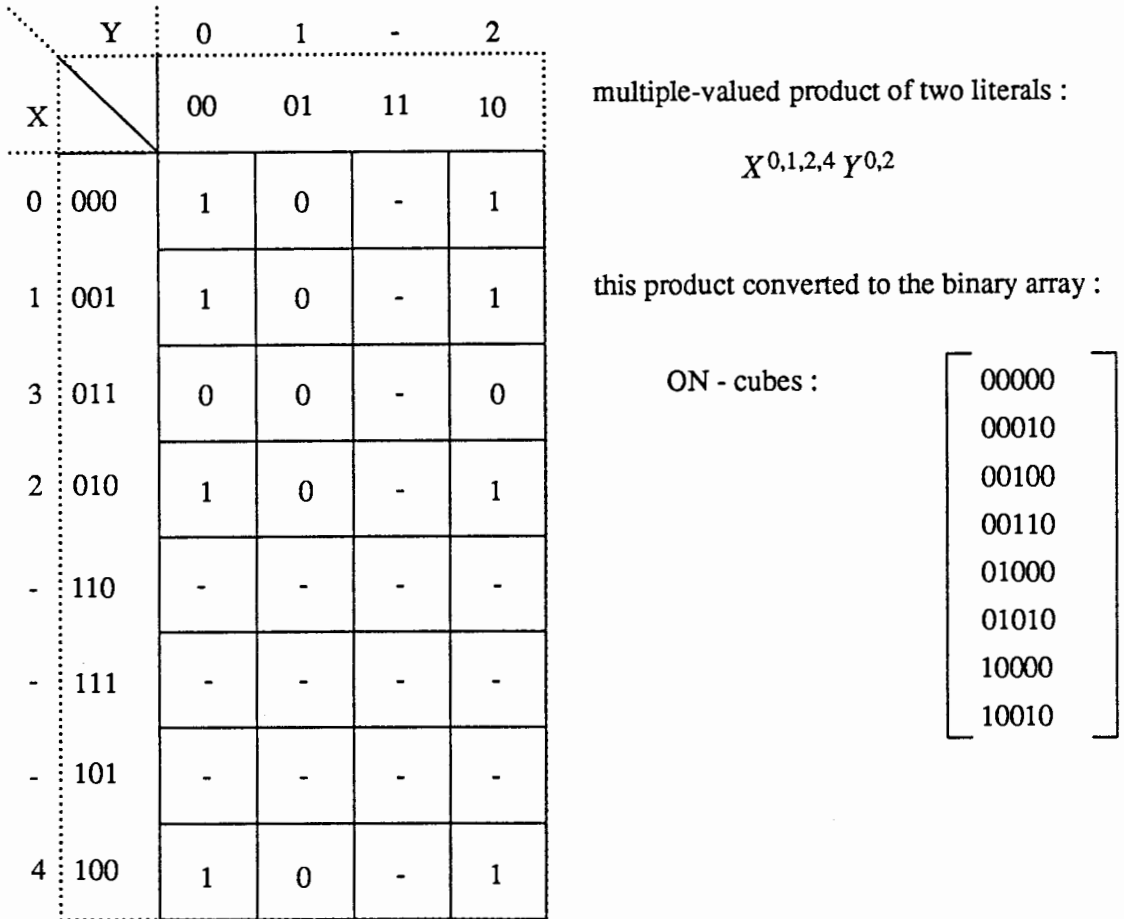
Figure 9. Conversion of multiple-valued product of literals.

TABLE XXII

BINARY REPRESENTATION OF A MULTIPLE-VALUED LITERAL

on cube	dc cube
000	110
001	111
010	101
100	

In order to obtain larger cubes, the cubes of each of these lists are merged (Table XXIII).

<div align="center">TABLE XXIII</div>

<div align="center">MERGED LIST OF THE BINARY REPRESENTATION
OF TABLE XXII</div>

on cube	dc cube
001	110
010	1-1
-00	

Now the binary representations of the cubes of literal X are shifted m positions to the left, where m is the number of bits necessary to represent the multiple-valued literal Y. Analogously, the cubes describing the literal Y are shifted n positions to the right, where n is the number of positions used in the description of the multiple-valued literal X. For our example, m = 3, and n = 2. The shifted positions in the cubes are filled with don't care symbols. The result at this stage is shown in Table XXIV.

<div align="center">TABLE XXIV</div>

<div align="center">SHIFTED BINARY REPRESENTATION OF TWO
MULTIPLE-VALUED LITERALS</div>

X on cubes	Y on cubes
001--	----0
010--	
-00--	

At the final stage, the cubes describing X and Y are intersected, and Figure 10 shows the complete conversion of the product of the two literals.

X \ Y		0	1	-	2
		00	01	11	10
0	000	1	0	-	1
1	001	1	0	-	1
3	011	0	0	-	0
2	010	1	0	-	1
-	110	-	-	-	-
-	111	-	-	-	-
-	101	-	-	-	-
4	100	1	0	-	1

multiple-valued product of literals :

$$X^{0,1,2,4}\, Y^{0,2}$$

converted product of literals :

ON - cubes :

$$\begin{bmatrix} -00-0 \\ 001-0 \\ 010-0 \end{bmatrix}$$

DC - cubes :

$$\begin{bmatrix} 1-1-- \\ 110-- \\ ---11 \end{bmatrix}$$
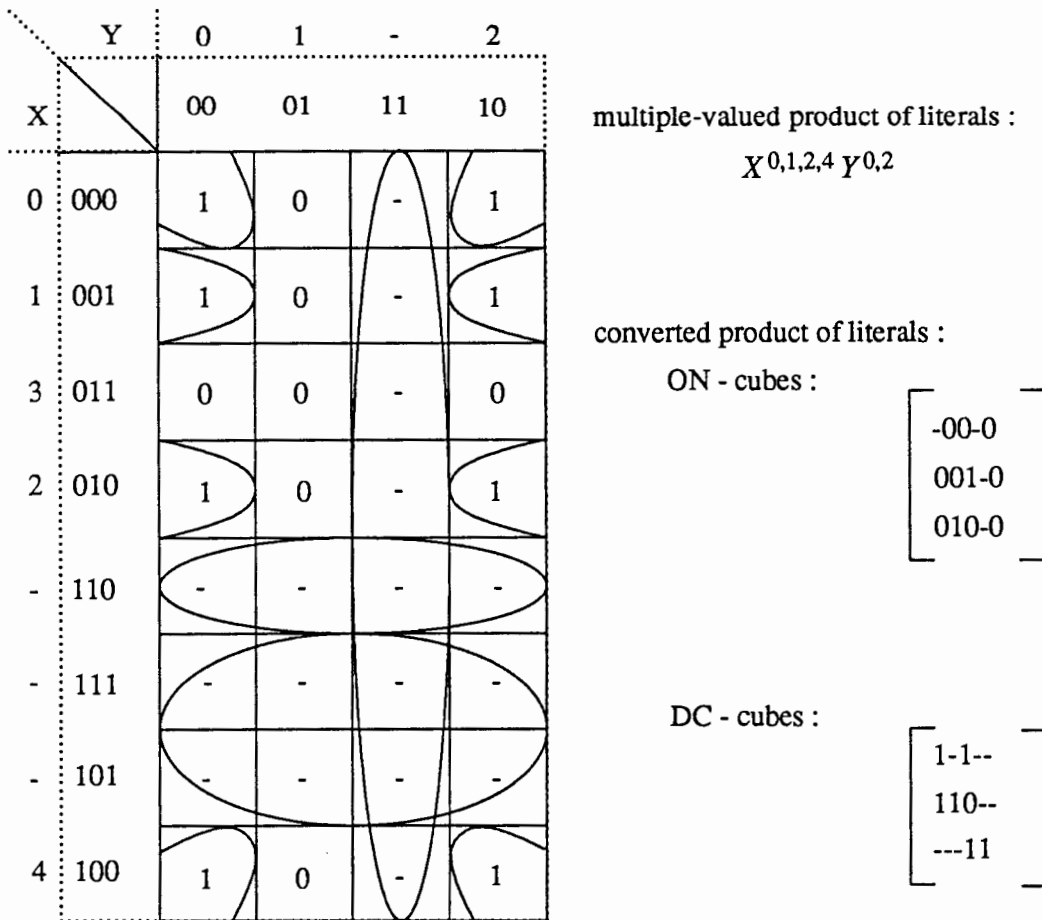
Figure 10. Converted product of literals to nondisjoint cubes.

A long variable (having 32 bits) is used to store any of those cubes. Generally, the following formula is valid for any product of multiple-valued literals X and Y, having m and n bits used for the description of their multiple values:

$$m + n = b,$$

where b is the number of bits in the total description of the binary cubes. Hence, in our implementation, the maximal value of b is 32. In the discussed example, b=5.

The algorithm which describes all steps of the conversion for multiple-valued input binary functions having an arbitrary number of literals is shown below. The notation used in this algorithm is as follows:

$literal_i$: i^{th} literal of a multiple-valued cube.

$literal_k$: k^{th} literal of a multiple-valued cube.

$values_i$: number of values of $literal_i$.

$values_k$: number of values of $literal_k$. .

map_i : necessary bits for the binary minterms to represent the multiple-valued $literal_i$.

map_k : necessary bits for the binary minterms to represent the multiple-valued $literal_k$.

Algorithm : Conversion from a mv-function to its Boolean representation

do for each multiple-valued input cube

    {*do* for each literal of the multi valued cube

        {generate the array of binary minterms according to ones in

        the positional representation of the literal;

        merge the minterms to cubes.}

    do for each product i,k of the generated arrays

        {append to the cubes of $literal_i$ map_k dc literals;

        append the cubes of $literal_k$ to map_i dc literals;

        put the intersections of the cubes to the solution list of the product i,k.

        do for i,k

        {if $values_{i,k} < map_{i,k}$

            {merge all minterms not covered by $literal_{i,k}$;

            append to the cubes of $literal_i$ map_k dc literals;

            append the cubes of $literal_k$ to map_i dc literals;

            put these dc cubes to the solution list of the product for i,k.}}}}

        The solution list of a product of two literals can have nondisjoint dc- and on-cubes. Therefore, before applying the Walsh Transformation the disjoint array of cubes has to be generated from the nondisjoint one (Figure 11, 13c).

