

**A METHOD TO FIND THE BEST MIXED POLARITY
REED-MULLER EXPANSION**

by

DMITRY MASLOV

M.Sc. (Mathematics) Lomonosov's Moscow State University, 1998

A thesis submitted in partial fulfillment of

the requirements for the degree of

MASTER OF SCIENCE

in

THE FACULTY OF COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

.....
.....
.....
.....
.....

THE UNIVERSITY OF NEW BRUNSWICK

June 2001

© Dmitry Maslov, 2001

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of New Brunswick, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature) _____

Faculty of Computer Science
The University of New Brunswick
Fredericton, Canada

Date _____

Abstract

In this thesis, we use the transeunt triangle in an efficient algorithm to find the minimum mixed polarity Reed-Muller expression of a given function. This algorithm runs in $\Theta(n^2 3^n)$ time and uses $\Theta(n 3^n)$ storage space. The algorithm is also designed for multiple output functions. Efficiency of this algorithm is demonstrated on benchmark functions.

Table of Contents

Abstract	ii
Table of Contents	iii
Chapter 1. Introduction	1
Chapter 2. Background	3
Chapter 3. Minimization Algorithm	11
3.1 Transeunt Triangle	11
3.2 Getting coefficients from the RTT	23
Chapter 4. Multiple Output Functions	28
Chapter 5. Experimental Results	30
Chapter 6. Further work	34
Chapter 7. Conclusion	38
Bibliography	39
Appendix A. C code for the program that finds the best mixed polarity Reed-Muller expansion	41

Chapter 1

Introduction

There are many ways to represent a Boolean function. Much research has been devoted to *sum-of-product* expressions in which *sum* is the Exclusive-OR function and *product* is the AND of variables or complements of variables [1]. It is known that such expressions require, on average, fewer product terms than OR sum-of-products [9]. Another advantage is ease of testability [7].

AND-EXOR circuits have been used in arithmetic, error correcting, and telecommunications applications [14], [10].

Such expressions can be used to classify a given function into an equivalence class of functions, where two functions are equivalent if one is transformed into the other by permuting variables, complementing variables, and/or complementing the function [13], and is useful for determining library cells in CAD (computer-aided design) tools. This is called *Boolean matching* and is important in the determination of library cells for use by CAD tools.

This thesis focuses on a special class of AND-EXOR circuits, called mixed polarity Reed-Muller (MPRM) expressions. In an MPRM expression of a given function $f(x_1, x_2, \dots, x_n)$, every variable appears either complemented, uncomplemented, or in

both forms, in which case all terms contain the variable. If all variables are uncomplemented (complemented), the MPRM expression is called the Positive (Negative) Polarity Reed-Muller or PPRM (NPRM) form. A fixed polarity Reed Muller (FPRM) expression is one where each variable appears either complemented or uncomplemented, but never in both forms. FPRM expression are a subset of MPRM expressions. MPRM expressions are *unique* [2] for a given polarity. Thus, only one representation exists for the PPRM or NPRM or indeed any MPRM of $f(x_1, x_2, \dots, x_n)$. This leads to the question of which of the MPRM's produces the fewest product terms [5], [6], [11].

Chapter 2

Background

In this section we give the formal definitions of the terms used in this thesis. One method to describe a Boolean function $f(x_1, x_2, \dots, x_n)$ of n variables is by a truth table. This construction is nothing, but a table with $(n + 1)$ columns and 2^n rows. In the rightmost column the value of the function for the input that is placed in the first n columns is given for each row. The number of different inputs for a function of n Boolean variables is 2^n , therefore the height of this construction is 2^n . In other words, the truth table has 2^n rows. It can be seen that the truth table requires large amount of storage, so in order to simplify it, the truth vector method is used. The truth vector for a function of n variables is the sequence of Boolean numbers of length 2^n , where the k -th number of this sequence is the value of the function on the input that is the binary representation of number k . In the following example we see the advantage of the truth vector method of representing a function compared with the truth table method.

Example 1. Consider the following truth table shown in Table 2.1. The truth vector for the same function is $[0, 0, 1, 1, 1, 0, 1, 1]$, which requires one-fourth of the storage space. For a function of n variables the truth vector method requires $(n + 1)$ times less storage than the truth table.

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 2.1: Truth table method

Now consider Boolean functions $f(x_1, x_2, \dots, x_n)$ of n variables and represent them as polynomials. This means that we consider the operations of conjunction (written as $\&$ or concatenation), negation ($\bar{}$) and exclusive or (\oplus). The popular notation $x_i^{\sigma_i}$ to represent x_i if $\sigma_i = 1$ and \bar{x}_i if $\sigma_i = 0$ will be also used. A proof that the system of operations $\{\&, \bar{}, \oplus\}$ is a basis can be easily obtained from Shannon's expansion:

$$f(x_1, x_2, \dots, x_n) = \bar{x}_1 f(0, x_2, \dots, x_n) \oplus x_1 f(1, x_2, \dots, x_n).$$

Now a polynomial that represents a function of n variables can be built. One of the simplest ways to do this is:

1. Take the truth vector and for each 1, say, at i -th place, build the product that has truth vector $[0, 0, \dots, 0, 1, 0, \dots, 0]$ where 1 is exactly at the i -th place. This product will look like $x_1^{\sigma_1} x_2^{\sigma_2} \dots x_n^{\sigma_n}$, where $i = (\sigma_1, \sigma_2, \dots, \sigma_n)$ is the binary decomposition of the natural number i .

2. The function is the “exclusive or” of all products obtained from the previous step.

By Shannon’s theorem this sum-of-products that is obtained from this algorithm is equal to the function. For example, the function shown in Table 2.1 (function with truth vector $[0, 0, 1, 1, 1, 0, 1, 1]$) can be expressed as: $\bar{x}_1x_2\bar{x}_3 \oplus \bar{x}_1x_2x_3 \oplus x_1\bar{x}_2\bar{x}_3 \oplus x_1x_2\bar{x}_3 \oplus x_1x_2x_3$.

There are many polynomials which represent the same function. For example,

$$x_1 \oplus \bar{x}_1x_2 \oplus x_1\bar{x}_2x_3, \tag{2.1}$$

$$x_2 \oplus x_1\bar{x}_2\bar{x}_3, \tag{2.2}$$

and

$$x_1 \oplus x_2 \oplus x_1x_2 \oplus x_1x_3 \oplus x_1x_2x_3 \tag{2.3}$$

represent the same function shown in Table 2.1. But in this thesis a function will not be represented as any polynomial, but as a polynomial of some special type (Reed-Muller canonical form). The first polynomial is not in such a form. This becomes clear from the following definition.

Definition 1. Mixed polarity Reed-Muller expression is an ESOP that has some polarity $P = (\rho_1, \rho_2, \dots, \rho_n)$. Let $P = (\rho_1, \rho_2, \dots, \rho_n)$ where $\rho_1, \rho_2, \dots, \rho_n \in \{0, 1, 2\}$ be the polarity of a mixed polarity Reed-Muller expression for the function $f(x_1, x_2, \dots, x_n)$, which means that:

- x_i appears complemented if $\rho_i = 0$,

Chapter 2. Background

- x_i appears uncomplemented if $\rho_i = 1$ and
- x_i appears mixed if $\rho_i = 2$.



For example, if $\rho_i = 2$, then f can be decomposed as

$$f(x_1, x_2, \dots, x_n) = x_i f_0(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) \oplus \bar{x}_i f_1(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n),$$

where $f_0(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$ and $f_1(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$ have polarity

$$(\rho_1, \dots, \rho_{i-1}, \rho_{i+1}, \dots, \rho_n).$$

According to Sasao's classification [8] the above is a Kronecker expression. The polarity P is said to be fixed, iff $\rho_1, \rho_2, \dots, \rho_n \in \{0, 1\}$. That is each variable appears either complemented or uncomplemented but never in both forms.

The polynomial $x_1 \oplus \bar{x}_1 x_2 \oplus x_1 \bar{x}_2 x_3$ is not a valid mixed polarity Reed-Muller expression because x_2 is found in all three possible forms: not present, present, and present with negation. The other two polynomials (2.2) and (2.3) are Reed-Muller expansions. From the definition, $x_2 \oplus x_1 \bar{x}_2 \bar{x}_3$ gives us polarity 1 on x_1 , 2 on x_2 and 0 on x_3 , which we write in short as polarity $(1, 2, 0)$. $x_1 \oplus x_2 \oplus x_1 x_2 \oplus x_1 x_3 \oplus x_1 x_2 x_3$ gives us polarity $(1, 1, 1)$ (so, according to commonly used notations, this is a fixed polarity Reed-Muller expression, namely positive polarity, which by the definition is the polarity where all variables appear uncomplemented).

The example below illustrates how a mixed polarity Reed-Muller expression is obtained given any function.

