812

Evolutionary design of digital circuits using Improved Multi Expression Programmig

Fatima Zohra Hadjam

Claudio Moraga

Lars Hildebrand

Table of contents

Evolutionary design of digital circuits using Improved Multi Expression Programming

Fatima Zohra Hadjam [1] Claudio Moraga [2, 3] Lars Hildebrand [2]

[1]Dept. Comp. Science, University of Djillali Liabes, Sidi Bel abbes, ALGERIA.

fatima.hadjam@udo.edu

[2]FB Informatik, Universität Dortmund 44221 Dortmund, Deutschland

[3]European Centre for Soft Computing, 33600 Mieres, Spain

claudio.moraga@udo.edu

hildebrand@uni-dortmund.de

Abstract

Evolutionary Electronics is a research area which involves application of Evolutionary Computation in the domain of electronics. It is seen as a quite promising alternative to overcome the drawbacks of conventional design. In this report we propose a methodology based on an Improved Multi Expression Programming (IMEP) to automate the design of combinational logic circuits in which we aim to reach the functionality and to minimize the total number of used gates. MEP is a Genetic Programming variant that uses linear chromosomes for solution encoding. A unique MEP feature is its ability of encoding multiples solutions of a problem in a single chromosome. These solutions are handled in the same time complexity as other techniques that encode a single solution in a chromosome. This report presents the main idea of an improved version of the MEP method, and shows positive preliminary experimental results.

Keywords: Evolutionary Computation, Genetic Programming, Multi Expression Programming, combinational circuits, computational effort.

1 Introduction

Automatic methods of digital circuit design are desirable, as a skilled human designer's time is often expensive. Some parts of the design process, such as finding the optimal set of component specifications to fulfill certain criteria, are often considered tedious, as they do not fully use a designer's skill.

Considering the increasing interest, research and application of evolutionary computation successfully used to solve search and optimization problems (particularly in cases when no efficient algorithms are known), a new field emerges: Evolutionary Electronics (EE). This new field is now seen as holding good chances for overcoming the drawbacks of conventional design techniques. EE considers the concept for automatic design of electronic systems, employing search algorithms to develop good designs.

The idea of electronic circuit design as a search task is summarized as follows [15]:

Imagine a design space where each point in that space represents the design of an electronic circuit. All possible electronics circuits are there, given the component types available to electronics engineer and the technological restrictions on how many components there can be and how they can interact. In this metaphor, we loosely visualize the circuits to be arranged in the design space so that similar circuits are close to each other. The idea of using Evolutionary Algorithms is not only to optimize digital chips layout, but also to accomplish the whole process of circuit design, including designing the circuit topology from scratches.

Recent research has begun to show that it is possible to design such circuits in a radically different way. One regards the problem of implementing the circuit as being equivalent to designing a black box with inputs and outputs. The content of the box is encoded into a chromosome and subject to the process of evolutionary algorithms. In this technique, the fitness of a particular chromosome is measured as the degree to which the black box outputs behave in the desired way.

Sushil and Rawlins [5] applied GAs to the combinational circuits design problem while John Koza [4] adopted genetic programming. Coello, Christiansen and Aguire [2] presented a computer program that automatically generates high quality circuit designs. Miller and Thomson [7] two of the pioneers in the field of evolvable digital circuits, used a special technique called Cartesian Genetic Programming (CGP). The results [8] show that CGP was able to evolve some digital circuits better than those designed by human experts.

2 Problem Statement

Design is first the process of deriving, from an input/output behavior specification, a structure (a combination of logic gates) that is functional (all combinations of the truth table are satisfied). Furthermore, we want this design to be optimum in terms of a certain set of specified constraints (e.g., the number of gates used, the depth of the produced circuit or expected power consumption).

Genetic Programming is an extension of John Holland's genetic algorithm (1975) in which the population consists of computer programs of varying sizes and shapes. Genetic Programming ordinarily evolves computer programs that are represented as rooted, point-labeled trees with ordered branches.

Multi Expression Programming (MEP) [10], [11] is a Genetic Programming (GP) variant that uses linear chromosomes for solution encoding. A unique MEP feature is its ability of encoding multiple solutions of a problem in a single chromosome. These solutions are handled within the same time complexity as other techniques that encode a single solution in a chromosome.

In our work, we are using an Improved Multi Expression Programming (IMEP) for evolving digital circuits. MEP uses linear chromosomes of fixed length. It has been documented [10] that MEP performs significantly better than other competitor techniques (such as Genetic Programming, Cartesian Genetic

Programming, Gene Expression Programming and Grammatical Evolution) for some well-known problems such as symbolic regression and even-parity.

In this report, we focus only on combinational logic circuits, which contain no memory elements and no feedback paths. However, the approach proposed is general enough as to allow its adaptation to more complex circuits.