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

multiple-valued product of two literals :

$$X^{0,1,2,4} Y^{0,2}$$

disjoint arrays of cubes representation :

ON - cubes :

$$\begin{bmatrix} -00-0 \\ 001-0 \\ 010-0 \end{bmatrix}$$

DC - cubes :

$$\begin{bmatrix} 1-1-- \\ 110-- \\ 0--11 \\ 10011 \end{bmatrix}$$
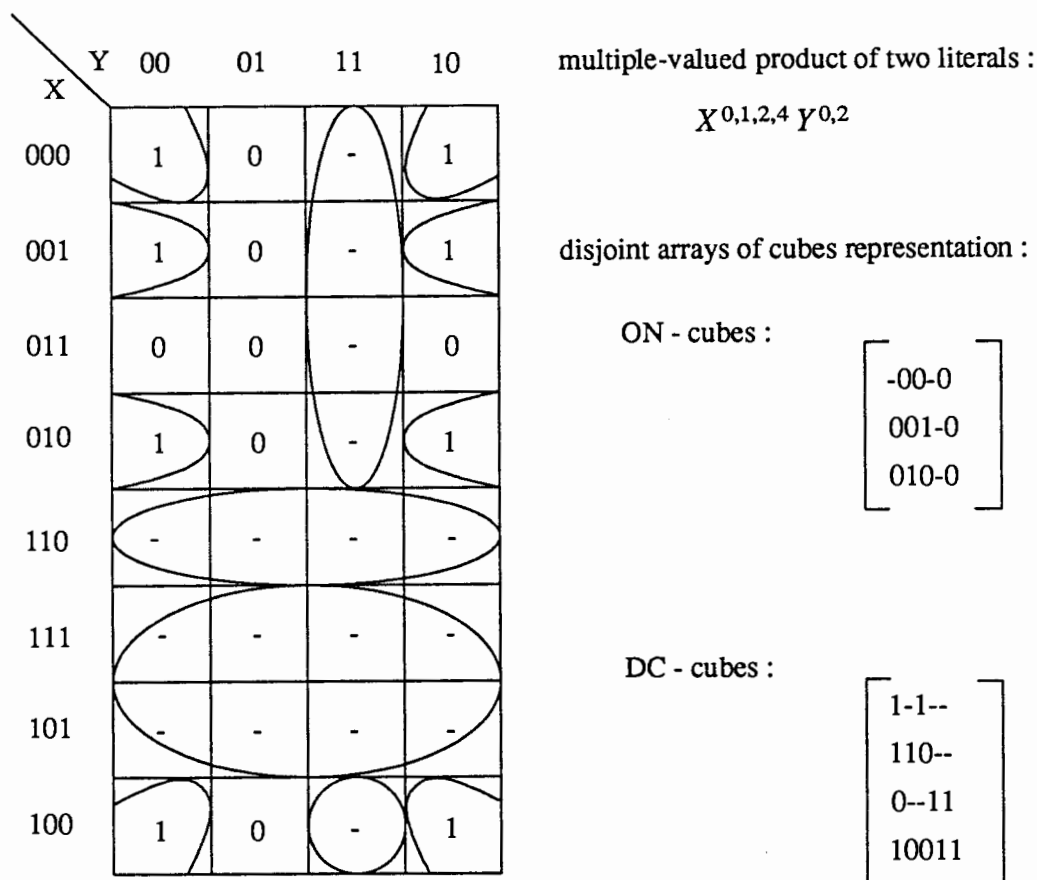
Figure 11. Disjoint cube representation of the product of two literals.

The final lists of on- and dc- cubes can be directly taken as the input data to generate the spectral coefficients with the program for the binary Rademacher-Walsh Transformation (Chapter II.6).

III.1.3 A complete example

In this part, an execution of the complete program will be illustrated. The input format is shown in Figure 12. In this example, the mv-function is composed of two terms each having three multiple valued literals: X, Y and Z. The number of the literals and their maximal logical values are declared first. Then, the terms are entered in the positional notation. The command .e stands for the end of the declaration list.

```
# example file for multiple-valued Walsh Transformation
# .mv [literals] [values_0]..[values_n]

.mv 3 5 3 4

#X   Y   Z
11101 101 0011
10110 011 1010
.e
```

Figure 12. An example of the input file for the multiple-valued Walsh Transformation program.

First, all the multiple-valued literals are converted to the binary representation. By using the previously described algorithm, the Boolean array of cubes is generated for all cubes of a given product of two literals. In the case of our example two Boolean arrays (illustrated here as Karnaugh maps) shown in Figure 13a, and Figure 13b are generated. Next, all the results for the same product of literals are merged together. Finally, a disjoint array is generated from the nondisjoint on (Figure 13c, where | stands for the bit-by-bit OR operation).

Similarly the binary representation for all product of literals, in our example the products of $literal_0$ and $literal_2$ and the product of $literal_1$ and $literal_2$, are generated. The final results for those product of literals is shown in Figure 14.

Now, the binary Walsh Transformation is applied to each of the cube representations in turn. An execution of the binary Walsh Transformation for the product YZ is shown below. Since the relationship between these two literals is described by the Karnaugh map of the dimension 4×4 then 16 spectral coefficients fully describe this map. The relationship between other products of literals is represented by the spectra having 32 spectral coefficients each. Thus, the binary function from our example is described by three vectors of binary spectra: two having 32 elements and one having 16 elements.

a. $X^{0,1,2,4} Y^{0,2}$

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

b. $X^{0,2,3} Y^{1,2}$

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

c. $X^{0,1,2,4}Y^{0,2} \mid X^{0,2,3}Y^{1,2}$

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

Figure 13. Conversion of the product of the literals X and Y to disjoint cube representation.

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

Y \ Z	00	01	11	10
00	0	0	1	1
01	1	0	0	1
11	-	-	-	-
10	1	0	1	1
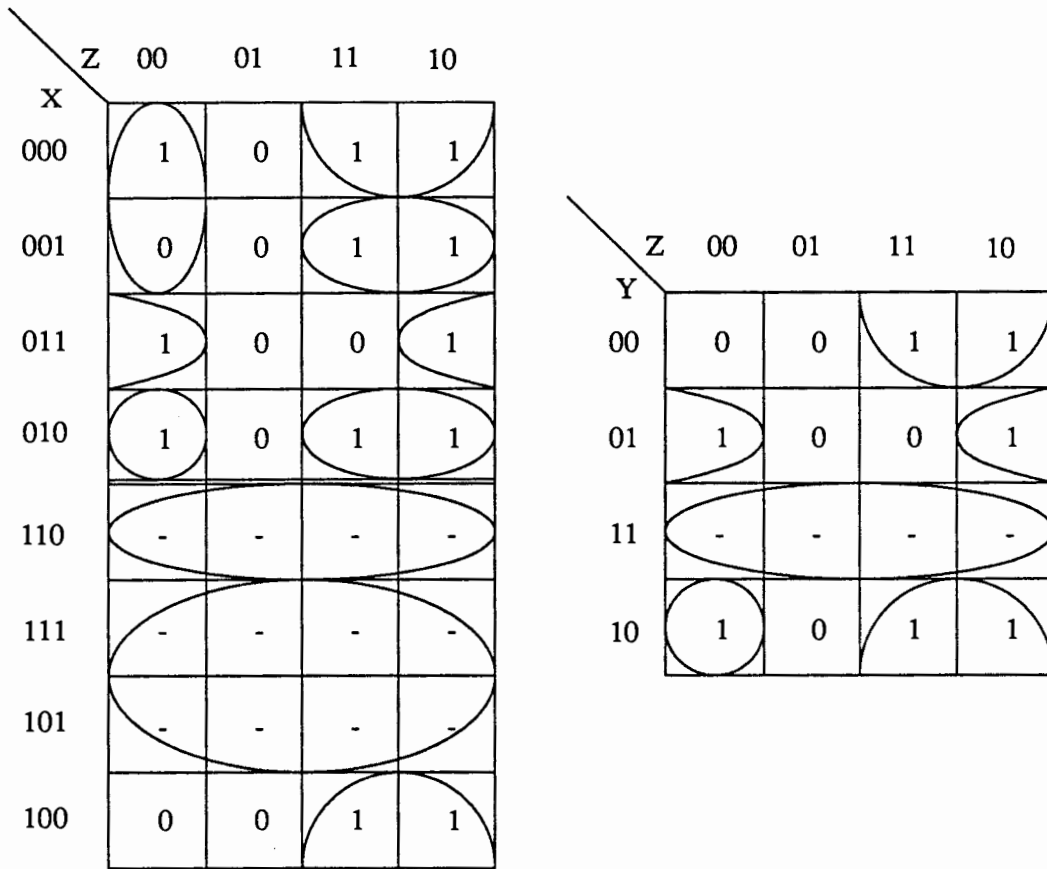
a. pair X-Z   b. pair Y-Z

Figure 14. Disjoint cube representation of other products of two literals.

TABLE XXV

FIRST ORDER COEFFICIENTS OF A PRODUCT OF
TWO MULTIPLE-VALUED LITERALS

		$R\ 0$	$R\ Z^{2,3}$	$R\ Z^{1,3}$	$R\ Y^{2,3}$	$R\ Y^{1,3}$
-01-	ON	8	0	-8	8	0
01-0	ON	4	-4	-4	8	-4
10-0	ON	2	-2	-6	6	-6
11--	DC	-2	2	-2	6	-6

In the algorithm, the coefficients of the dc coefficient (R_0) and the first order

coefficients are generated first. It is shown in Table XXV. For the notation used in the spectrum see (30).

After that, the next consecutive order (second one in this case) is generated.

TABLE XXVI

SECOND ORDER COEFFICIENTS

$R\ Z^{1,2}$	$R\ Z^{2,3}+Y^{2,3}$	$R\ Z^{2,3}+Y^{1,3}$	$R\ Z^{1,3}+Y^{2,3}$	$R\ Z^{1,3}+Y^{1,3}$	$R\ Y^{1,2}$
0	0	0	8	0	0
4	0	-4	8	4	0
6	2	-2	6	2	-2
2	2	-2	6	2	-2

The last two orders are generated in a similar way and one obtains the complete spectrum of the literal product YZ shown in Table XXVII.

TABLE XXVII

COMPLETE SPECTRUM OF THE
MULTIPLE-VALUED FUNCTION

$R\ 0$	$R\ Z^{2,3}$	$R\ Z^{1,3}$	$R\ Y^{2,3}$	$R\ Y^{1,3}$
-2	2	-2	6	-6

$R\ Z^{1,2}$	$R\ Z^{2,3}+Y^{2,3}$	$R\ Z^{2,3}+Y^{1,3}$	$R\ Z^{1,3}+Y^{2,3}$	$R\ Z^{1,3}+Y^{1,3}$	$R\ Y^{1,2}$
2	2	-2	6	2	-2

$R\ Z^{1,2}+Y^{2,3}$	$R\ Z^{1,2}+Y^{1,3}$	$R\ Z^{2,3}+Y^{1,2}$	$R\ Z^{1,3}+Y^{1,2}$	$R\ Z^{1,2}+Y^{1,2}$
2	6	2	-2	2

III.1.4 Execution times

The examples tested to evaluate the execution time have either 5 or 10 values per literals. The corresponding numbers of cubes, values and literals can be read from the Table. The notations follows the one from Chapter II.2.4.