Chapter 2. Background

Example 2. Given the three-variable function with the truth vector $[0, 0, 1, 1, 1, 0, 1, 1]$. Find the mixed polarity expression with polarity $(1, 2, 0)$. To build it, we construct the polynomial by the method described above. We have:

$$\bar{x}_1 x_2 \bar{x}_3 \oplus \bar{x}_1 x_2 x_3 \oplus x_1 \bar{x}_2 \bar{x}_3 \oplus x_1 x_2 \bar{x}_3 \oplus x_1 x_2 x_3 \quad (2.4)$$

The polynomial is given with $(2, 2, 2)$ -polarity, since all the variables appear in each term and can be found complemented as well as not complemented. In general, when we build this simplest polynomial for a function of n variables (which is also called Shannon's complete expansion or perfect exclusive-or normal form) we have a Reed-Muller expression of polarity $(2, 2, \dots, 2)$. Returning to our example we notice that in order to have a polynomial of polarity $(1, 2, 0)$ we have to change polarity of x_1 from 2 to 1 and polarity of x_3 from 2 to 0. There's no need to change polarity of x_2 since it is already correct, so we break the procedure into two parts:

1. Change the polarity of x_1 . To do this we use formula $\bar{x}_1 = x_1 \oplus 1$ and apply it to each \bar{x}_1 presented in the expression:

$$\begin{aligned} & \bar{x}_1 x_2 \bar{x}_3 \oplus \bar{x}_1 x_2 x_3 \oplus x_1 \bar{x}_2 \bar{x}_3 \oplus x_1 x_2 \bar{x}_3 \oplus x_1 x_2 x_3 = \\ & = (x_1 \oplus 1) x_2 \bar{x}_3 \oplus (x_1 \oplus 1) x_2 x_3 \oplus x_1 \bar{x}_2 \bar{x}_3 \oplus x_1 x_2 \bar{x}_3 \oplus x_1 x_2 x_3 = \\ & = x_1 x_2 \bar{x}_3 \oplus x_2 \bar{x}_3 \oplus x_1 x_2 x_3 \oplus x_2 x_3 \oplus x_1 \bar{x}_2 \bar{x}_3 \oplus x_1 x_2 \bar{x}_3 \oplus x_1 x_2 x_3 = \\ & = x_2 \bar{x}_3 \oplus x_2 x_3 \oplus x_1 \bar{x}_2 \bar{x}_3 \end{aligned}$$

2. After simplification the obtained polynomial has polarity $(1, 2, 2)$. Now we come with the procedure that changes the polarity of x_3 . We have to substitute every x_3 with $(\bar{x}_3 \oplus 1)$:

$$\begin{aligned}
 & x_2\bar{x}_3 \oplus x_2x_3 \oplus x_1\bar{x}_2\bar{x}_3 = \\
 & = x_2\bar{x}_3 \oplus x_2(\bar{x}_3 \oplus 1) \oplus x_1\bar{x}_2\bar{x}_3 = \\
 & = x_2\bar{x}_3 \oplus x_2\bar{x}_3 \oplus x_2 \oplus x_1\bar{x}_2\bar{x}_3 = \\
 & = x_2 \oplus x_1\bar{x}_2\bar{x}_3.
 \end{aligned}$$

The last Reed-Muller expression $(x_2 \oplus x_1\bar{x}_2\bar{x}_3)$ has polarity $(1, 2, 0)$.

As we see from the example above, different polarity Reed-Muller expressions have different number of terms. At this point we are ready to give the definition for the cost of a representation. In general, there are two common ways to understand the cost. The first approach is to consider the cost to be the number of literals used to create the polynomial. This understanding of the cost is widely used when one considers contact schemes [?]. In this method, every time we use a literal, we add one contact which adds one to the final cost. The geometry of the net consisting of these contacts defines the function. The second common understanding of the cost is a Programmable Logic Array(PLA)-like structure. PLA itself consists of two arrays: “AND”-array and “EXOR”-array. In the first part of a PLA, “AND”-array, we create all necessary products that are used in the “EXOR”-array part. Therefore, according to the theory of PLA design we care more about the number of products, than the number of literals in each product. In this work we chose the PLA-oriented definition of the cost, which is formally given below:

Definition 2. The cost of a Reed-Muller expression is the number of all non-zero coefficients of the corresponding polynomial. ►

In case we want to realize the function with some device, two questions are interesting: is there an efficient algorithm to find the expression with the smallest cost, and what is the maximum cost for a n -variable function. The main issue of the present thesis is to find an efficient algorithm to find the best mixed polarity Reed-Muller expansion and compare its efficiency with the efficiency of previous known algorithms. The second problem was not forgotten, and one of the hard functions was found. The problem of finding the hardest functions to represent will be generalized into the case of Cohn's inconsistent forms, but this issue refers to future work, which is introduced in Chapter 6.

Several algorithms have been created to solve the problem of finding a minimal mixed polarity Reed-Muller expression. We comment on the following three.

The first, Green [4]. The method he used is the closest to our approach. Green considered ternary maps with 3^n elements that are interpreted as rectangles. The method to find the best mixed polarity Reed-Muller expansion is very similar to ours: first, we fill the map by specially described rules. After filling the map, the author proceeds to add numbers inside the structure, and then the costs for all 3^n polarities are obtained in all 3^n different cells. Finally, Green finds the minimum cost among all 3^n costs he obtained.

The second, Lui and Muzio [5]. The authors used the Kronecker product of elementary matrices to describe the generalized Boolean matrix transforms for modulo-2 fixed

and mixed polarity Reed-Muller expansions of a Boolean function. A method for fast Boolean matrix transformations was considered to generate exhaustively all the 2^n (3^n) fixed (mixed) polarity Reed-Muller expansions. Then, the expansion with the least number of terms was selected.

The third, Drechsler, Theobald and Becker's [3]. The core of the data structure they consider is the ordered functional decision diagram (OFDD), which is a restricted type of a decision diagram (decision tree). There exists a one-to-one correspondence between so called one-paths and terms in fixed polarity Reed-Muller expansions. This allows to say that a method that minimizes the number of one-ways in OFDD gives the cost for the shortest fixed polarity Reed-Muller expansion. For the minimization the authors build an OFDD with only positive Davio-nodes and transform it step by step to an OFDD with only negative Davio-nodes. Drechsler *et al.* constructs 2^n OFDDs, determining the number of one-paths and store the best result. To avoid the construction of the same OFDD twice, they use the Gray code.

Chapter 3

Minimization Algorithm

3.1 Transeunt Triangle

Let $f(x_1, x_2, \dots, x_n)$ be a switching function of n variables, where $x_i \in \{0, 1\}$. We seek a mixed polarity Reed-Muller expression for f with minimal cost. A design algorithm based on the transeunt triangle, previously applied to symmetric functions, [?], [?], [?], [12], is also useful for arbitrary functions.

Definition 3. The **transeunt triangle** for $f(x_1, x_2, \dots, x_n)$ is a triangle of 0's and 1's where the bottom row is the truth vector of f . The j -th element in the i -th row of the triangle is denoted by $e_{i,j}$, where the indices run from top-to-bottom and left-to-right starting with $i = 0$ and $j = 1$, respectively. The truth vector corresponds to the elements $e_{2^n,0}, e_{2^n,1}, \dots, e_{2^n,2^n-1}$. The element $e_{2^n,k}$ corresponds to $f(\alpha_1, \alpha_2, \dots, \alpha_n)$, where $k = (\alpha_1, \alpha_2, \dots, \alpha_n)$ is the binary representation of integer k . Other elements are related by $e_{i,j} = e_{i+1,j} \oplus e_{i+1,j+1}$. ►

Example 3. The transeunt triangle for $f(x_1, x_2) = 1 \oplus x_1 \oplus x_2$ is shown in Fig. 3.1.

Note that a transeunt triangle for an n -variable function has a width of 2^n and a height of 2^n . There are $(4^n + 2^n)/2$ elements in total. Besides the defining relation, there are

$$\begin{array}{cccc}
 & & & 0 \\
 & & & 1 & 1 \\
 & & 1 & 0 & 1 \\
 & 1 & 0 & 0 & 1
 \end{array}$$

Figure 3.1: Transeunt triangle for $f(x_1, x_2) = 1 \oplus x_1 \oplus x_2$.

other relations among elements in the transeunt triangle, as shown below.

Lemma 1. For the transeunt triangle of any function $f(x_1, x_2, \dots, x_n)$, $e_{i,j} = e_{i+2^k,j} \oplus e_{i+2^k,j+2^k}$ where $k \geq 0$.

Proof. By induction: If $k = 0$, then $e_{i,j} = e_{i+1,j} \oplus e_{i+1,j+1}$ is true by the definition of transeunt triangle. Assume the hypothesis is true for $k = m - 1$. Therefore,

$$e_{i,j} = e_{i+2^{m-1},j} \oplus e_{i+2^{m-1},j+2^{m-1}} \quad (3.1)$$

Similarly,

$$e_{i,j+2^{m-1}} = e_{i+2^{m-1},j+2^{m-1}} \oplus e_{i+2^{m-1},j+2^{m-1}+2^{m-1}} \quad (3.2)$$

and

$$e_{i+2^{m-1},j+2^{m-1}} = e_{i+2^{m-1}+2^{m-1},j+2^{m-1}} \oplus e_{i+2^{m-1}+2^{m-1},j+2^{m-1}+2^{m-1}} \quad (3.3)$$

Substituting (3.2) and (3.3) into (3.1) yields the hypothesis. ■

Since the truth vector for functions with one or more variables has an even number of entries, it can be divided evenly into two equal parts. Each part produces, on its own,

two subtriangles. Indeed, we can identify three subtriangles, as shown in Fig. 3.2.