3 Multi Expression Programming (MEP)

The Multi Expression Programming (MEP) technique is described in this section:

3.1 MEP algorithm

The standard MEP algorithm [9] uses a steady state as its underlying mechanism. The MEP algorithm starts by creating a random population of individuals. The following steps are repeated until a stop condition is reached. Two parents are selected using a selection procedure. The parents are recombined in order to obtain two offsprings. The offsprings are considered for mutation. The best offspring replaces the worst individual in the current population if the offspring is better than the worst individual. The algorithm returns as its answer the best expression evolved along a fixed number of generations :

S1. Randomly create the initial population P(0)

S2. for t = 1 to Max Generations do

S3. for k = 1 to |P(t)| / 2 do

S4. p1 = Select(P(t)); // select one individual from the current population

S5. p2 = Select(P(t)); // select the second individual

S6. Crossover (p1, p2, o1, o2); // crossover parents p1 and p2 // offsprings o1 and o2 are obtained

S7. Mutation (o1); // mutate the offspring o1

S8. Mutation (o2); // mutate the offspring o2

S9. Select best the individual from {o1,o2, the worst individual} based on the fitness;

S10. endfor

S11. endfor

3.2 MEP representation

MEP genes are represented by substrings of a variable length. The number of genes per chromosome is constant. This number defines the length of the chromosome. Each gene encodes a terminal or a function symbol. A gene encoding a function includes pointers towards the function arguments. Function arguments always have indices of lower values than the position of that function in the chromosome which ensures that no cycle arises while the chromosome is decoded. According to the proposed representation scheme [12], [9] the first symbol of the chromosome must be a terminal symbol. In this way only syntactically correct programs (MEP individuals) are obtained. Offsprings obtained by crossover and mutation are always syntactically correct. Thus, no extra processing for repairing newly obtained individuals is needed (see Section 4.4). Example: the following sets are used :

F ={ AND, OR, XOR }: Function set, T = {a, b, c}: Terminal set.

An example of a chromosome using the sets F and T is given below (the labels shown in the example do not belong to the chromosome):

1: a

2: b

3: AND 1, 2

4: c

5: OR 1, 2

6: XOR 3, 5

7: AND 4, 6

3.3 MEP phenotype's transcription

This section describes the way in which the MEP individuals are translated into computer programs. A terminal symbol specifies a simple expression. A function symbol specifies a complex expression (formed by linking the operands specified by the argument positions with the current function symbol).

For instance, genes 1, 2 and 4 in the previous example encode simple expressions formed by a single terminal symbol. These expressions are: E1 = a; E2 = b; E4 = c.

Gene 3 indicates the operation AND on the operands located at positions 1 and 2 of the chromosome. Therefore gene 3 encodes the expression: E3 = a AND b.

Gene 5 indicates the operation OR on the operands located at positions 1 and 2 of the chromosome. Therefore gene 5 encodes the expression: E5 = a OR b.

Gene 6 indicates the operation XOR on the operands located at positions 3 and 5 of the chromosome. Therefore gene 6 encodes the expression: E6 = (a AND b) XOR (a OR b).

and finally, Gene 7 indicates the operation AND on the operands located at positions 4 and 6 of the chromosome. Therefore gene 7 encodes the expression: E7 = c AND ((a AND b) XOR (a OR b)).

The expression associated to each chromosome position is obtained by reading the chromosome bottom-up from the current position, by following the links provided by the functions pointers. The fitness of each expression encoded in a MEP chromosome is computed. The best expression encoded in a MEP chromosome is chosen to represent the chromosome (the fitness of a MEP individual equals the fitness of the best expression encoded in that chromosome).

Due to its multi expression representation, each MEP chromosome may be viewed as a forest of trees rather than as a single tree, which is the case of Genetic Programming. Figure 1 shows the forest of expressions encoded by the previously presented MEP chromosome.

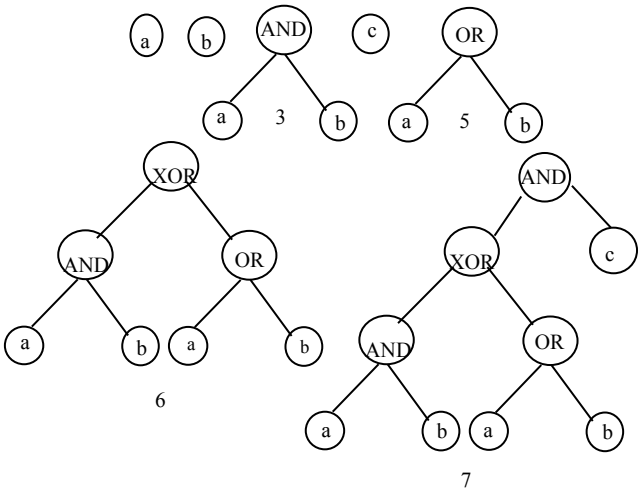

Figure 1: Expressions encoded by a MEP chromosome represented as trees

4 IMEP for evolving digital circuits

In this section, we show how the MEP method is improved:

The new representation is based on rearranging the nodes: we keep all terminals in the first positions (genes) and no other genes containing terminals are allowed in the rest of the chromosome. We have done this change to improve efficiency (see below, mutation). For example:

Old representation	**New representation**
1: a	1: a
2: b	2: b
3: AND 1, 2	3: c
4: c	4: AND 1, 2
5: OR 1, 2	5: OR 1, 2
6: XOR 3, 5	6: XOR 4, 5
7: AND 4, 6	7: AND 3, 6

4.1 IMEP algorithm

S1. Randomly create the initial population P(0) // keeping all terminals in the first positions.