TABLE XXVIII

EXECUTION TIMES FOR THE MULTIPLE-VALUED
WALSH TRANSFORMATION

cubes	values	5 literals		10 literals	
		u	s	u	s
1	5	1.6	3.9	7.6	16.1
1	10	5.1	3.7	31.7	27.9
5	5	2.8	4.6	13.6	20.4
5	10	26.9	6.2	235.9	31.6
10	5	4.9	5.7	24.3	25.2
10	10	61.0	6.2	472.3	39.5

The results from the table confirm the theory. The execution time increases

- nearly linearly with the number of cubes.

- exponentially with the number of values, because the number of spectral coefficients that have to be calculated increases exponentially.

- linearly with the number of spectra that have to generated according to the number of literals (see (30)).

III.2 MULTIPLE-VALUED GENERALIZED REED-MULLER TRANSFORM

III.2.1 Description

The new concept of a mixed polarity multiple-valued input Generalized Reed-Muller expression (MRME) (32) is a generalization of the GRME to multiple-valued algebra. It finds several applications in logic design, pattern recognition, and other areas. In logic design it can be primarily used for the minimization of EXOR-based

counterparts of PLAs with input decoders (16,17,31,33). It will be investigated in this Chapter, how to apply the method of pattern matching to perform the transformation from a disjoint multiple-valued SOPE or MRME, to another MRME.

The method for the generation of the MRME consists of two basic parts. First, each multiple-valued literal of the mv-input function has to be transformed to a chosen polarity (defined below). In the second part the transformed literals for each term are taken to calculate the final MRME.

The theory to change the polarity of a singular multiple-valued literal is defined by the two following Theorems.

Theorem III.1

Any value of a multiple-valued variable $X_i^{S_i}$ with the set of truth values $P_i = \{0,1,...,p_i-1\}$ can be created by p_i variables $V_{ir}^{T_{ir}}$ with the set of truth values $T_{ir} \subseteq A_i = \{0,1,...,p_i-1\}$, where the p_i vectors A_{ir} for the mv-variable $X_i^{S_i}$ form the row vectors of an orthogonal $p_i \times p_i$ matrix A_i.

Proof:

One property of an orthogonal $m \times m$ matrix O with the elements $o_{ij} \in (0,1)$ is, that any vector $U(u_0, u_1, ..., u_{m-1})$ with $u_j \in (0,1)$ can be represented by a super-position (performing of bit-by-bit arithmetic sum operation) of row vectors O_i of the matrix O. Hence, any set S of truth values $P = \{0,1,...,p-1\}$ of a mv-literal X^S can be represented by a superposition (or in the GF(2) field the bit-by-bit EXOR operation) of the values from the orthogonal set of truth values $A = \{0,1,...,p-1\}$, where V_r is a row vector of the orthogonal $p \times p$ matrix A.

Example III.1

To illustrate Theorem III.1 all possible sets of truth values $S \subseteq P = \{0,1,2\}$ of a three-valued literal X^S are calculated from the representation of the 3×3 orthog-onal matrix shown in Figure 15.

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix}$$

Figure 15. Example of a 3×3 orthogonal matrix.

In the following Table the calculation of all possible sets of truth values S by exoring the rows of the orthogonal matrix A (Figure 15) is shown, where the rows are named T_1, T_2 and T_3.

TABLE XXIX

ALL POSSIBLE COMPOSITIONS OF POLARITY LITERALS

truth values S	binary code	composition of matrix rows
{2}	001	$T_1 \oplus T_2$
{1}	010	T_2
{1,2}	011	T_1
{0}	100	T_3
{0,2}	101	$T_1 \oplus T_2 \oplus T_3$
{0,1}	110	$T_2 \oplus T_3$
{0,1,2}	111	$T_1 \oplus T_1$

For the use of less restricted forms Theorem III.1 can be easily generalized with Lemma III.1.

Definition III.4

The matrix describing the vectors T_r of the mv-literals $V_r{}^{T_r}$, where by exoring of those literals every possible set $S \subseteq P = \{0,1,...,p-1\}$ of a mv-variable X^S can be described will be denoted as matrix C.

Lemma III.1

The Theorem III.1 can be generalized to a non canonical form by using instead of the orthogonal matrix A the matrix C according to Definition III.4, where the orthogonal matrix A is a proper subset of the matrix C.

A restricted orthogonal matrix A will further be used to represent the set of literals V_{ir} defined in Theorem III.1.

Theorem III.2

Any value of a multiple-valued variable $X_i{}^{S_i}$ with the set of truth values $P_i = \{0,1,..,p_i-1\}$ can be created by $p_i - 1$ variables V_{ir} with the set of truth values $A_i = \{0,1,..,p_i-1\}$, where one of the values of the truth values P_i is not used in the set of truth values A_i. The set or truth values have to form a orthogonal $(p - 1) \times (p - 1)$ matrix B. To expand this matrix again to a orthogonal $p \times p$ matrix A a row corresponding to the dc-literal is added.

Proof:

The proof is similar to the proof of Theorem III.1 because again an orthogonal $p \times p$ matrix A is used. It is obvious, that an orthogonal $(p - 1) \times (p - 1)$ matrix B can be expanded to an orthogonal $p \times p$ matrix A by adding a row containing only 1's, and filling up the new positions in the matrix A with 0's.

Example III.2

The Example III.1 will be now used for the orthogonal matrix A shown in Figure 16, where the first row represents the dc literal. The set $S = \{1\}$ of the truth values P is not used by the truth values A for the literals V_r.

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix}$$

Figure 16. Restricted orthogonal matrix A.

TABLE XXX

TRUTH VALUES S FOR RESTRICTED ORTHOGONAL MATRIX A

truth values S	binary code	composition of matrix rows
{2}	001	T_2
{1}	010	$T_1 \oplus T_3$
{1,2}	011	$T_1 \oplus T_2 \oplus T_3$
{0}	100	$T_2 \oplus T_3$
{0,2}	101	T_3
{0,1}	110	$T_1 \oplus T_2$
{0,1,2}	111	T_1

Definition III.5

The set of truth values A_i for the literals V_{ir} defined in Theorem III.2.2 forming

the orthogonal $p_i \times p_i$ matrices A_i, is the *restricted polarity* for the literal $X_i{}^{S_i}$

with the truth values $P_i = \{0,1,..,p_i - 1\}$. Because in the sequel only the

restricted polarity is used, it will just be called the *polarity* of literal $X_i{}^{S_i}$.

Definition III.6

The literals V_{ir} with the truth values A_i defined in Theorem III.2.2 are called the

polarity literals.

Example III.3

The four-valued literal X^{023} can be represented by the set of polarity literals

shown in Table XXXI. Below the representation of the truth values of the polarity

literals are illustrated as a orthogonal matrix. The chosen set of polarity literals

representing any possible four-valued literal are: $V_1{}^{T_1} = X^{0123}$, $V_2{}^{T_2} = X^{13}$, $V_3{}^{T_3} =$

X^{23}, and $V_4{}^{T_4} = X^{123}$.

TABLE XXXI

POLARITY LITERALS OF A GIVEN POLARITY

polarity literal	binary representation
$V_1^{T_1} = X^{0123} = 1$	1111
$V_2^{T_2} = X^{13}$	0101
$V_3^{T_3} = X^{23}$	0011
$V_4^{T_4} = X^{123}$	0111

$$A = \begin{bmatrix} 1111 \\ 0101 \\ 0011 \\ 0111 \end{bmatrix} = \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix}$$

Figure 17. Polarity matrix for polarity of Table XXXI.

Example:

$$X^{023} = 1 \oplus V_3^{T_3} \oplus V_4^{T_4}$$

in the binary form representing :

$$\begin{array}{r} 1111 \\ 0011 \\ \oplus \quad 0111 \\ \hline 1011 \end{array}$$

According to Theorem III.2 and Definition III.5 it is possible to transform each variable of a mv-function f to another polarity.

Let us recall the notation presented until now in the above Definitions and Theorems:

TABLE XXXII

NOTATION FOR THE MRME

$X_1{}^{S_1}$	$X_2{}^{S_2}...$	$X_i{}^{S_i}...$	$X_n{}^{S_n}$
$P_1=(0,1,..,p_1-1)$		$P_i=(0,1,..,p_i-1)$	$P_n=(0,1,..,p_n-1)$
$V_{11}{}^{T_{11}}...V_{1r}{}^{T_{1r}}...V_{1p_1}{}^{T_{1p_1}}$	...	...	$V_{n1}{}^{T_{n1}}...V_{nr}{}^{T_{nr}}...V_{np_n}{}^{T_{np_n}}$
A_1		A_i	A_n

$$A_1 \;=\; \begin{bmatrix} T_{11} \\ \cdots \\ T_{1r} \\ \cdots \\ T_{1p_r} \end{bmatrix} \qquad A_n \;=\; \begin{bmatrix} T_{n1} \\ \cdots \\ T_{nr} \\ \cdots \\ T_{np_r} \end{bmatrix}$$

Figure 18. Description of polarity matrices.

In the first row of Table III.4 the multiple-valued literals $X_i{}^{S_i}$ of a function $F(X_1, X_2,...,X_n)$ are shown. Below the set of truth tables P_i for those literals are given. Next the polarity literals V_{ir} are shown. Finally in Figure 18 the polarity matrices consisting of the value vectors T_{ir} are illustrated.

Example III.4

The term $X_1{}^{023} X_2{}^1$ (where X_1 is the same literal as in Example III.3 and X_2 is a five-valued literal) shall be represented with the polarity literals denoted as in Table XXXII. The polarity for X_1 is the same as in Example III.3: $V_{11}=X_1{}^{0123}$, $V_{12}=X_1{}^{13}$, $V_{13}=X_1{}^{23}$, $V_{14}=X_1{}^{123}$. The polarity for X_2 is ($V_{21}=X_2{}^{13}$, $V_{22}=X_2{}^{23}$, $V_{23}=X_2{}^{123}$, $V_{24}=X_2{}^4$).

$$X_1{}^{023}X_2{}^1= (1 \oplus V_{13} \oplus V_{14}) (V_{22} \oplus V_{23})$$

$$= 1 \oplus V_{22} \oplus V_{23} \oplus V_{13}V_{22} \oplus V_{13}V_{23} \oplus V_{14}V_{22} \oplus V_{14}V_{23}$$

It will be now investigated how to describe the MRME spectrum M and apply the Cube Comparison method to calculate the final MRME from the polarity representation of the literals $X_i{}^{S_i}$. It will be shown, that the Cube Comparison method can be used instead of the product multiplication of the two products in Example III.4. The Definition II.2 defining the Reed-Muller spectrum M for Boolean functions is extended here to one of the spectrum for the multiple-valued Generalized Reed-Muller Transformation.