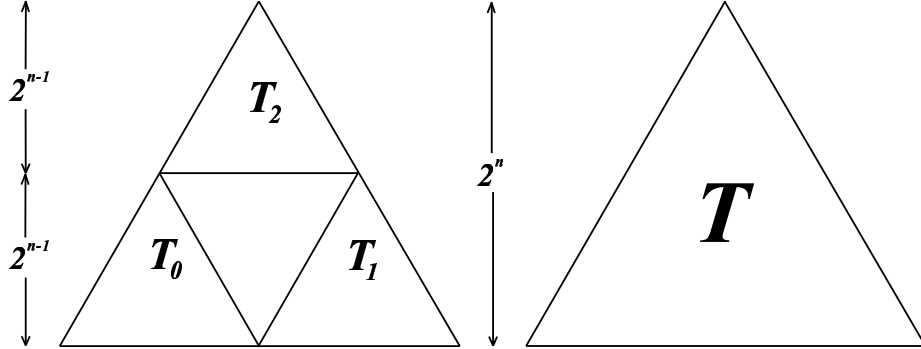


Figure 3.2: Decomposition of the transeunt triangle T .

Lemma 2. Let T be transeunt triangle for a switching function $f(x_1, x_2, \dots, x_n)$. If we divide T into 3 triangles T_0, T_1 and T_2 as shown in Fig. 3.2, then

- T_0 is the transeunt triangle for the function $f(0, x_2, \dots, x_n)$,
- T_1 is the transeunt triangle for the function $f(1, x_2, \dots, x_n)$ and
- T_2 is the transeunt triangle for the function $f(0, x_2, \dots, x_n) \oplus f(1, x_2, \dots, x_n)$.

Proof. First, denote $e_{i,j}^0$ to be the (i, j) -th element from T_0 , $e_{i,j}^1$ to be the (i, j) -th element from T_1 and $e_{i,j}^2$ to be the (i, j) -th element from T_2 . T_0 can be considered as the transeunt triangle for $f(0, x_2, \dots, x_n)$ because it's bottom row is the truth vector of $f(0, x_2, \dots, x_n)$ and every element $e_{i,j}^0 = e_{i+1,j}^0 \oplus e_{i+1,j+1}^0$ by the construction of T . Similarly, T_1 is the transeunt triangle for $f(1, x_2, \dots, x_n)$.

Note, that by Lemma 1 and the statement that the height of transeunt triangle is a power of two $e_{i,j}^2 = e_{i,j}^0 \oplus e_{i,j}^1$. Therefore, the bottom of T_2 will be “exclusive or” of the elements of the bottom row of transeunt triangles for $f(0, x_2, \dots, x_n)$ and $f(1, x_2, \dots, x_n)$,

which is the “exclusive or” of truth vectors of $f(0, x_2, \dots, x_n)$ and $f(1, x_2, \dots, x_n)$, which is the truth vector for $f(0, x_2, \dots, x_n) \oplus f(1, x_2, \dots, x_n)$. Other (non-bottom) elements of T_2 , by the construction of T , are corresponding sums. So, T_2 is the transeunt triangle for the function $f(0, x_2, \dots, x_n) \oplus f(1, x_2, \dots, x_n)$. ■

The algorithm to find the best, from the point of view of number of terms, mixed polarity Reed-Muller expansion can be described intuitively, as follows. First, given a function $f(x_1, x_2, \dots, x_n)$ as a truth vector and a polarity $P = (\rho_1, \rho_2, \dots, \rho_n)$, construct the corresponding transeunt triangle. To obtain the cost, divide the triangle into parts, extracting elements. First, divide the transeunt triangle T into the three triangles T_0, T_1 and T_2 . Consider the variable x_1 and the corresponding ρ_1 in the polarity P . If $\rho_1 = 0$ retain T_1 and T_2 . If $\rho_1 = 1$ retain T_0 and T_2 . And, if $\rho_1 = 2$ retain T_0 and T_1 . In other words, if $\rho_1 = i$ where $i \in \{0, 1, 2\}$, then delete T_i . Next, repeat this process with x_2 (and ρ_2). As a result we have four triangles. Continue this operation for all variables until each triangle is a single element. Such a one-element triangle is called an *elementary triangle*. There will be 2^n of them. The sum of all these 2^n values is the cost for the given polarity (view logic values as integers and sum as integer sum).

A formal version of the algorithm is shown in Fig.3.3.

Theorem 1. *Given function $f(x_1, x_2, \dots, x_n)$, Algorithm 1 finds the cost for a given polarity P .*

Proof. The proof is by induction.

1. If $n = 1$ write $f(x_1) = \bar{x}_1 f(0) \oplus x_1 f(1)$ which gives us polarity $P = (2)$, or $f(x_1) =$

Algorithm 1

```

GetCost( $T, P$ )

    /* Calculate the cost for polarity  $P$  */

    /* for a function  $f$  given as transeunt triangle  $T^*$  */

    if (  $|P| = 0$  ) then

        return  $e_{0,0}$  /*  $T$  is a triangle consisting of one element */

    if (  $\rho_1 = 0$  ) then

        return GetCost( $T_1, P'$ ) + GetCost( $T_2, P'$ )

        /* where  $P'$  is  $P$  with  $\rho_1$  deleted */

    if (  $\rho_1 = 1$  ) then

        return GetCost( $T_0, P'$ ) + GetCost( $T_2, P'$ )

    if (  $\rho_1 = 2$  ) then

        return GetCost( $T_0, P'$ ) + GetCost( $T_1, P'$ )

```

Figure 3.3: Recursive algorithm to find the cost for a given polarity.

$f(0) \oplus x_1(f(0) \oplus f(1))$ which gives us polarity $P = (1)$, or $f(x_1) = f(1) \oplus \bar{x}_1(f(0) \oplus f(1))$ which gives us polarity $P = (0)$. In this simple case the cost of the polarity $P = (2)$ is $f(0) + f(1)$, the cost of the polarity $P = (1)$ is $f(0) + (f(0) \oplus f(1))$ and the cost of the polarity $P = (0)$ is $f(1) + (f(0) \oplus f(1))$. The transeunt triangle for the given function will look like:

$$\begin{array}{ccc}
 & f(0) \oplus f(1) & \\
 f(0) & & f(1)
 \end{array}$$

Note, that $T_0 = f(0)$ is the transeunt triangle for $f(0)$, $T_1 = f(1)$ is the transeunt triangle for $f(1)$ and $T_2 = f(0) \oplus f(1)$ is the transeunt triangle for $f(0) \oplus f(1)$. By the algorithm, if $\rho_1 = 0$ we should consider T_1 and T_2 . These triangles are elementary, so the sum of their elements is taken. It is equal to the $f(1) + (f(0) \oplus f(1))$ which is actually the cost for the given polarity as it was shown before. If $\rho_1 = 1$ consider T_0 and T_2 and conclude that the sum of their elements is equal to the cost of polarity $P = (1)$ as it was shown. And the same goes for $P = (2)$.

2. Assume the theorem holds for $n = k - 1$. We can decompose $f(x_1, x_2, \dots, x_k)$ as follows:

$$f(x_1, x_2, \dots, x_k) = \bar{x}_1 f(0, x_2, \dots, x_k) \oplus x_1 f(1, x_2, \dots, x_k) \quad (3.4)$$

$$= f(0, x_2, \dots, x_k) \oplus x_1 (f(0, x_2, \dots, x_k) \oplus f(1, x_2, \dots, x_k)) \quad (3.5)$$

$$= f(1, x_2, \dots, x_k) \oplus \bar{x}_1 (f(0, x_2, \dots, x_k) \oplus f(1, x_2, \dots, x_k)) \quad (3.6)$$

From this formula it can be concluded that the final cost for every polarity $P = (2, \rho_2, \dots, \rho_k)$ will be the sum of the costs of polarity $P' = (\rho_2, \rho_3, \dots, \rho_k)$ for functions $f(0, x_2, \dots, x_k)$ and $f(1, x_2, \dots, x_k)$. The algorithm leads one to consider triangles T_0 and T_1 , which are transeunt triangles for $f(0, x_2, \dots, x_k)$ and $f(1, x_2, \dots, x_k)$ correspondingly. By induction, the algorithm works for functions $f(0, x_2, \dots, x_k)$ and $f(1, x_2, \dots, x_k)$ as functions from $(k - 1)$ variables, and the cost for the given polarity P' will be the sum of given by the algorithm elementary triangles inside T_0 and T_1 . But this sum is equal to the sum of all given by the algorithm elementary triangles of transeunt triangle for

$f(x_1, x_2, \dots, x_k)$ because the first step takes T_0 and T_1 due to the algorithm. And, finally, the number which was obtained can be considered as the sum of costs of polarity P' for $f(0, x_2, \dots, x_k)$ and $f(1, x_2, \dots, x_k)$ which is the cost of polarity $P = (2, \rho_2, \dots, \rho_k)$ for $f(x_1, x_2, \dots, x_k)$ as it was shown before.