S2. for t = 1 to Max Generations do

S3. for k = 1 to |P(t)| / 2 do

S4. p1 = Select(P(t)); // select one individual from the current population

S5. p2 = Select(P(t)); // select the second individual

S6. Crossover (p1, p2, o1, o2); // crossover parents p1 and p2 // offsprings o1 and o2 are obtained

S7. Mutation (o1); // mutate the offspring o1

S8. Mutation (o2); // mutate the offspring o2

S9. Select best 2 individuals from {p1, p2, o1,o2} based on the fitness

S10. endfor

S11. Mutate a copy of the Best Individual

S12. Replace randomly an individual, except for the best one, with the mutated copy
// avoid to lose the fittest

S13. Mutation(Worst Individual)

S14. Replace the worst individual with the worst mutated

S15. endfor

Notice that S13 and S14 were added to the algorithm because some times the worst individual may contain good genes to be exploited, so by mutating this individual its fitness may improve and therefore it will have better chances to be selected.

4.2 Fitness computation

Each circuit has one or more inputs (denoted by NI) and one or more outputs (denoted by NO).

When multiple genes are required as outputs we have to select those output genes which minimize the difference between the obtained results and the expected output. Each of the IMEP chromosome expressions is considered as being a potential solution of the problem. Partial results are computed by Dynamic Programming [1]. A terminal symbol specifies a simple expression (a variable: circuit input). A function symbol specifies a complex expression obtained by connecting the operands specified by the argument positions with the current function symbol. The fitness of each sub-expression (gene) is calculated by computing this sub-expression for each case (truth table input combinations) and then comparing with the corresponding target value (truth table outputs): the fitness value is given by the number of not matching values. The chromosome fitness is defined as the fitness of the best expression(s) encoded by that chromosome. Fitness = 0 means that 100% of target values match with the values given by this (these) sub-expression(s).

The quality of the gene for a given output is given by [9]:

$$f(E_i, q) = \sum_{k=1}^{n} \left(o_{i,k} - w_{k,q}\right) \qquad (1)$$

Where $o_{i,k}$ is the computed value for the gene i (Expression i) and for the combination k and $w_{k,q}$ is the target value for the combination k and the output q.

The minimized fitness for a given output is given by [9]:

$$f(O) = \min_{i_1, i_2, \ldots, i_{NO}} \sum_{q=1}^{NO} f\left(E_{i_q}, q\right) \qquad (2)$$

In our case, the minimized fitness is chosen with respect to the minimum number of not matching values (Eq 2), then to the max number of correct outputs in the same chromosome and finally to the minimum of the total number of gates (circuit devices). The observation of the correct outputs in the same chromosome covers the case when during evolution a chromosome might possibly lead to a circuit giving some correct outputs, but at permuted positions.

4.3 The evolution operators

The evolution operators used within IMEP algorithm are Selection, Crossover and Mutation. As explained earlier (Section 3), they preserve the chromosome structure thus, all offsprings are syntactically correct expressions.

Selection: we use the tournament with variable size. The size is dependent on the population size. The most used value is 2 .

Crossover: two parents are selected and recombined to produce offsprings. In our experiments, we have considered two kinds of crossover: One cut crossover and multi-cut crossover. Cut points are chosen randomly.

Example: Let us consider the two parents P_1 and P_2 given below. If the multi-cut crossover is used with the selected exchange points 3, 5, 9 and 10 then two offspring O_1 and O_2 are obtained:

Parent P_1		**Parent P_2**		**Offspring O_1**		**Offspring O_2**	
0: x_0	**0 0**	0: x_0	0 0	**0: x_0**	0 0	0: x_0	0 0
1: x_1	**0 0**	1: x_1	0 0	**1: x_1**	0 0	1: x_1	0 0
2: x_2	**0 0**	2: x_2	0 0	**2: x_2**	0 0	2: x_2	0 0
3: xor	**0 0**	3: and not	2 0	3: and not	2 0	3: xor	0 0
4: and	**3 2**	4: and	1 3	**4: and**	3 2	4: and	1 3
5: and	**1 4**	5: xor	4 2	5: xor	4 2	**5: and**	1 4
6: and	**1 5**	6: and not	2 5	**6: and**	1 5	6: and not	2 5
7: xor	**4 6**	7: xor	6 0	**7: xor**	4 6	7: xor	6 0
8: and not	**4 0**	8: xor	7 4	**8: and not**	**4 0**	8: xor	7 4
9: xor	**4 7**	9: and not	7 2	9: and not	7 2	**9: xor**	**4 7**
10: and	**4 8**	10: and not	5 5	10: and not	5 5	**10: and**	**4 8**
11: and not	**5 9**	11: and	10 5	11: and not	5 9	11: and	10 5

If one cut point crossover is used (for instance at 6) then two offsprings O_3 et O_4 are obtained:

Parent P_1		**Parent P_2**		**Offspring O_3**		**Offspring O_4**	
0: x_0	**0 0**	0: x_0	0 0	**0: x_0**	0 0	0: x_0	0 0
1: x_1	**0 0**	1: x_1	0 0	**1: x_1**	0 0	1: x_1	0 0
2: x_2	**0 0**	2: x_2	0 0	**2: x_2**	0 0	2: x_2	0 0
3: xor	**0 0**	3: and not	2 0	**3: xor**	0 0	3: and not	2 0
4: and	**3 2**	4: and	1 3	**4: and**	3 2	4: and	1 3
5: and	**1 4**	5: xor	4 2	**5: and**	1 4	5: xor	4 2
6: and	**1 5**	6: and not	2 5	**6: and**	1 5	6: and not	2 5
7: xor	**4 6**	7: xor	6 0	7: xor	6 0	**7: xor**	**4 6**
8: and not	**4 0**	8: xor	7 4	8: xor	7 4	**8: and not**	**4 0**
9: xor	**4 7**	9: and not	7 2	9: and not	7 2	**9: xor**	**4 7**
10: and	**4 8**	10: and not	5 5	10: and not	5 5	**10: and**	**4 8**
11: and not	**5 9**	11: and	10 5	11: and	10 5	**11: and not**	**5 9**