Definition III.7

The Multiple-valued input Generalized Reed-Muller form (MRME) of a function $F(X_1,X_2,...,X_n)$ over the Galois Field 2 (GF(2)), where X_i, i=1,2,...,n are the literals according to Definition III.1, can be described by its spectrum M that is analogous to the spectrum M defined in Equation (1) from Chapter II.2.1.

$F(X_1,X_2,...,X_n) = a_0 \oplus$

$\quad a_1 V_{11} \oplus ... \oplus a_r\ V_{1r} \oplus ... a_{p_1} V_{1p_1} \oplus a_{p_1+1} V_{21} \oplus ... \oplus a_{p_1+ \cdots +p_n} V_{np_n} \oplus$

$\quad a_{o_1+1} V_{11} V_{21} \oplus a_{o_1+2} V_{11} V_{31} \oplus ... \oplus a_{o_2} V_{n-1p-1} V_{np_n} \oplus$

$\quad ....$

$\quad a_{o_{n-1}+1} V_{11} ... V_{n1} \oplus ... \oplus a_{o_n} V_{1p_1} ... V_{np_n}$

The above formula is very general, because every polarity can have a different number of polarity literals. The total number of coefficients $a_i \in (0,1)$ is dependent on the total number of polarity literals, as well as on their number in one polarity. For the Definition of the spectrum M still the same Definition II.2.2 for Boolean functions is valid, the basic vectors (standard trivial functions) are now the EXOR combinations of the polarity literals.

Because of the complexity of the multiple-valued spectrum M only an example of a simple spectrum is shown below.

Example III.5

The spectrum M of the two variable mv-function $F(X_1,X_2) = X_1^{023} X_2^{01}$ is used to show the spectrum and the standard trivial functions of the chosen polarity. The chosen polarity for the two literals is shown in Figure 19. In the Figure 20 the set of all standard trivial functions for these polarities of X_1 and X_2 is given. Below the transformation of the product $X_1^{023} X_2^{01}$ is shown:

$$X_1^{023} X_2^{01} = (V_{11} \oplus V_{13} \oplus V_{14})(V_{21} \oplus V_{23})$$

$$= (1 \oplus V_{13} \oplus V_{14})(1 \oplus V_{23})$$

$$= 1 \oplus V_{13} \oplus V_{14} \oplus V_{23} \oplus V_{13}V_{23} \oplus V_{14}V_{23}$$

For a comparison of the product $X_1^{023} X_2^{01}$ with the standard trivial functions in Figure 20, the Karnaugh map of this product of literals is shown in Figure 21. The reader can observe that the above shown formula can be directly related to the Karnaugh maps in Figure 20, and 21.

$$A_1 = \begin{bmatrix} 1111 \\ 0101 \\ 0011 \\ 0111 \end{bmatrix} = \begin{bmatrix} T_{11} \\ T_{12} \\ T_{13} \\ T_{14} \end{bmatrix} \qquad A_2 = \begin{bmatrix} 111 \\ 100 \\ 001 \end{bmatrix} = \begin{bmatrix} T_{21} \\ T_{22} \\ T_{23} \end{bmatrix}$$

Figure 19. Polarity matrices.

$1=V_{11}=V_{21}$

X_1\\X_2	0	1	2
0	1	1	1
1	1	1	1
2	1	1	1
3	1	1	1

V_{12}

X_1\\X_2	0	1	2
0	0	0	0
1	1	1	1
2	0	0	0
3	1	1	1

V_{13}

X_1\\X_2	0	1	2
0	0	0	0
1	0	0	0
2	1	1	1
3	1	1	1

V_{14}

X_1\\X_2	0	1	2
0	0	0	0
1	1	1	1
2	1	1	1
3	1	1	1

V_{22}

X_1\\X_2	0	1	2
0	1	0	0
1	1	0	0
2	1	0	0
3	1	0	0

V_{23}

X_1\\X_2	0	1	2
0	0	0	1
1	0	0	1
2	0	0	1
3	0	0	1

$V_{12}V_{22}$

X_1\\X_2	0	1	2
0	0	0	0
1	1	0	0
2	0	0	0
3	1	0	0

$V_{12}V_{23}$

X_1\\X_2	0	1	2
0	0	0	0
1	0	0	1
2	0	0	0
3	0	0	1

$V_{13}V_{22}$

X_1\\X_2	0	1	2
0	0	0	0
1	0	0	0
2	1	0	0
3	1	0	0

$V_{13}V_{23}$

X_1\\X_2	0	1	2
0	0	0	0
1	0	0	0
2	0	0	1
3	0	0	1

$V_{14}V_{22}$

X_1\\X_2	0	1	2
0	0	0	0
1	1	0	0
2	1	0	0
3	1	0	0

$V_{14}V_{23}$

X_1\\X_2	0	1	2
0	0	0	0
1	0	0	1
2	0	0	1
3	0	0	1

Figure 20. Standard trivial functions for polarity in Figure 19.

X_1\\X_2	0	1	2
0	1	1	0
1	0	0	0
2	1	1	0
3	1	1	0

Figure 21. Karnaugh map of product $X_1^{023} X_2^{01}$.

The new expression can be now represented in the form of spectral coefficients, where the indices are again the representation of the standard trivial functions.

TABLE XXXIII

SPECTRUM OF THE PRODUCT X_1X_2

product	M_0	$M_{V_{12}}$	$M_{V_{13}}$	$M_{V_{14}}$	$M_{V_{22}}$	$M_{V_{23}}$
$X_1{}^{023}X_2{}^{01}$	1	0	1	1	0	1

$M_{V_{12}V_{22}}$	$M_{V_{12}V_{23}}$	$M_{V_{13}V_{22}}$	$M_{V_{13}V_{23}}$	$M_{V_{14}V_{22}}$	$M_{V_{14}V_{23}}$
0	0	0	1	0	1

The polarity matrices in the further presentation will be restricted for implementation reasons: one variable in the polarity matrix has to be zero for all the literals representing the orthogonal matrix, except the dc-literal (see Example III.3, III.4).

III.2.2 Implementation

Analogously to the theory from Chapter III.2.1 the method for the implementation of the MRME Transformation consists of two algorithms. With the first one the mv-literals from the mv-function F are transformed to their chosen polarity. This algorithm is basically a change of the binary representation of the initial literals of the function F to another one representing the EXOR combination of polarities literals. This new code is chosen in such a way, that the second algorithm can accept it without respect to the chosen polarity.

The second algorithm performs the Final Transformation, such as shown in Example III.5 for a classical method, with the Cube Comparison method.

The used pointer structures in this procedures are:

unsigned long   *cube;   /* represents the literals of the cube */

where cube[n] represents an array of n literals (using calloc()). Therefore the number of values is limited to 32.

unsigned int*cube_list; /* field to store the addresses of the cubes   */

where cube_list[n] represents an array of addresses for n cubes.

Finally

```
struct polarity
{
        unsigned int       po_add;      /* address to field of polarity literals */
        unsigned long      notused;/* value not used in the polarity literals   */
        unsigned short     no;   /* number of polarity literals                 */
}*pol;
```

is the structure to store the *polarity literals* for each literal of the cube. Where *po_add* contains the address of the field of polarity literals stored in *pol_lit*.

unsigned long   *pol_lit;

The additional array *coeff[n]* is to store the cube for the current created coefficient, the array *index[m]* contains the normalized index of the coefficient which has to be compared with the resultant code for the initial literals of each cube derived from the algorithm to transform one literal (Chapter III.2.2.1).

III.2.2.1 Transformation for one multiple-valued literal. The basic steps of the algorithm for the transformation of a multiple-valued literal to its representation of polarity literals will be illustrated on an example. In Table XXXIV the set of transformations of a three-valued literal X for the polarity used in Example III.3 is shown. In the first row the possible combinations of polarity literals are shown. In the second row the result of the EXOR operation is shown in the binary representation. In the last row of the table the new code called the *normalized code* is given. It determines the combination of the polarity literals.

TABLE XXXIV

SET OF TRANSFORMATIONS FOR A
MULTIPLE VALUED LITERAL

literal	1	V_2	V_3	V_4	$V_2 \oplus V_3$	$V_2 \oplus V_4$	$V_3 \oplus V_4$	$V_2 \oplus V_3 \oplus V_4$
	1111	0101	0011	0111	0110	0010	0100	0001
X^3								1
X^2						1		
X^{23}			1					
X^1							1	
X^{13}		1						
X^{12}					1			
X^{123}				1				
X^0	1			1				
X^{03}	1				1			
X^{02}	1	1						
X^{023}	1						1	
X^{01}	1		1					
X^{013}	1					1		
X^{012}	1							1
X^{0123}	1							
code	1000	0100	0010	0001	0110	0101	0011	0111

As mentioned above, the code for the representation of the GRME of one literal is chosen in a way, that this normalized code can be directly used to perform the Final Transformation. This can be done by using the same representation of the polarity literals V_1, V_2, ... for every chosen polarity. The code determines of what polarity literal/s the initial literal is composed of.

If a literal has to be represented by a combination of the orthogonal subset and the dc coefficient, the new code is calculated by the supercube operation of the code for the dc coefficient and the code of the subset combination.

Example III.6

The literal X^1 with the internal representation 0100 is changed to the new code 1101 for the polarity chosen in Example III.3. The new code representing this combination of polarity literals can be derived from Table XL, which is obtained from the normalized code shown in Table XXXIV.

TABLE XL

CODE REPRESENTATION FOR THE
POLARITY LITERALS

code	polarity
1000	$V_1 = 1$
0100	V_2
0010	V_3
0001	V_4

In Table XL the normalized code for the possible EXOR combinations of rows from Table XXXIV is shown, where according to the indices numbers of the polarity literals the normalized code cube has a "1". To obtain the code for a combination of those polarity literals one has just to perform the supercube operation on the code from the selected polarity literals.