For cases $P = (1, \rho_2, \dots, \rho_k)$ and $P = (0, \rho_2, \dots, \rho_k)$ the induction step is similar to the previous proof. ■

Example 4. Given the function $f(x_1, x_2, x_3)$ whose truth vector is $(0, 0, 1, 0, 1, 0, 1, 1)$ find the cost of polarity $P = (1, 2, 0)$.

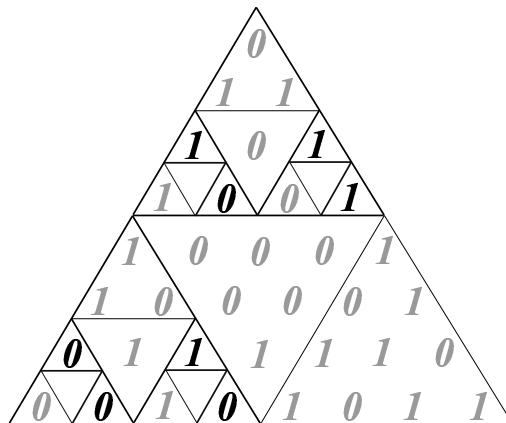


Figure 3.4: transeunt triangle for the example

First, build the transeunt triangle, then identify the MPRM coefficients associated with the given polarity according to Algorithm 1. The coefficients are shown in Fig.3.4 in black. Finally, the coefficients are summed to obtain the cost 4.

A computer program can be built to find the best mixed polarity Reed-Muller expansion which runs with the complexity of $k * n3^n$, $k \leq 4$ and requires 3^n memory places for n -bit naturals ($n * 3^n$ bits) of memory. Notice that every time the given algorithm is applied,

some triangles are divided into the 4 parts and only 3 of them are considered: all, except the central one which is never used. Therefore, at each step of the algorithm the number of elements, equal to the number of the elements inside all the central triangles considered are not used. This number of elements for the first step is $2^{2n-3} - 2^{n-2}$, for the second - $3 * (2^{2n-5} - 2^{n-3})$, and so on up to the $(n - 1)$ -th step, where we are not interested in 3^{n-2} elementary triangles. In sum it gives us:

$$\begin{aligned} & (2^{2n-3} - 2^{n-2}) + 3 * (2^{2n-5} - 2^{n-3}) + \dots + 3^{n-2} = \\ & = (2^{2n-3} + 3 * 2^{2n-5} + \dots + 3^{n-2} * 2) - (2^{n-2} + 3 * 2^{n-3} + \dots + 3^{n-2}) = \\ & = 2 * (2^{2n-4} + 3 * 2^{2n-6} + \dots + 3^{n-2}) - (2^{n-2} + 3 * 2^{n-3} + \dots + 3^{n-2}) \end{aligned}$$

Denote $m = n - 2$ and continue:

$$\begin{aligned} & 2 * (2^{2n-4} + 3 * 2^{2n-6} + \dots + 3^{n-2}) - (2^{n-2} + 3 * 2^{n-3} + \dots + 3^{n-2}) = \\ & = 2 * (4^m + 3 * 4^{m-1} + \dots + 3^m) - (2^m + 3 * 2^{m-1} + \dots + 3^m) = \\ & = 2 * 3^m \left(\left(\frac{4}{3} \right)^m + \left(\frac{4}{3} \right)^{m-1} + \dots + \left(\frac{4}{3} \right)^0 \right) - 2^m \left(\left(\frac{3}{2} \right)^0 + \frac{3}{2} + \dots + \left(\frac{3}{2} \right)^m \right) = \\ & = 2 * 3^m * \frac{\left(\frac{4}{3} \right)^{m+1} - 1}{\frac{4}{3} - 1} + 2^m * \frac{\left(\frac{3}{2} \right)^{m+1} - 1}{\frac{3}{2} - 1} = \\ & = 2 * 3 * 3^m \left(\left(\frac{4}{3} \right)^{m+1} - 1 \right) + 2 * 2^m \left(\left(\frac{3}{2} \right)^{m+1} - 1 \right) = 2 * 4^{m+1} - 2 * 3^{m+1} - 3^{m+1} + 2^{m+1} = \\ & = 2 * 4^{n-1} - 3^n + 2^{n-1} \end{aligned}$$

Now, the number of useful elements is equal to the number of all elements minus the number of non-useful elements, that gives us $2 * 4^{n-1} + 2^{n-1} - (2 * 4^{n-1} - 3^n + 2^{n-1}) = 3^n$.

In the computer program implementation of the algorithm, only a transeunt triangle that has these 3^n elements is considered. We call this a *restricted transeunt triangle*

(RTT). The construction of a RTT proceeds as follows: enumerate elements $e_{i,j}$ of transeunt triangle. The ternary number $(\gamma_1, \gamma_2, \dots, \gamma_n)$ which corresponds to the element $e_{i,j}$ of the RTT for the function $f(x_1, x_2, \dots, x_n)$ is built as follows: first, γ_1 is k , if $e_{i,j} \in T_k$, where $k \in \{0, 1, 2\}$. Second, γ_2 , is zero if $e_{i,j}$ lies in the bottom-left triangle, one if $e_{i,j}$ lies in the bottom-right triangle and two if $e_{i,j}$ lies in the top triangle inside the chosen T_k . Continue this operation until all values are obtained. For example,

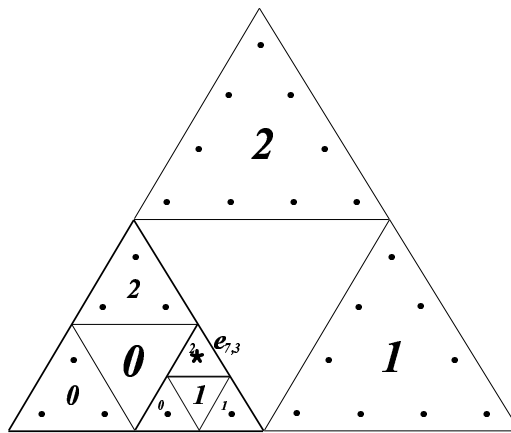


Figure 3.5: The example of building the ternary number

$e_{7,3}$ in (Fig. 3.5) has the ternary number $(0, 1, 2)$, because it is in the second smallest triangle, which is inside a bigger triangle with number 1, which in its turn, lies in the bigger triangle number 0.

Lemma 3. If the element $e_{i,j}$ with the ternary number $(\gamma_1, \gamma_2, \dots, \gamma_n)$ has $\gamma_k = 2$ for $k \in \{1, 2, \dots, n\}$, then it is equal to the exclusive or of two elements $e_{s,t}$ and $e_{u,v}$ of the RTT with ternary numbers $(\gamma_1, \dots, \gamma_{k-1}, 0, \gamma_{k+1}, \dots, \gamma_n)$ and $(\gamma_1, \dots, \gamma_{k-1}, 1, \gamma_{k+1}, \dots, \gamma_n)$ respectively.

Proof. Notice, that by the construction of a ternary number for $e_{i,j}$ the change of k -th

position from 2 to 0 gives us element $e_{s,t}$, where $s = i + 2^{n-k}$ and $t = j$. For the same reason the change of k -th position of $e_{i,j}$ from 2 to 1 gives us element $e_{u,v}$, where $u = i + 2^{n-k}$ and $v = j + 2^{n-k}$. Now we can apply Lemma 1 to prove the statement. ■

The fact that all elements with ternary numbers $(\gamma_1, \gamma_2, \dots, \gamma_n)$ where $\gamma_i \in \{0, 1\}$ lie in lexicographic order in the bottom of the RTT together with the definition of transeunt triangle gives us the following lemma.

Lemma 4. All elements with ternary numbers $(\gamma_1, \gamma_2, \dots, \gamma_n)$ where $\gamma_i \in \{0, 1\}$ are equal to $f(\gamma_1, \gamma_2, \dots, \gamma_n)$.

Given the truth vector of a function (the bottom row) we need $3^n - 2^n$ “exclusive or” operations to build the remaining elements in the RTT. That is, there are 3^n elements in the RTT, of which 2^n are the truth vector. Next, to find the best mixed polarity Reed-Muller expression in our computer program implementation of the algorithm (Figure 3.6), switching variables are interpreted as 0, 1 integers. The resulting costs can be calculated within the RTT so that the elements in T will contain the costs. First, take 3^{n-1} 3-element triangles of the RTT. Each of them has elements $e_{i,j}, e_{i+1,j}, e_{i+1,j+1}$ and we

1. set $e_{i,j}$ to $e_{i+1,j} + e_{i+1,j+1}$,
2. set $e_{i+1,j}$ to $e_{i,j} + e_{i+1,j+1}$,
3. set $e_{i+1,j+1}$ to $e_{i,j} + e_{i+1,j}$.