Mutation: According to the new representation, the mutation process has been modified. The first genes representing the problem variables are immune against mutation. The function symbols can be mutated only into other function symbols and the links (pointing to the function arguments) can also be mutated into other links, however satisfying the constraint that function arguments always have indices of lower values than the

position of that function in the chromosome. This reduces the probability of generating redundant individuals.

Example:

Chromosome C		**Offspring O**	
0: x_0	0 0	0: x_0	0 0
1: x_1	0 0	1: x_1	0 0
2: x_2	0 0	2: x_2	0 0
3: xor	**0 1**	**3: and**	**0 1**
4: and not	2 3	4: and not	2 3
5:and	**0 4**	**5: xor**	**0 2**
6: and	5 2	6: and	5 2
7: xor	4 6	7: xor	4 6
8: and not	**5 7**	**8: and**	**4 3**
9: xor	4 5	9: xor	4 5
10: and	8 9	10: and	8 9
11: and not	10 7	11: and not	10 7

The original method [9] describes the mutation as follows : each symbol (terminal, function or link) in the chromosome may be target of mutation operator (a terminal may become a function and function may become a terminal). The first gene of the chromosome must always encode a terminal symbol.

5 Numerical experiments

In this section, numerical experiments with Improved MEP for evolving digital circuits, are performed. For this purpose several well-known test problems [8] are used.

To assess the performance of the algorithms, we consider two statistics : the success rate and the computational effort:

Success rate = Number of successful runs / the total number of runs

Computational effort: in [4] Koza describes a method to compare the results of different evolutionary methods. The so called Computational Effort is calculated as the number of fitness evaluations needed to find a solution of a problem with a probability of success z of at least z = 99%. We have to use relative frequencies instead of probabilities for finding the solution after a certain number of fitness evaluations. One first calculates P(M,i), the probability of success by generation i using a population of size M. For each generation i this is simply the total number of runs that succeeded on or before the ith generation, divided by the total number of runs conducted. One then calculates I(M,i,z), the number of individuals that must be processed to produce a solution by generation i with probability greater than z (where z is usually 99%). The minimum of I(M,i,z) over the range of i is defined as the "computational effort" required to solve the problem.

Koza defined the following equation:

$$I(M,z) = \min_i \left(M * i * ceil\left[\ln(1-z) / \ln(1-P(M,i)) \right] \right) \qquad (3)$$

P(M,i) = $N_s(i)$ / N_{total}, where $N_s(i)$ represents the number of successful runs at generation i and N_{total} represents the total number of runs.

Before presenting our examples, we first introduce an integer-coded form of the truth table, which we have developed to reduce the space and processing time.

Before being used, the truth table is processed in order to reduce the number of combinations checked during the evolution process. It is a kind of parallelism of data. The idea is to group each column in one word (16 or 32 bits) depending on the number of combinations (2^{NI}, where NI is the number of the circuit inputs). Words will be interpreted as the binary representation of a (non-negative) integer and will be coded by the corresponding integer value.

The truth table given by table 1 represents, for instance, the Two Bits Multiplier.

Table 1. Truth table for Two Bits Multiplier

A1	**A0**	**B1**	**B0**	**P3**	**P2**	**P1**	**P0**
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

The new truth table of the Two Bits Multiplier is given by table 2.

Table 2. The new truth table of the Two Bits Multiplier

A1	**A0**	**B1**	**B0**	**P3**	**P2**	**P1**	**P0**
255	3855	13107	21845	1	50	854	1285

When the number of combinations is greater than 32, each column is divided into blocks of 32 bit and each block will be coded by the corresponding integer. This strategy enables us to divide the computing time by a factor of 32 when NI > 4, else by a factor of 2^{NI}.

For Example the truth table of the Three Bits Multiplier is given by the Table 3 below . The original one consists of 64 rows instead of 2.

Table 3. The new truth table of the Three Bits Multiplier :

A2	**A1**	**A0**	**B2**	**B1**	**B0**
0	65535	16711935	252645135	858993459	1431655765
4294967295	65535	16711935	252645135	858993459	1431655765
P5	**P4**	**P3**	**P2**	**P1**	**P0**
0	3	3868	996141	3364198	5570645
66311	252582937	859188522	1431987832	3364198	5570645

5.1 Experiment details

The performance of the IMEP was tested on four different classes of problems shown in Table4: Digital Adder, Digital Multiplier, Digital Comparators and N_Bit Even Parity problem.

Table 4: The experimental problems used to test the performance of IMEP.