The algorithm to generate the new code is shown below. It has the same structure as the algorithm shown in Chapter II.4.2 for the generation of one order of coefficients for the GAD- and GAR-Transformations. Because the new code here is the same as the one used for the indices in those transformations, the same basic procedure could be taken. Again, this procedure creates one order (determined by the number of EXOR-ed

basic polarity literals). In Table XXXIV every order of the new code is indicated by vert-ical double lines. The following notation is used in addition to the one of Chapter II.4.2.

Notation

ind: determines the literal of the cube that has to be transformed.

code: beginning number of the code for a new order, new code..

value: the value of the exored binary representation of the polarity literals.

pol_list: array of the binary representation of the polarity literals.

change_code:

substitutes the old code of the literal by the new generated one.

Algorithm: Transformation of one multiple-valued literal

```
unsigned short order;    /* order that should be generated        */
unsigned long code;      /* new code for the initial literal      */
unsigned long pol_lit;   /* address of the polarity literals      */
unsigned long value;     /* result of EXOR-ed polarity literals   */
int ind;                 /* which literal of the cube       */

gen_lit ( from,to,times)
unsigned short from;     /* initial 0                         */
unsigned short to; /* number of different literals minus order */
short times;             /* determines depth of recursion     */
{
        int i;

        times++;
        for ( i = from ; i <= to ; i ++ )
        {
                /* first bit for 1     */
                code = code + (1<<(30-i));
                /* generate EXOR list of polarity literals    */
                value = value ^ pol_lit[i];
                if ( times < order )
                        gen_lit(i+1, to+1 , times);
                else
                        change_code ( value,ind,code );
                code = code - ( 1<<(30-i) );
                value = value ^ pol_lit[i];
        }
}
```

In the subroutine change_code() of the shown above algorithm, the representation of the initial literals of all cubes determined by *ind*, which are equal to *value* will be changed to the new *code*.

III.2.2.2 Transformation of a multiple-valued function. After the transformation of each literal, as described above, the whole set of multiple-valued cubes has to be changed to the MRME. Again, the basic steps of the algorithm will be explained on examples.

Example III.7

The possible spectral coefficients / standard trivial functions for a spectrum representing a function having three distinct literals is shown in Table XXXVI. The literal X being four valued is represented by three polarity literals X_1, X_2 and X_3. The second literal X_2 being five valued is represented by the polarity literals V_{21}, V_{22}, V_{23} and V_{24}. Similarly for the third four-valued literal X_3.

TABLE XXXVI

COMPLETE MRME SPECTRUM

cube	dc	$M_{V_{12}}$	$M_{V_{13}}$	$M_{V_{22}}$	$M_{V_{23}}$	$M_{V_{24}}$	$M_{V_{32}}$	$M_{V_{33}}$
code	100	110	101	1100	1010	1001	110	1010

$M_{V_{12}V_{22}}$	$M_{V_{12}V_{23}}$	$M_{V_{12}Y_{24}}$	...	$M_{V_{24}V_{33}}$
110 1100 100	110 1010 100	110 1001 100	...	100 1001 101

$M_{V_{12}V_{22}V_{32}}$	$M_{V_{12}V_{22}V_{33}}$	...	$M_{V_{13}V_{24}V_{33}}$
110 1100 110	110 1100 101	...	101 1001 101

The same code as shown in Example III.6 is used for each literal, but here only one polaritity literal for each literal X_1, X_2, and X_3 can occur. The complete code for the shown first order in Table XXXVI has additionally the code 1000, 100 for the not used literals, analogously as shown above for the second order.

Code specification:

      1. the indices of the literals are represented in a positional notation.

      2. first bit is always one (code for the dc literal).

According to this we are now able to formulate an algorithm to perform the transformation.

Algorithm: MRME Transformation of a multiple-valued function

do for all spectral coefficients

    {*do* for every multiple-valued literal in normalized code

      {if intersection of normalized code and index is not empty

        {add "1" to the value of the spectral coefficient.}}

    if the value of the spectral coefficient is odd

      index of spectral coefficient determines polarity literals

      that form the solution cube.}

The algorithm which performs the above described transformation consists of two basic procedures. The first procedure P1 generates the possible combinations of literals of the initial cube, the second procedure P2 takes such a combination and generates all possible combinations of polarity literals within this combination. The C source-code for these procedures can be found in Appendix D.

CHAPTER IV

CANONICAL NOR EXOR SYNTHESIZER (CANNES)

IV.1 DESCRIPTION OF THE CRMPE

In (33) a new kind of EXOR-based PLAs was proposed. It makes use of both uncomplemented and complemented variables. Hence, it can realize any ESOPE without restrictions. The advantage of such an approach is that on the average the PLA structure requires lesser rows (terms) than the standard PLA. Since there is no exact minimization method or a reliable method that produces a quasi minimum solution of ESOPEs so far, a new concept of Canonical Restricted Mixed Polarity Exclusive Sum of Products forms (CRMPE) has been recently introduced (34). It has been proven there, that the upper bound on the number of terms in the CRMPE is smaller than that in the conventional forms and equal to that of the ESOPEs.

Let us recall the Definition of the CRMPE (34).

Definition of CRMPE:

The form:

$$F(X_0, X_1, \ldots, X_{m-1}) = a_0 \oplus a_1 X_0 \oplus a_2 X_1 i \oplus a_3 X_0 X_1 \oplus \cdots \oplus a_{2^m - 1} X_0 \cdots X_{m-1}$$

(similar to Reed-Muller form (Chapter II.2.1)) where every variable can be both complemented and not complemented, but where there is exactly one coefficient for each basic vector $(X_0, X_1, ..., X_0 \cdots X_{m-1})$ of the variables of a term, is called the *canonical restricted mixed polarity form.*

Example IV.1:

$1 \oplus x_1 \oplus \overline{x}_2 \oplus \overline{x}_1 x_2$ is a CRMP form, because there exists only one term for each basic vector of variables.

It was shown (34) that this form leads to the minimum solution only if the function is dense. The Definition of a dense function is, that the Hamming distance between the cubes describing the function is less than 3. Hence, the initial Boolean function being an array of disjoint cubes has to be decomposed to arrays of dense functions.

The very generic description for minimization of CRMPE described in (34) takes all possible branches of a tree search to find the minimal solution.

Hence, a fast straight forward algorithm based on heuristical mechanisms which will be introduced here has been developed. In the sequel this algorithm will be called CANonic Nor Exor Synthesizer (CANNES). The algorithm makes use of a completely or incompletely specified Boolean function represented in the form of array of cubes. One basic procedure of the CANNES-algorithm is performing the GRME Transformation on parts of the array for all possible polarities to find the minimal GRME. For the definitions of *prime term* and *subcombination* please see Theorem 2.3 and Definition 4.1 in (34).

Definition IV.1

An *essential prime term* of a function f is the term of the GRME of this function, that has the most subcombinations.

Definition IV.2

A *subset of a prime term* (for short, *subset*) consists of the prime term itself and all of its subcombinations.

The algorithm of the complete CANNES program performing the minimization of SOPE to CRMPE is shown below. The notation as defined in the above Definitions is

used.

Algorithm : Minimization to a CRMPE

    generate disjoint array of cubes;

    decompose array to dense arrays;

    do for each dense array

        { *do*

            { find essential prime term;

            if order of essential prime term < 2

                { final CRMPE found;

                stop}

            find minimal GRME of this subset;

            if (minimal GRME of subset < subset)

                substitute subset by minimal GRME;

        else

            essential prime term is term of CRMPE;

    while (TRUE)}

IV.2 IMPLEMENTATION

For the implementation of the CANNES algorithm is separated in several subroutines. The structure is shown in Figure 22.

One can observe, that two procedures of the program are already known. The algorithm for the generation of disjoint cubes from nondisjoint ones (Chapter II.1) and the procedure for the General Reed-Muller transform (Chapter II.2). Therefore in the sequel only the decomposition algorithm and the minimization algorithm will be described in detail.

The GRME Transformation is the main procedure used, therefore the same structures as for the GRME Transformation implementation (Chapter II.2.2) are taken.

```
┌─────────────────────────────────────────────────────┐
│                    C A N N E S                        │
│   ┌─────────────────────────────────────────────┐    │
│   │  DISJOINT ARRAY                               │    │
│   └─────────────────────────────────────────────┘    │
│                                                       │
│   ┌─────────────────────────────────────────────┐    │
│   │  DECOMPOSITION                                │    │
│   └─────────────────────────────────────────────┘    │
│                                                       │
│   ┌─────────────────────────────────────────────┐    │
│   │  GENERAL REED MULLER                          │    │
│   └─────────────────────────────────────────────┘    │
│                                                       │
│   ┌─────────────────────────────────────────────┐    │
│   │  MINIMIZATION                                 │    │
│   │   ┌─────────────────────────────────────┐     │    │
│   │   │  POLARITIES                          │     │    │
│   │   │    ┌───────────────────────────┐      │     │    │
│   │   │    │  GENERAL REED MULLER       │      │     │    │
│   │   │    └───────────────────────────┘      │     │    │
│   │   └─────────────────────────────────────┘     │    │
│   └─────────────────────────────────────────────┘    │
└─────────────────────────────────────────────────────┘
```
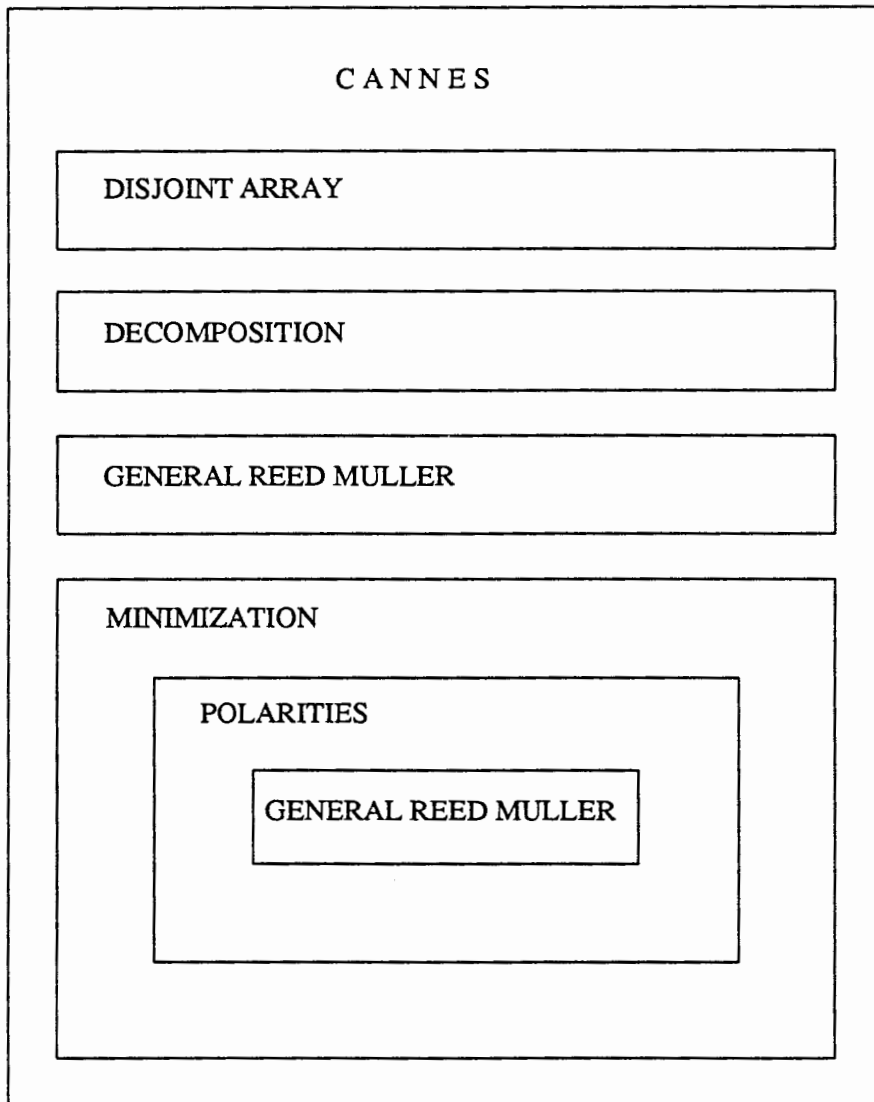
Figure 22. Structogram of CANNES.