Note: the operations are performed simultaneously. To do this for the single 3-element triangle requires 3 addition operations. For the first step $3 * 3^{n-1} = 3^n$ addition op-

erations are needed. In general, for the k -th step take 3^{n-k} of 3^k -element triangles (decomposition of T into smaller triangles as shown in the Fig. 3.7) and, within each of them by the same rules, obtain: element-wise sum of $T_{\beta_1, \beta_2, \dots, \beta_{k-1}, 0}$ and $T_{\beta_1, \beta_2, \dots, \beta_{k-1}, 1}$ in $T_{\beta_1, \beta_2, \dots, \beta_{k-1}, 2}$, element-wise sum of $T_{\beta_1, \beta_2, \dots, \beta_{k-1}, 1}$ and $T_{\beta_1, \beta_2, \dots, \beta_{k-1}, 2}$ in $T_{\beta_1, \beta_2, \dots, \beta_{k-1}, 0}$ and element-wise sum of $T_{\beta_1, \beta_2, \dots, \beta_{k-1}, 2}$ and $T_{\beta_1, \beta_2, \dots, \beta_{k-1}, 0}$ in $T_{\beta_1, \beta_2, \dots, \beta_{k-1}, 1}$, where $\beta_i \in \{0, 1, 2\}$ for $i \in \{1, 2, \dots, k-1\}$. To do this $3^k * 3^{n-k} = 3^{k+n-k} = 3^n$ addition operations are required. After all the steps (there will be $(n-1)$ of them) are done, costs of all 3^n different mixed polarities are represented by the elements in the RTT. It is easy to see, that for polarity $P = (\rho_1, \rho_2, \dots, \rho_n)$ its cost is in the cell with ternary number $(\gamma_1, \gamma_2, \dots, \gamma_n)$. To find the minimal cost any known method which finds the minimal element of the array may be applied.

The final complexity is:

- $3^n - 2^n$ binary additions to build the RTT,
- $(n-1)3^n$ natural addition operations to find all costs,
- $3^n - 1$ comparisons to find the minimal cost.

Needed memory is:

- 3^n integers plus a few integer variables for intermediate assignments (3 were used).

The algorithm is best illustrated with an example.

Example 5. Consider the same function used in the previous example. Build the RTT first, which already has been done, therefore just use the result as it is shown in Fig.3.8A. Note, that those elements that are not used are initially colored gray in Fig.3.8A and

will be eliminated from the future parts of the figure. Then corresponding sums inside the small 3-element triangles are found and placed in the triangle as follows:

- the sum of bottom elements is stored in the top of the triangle;
- the sum of leftmost bottom and top element is stored in the rightmost bottom place;
- the sum of rightmost bottom and top element is stored in the leftmost bottom place.

Assume that these three steps are done in parallel. The procedure is shown by arrows on the rightmost bottom triangle in Fig.3.8A. The procedure is repeated for each 3-element triangle colored black. The result of this step is shown in Fig.3.8B. Then take the 9-element triangles and proceed with element-wise summation of elements of the 3-element triangles inside each of them. The order of this summation is also shown for one of the triangles in Fig.3.8B, but is done for all 3 triangles that exist at this step. Fig.3.8C gives us the result of our operation. Now we take the element-wise sum of sub-triangles T_0, T_1 and T_2 of the whole triangle T and obtain the final result as shown in Fig.3.8D. Each cost of the polarity $P = (\rho_1, \rho_2, \rho_3)$ is given by the element with the ternary number (ρ_1, ρ_2, ρ_3) . From the figure we see that the minimum cost is 3. The polarity of the minimal cost circled in Fig.3.8D corresponds to $P = (1, 2, 2)$, which is not unique. There are 3 other polarities with corresponding polynomials that have the same cost.

3.2 Getting coefficients from the RTT

Given a RTT one can find all the elements inside it which are needed to obtain the cost of any given polarity as it is shown in the previous chapter. The question that one may ask is: can the whole expansion be obtained and not only its cost from the transeunt triangle? The answer is “Yes” and in order to obtain the polynomial we need to enumerate all the needed elementary triangles. The enumeration is as follows: for a given polarity $P = (\rho_1, \rho_2, \dots, \rho_n)$ and triangle $T_{\beta_1, \beta_2, \dots, \beta_n}$ its binary number as the needed for the given polarity element is $(\epsilon_1, \epsilon_2, \dots, \epsilon_n)$ where ϵ_i for $i = 1, 2, \dots, n$ can be obtained from the table:

ρ_i	β_i	ϵ_i	$\varphi(\rho_i, \epsilon_i, i)$
0	1	0	1
0	2	1	\bar{x}_i
1	2	0	x_i
1	0	1	1
2	0	0	\bar{x}_i
2	1	1	x_i

Table 3.1: table of meanings for $\rho_i, \beta_i, \epsilon_i$ and $\varphi(\rho_i, \epsilon_i, i)$

Denote every coefficient of a term as $E(\epsilon_1, \epsilon_2, \dots, \epsilon_n)$ due to its binary number $(\epsilon_1, \epsilon_2, \dots, \epsilon_n)$.

Then, the polynomial for a mixed polarity $P = (\rho_1, \rho_2, \dots, \rho_n)$ Reed-Muller expansion will look like

$$\bigoplus_{\epsilon_1, \epsilon_2, \dots, \epsilon_n \in \{0,1\}} E(\epsilon_1, \epsilon_2, \dots, \epsilon_n) \& \varphi(\rho_1, \epsilon_1, 1) \& \varphi(\rho_2, \epsilon_2, 2) \& \dots \& \varphi(\rho_n, \epsilon_n, n),$$

where the meaning of $\varphi(\rho_i, \epsilon_i, i)$ is taken from the Table 3.2. The proof of correctness of this formula is similar to the proof of Theorem 1 and can be done easily by induction and usage of formulas (3.4). The formula will now be illustrated by the following example.

Example 6. Take the function and polarity from the previous example. The transeunt triangle is already built, so the next step is to get all $E(\epsilon_1, \epsilon_2, \dots, \epsilon_n)$ with their numbers as shown in Fig.3.9 (identify numbers ϵ_i shown in the figure by the Table 3.2, interpreting it as the truth table for the function $\epsilon_i(\rho_i, \beta_i)$). Finally, using the Reed-Muller expansion polynomial and Fig.3.9 there will be:

$$\begin{aligned}
 & \bigoplus_{\epsilon_1, \epsilon_2, \epsilon_3 \in \{0,1\}} E(\epsilon_1, \epsilon_2, \epsilon_3) \& \varphi(\rho_1, \epsilon_1, 1) \& \varphi(\rho_2, \epsilon_2, 2) \& \varphi(\rho_3, \epsilon_3, 3) = \\
 & = E_{0,0,0} \& \varphi(1, 0, 1) \& \varphi(2, 0, 2) \& \varphi(0, 0, 3) \oplus E_{0,0,1} \& \varphi(1, 0, 1) \& \varphi(2, 0, 2) \& \varphi(0, 1, 3) \oplus \\
 & \oplus E_{0,1,0} \& \varphi(1, 0, 1) \& \varphi(2, 1, 2) \& \varphi(0, 0, 3) \oplus E_{0,1,1} \& \varphi(1, 0, 1) \& \varphi(2, 1, 2) \& \varphi(0, 1, 3) \oplus \\
 & \oplus E_{1,0,0} \& \varphi(1, 1, 1) \& \varphi(2, 0, 2) \& \varphi(0, 0, 3) \oplus E_{1,0,1} \& \varphi(1, 1, 1) \& \varphi(2, 0, 2) \& \varphi(0, 1, 3) \oplus \\
 & \oplus E_{1,1,0} \& \varphi(1, 1, 1) \& \varphi(2, 1, 2) \& \varphi(0, 0, 3) \oplus E_{1,1,1} \& \varphi(1, 1, 1) \& \varphi(2, 1, 2) \& \varphi(0, 1, 3) = \\
 & = 0 \& x_1 \& \bar{x}_2 \& 1 \oplus 1 \& x_1 \& \bar{x}_2 \& \bar{x}_3 \oplus 1 \& x_1 \& x_2 \& 1 \oplus 1 \& x_1 \& x_2 \& \bar{x}_3 \oplus \\
 & \oplus 0 \& 1 \& \bar{x}_2 \& 1 \oplus 0 \& 1 \& \bar{x}_2 \& \bar{x}_3 \oplus 0 \& 1 \& x_2 \& 1 \oplus 1 \& 1 \& x_2 \& \bar{x}_3 = \\
 & = x_1 \bar{x}_2 \bar{x}_3 \oplus x_1 x_2 \oplus x_1 x_2 \bar{x}_3 \oplus x_2 \bar{x}_3.
 \end{aligned}$$