Problem	Inputs	Outputs	Description
2_Bit Adder	5	3	The sum of two 2_bit numbers and 1_bit carry to produce 2_bit number and 1_bit carry
2_Bit Multiplier	4	4	The Product of two 2_bit numbers to produce 4_bit number
3_Bit Multiplier	6	6	The Product of two 3_bit numbers to produce 6_bit number
1_Bit Comparator	2	3	Compares two 1_bit numbers for <,=,>
2_Bit Comparator	4	3	Compares two 2_bit numbers for <,=,>
3_Bit Comparator	6	3	Compares two 3_bit numbers for <,=,>
N-Bit Even Parity problem	N	1	The function returns True if an even number or none of its arguments are True.

5.2 Results

The results show the contribution of the changes introduced to the MEP algorithm and in the mutation process. These results are compared to those published in [8], [12], [14], [13], [3] depending on the studied case.

5.2.1. **Standard MEP versus Improved MEP**

First, two examples were evolved: 2_bit Adder and 2_bit Multiplier and compared to those published in [8]. The parameters used are given by Table 5.

Table 5 : The parameter settings used in experiments of the first part

Parameters	Values
Number of Runs	100
Selection	Binary Tournament
Crossover	multi-cut crossover
Crossover Probability	0.9
Mutation Probability	3 genes / chromosome
Functions Set 1 for Multiplier	A AND B, A AND NOT B, A XOR B
Functions Set 3 for Adder	MUX(A,B,C), A XOR B

In [8], two experiments have been done on both 2_bit multiplier and 2_bit adder. The first experiment kept the chromosome length fix and equal to 20 genes and the population size was varied from 10 to 300 individuals . In the second one, the population size was kept fix equal to 20 and the chromosome length was varied from 10 to 100 genes. The number of generations used in all experiments [8] was **150,000**. The results obtained with the IMEP algorithm, compared to the results of [8], are given in the tables 6 and 7 below.

Table 6: Fixed length chromosome

Fixed length chromosome = 20 genes			
2_bit Adder with carry		**2_bit Multiplier**	
Standard MEP	**Improved MEP**	**Standard MEP**	**Improved MEP**
A population size of 270 individuals yields over 90% successful runs. After this value, the success rate does not increase significantly.	A population size of **50** individuals yields 99% successful runs after **10,000** generations only.	A population size of 90 individuals yields 100% successful runs.	A population size of **20** individuals yields 100% successful runs after **2,000** generations only.

Table 7: Fixed population size

Fixed Population Size = 20 individuals			
2_bit Adder with carry		**2_bit Multiplier**	
Standard MEP	**Improved MEP**	**Standard MEP**	**Improved MEP**
A chromosome length of 80 genes yields over 90% successful runs. After this value, the success rate does not increase.	A chromosome length of **50** genes yields over 97% successful runs after **5,000** generations only.	A chromosome length of 100 genes yields over 90% successful runs.	A chromosome length of **50** genes yields **100%** successful runs after **5,000** generations only.

Four other examples were evolved: Parity problems (3_bit, 4_bit, 5_bit and 6_bit) and compared with the results published in [13]. The parameters used in [13] are given in Table 8. According to [4] and [8] , the boolean even Parity problem appears to be extremely difficult to evolve using standard logic gates AND, NAND, OR, NOR. According to [3], the Even Parity problem is a very hard classification problem for GP to solve; increasing rapidly in difficulty and solution size with N (N is the number of problem inputs). Koza has shown that N = 5 represents, in effect, an upper limit for standard GP, even with a large population size of 8000 [4]. To solve the problem for N = 6 and higher, large populations and Automatically Defined Functions (**ADF**) [4] are required.

Table 8: The parameters setting according to [13]

Parameters	**Values**
Number of Runs	100
Number of Generations	51
Selection	q-Tournament (q= 10% of the Population Size)
Crossover	multi-cut crossover
Mutation Probability	0.1
Functions Set	A AND B, A NAND B, A OR B, A NOR B

In [13], two experiments have been done on parity problems (3_bit and 4_bit). The first experiment kept the chromosome length fixed to 200 genes and the population size was varied from 20 to 400 individuals . In the second one, the population size was kept fixed to 100 and the chromosome length was varied from 50 to 500 genes.

For the 3_bit parity problem, we have used the same parameters given by the table above. The results are shown in the table 9.

Table 9: The comparative results of 3_bit Parity

3_bit Parity	
Standard MEP	**Improved MEP**
A chromosome length of **270** genes with population size of **100** individuals yields a 100% successful runs. Also A chromosome length of **200** genes with population size of **300** individuals yields a 100% successful runs.	A chromosome length of **180** genes with population size of **100** individuals yields a 100% successful runs.

We can see that the Improved MEP outperforms the standard one. We have tried also to evolve the same problem using different parameters which are given by the following table 10. We have noticed that the size of the tournament used causes a high pressure so premature convergence was attained. And we have concluded that a mutation = 0.1 causes the best individual to be lost during the evolution. We have noticed also according to our experiments that a small population size with a large number of generations gives better solutions in quality and time because as argued before (Section 4.3), after the fitness = 0 is attained, the evolution system tries to minimize the number of gates.