In the CANNES program the algorithm to generate disjoint cubes from nondisjoint ones is first performed, because for the decomposition algorithm (Chapter IV.2.1) the Boolean function has to be represented as an array of disjoint cubes. The minimization algorithm is based on the representation of the function in any GRME. The best result is obtained, when the minimal GRME is selected. Hence, for each dense array the

minimal GRME is found by generating all possible GRMEs. Next every dense array being represented by its minimal GRME is minimized by the minimization algorithm (Chapter IV.2.2).

IV.2.1 Decomposition of a sparse function

To be able to obtain a quasi minimal solution for a sparse function, it has to be decomposed to a representation of sets of dense functions (34). To perform this task a simple algorithm has been designed.

Example IV.2.1

Figure 23 presents a sparse function f represented in a Karnaugh map. To generate the dense arrays for this function it is separated into parts having smaller Hamming distance than three.
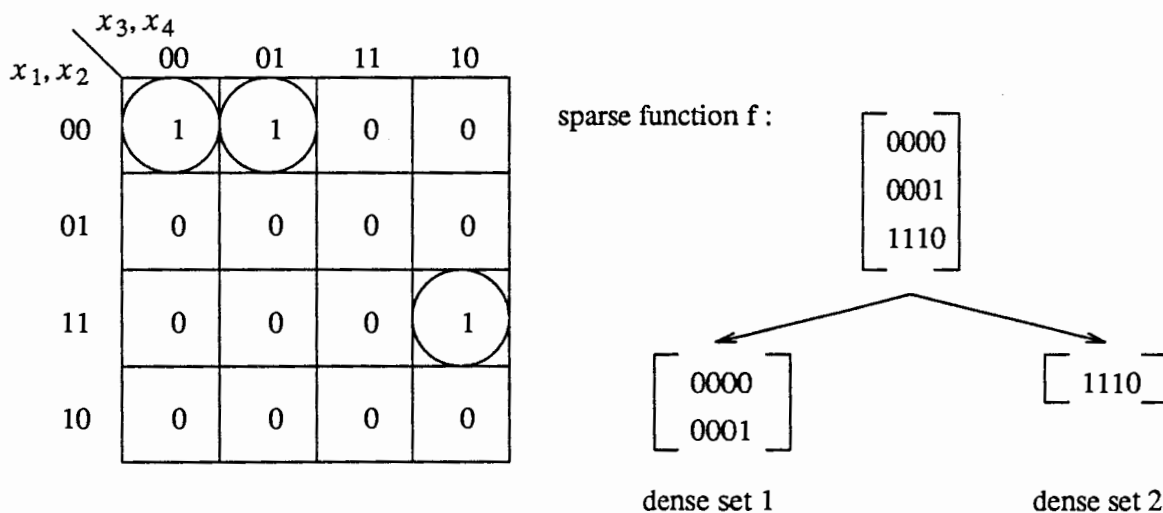

Figure 23. Decomposition of sparse function.

The basic idea of the decomposition algorithm is to take one cube of the input array and perform the supercube operation (bit-by-bit OR operation between two cubes, in the algorithm denoted as |) with all other cubes to which the Hamming distance is smaller than three. This has to be repeated, as shown in the algorithm below, until one obtains different arrays of cubes, where every array represents a dense function.

Notation:

cubes   number of cubes in the input array.

supercube

> variable that contains the result of the supercube operation.

H(supercube,cube)

> function that gives the Hamming distance between the supercube and the cube.

TRUE flag to determine that an additional cube with H<3 has been found.

Algorithm : decomposition of a sparse input function

do

> {supercube = first cube of the input array;
>
> cubes = cubes - 1;
>
> *do* for all remaining cubes
>
>> {NEW = FALSE;
>>
>> *do* for all cubes:
>>
>>> {if H(supercube,cube) < 3
>>>
>>>> {supercube = supercube | cube;
>>>>
>>>> cubes = cubes - 1;
>>>>
>>>> NEW = TRUE}}
>
> *while* (NEW == TRUE)}
>
> move all cubes covered by the supercube to a new array;

while (cubes > 0)}

IV.2.2 Minimization of a dense function

In the complete algorithm of the CANNES program (Chapter IV.1) the general method of the minimization was described. For the implementation, only the procedures to handle the array operations such as removing of the subsets of an essential prime term

have to be added. Therefore, a more detailed algorithm is given below. For the complete description of the CANNES program one can refer to its source code.

Notation :

input array   : array that contains the cubes of a GRME of the dense function.

output array   : array that contains the final minimized CRMPE.

Algorithm : Minimization of a dense function in GRME

do

    {find essential prime term;

    if order of essential prime term < 2

        put all terms to output array;

        break;

    extract the subset of the essential prime term from the array;

    generate all possible GRMEs of this subset;

    take the minimal GRME of this subset;

    if (CARD(minimal GRME of subset) < CARD(subset))

        substitute subset with its minimal GRME in the input array;

    else

        put essential prime term to output array;

while (TRUE)}

In the next Chapter this algorithm is illustrated on an example.

IV.3 A COMPLETE EXAMPLE OF AN EXECUTION OF THE CANNES PROGRAM

For the description of the final algorithm for the implementation the function F shown in Figure 24 was taken. As one can observe, the function is dense, hence, only one array of cubes representing the whole function has to be minimized.
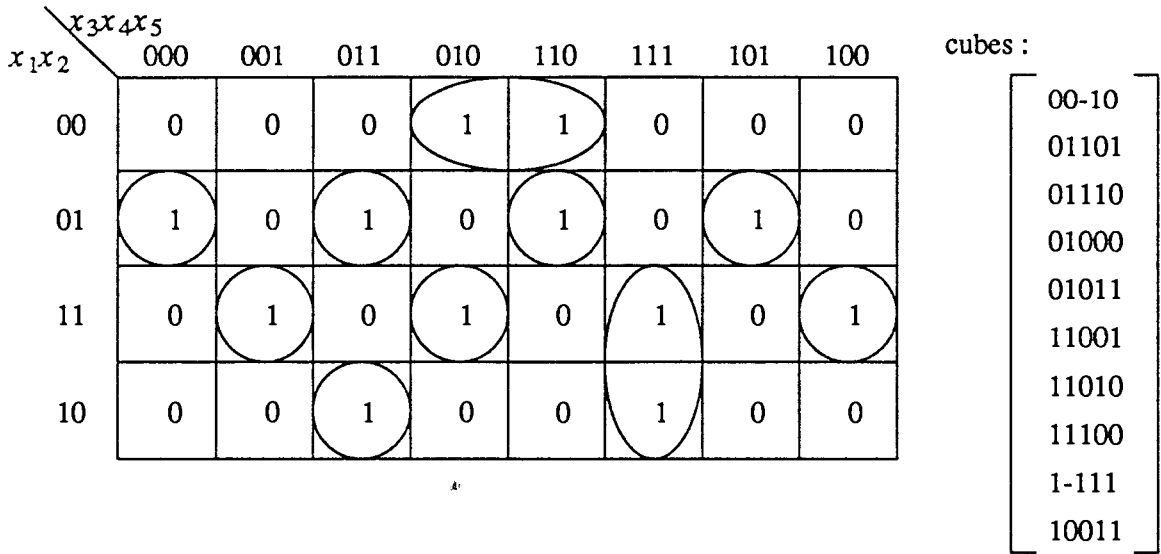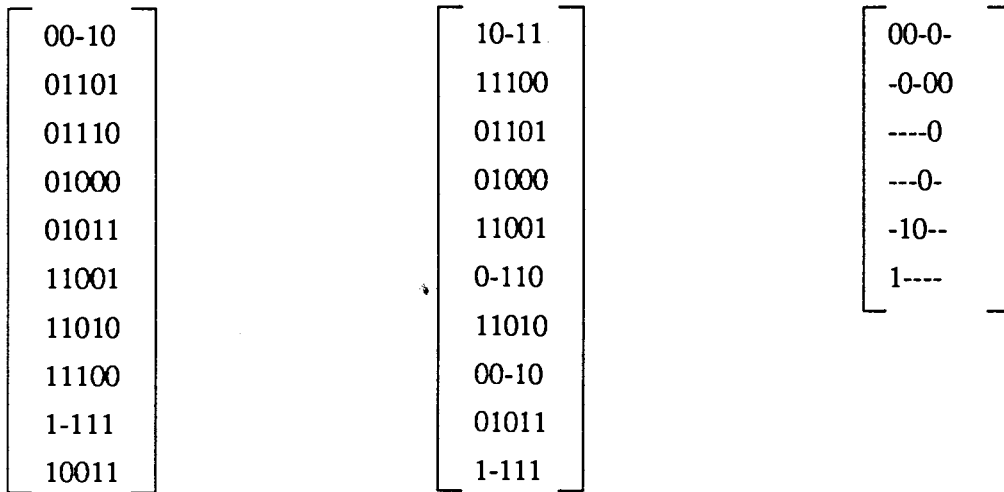
Figure 24. Test function.