Algorithm 2

```

CalcCostTriangle( $T_{\text{cost}}(f)$ ,  $f$ )

/* Returns the cost triangle  $T_{\text{cost}}(f)$  of function  $f$ . */

if (  $n > 1$  ) then

     $T_{\text{cost}}(f) = T_{\text{cost}}(f_1) + T_{\text{cost}}(f_0 \oplus f_1)$ ,

         $T_{\text{cost}}(f_0) + T_{\text{cost}}(f_0 \oplus f_1)$ ,

         $T_{\text{cost}}(f_0) + T_{\text{cost}}(f_1)$ 

else if (  $n = 1$  ) then

    if  $f = 0$  then  $T_{\text{cost}}(f) = 0, 0, 0$ 

    if  $f = x$  then  $T_{\text{cost}}(f) = 2, 1, 1$ 

    if  $f = \bar{x}$  then  $T_{\text{cost}}(f) = 1, 2, 1$ 

    if  $f = 1$  then  $T_{\text{cost}}(f) = 1, 1, 2$ 

else  $T_{\text{cost}}(f)$  is undefined.

end CalcCostTriangle

/* Note: Composition of triangles, as in

 $T_{\text{cost}} = T_{\text{cost}0}, T_{\text{cost}1}, T_{\text{cost}2}$ , places  $T_{\text{cost}0}$  in the lower

left corner,  $T_{\text{cost}1}$  in the lower right corner, and

 $T_{\text{cost}2}$  in the apex. */

```

Figure 3.6: Recursive algorithm to find costs for all mixed polarities for a given function.

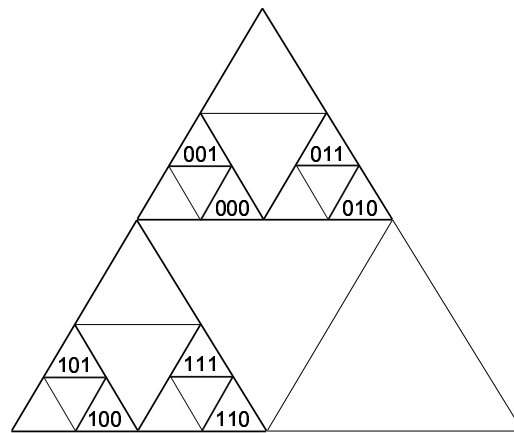


Figure 3.9: Binary numbers for needed elements

Chapter 4

Multiple Output Functions

The method developed in the previous section also works for multiple output functions. Recall that a multiple output function is nothing more, but a Boolean vector-function $(f_1(x_1, x_2, \dots, x_n), f_2(x_1, x_2, \dots, x_n), \dots, f_k(x_1, x_2, \dots, x_n))$ of some dimensionality k . To build Reed-Muller expansion for this multiple output function we need to build expansions for all functions $f_1(x_1, x_2, \dots, x_n), f_2(x_1, x_2, \dots, x_n), \dots, f_k(x_1, x_2, \dots, x_n)$. There are two different approaches to define the cost of this expansion. One is to realize each function separately as polynomials and add all costs. From the point of view of present work the cost defined by this rule is easy to find, and the method to find the best mixed polarity Reed-Muller expansion has minimal adaptations, namely, create k transeunt triangles and, after obtaining all the costs for all functions $f_1(x_1, x_2, \dots, x_n), f_2(x_1, x_2, \dots, x_n), \dots, f_k(x_1, x_2, \dots, x_n)$ take element-wise sum of all k triangles.

The more interesting case is if a term that has been used once, can be used again without increasing the final cost. In this case one needs to minimize the number of distinct terms used to realize the set of functions $f_1(x_1, x_2, \dots, x_n), f_2(x_1, x_2, \dots, x_n), \dots, f_k(x_1, x_2, \dots, x_n)$. This understanding of cost is supported by existence of PLAs, that, in fact, consist of two arrays: “AND”-array and “EXOR”-array. In the first part of a PLA, “AND”-array,

we create all necessary terms that are used in the “EXOR”-array part as many times as it is needed. Therefore, in the theory of PLA design we care about the number of terms, not the number of times the term is used.

For our algorithm, this means that the multiple output function

$F(x_1, x_2, \dots, x_n) = (f_1(x_1, x_2, \dots, x_n), f_2(x_1, x_2, \dots, x_n), \dots, f_k(x_1, x_2, \dots, x_n))$ the truth vector is given as a list of words (i.e. is vectors made of ones and zeros of length k) rather than simple bits. To build the transeunt triangle the words inside the triangle are element-wise “exclusive or”-ed. Before the cost is calculated, all entries in the transeunt triangle are set to 1 if the word contains at least one nonzero bit and are set to 0 otherwise. The cost of the best polarity is calculated as before.

The correctness of the algorithm for the multiple output functions can be explained as follows. First proceed by setting the numbers inside the triangle structure for the whole multiple output function, a 1 identifies that the word contains at least one 1. Note that it doesn't matter how many ones were in the word. If there is at least one 1 in the word and the cell was chosen, then the product term must be created. It doesn't matter how many times it will be used, since we are interested not in the number of times it is used, but in the number of terms needed. 0 represents the case when the word consists of zero's and if the cell is chosen, then no term should be constructed.

Chapter 5

Experimental Results

The C programming language was used to implement the algorithm. The complete program listing is in the Appendix. The RTT is represented as a flat array of the length 3^n (variable `int* vec`). This structure helps to work with small triangles inside the initial big one and simplifies the work with recursive functions. Logically, the algorithm consists of three parts. In the first, “fill” part, we

- allocate memory for our transeunt triangle,
- set values in the truth vector and
- the recursive function `void fill()` will set the meanings of the transeunt triangle.

The second part, “normalize”, is needed only for multiple output functions. Here all non-zero elements are changed to 1 since each term that is used more than once adds only 1 to the overall cost (a term can be reused). The last part, “cost”, takes the transeunt triangle that was built in the previous part of the program, interprets boolean constants as integers and recursively calculates costs of all polarities. Numbers for the costs of polarities are put inside the triangle, so that in some sense the algorithm is destructive, since the costs were found the initial structure of the transeunt triangle is lost. This third part is performed by the `void calcMixedCost`, according to algorithm 2. Note

that this algorithm is destructive in that at the end of it we don't have RTT unless a copy of it is made before the third part of the algorithm is activated. Finally, the minimal cost is achieved by calling **minimum(vec, length)** that returns the minimal polarity cost.

The program was applied to several benchmark functions running on a Sun Enterprise 250 with two 400Mhz Ultra Sparc II processors and 1GB of main memory. The results are summarized in Table 5. In the first column, *name*, denotes the name of the function (from MCNC benchmarks). Note: *co14* is a symmetric function where the output is one if exactly 1 input variable is 1. In general *con* is an n variable symmetric function whose output is 1 if exactly one input is 1. Function *hard_n* is explained in the following section. The number of inputs (outputs) are given in *in (out)*. The minimum number of terms required for a fixed (mixed) polarity Reed Muller expression are shown in *cost fixed (cost mixed)*. *OFDD time* denotes the cpu time in seconds for the OFDD implementation reported in [3] (run on a HP Apollo series 700 workstation). *RTT time* gives the cpu time in seconds for our algorithm. Our results compare favorably with previous implementations. The program is able to minimize any function with up to 18 variables. Memory requirements make it difficult to minimize functions with more than 18 variables.

The Internet was used to obtain the performances of the Sun Enterprise 250 with two 400Mhz Ultra Sparc II processors and the HP Apollo series 700 workstation for comparison purposes. From the official Sun web site it was found that the SPECint95 performance of the Sun machine is 19.7 [?] while the performance of the Hewlett Packard

CPU 744 SPECint95 is 5.90 [?]. As the result of this comparison one may see that to be fair the results for OFDD should be factored by $4 \approx 19.7/5.90$, but as we see from the table results for our method is still significantly better. This comparison of computers is only an approximation, because of a lack of enough information about the computer used by Drechsler *et al.*

Green's [4] algorithm (which is based on Karnaugh-like maps and has the same complexity) has never been implemented. In the paper there's no algorithm for getting coefficients of the corresponding Reed-Muller expansion. Partially symmetric functions were not discussed. The author also claims that the method designed in this thesis is easier to program.

Drechsler, Theobald and Becker's [3] implementation is significantly slower than our program and their algorithm considers only fixed polarity expressions. Drechsler *et al.* do not provide a complexity analysis for their algorithm. However, the execution time depends on the function as well as on the number of variables, whereas our algorithm depends only on the number of variables.

Lui and Muzio [5] describe an algorithm based on Boolean matrix transforms. It has space complexity of $\Theta(2^n)$ and time complexity of $\Theta(6^n)$.