Table 10: The new parameters setting

Parameters	Values
Number of Runs	100
Number of Generations	200
Chromosome Length	100
Population Size	60
Selection	Binary Tournament
Crossover	multi-cut crossover
Mutation rate	3 genes / chromosome
Functions Set	A AND B, A NAND B, A OR B, A NOR B

We have obtained also a 100% successful runs, but in less time than by other methods. The run time over 100 runs was equal to **5:49:920** (M:S:mS) for the parameters used in [13] and equal to **3:19:984** using our parameters. Then we have decided to use new parameters to evolve the parity problems (4_bit, 5_bit and 6_bit). The results are given by table 11, 12 and 13 respectively.

Table 11: The comparative results of 4_bit Parity

4_bit Parity		
CGP (Cartesian Genetic Programming)	**Standard MEP**	**Improved MEP**
The best result was found after 1,000,000 generations of (1+4) ES (mutation equal to 2 genes per genotype). 15 successful runs over 100 (15%).	The best result was 42% successful runs obtained by a chromosome length of **200** genes with population size of **300** individuals	A chromosome length of **30** genes with population size of **50** individuals during 30,000 generations yields **70%** successful runs. A chromosome length of **100** genes with population size of **100** individuals during 10,000 generations yields **96%** successful runs.

Table 12: the comparative results of 5_bit Parity

5_bit Parity		
Standard GP	**Standard MEP**	**Improved MEP**
The best result found was 1 successful run over 8 (12.5%) using a population of **8000** individuals	The best result found was 5 successful runs over 30 (16.66%) using a population of **1000** individuals having **600** genes each.	A chromosome length of **100** genes with population size of **50** individuals during 40,000 generations yields **20** successful runs over **50** (**40%**). A chromosome length of **150** genes with population size of **50** individuals during 40,000 generations yields **35** successful runs over **50** (**70%**)

Table 13: the comparative results of 6_bit Parity

6_bit Parity		
Standard GP	**Standard MEP**	**Improved MEP**
Not Solvable without ADF	Not Solvable without ADF	A chromosome length of **150** genes with population size of **50** individuals during 100000 generations yields **8** successful runs over **50** (**16%**). A chromosome length of **150** genes with population size of **100** individuals during 100000 generations yields **20** successful runs over **50** (**40%**).

5.2.2. **IMEP versus CGP and ECGP**

The Cartesian Genetic Programming and Embedded Genetic Programming Methods were introduced respectively, by Miller and Thomson in [7] and by Walker and Miller in [14].

Originally CGP used a program topology defined by a rectangular grid of nodes with a user defined number of rows and columns. The genotype is a fixed length representation and consists of a list of integers which encode the function and connections of each node in the directed graph.

ECGP is an extension of CGP that can automatically acquire, evolve and re-use partial solutions in the form of modules.

The performance of IMEP was tested on two different classes of problems: digital adders (1_bit, 2_bits and 3_bits) digital multipliers (2_bits and 3_bits) and digital comparators (1_bit, 2_bits and 3_bits) (see table 4 above). The computational effort spent by IMEP was compared to the one spent by CGP and ECGP tested in [14] on the same problems cited above. The parameter settings used for CGP and ECGP [14] in all the experiments are: a (1+4) ES, 300 genes as initial genotype size and a genotype point mutation rate equal to 6 genes (2%). Other proper parameters are given in [14]. The parameter settings used for IMEP are: a population size varying between 5 and 50, where the number of genes was varying between 100 and 300 (depending on the studied case) with a crossover probability equal to 0.9. Over all five problems tested, IMEP, CGP and ECGP produced 100% successful solutions over 50 independent runs. The functions set used to evolve the comparators and the adders is {AND, NAND, OR, NOR} and the functions set used to evolve the multipliers is {A AND B, A AND NOT B, A XOR B}(see [14]). The results are given in the table 14.

Table 14:The computational effort figures for IMEP, CGP and ECGP for digital multipliers and digital comparators.

	IMEP	**CGP**	**ECGP**
1_Bit adder	5,460	26,720	35,840
2_Bit adder	303,600	493,760	203,520
3_Bit adder	3,916,350	2,599,360	1,530,880
2_Bit Multiplier	2,180	35,840	35,520
3_Bit Multiplier	1,864,450	8,659,840	1,917,760
1_Bit Comparator	15	2,880	3,200
2_Bit Comparator	24,000	78,880	87,360
3_Bit Comparator	670,220	466,880	520,320

It may be seen that in most cases IMEP outperforms CGP and ECGP, except for the relatively more complex problems such as the 3_Bit Adder and the 3_Bit Comparator.

Some of the evolved circuits relative to the examples given in this paper are given by figures 2 through 8. Input or output terminals marked with a black dot are meant to be holding a logical 1; otherwise, a logical 0.

6 Conclusion and future works

In this report, an Improved Multi Expression Programming (IMEP) has been used for evolving digitals circuits. It has been shown that the new algorithm with the modified representation and the mutation process have improved the standard MEP. In the different outperformed experiments, all the circuits were evolved from scratch. Well known benchmark problems such as multipliers, comparators, full adders and even parity problems, and comparative studies with other methods, were used to asses the performance of the improved MEP. The results show that IMEP outperforms :

- ✓ MEP in the both studied cases: 2_bit multiplier and 2_bit adder.
- ✓ MEP, GP and CGP in the case of the even parity problem.