For comparison, the solution given by the well-known minimization program ESPRESSO (21) and the solution calculated by CANNES are shown in Figure 25.

a. input array   b. minimization by ESPRESSO   c. minimization by CANNES

Figure 25. Comparison ESPRESSO with CANNES.

As one can observe from Figure 24 the input function is already disjoint and dense. Hence, the GRME Transformation to obtain the ESOPE for the final minimization can be directly executed.

First all possible GRMEs are generated for the input function. The GRME with the minimal number of terms is selected (Figure 26).

```
GRME
-----
0----
--0--
-00--
---0-
00-0-
----0
-0-00
```

Figure 26. GRME of the non sparse array.

The different steps of the now performed minimization are illustrated in Figure 28. The prime term with the most subcombinations in Figure 26 is 00-0-. The list of its subcombinations is shown in Figure 27, and as a subset in Figure 28a.
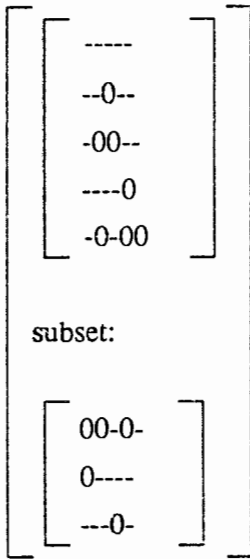
```
subcombination
00-0-
0----
---0-
```
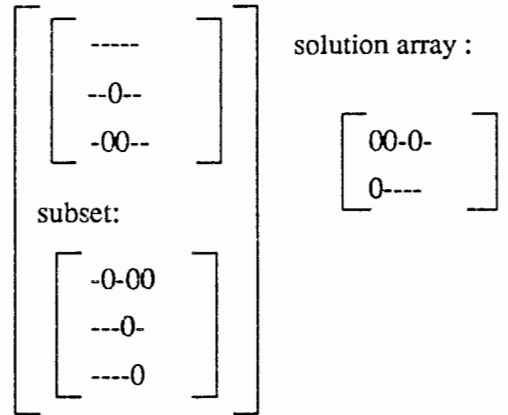
Figure 27. Subcombinations.

Next the minimal GRME of this subset has to be generated, by calculation of all possible GRMEs. Because all GRMEs do not have less cubes than the initial subset, the essential prime term is a final solution term. Therefore it is stored in the solution array (first cube in solution array Figure 28b). The term 0--- is now a prime term in the remaining array. So it is also given to the solution array.

Again, the prime term with the most subcombinations (essential prime term) for the remaining cubes has to be found. This essential prime term -0-00 with its subcombinations (shown as a subset in Figure 28b) does also not lead to a more minimal solution. Therefore the essential prime term is also a final solution term and has to be put with the prime terms of the remaining array to the solution array (Figure 28c). Again, the next essential prime term has to be found.
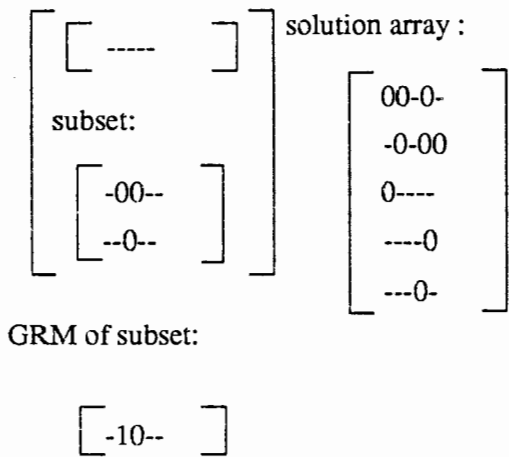
a. first loop

b. second loop

$$
\begin{bmatrix}
\begin{bmatrix}
\text{-----} \\
\text{--0--} \\
\text{-00--} \\
\text{----0} \\
\text{-0-00}
\end{bmatrix} \\
\text{subset:} \\
\begin{bmatrix}
\text{00-0-} \\
\text{0----} \\
\text{---0-}
\end{bmatrix}
\end{bmatrix}
$$

$$
\begin{bmatrix}
\begin{bmatrix}
\text{-----} \\
\text{--0--} \\
\text{-00--}
\end{bmatrix} \\
\text{subset:} \\
\begin{bmatrix}
\text{-0-00} \\
\text{---0-} \\
\text{----0}
\end{bmatrix}
\end{bmatrix}
$$

solution array :

$$
\begin{bmatrix}
\text{00-0-} \\
\text{0----}
\end{bmatrix}
$$

c. third loop

d. fourth loop

$$
\begin{bmatrix}
\begin{bmatrix}
\text{-----}
\end{bmatrix} \\
\text{subset:} \\
\begin{bmatrix}
\text{-00--} \\
\text{--0--}
\end{bmatrix}
\end{bmatrix}
$$

solution array :

$$
\begin{bmatrix}
\text{00-0-} \\
\text{-0-00} \\
\text{0----} \\
\text{----0} \\
\text{---0-}
\end{bmatrix}
$$

GRM of subset:

$$
\begin{bmatrix}
\text{-10--}
\end{bmatrix}
$$

solution array :

$$
\begin{bmatrix}
\text{-----} \\
\text{-10--} \\
\text{00-0-} \\
\text{-0-00} \\
\text{0----} \\
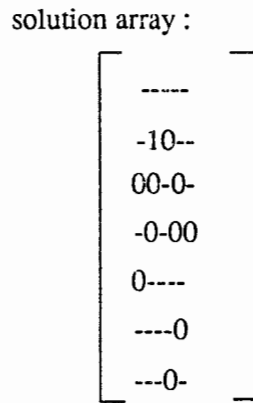\text{----0} \\
\text{---0-}
\end{bmatrix}
$$

Figure 28. Step by step minimization of the array shown in a.

In Table XXXVII, subset in Figure 28c, the subset of this prime term and the minimal GRME is shown.

TABLE XXXVII

MINIMAL GRM OF A SUBSET

subset	minimal GRME
-00--	-10--
--0--	

The final two cubes are prime terms and so they are immediately put to the solution array (Figure 28d). As one can observe this solution is not the final solution shown in Figure 25c. The general minimization according to the algorithm shown in Chapter IV.3.1 is finished. As a last minimization step it is tried to substitute a possible dc term (-----). In our example ----- ^ 0---- = 1----. With this last interaction we get the result as shown in Figure 25c.

CHAPTER V

CONCLUSION

The basic ideas of the new Cube Comparison Method have been shown for several discrete spectral transformations of logic functions. The first part of the Cube Comparison Method has been to compare the disjoint cube representation of a logic function with all indices of the spectral coefficients. Secondly the signs of the values have been calculated according to the order of the spectral coefficient. Finally the last part of the Cube Comparison method has been illustrated for the Rademacher Walsh Transformation, where the values of the spectral coefficients depend on the number of dc-literals in the cubes. The internal memory requirement, that limits all other existing implementations, has been overcome by the Cube Comparison Method. It has been also possible to generate only single spectral coefficients of the whole spectrum. This is for example useful for cases, where only those coefficients with the largest magnitude are necessary (25,26). Additionally, without any speed up methods, the implemented transformations are up to several times faster than the current existing ones.

Still, there are other speed up possibilities. In the current implementations, every cube is compared with all the indices of the spectral coefficients. Many cubes of the array of cubes have no contribution to certain orders of spectral coefficients. Hence, by sorting the cubes of the input array according to their smallest/highest order of spectral coefficients that are not equal to zero the calculation time can be sped up several times. Another possibility is to generate the spectral coefficients directly from the cube representation of the function. This approach will speed up the processing time, but its disadvantage is, that the whole spectrum has to be stored in the computer memory.

On top of that, all the algorithms can be performed in parallel. Because every spectral coefficients can be calculated directly from the cube representation, all spectral coefficients and the spectra for every cube can be processed in parallel. There are hardly any limitations for the parallelization of the algorithms. Thus, efficient implementations on pipelined vector processors and recently introduced DSP-coprocessors for personal computers are possible. It makes it also very well suited for systolic VLSI realizations.

The two advantages, no memory requirements and the shorter processing time, gives the new Cube Comparison Method for spectral transformation the advantage over all other currently existing methods yielding solutions to problems of very high dimensions.

REFERENCES

(1)   K. G. Beauchamp, "Applications of Walsh and Related Functions." New York, NY: Academic Press, 1984.

(2)   K. G. Beauchamp, "Transforms for Engineers." Oxford:Clarendon Press, 1987

(3)   J. C. Muzio, S. L. Hurst, "The computation of complete and reduced sets of orthogonal spectral coefficients for logic design and pattern recognition purposes," Comput. & Elect. Engng., vol. 5, pp. 231-249, 1978.

(4)   S. L. Hurst, D. M. Miller, J. C. Muzio, "Spectral Techniques in Digital Logic." London: Academic Press, 1985.

(5)   S. L. Hurst, "Use of linearization and spectral techniques in input and output compaction testing of digital networks," IEE Proc. Comput. & Digital Tech., vol. 136, pp. 48-56, Jan. 1989.

(6)   M. G. Karpovsky, "Finite Orthogonal Series in Design of Digital Devices." New York: Wiley, 1976.

(7)   M. G. Karpovsky, (ed), "Spectral Techniques and Fault Detection." Orlando: Academic Press, 1985.

(8)   S. S. Agaian, "Hadamard Matrices and Their Applications." Berlin: Springer-Verlag, 1980.

(9)   B. R. K. Reddy, A. L. Pai, "Reed-Muller Transform Image Coding," Computer Vision, Graphics, and Image Processing, vol. 42, pp. 48-61, 1988.

(10)   L. P. Yaroslavsky, "Digital Picture Processing - An Introduction." Berlin: Springer Verlag, 1985.

(11)   D. F. Elliott, K. R. Rao, "Fast Transforms: Algorithms, Analysis, Applications." London: Academic Press, 1982.

(12)   B. J. Falkowski, M. A. Perkowski, "Yet another method for the calculation of Hadamard-Walsh spectrum for completely and incompletely specified Boolean functions," accepted for publication in Int. J. of Electronics, August 1990.

(13)   B. J. Falkowski, M. A. Perkowski, "Algorithms for the calculation of Hadamard-Walsh spectrum for completely and incompletely specified Boolean functions," Proc. of IEEE 9th Int. Phoenix Conf. on Comp. & Comm., pp. 868-869, Scottsdale, AR, March 1990.

(14)   B. J. Falkowski, I. Schäfer, M. A. Perkowski, "Effective computer methods for the calculation of Rademacher-Walsh spectrum for completely and incompletely specified Boolean functions," submitted to IEEE Int. Conf. on Computer Aided Design, ICCAD 1990.