<i>name</i>	<i>in</i>	<i>out</i>	<i>cost fixed</i>	<i>cost mixed</i>	<i>OFDD time (secs.)</i>	<i>RTT time (secs.)</i>
rd53	5	3	20	20	0.5	< 0.01
rd73	7	3	63	63	2.3	< 0.01
rd84	8	4	107	107	5.5	< 0.01
root	8	5	118	83	8.8	< 0.01
dist	8	5	185	157	12.5	< 0.01
9sym	9	1	173	173	8.1	< 0.01
sao2	10	4	100	76	8.8	0.02
add6	12	7	132	132	295.1	0.17
co14	14	1	14	14	448.4	1.99
tial	14	8	3683	2438	8480.4	2.04
table3	14	14	1845	407		2.01
misex3	14	14	3536	1421		2.02
co15	15	1	15	15		6.68
gary	15	11	349	242	16216.1	6.44
co16	16	1	16	16		20.41
co17	17	1	17	17		64.51
table5	17	15	2458	559		65.88
co18	18	1	18	18		214.11
<i>hard</i> ₁₀	10	1	252	252		0.01
<i>hard</i> ₁₂	12	1	924	924		0.16
<i>hard</i> ₁₄	14	1	3432	3432		1.95
<i>hard</i> ₁₆	16	1	12870	12870		20.05
<i>hard</i> ₁₈	18	1	48620	48620		211.37

Table 5.1: Experimental results

Chapter 6

Further work

The first part of the proposed work will be to reduce the size of RTT for partially symmetric functions. Without loss of generality consider the function $f(x_1, x_2, \dots, x_n)$, partially symmetric in its first 2 variables. Build its RTT and consider triangles $T_{0,1}$ and $T_{1,0}$ shown on the Fig.6.1. It is known that triangle $T_{0,1}$ is the RTT for the $f(0, 1, x_3, \dots, x_n)$

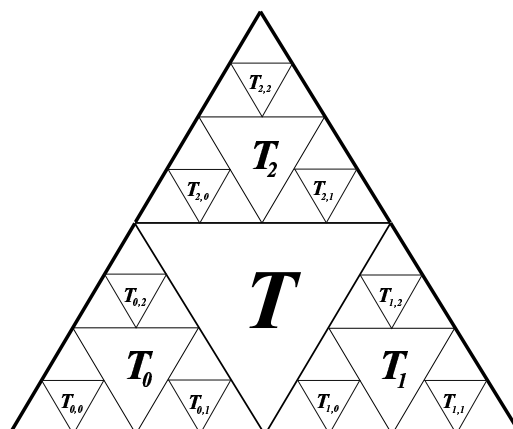


Figure 6.1: RTT for the function

and $T_{1,0}$ is the RTT for the $f(1, 0, x_3, \dots, x_n)$. Because of the symmetry of the function in the first 2 arguments these triangles have the same bottom, therefore they are equivalent. It is also known that $T_{2,0}$ is element-wise “exclusive or” of triangles $T_{0,0}$ and $T_{1,0}$ and $T_{0,2}$ is element-wise “exclusive or” of triangles $T_{0,0}$ and $T_{0,1}$, so they are equal

because of the identity of $T_{0,1}$ and $T_{1,0}$. The same goes for triangles $T_{1,2}$ and $T_{2,1}$ that are equal to “exclusive or” of $T_{1,0}$, $T_{1,1}$ and $T_{0,1}$, $T_{1,1}$ correspondingly. It can be seen that there’s no need to hold both $T_{0,1}$ and $T_{1,0}$, $T_{0,2}$ and $T_{2,0}$, $T_{1,2}$ and $T_{2,1}$ in memory. Therefore, only one from each pair can be held, and the storage space for function partially symmetric in two variables is reduced by 6/9 (keep 6 out of 9 triangles). The new storage space is $6 * 3^{n-2}$.

It should not be too difficult to prove that for partially symmetric functions in k variables one can use a storage space of $\frac{(k+1)(k+2)}{2} * 3^{n-k}$. The proof of this conjecture is under investigation right now. The proof can be obtained by combining of the method suggested by Butler *et al.* for the symmetric functions and the method for general functions, which is discussed in the given work. The implementation of such an algorithm must be more difficult.

The symmetry is exploited by Drechsler’s algorithm [3] in terms of storage. This is due to the fact that OBDDs take advantage of symmetry naturally. However the symmetry is not exploited during the path enumeration.

Cohn conjectured that any boolean function $f(x_1, x_2, \dots, x_n)$ can be realized by a polynomial with not more than $C_n^{\lfloor n/2 \rfloor}$ terms [1]. Consider a polynomial in the basis of conjunction, negation and exclusive or which represents Cohn’s inconsistent form. In other words, one can consider polynomials such that for any set of indices $\{i_1, i_2, \dots, i_k\}$, where $i_1, i_2, \dots, i_k \in \{1, 2, \dots, n\}$ there is at most one term which contains exactly the variables $x_{i_1}, x_{i_2}, \dots, x_{i_k}$, each variable appearing either complemented or not. There’s still no proof or counterexample for this conjecture, although the problem is easy to

Chapter 6. Further work

understand. The boundary $C_n^{\lfloor n/2 \rfloor}$ is achievable for the function $hard_n$ of even number of variables $n = 2k$ that is equal to one iff equally half of its' inputs are one's. In other words, $hard_n = \bigoplus_{1 \leq s_1 < s_2 < \dots < s_k \leq n} x_{s_1} x_{s_2} \dots x_{s_k}$ which is a shortest form of polynomial for our function (from the point of view that no better can be obtained).

To prove this statement one needs the following definitions and lemma.

Definition 4. The degree of the polynomial, $deg(p(x_1, x_2, \dots, x_n))$, is the number of variables in the longest term $x_{i_1}^{\sigma_1} x_{i_2}^{\sigma_2} \dots x_{i_k}^{\sigma_k} \in p(x_1, x_2, \dots, x_n)$. ►

Definition 5. Two functions (polynomials) are called equal if they have equivalent truth vectors. ►

Lemma 5. Given two equal polynomials $p_1(x_1, x_2, \dots, x_n)$ and $p_2(x_1, x_2, \dots, x_n)$ which are both in Cohn's inconsistent form. One can transform one into the other by applying the following operations:

1. $x_i \longrightarrow \bar{x}_i \oplus 1$;
2. $\bar{x}_i \longrightarrow x_i \oplus 1$;
3. $x_{i_1}^{\sigma_1} x_{i_2}^{\sigma_2} \dots x_{i_t}^{\sigma_t} \oplus x_{i_1}^{\sigma_1} x_{i_2}^{\sigma_2} \dots x_{i_t}^{\sigma_t} \longrightarrow 0$;

Proof. First, take all the terms of $p_1(x_1, x_2, \dots, x_n)$ that have degree $deg(p_1(x_1, x_2, \dots, x_n))$.

Notice that for any $x_{i_1}^{\sigma_1} x_{i_2}^{\sigma_2} \dots x_{i_k}^{\sigma_k}$ from $p_1(x_1, x_2, \dots, x_n)$ $x_{i_1}^{\delta_1} x_{i_2}^{\delta_2} \dots x_{i_k}^{\delta_k}$ from $p_2(x_1, x_2, \dots, x_n)$

- a term of the same degree with the same set of variables in it can be found. Prove this statement by contradiction. Suppose that it is not so. Then there exist $x_{i_1}^{\sigma_1} x_{i_2}^{\sigma_2} \dots x_{i_k}^{\sigma_k}$

Chapter 6. Further work

from $p_1(x_1, x_2, \dots, x_n)$ such that there's no pair (in the sense described above) for it. Create the sum $p_1 \oplus p_2$, which must be equal zero as the exclusive or of equal things. Apply operation 2 wherever possible and open all the brackets. After doing this we have the sum of $x_{i_1}x_{i_2}\dots x_{i_k}$ and other terms of smaller degree and equal degree, but having other sets of indices. Therefore, after applying operation 3, we find that $x_{i_1}x_{i_2}\dots x_{i_k}$ remains. This mean that $p_1 \oplus p_2 \neq 0$, which is a contradiction. Therefore, knowing that for each term of highest degree a pair exists (another term with the same set of variables), apply operations 1 and 2 to change $x_{i_1}^{\sigma_1}x_{i_2}^{\sigma_2}\dots x_{i_k}^{\sigma_k}$ to $x_{i_1}^{\delta_1}x_{i_2}^{\delta_2}\dots x_{i_k}^{\delta_k}$. Some new terms may appear during this procedure, but they all will be of the smaller degree. Now, the parts of degree k of two polynomials are equal. After this is done, proceed with the same operation on all the terms of degree $(k-1)$, and so on. Finally, transform $p_1(x_1, x_2, \dots, x_n)$ into $p_2(x_1, x_2, \dots, x_n)$. ■

Now, from the proof of previous Lemma we see that if we take $hard_n$ as the positive polarity Reed-Muller expression and that any other Cohn's inconsistent form polynomial which represents the same function, the second will be of the same degree and will contain all the terms of degree k that have the same set of variables as a term from the first polynomial. Therefore, the number of terms in the second polynomial can not be smaller than those for the first one. This function is used in our computer program. This is a good test function, since it is complex, yet the minimal result is known. The last result can be formulated as the following theorem:

Theorem 2. *The presented function $hard_n$ proves that $C_n^{n/2}$ gives the lower boundary for the number of terms needed to realize a Boolean function of n variables as a Cohn's inconsistent form.*

Chapter 7

Conclusion

A new algorithm to minimize fixed and mixed polarity Reed Muller expressions is presented in this thesis. The algorithm is easy to understand and to program. One can apply geometric intuition dealing with RTT. The results compare favorably to previous algorithms, although the complexity of $\Theta(3^n)$ in either storage and execution time have not been reduced.