Another comparative study was done between IMEP, CGP and ECGP. IMEP shows significant speedup when compared with non modular CGP and even with ECGP in the most of cases but did not perform as well as CGP and ECGP on the digital 3_bit comparator problem. (Notice that also CGP gave better results than ECGP in the case of comparators). This phenomenon found on the comparators will be investigated further in future work. Perhaps a potential reason can be the limit values of certain parameters like the chromosome length and the population size, we intend to use parallelism (Distributed IMEP) to overcome this drawback in our future works Parallelism may contribute to decrease the average number of evaluations required by each algorithm to achieve their best possible fitness value under the principle of "divide to conquer".

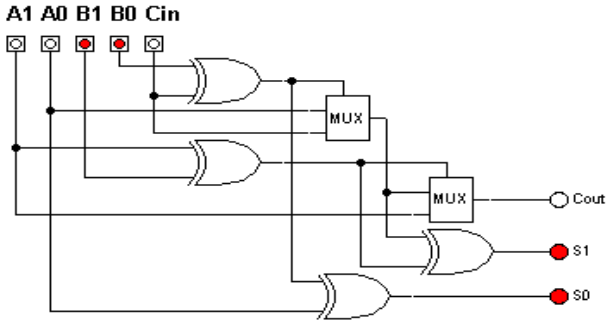

Figure 2: Evolved 2-bits adder with carry : 6 gates with 3 levels using {mux, xor}

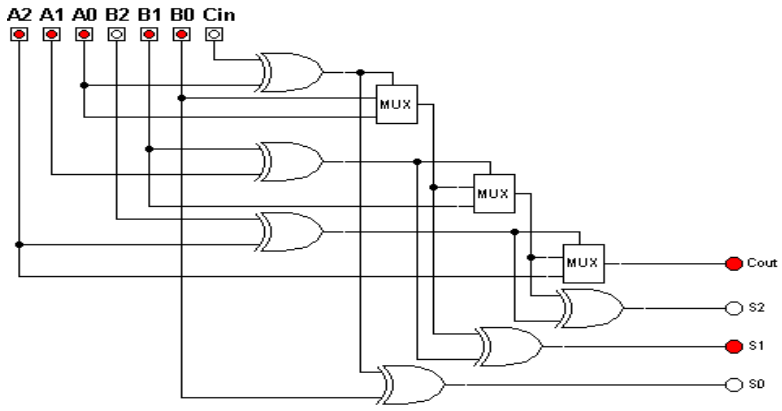

Figure 3: Evolved 3-bits adder with carry : 9 gates with 4 levels using {mux, xor}

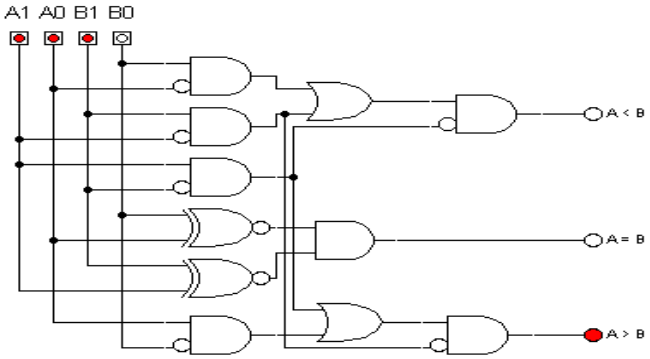

Figure 4 : Evolved 2-bits comparator : 11 gates using
{and, and with one input inverted, or, nxor}

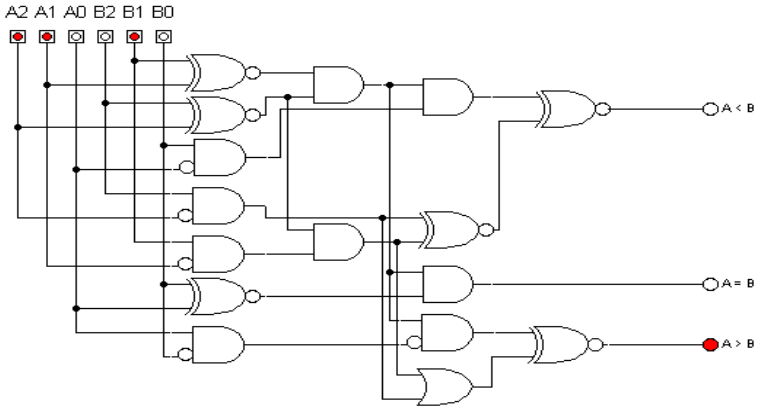

Figure 5 : Evolved 3-bits comparator : 16 gates using
{and, and with one input inverted, or, nxor}

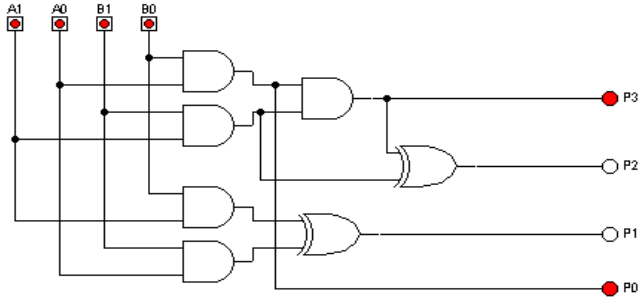

Figure 6 : Evolved Evolved 2-bits multiplier : 7 gates with 3 levels, using {and, xor}