(15)   B. J. Falkowski, I. Schäfer, M. A. Perkowski, "Effective Computer methods for the calculation of Rademacher-Walsh spectrum with fast generation of disjoint cubes for completely and incompletely specified Boolean functions," submitted to IEEE Trans. on Computer Aided Design, April 1990.

(16)   Ph. W. Besslich, "Efficient Computer Method for EXOR Logic Design," IEE Proc. E., Comput. & Digital Tech., vol. 130, No6, pp. 203-206, Nov. 1983.

(17)   Ph. W. Besslich, "Spectral Processing of Switching Functions Using Signal-Flow Transformations, in Spectral Techniques and Fault Detection." (M.G. Karpovsky, ed.), Orlando: Academic Press, 1985.

(18)   L. Nguyen, M. Perkowski, N. Goldstein, "PALMINI - fast Boolean minimizer for personal computers," Proc. of 24th ACM/IEEE Design Automation Conference, pp. 615-621, 1987.

(19)   B. J. Falkowski, M. A. Perkowski, "An algorithm for the calculation of disjoint cube representation of completely and incompletely specified Boolean functions," accepted for publication in Int. J. of Electronics, 1990.

(20)   B. J. Falkowski, I. Schäfer, M. A. Perkowski, "A fast computer algorithm for the generation of disjoint cubes for completely and incompletely specified Boolean functions." submitted to IEEE Midwest Conf. on Circuits and Systems, Calgary: August 1990.

(21)   R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis." Hingham, MA: Kluwer Academic Publishers, 1985.

(22)   D. L. Dietmayer, "Logic Design of Digital Systems." Boston, MA: Allyn and Bacon, 1978.

(23)   J. P. Roth, "Computer Logic, Testing and Verification." Potomac, MD: Computer Science Press, 1980.

(24)   B. W. Kernighan, D. M. Ritchie, "The C Programming Language." Englewood Cliffs, NJ: Prentice-Hall, 1978.

(25)   M. Davio, J. P. Deschamps, A. Thayse, "Discrete and Switching Functions," New York: McGraw-Hill, 1978.

(26)   D. Green, "Modern Logic Design." Workingham: Addison-Wesley, 1986.

(27)   B. J. Falkowski, M. A. Perkowski, "A family of all essential radix-2 addition/subtraction multi-polarity transforms: algorithms and interpretation in Boolean domain," Proc. of IEEE Int. Symp. on Circuits & Systems, pp 2913-2916, May 1990.

(28)   E. A. Trachtenberg, D. Varma, "A design automation system for spectral logic synthesis," Proc. of Int. Workshop on Logic Synthesis, Research Triangle Park, North Carolina, May 12-15, 1987.

(29)   D. Varma, E. A. Trachtenberg, "Design automation tools for efficient implementation of logic functions by decomposition," IEEE Trans. Computer-Aided Design, vol. CAD-8, pp. 901-916, Aug. 1989.

(30)   B. J. Falkowski, M. A. Perkowski, "Walsh type transforms for completely and incompletely specified multiple-valued input binary functions", Proc. of Int. Symp. on Multiple-Valued logic, Charlotte, NC, May 1990.

(31)   T. Sasao, "An application of multiple-valued logic to a design of programmable logic arrays," Proc. of 8th Int. Symp. on Multiple-Valued Logic, 1979.

(32)   M. A. Perkowski, P. Dysko, B. J. Falkowski, "Two learning methods for a tree-search combinatorial optimizer," Proc. of IEEE 9th Int. Phoenix Conf. on Comp. & Comm., pp. 606-613, Scottsdale, AR, March 1990.

(33)   T. Sasao, Ph. W. Besslich, "On the Complexity of Mod-2 Sum PLA's'" IEEE Trans. on Comp., vol. 39, No.2, Feb. 1990.

(34)   L. Csanky, M. Perkowski, I. Schäfer, "Canonical restricted mixed-polarity exclusive sums of products and the efficient algorithm for their minimization," PSU internal report, Dec. 1989.

APPENDIX A

BASIC DEFINITIONS

Definition 1

A *literal* is a variable x_i which can either be positive ($x_i/1$) or negative ($\overline{x_i}/0$). A *literal* that can be in both forms is called a don't care (dc) literal (/-).

Definition 2

A *term* t of degree m is a product of m distinct literals, not including dc literals. For example the term $ab\overline{c}$ has the degree 3. The number of dc literals in term t is denoted by p.

Definition 3

A *cube* of degree m is the representation of a term t of degree m in binary form. Assuming three variables of order a,b,c for the term $ab\overline{c}$ the binary form is 110.

Definition 4

In a n-variable combinational logic function F, n is the number of distinct literals. Then, $n = m + p$.

Definition 5

A *minterm* *mt* of a n-variable combinational logic function F is a *term* that has n literals and no dc literal. All minterms necessary to describe this function F are so called *on* minterms, minterms for that the function F is not specified are the so-called *dc* minterms and finally all other minterms, that are not used for the function F are *off* minterms.

Example:

The literals that occur in a function f are a,b,c and d. Therefore the term $ab\overline{c}$ is described by the cube representation 110-, where '-' stands as a dc literal for the literal d not used in this term.

APPENDIX B

INPUT FORMAT FOR DISCRETE BINARY TRANSFORMATIONS

In general all programs introduced here use the ESPRESSO input format (30). An example of this format is shown below.

```
# example of a general input file
.i 4
.o 1
1-00 1
1-1- 0
--11 -
-1-- 1
.e
```

The number after *.i* determines the number of input literals, where the input literals can be on-(1), off-(0), or dc-(-) literals. The symbol *.e* at the end of the file determines the end of the list of input cubes. All programs for Boolean functions are limited to 32 input literals, except for the disjoint and ESOPE to SOPE implementation. These last two program have no input limitation basing on their different pointer structures.

The number after *.o* determines the number of output literals. All programs for Boolean functions are limited to 1 output literal. For the GRME transformation this value is ignored, because the program works only for on-cubes. The type is similar coded to the literals (on(1), off(0), dc(-)).

In the above example the first four bits (i.e. 1-00) of one line determine the four input literals. The one that follows that four positions determines the type of the cube.

In the follows the additional notation necessary for certain programs are described:

GRME Transformation

For the GRME Transformation the first cube in the input file will be read as the polarity. Hence, no dc-literal (-) is allowed.

GARD Transformation

Similar to the GRME Transformation the first cube of the input file will be read as the polarity. Additionally the following Notations are used:

.t+/.t-

The letter *t* stands for transformation. The following letter "+" determines that the Adding transformation is performed. If the letter "-" follows the Arithmetic transformation is performed.

.R/.S

The letters *R/S* specify the chosen coding of the spectrum. The letter R determines that the spectrum according to the coding of the R spectrum is generated. Similar for S. In the current version of the program the R spectrum is not implemented, but the subprogram has been written in a way to expand it easily.

INGARD Transformation

The input format of the INGARD implementation is the same as the output format of the GARD transformation. Where the specification of the transformation is the same as for the GARD transformation, including the polarity cube. The different orders are separated by *.n [ordernumber]*. An example is shown below. The specification *.o [outputliterals]* is not necessary.

example output file of the GARD transformation
which is the input format of the INGARD transformation

.i 4
.o 1
.R
.t+

1111

.n 0
0
.n 1
0
0
0
1
.n 2
0
1
2
0
2
1
.n 3
2
3
3
2.5
.n 4
6
.e

APPENDIX C

INPUT FORMAT FOR THE MULTIPLE-VALUED TRANSFORMATIONS

The input format for the multiple-valued Walsh transformation is shown in Chapter III.1.3. It is similar to the input format of ESPRESSO (30).

For the MRME Transformation implementation additionally the polarity has to be specified. Therefore, first the polarity cubes have to be read, such as for the GRME Transformation An example of an input file of the MRME Transformation is shown below. The number of literals per cube determines the number of polarity cubes that have to be read. In the below example this means two polarity cubes.

```
# example file for the MRME Transformation
# .mv [literals] [values 0]..[values n]

.mv 2 5 3 4

# now the two polarity cubes follow
11101 101 0011
10110 011 1010
# now the cubes follow
10011 011 1101
11001 001 0010
.e
```

APPENDIX D

PROCEDURES OF THE MRM IMPLEMENTATION

Algorithm: GRME for a multiple-valued input function

Procedure P1:

```
gen_order ( from,to,order,times)
unsigned short from;      /* initial 0                          */
unsigned short to;    /* number of different literals minus order     */
unsigned short order;      /* order that should be generated           */
unsigned short times;       /* to determine the loop of the procedure */
{
      int i,k;

      times++;

      for ( i = from; i <= to ; i++ )
      {
            which_lit[times-1] = i;
            if ( times < order )
                  gen_order ( i+1,to+1,order,times );
            else
            {
                  /* allocate memory for  Procedure P2      */
                  coeff = (LONG*)calloc(literals,sizeof(LONG));
                  index = (LONG*)calloc(literals,sizeof(LONG));
                  for ( k = 0; k < literals ; k++ )
                        index[k] = first[31];
                  gen_coeff(order,0);
                  free ( coeff );
                  free ( index );
            }
      }
}
```

Procedure 2:

```
gen_coeff ( order, times )
unsigned short order;
unsigned short times;
{
        int i,k,l;
        int final;       /* to determine if even or odd coefficient    */
        short match;/* check if all literals have an intersection    */

        times ++;
        for ( i = 0 ; i < pol[which_lit[times-1]].no ; i++ )
        {
                pol_lit = pol[which_lit[times-1]].po_add;
                coeff[which_lit[times-1]] = pol_lit[i];
                /* first bit is used to determine if + 1 in algorithm 1      */
                index[which_lit[times-1]] = 1 << (30-i);
                if ( times < order )
                        gen_coeff ( order,times );
                else
                {
                        /* check if intersection for all literals is not empty        */
                        /* for 1 has to be done outside this loop !             */
                        final = 0;
                        for ( l = 0 ; l < cubes ; l++ )
                        {
                                /* check current ESOP form for all cubes    */
                                cube = cube_list[l];
                                match = TRUE;
                                for ( k = 0 ; k < literals ; k++ )
                                        if ( ( coeff[k] & cube[k] ) == 0 )
                                                match = FALSE;
                                if ( match == TRUE )
                                        if ( final == 0 )
                                                final = 1;
                                        else
                                                final = 0;
                        }
                        if ( final == 1 )
                                for ( k = 0 ; k < order ; k++ )
                                        output_value(coeff[which_lit[k]]);

                }
        }
}
```