Initially, an algorithm involving transeunt triangles has been devised to find the best polarity for totally symmetric functions in $\Theta(n^3)$ operations and $\Theta(n^2)$ storage space [?].

A dramatic reduction in complexity can be achieved for partially symmetric functions.

This area is currently under investigation.

Bibliography

- [1] M. Cohn. Inconsistent canonical forms of switching functions. *IRE Transactions of Electronic Computers*, EC-11:284–285, 1962.
- [2] M. J. Davio, P. Deschamps, and A. Thayse. Discrete and switching functions. *McGraw-Hill Int. Book Co.*, 1978.
- [3] R. Drechsler, M. Theobald, and B. Becker. Fast OFDD-based minimization of fixed polarity Reed-Muller expressions. *IEEE Transactions on Computers*, 45:1294–1299, Nov. 1996.
- [4] D. H. Green. Reed-Muller canonical forms with mixed polarity and their manipulations. *IEE Proceedings, Part E.*, 137:103–113, Jan. 1990.
- [5] P. K. Lui and J. C. Muzio. Boolean matrix transforms for the minimization of modulo-2 canonical expressions. *IEEE Transactions on Computers*, 41:342–347, Mar. 1992.
- [6] A. Mukhopadhyay and G. Schmitz. Minimization of exclusive-or and logical-equivalence switching circuits. *IEEE Trans. on Computers*, pages 132–140, 1970.
- [7] S. M. Reddy. Easily testable realisations for logic functions. *IEEE Trans. on Computers*, pages 1183–1188, 1972.
- [8] T. Sasao. *Logic Synthesis and Optimization*. Kluwer Academic Publisher, 1993.
- [9] T. Sasao and Ph. W. Besslich. On the complexity of mod-2 sum pla's. *IEEE Trans. on Computers*, C-29:262–266, Feb. 1990.
- [10] J. Saul. Logic synthesis for arithmetic circuits using the Reed-Muller representation. *Proc. Euro. Conf. Design Automation*, pages 109–113, 1992.
- [11] I. Schafer and M. A. Perkowski. Multiple-valued input generalized Reed-Muller forms. *Proc. of the Inter. Symp. on Multiple-Valued Logic*, pages 40–48, 1991.

Bibliography

- [12] V. P. Suprun. Fixed polarity Reed-Muller expressions of symmetric Boolean functions. *Proc. IFIP WG 10.5 Workshop on Application of the Reed-Muller Expansions in Circuit Design*, pages 246–249, 1995.
- [13] Chien-Chang Tsai and M. Marek-Sadowska. Boolean Functions Classification via Fixed Polarity Reed-Muller Forms. *IEEE Trans. on Computers*, C-46(2):173–186, Feb. 1997.
- [14] T.Sasao. Switching theory for logic synthesis. *Kluwer Academic Publishers, Norwell, MA*, 1999.

Appendix A

C code for the program that finds the best mixed polarity Reed-Muller expansion

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <sys/resource.h>

/*****/
/* prototypes */
/*****/
int minimum(int *arr, int sz);
int index(int *vars);
void setTerm(char *term, int ovalue);
void fill(int *v, int size);
void normalize(int *v, int size);
void calcMixedCost(int *func, int tn);

/*****/
/* global variables */
/*****/
int* vec; /* all values in the triangle */
```

Appendix A. C code for the program that finds the best mixed polarity Reed-Muller expansion

```
int n; /* number of variables */

/* returns elapsed user CPU time in usec. */
long usertime()
{
    struct rusage *usage;

    usage=malloc(sizeof(struct rusage));
    getrusage(RUSAGE_SELF,usage);
    return(usage->ru_utime.tv_sec*1000000+usage->ru_utime.tv_usec);
}

/* Display a time value in msec. */
void printtime(long x)
{
    printf("%8.3f msec\n", (float)x/1000.0);
}

/*****
find the minimum value in an array
*****/
int minimum(int *arr, int sz){
    int res = arr[0], i;
    for(i = 1; i < sz; i++)
        if(res > arr[i])
            res = arr[i];
    return res;
}
```

Appendix A. C code for the program that finds the best mixed polarity Reed-Muller expansion

```

/*****
    convert the triangle index to a flat one
*****/
int index(int *vars){
    int res = 0, i;
    for(i = 0; i < n; i++){
        res = (res * 3) + vars[i];
    }
    return res;
}

/*****
    Set the truth values for a single term
*****/
void setTerm(char *term, int ovalue){
    int *vars, *dashes, ndash = 0, i, done;
    vars = malloc(n*sizeof(int));
    dashes = malloc(n*sizeof(int));

    for(i = 0; i < n; i++){
        vars[i] = term[i] == '1' ? 1 : 0;
        if(term[i] == '-'){
            dashes[ndash] = i;
            ndash++;
        }
    }

    done = 0;
    while(!done){
        vec[index(vars)] = vec[index(vars)] | ovalue;
        i = 0;
    }
}

```

Appendix A. C code for the program that finds the best mixed polarity Reed-Muller expansion

```
        while(i < ndash && vars[dashes[i]] == 1 ){
            vars[dashes[i]] = 0;
            i++;
        }
        if(i == ndash)
            done = 1;
        else
            vars[dashes[i]] = 1;
    }
    free(vars);
    free(dashes);
}

/*****
    fill all values in vec
    must be called after the truth values are set
*****/
void fill(int *v, int size){
    int i, s3 = size/3;
    if(size > 3){
        fill(v, s3);
        fill(v+s3, s3);
    }
    for(i = 0; i < s3; i++)
        v[i+2*s3] = v[i] ^ v[i+s3];
}

/*****
    change all non-zero elements to 1
    used for multiple output functions
*****/
void normalize(int *v, int size){
```

Appendix A. C code for the program that finds the best mixed polarity Reed-Muller expansion

```
    int i;
    for(i = 0; i < size; i++)
        v[i] = v[i] >= 1;
}

/*****
    calculate the cost for all mixed polarities

    destructive ==> the values in func are overwritten
*****/

void calcMixedCost(int *func, int tn){
    register i, t1, t2, t3;
    int sz = pow(3,tn-1);
    int sz2 = 2 * sz;

    if(tn > 1){
        calcMixedCost(func,tn-1);
        calcMixedCost(func+sz,tn-1);
        calcMixedCost(func+sz2,tn-1);
    }

    for(i = 0; i < sz; i++){
        t1 = func[i] + func[i + sz2];
        t2 = func[i + sz] + func[i+ sz2];
        t3 = func[i] + func[i + sz];
        func[i] = t2;
        func[i + sz] = t1;
        func[i + sz2] = t3;
    }
}
```

Appendix A. C code for the program that finds the best mixed polarity Reed-Muller expansion

```
}

int main(int argc, char *argv[])
{
    long time1, time2;
    int o, pos, ch, ovalue;
    char parm[12];
    char *finputs, *foutputs;
    int length, clength, i;
    int *cost, bestCost;
    FILE *fin;

    if( !(fin = fopen(argv[1], "r"))){
        printf("Could not open file \n");
        exit(1);
    }

    fscanf(fin,"%s %d", &parm, &n);
    if(strcmp(".i",parm) != 0){
        printf("ERROR expecting .i\n");
        exit(1);
    }

    length = pow(3,n);
    vec = malloc(length*sizeof(int));
    for( i = 0; i < length; i++)
        vec[i] = 0;
    fscanf(fin,"%s %d", &parm, &o); /* read number of ouputs */
    if(strcmp(".o",parm) != 0){
        printf("ERROR expecting .o\n");
    }
}
```

Appendix A. C code for the program that finds the best mixed polarity Reed-Muller expansion

```
        exit(1);
    }

    finputs = malloc(n*sizeof(char) + 1);
    foutputs = malloc(o*sizeof(char) + 1);
    finputs[n] = 0;
    foutputs[o] = 0;

    /* read all the terms          */
    /* very curde: strict input format */
    ch = getc(fin);
    while(EOF != (ch = getc(fin))){
        for(i = 0; i < n ; i++){
            finputs[i] = (char) ch;
            ch = getc(fin);
        }
        ch = getc(fin);
        ovalue = 0;
        for(i = 0; i < o ; i++){
            foutputs[i] = (char) ch;
            ovalue = ovalue * 2 + (ch == '1');
            ch = getc(fin);
        }
        setTerm(finputs, ovalue);
    }

    time1 = usertime();
    fill(vec, length);
    normalize(vec, length);
    calcMixedCost(vec,n);
    bestCost = minimum(vec, length);
```

Appendix A. C code for the program that finds the best mixed polarity Reed-Muller expansion

```
printf("%s best mixed polarity cost = %d\n ",argv[1], bestCost);
time2 = usertime();
printf("time elapsed = ");
printtime(time2 - time1);
return 0;
}
```