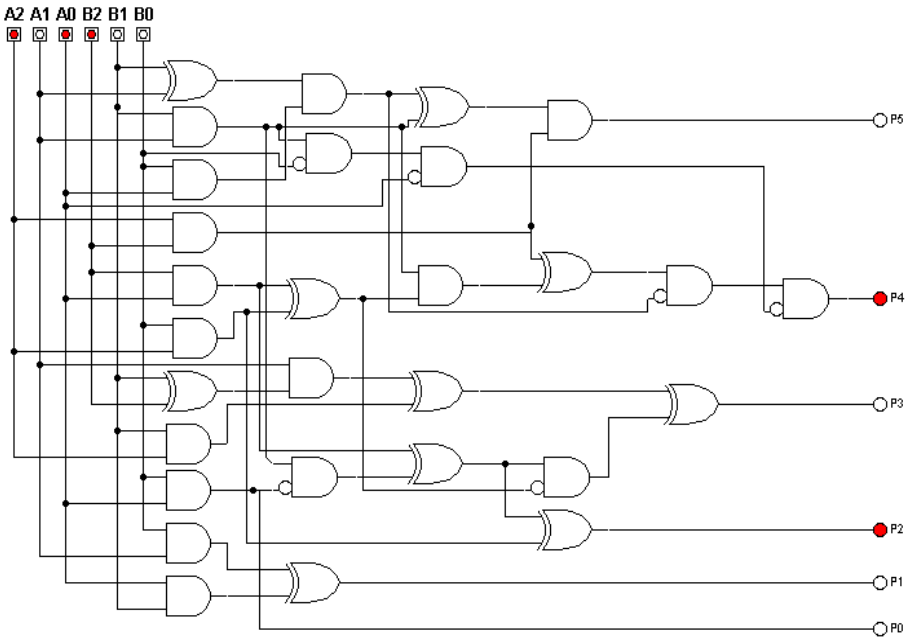

Figure 7 : Evolved 3-bits multiplier : 29 gates with 6 levels, using {and, and with one input inverted, xor}

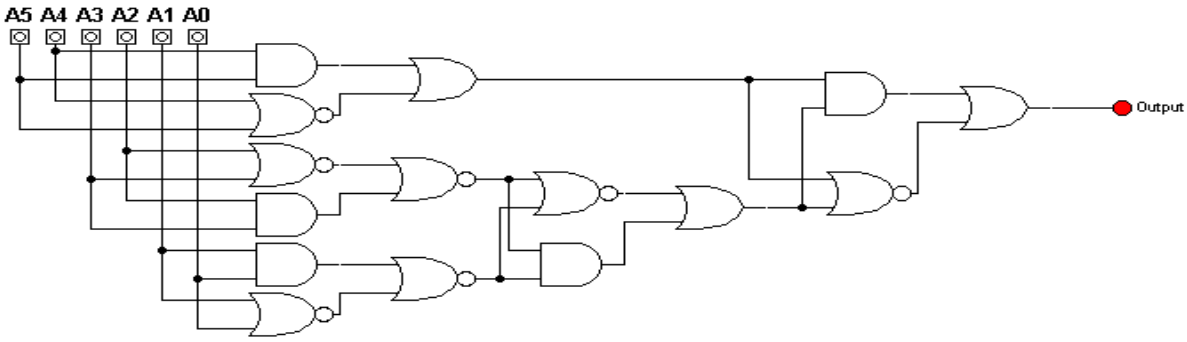

Figure 8 : Evolved 6_bits parity problem : 15 gates using {and, or, nor}

References

[1] Bellman, R., *Dynamic Programming*, Princeton University Press, New Jersey, 1957.

[2] Coello, C. A., Christiansen, a. D. and Aguire, A. H., Using Genetic Algorithms to Design Combinational Logic Circuits, *Intelligent Engineering through Artificial Neural Networks*. Vol. 6, pp 391-396, 1996

[3] Gathercole, C. and Ross, P. ,Tackling the Boolean even N parity problem with genetic programming and limited-error fitness, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), pp 119–127, Stanford University, CA, USA: Morgan Kaufmann. 1997

[4] Koza, J.R., *Genetic Programming. On the Programming of Computers by means of Natural*

Selection, MIT Press, 1992.

[5] Louis, S.J. And Rawlins, G. J., Designer Genetic Algorithm : Genetic Algorithms in Structure Design. *Proc. of the fourth Int. Conference on Genetic Algorithms*, pp. 53-60, 1991.

[6] Miller, J.F. and Thomson, P., Aspects of Digital Evolution: Evolvability and architecture. *Proc. of the Parallel Problem Solving from Nature V*, pp 927-936, Springer, 1998.

[7] Miller, J.F., and Thomson, P., Cartesian Genetic Programming. *Proc. of the 3rd International Conference on Genetic Programming* (EuroGP2000), *LNCS* 1082, SpringerVerlag, Berlin, pp. 15-17, 2000.

[8] Miller, J.F., Job. D. and Vassilev, V.K., Principles in the Evolutionary Design of digitals circuits-Part I, *Genetic Programming and Evolvable Machines*, **1**, (1), pp 7-35, Kluwer Academic Publishers, 2000.

[9] Oltean Mihai, Multi Expression Programming, Technical Report, Babes-Bolyai Univ, Romania, 2006.

[10] Oltean, M., Grosan, C., A Comparison of Several Linear Genetic Programming Techniques, *Complex-Systems*, **14**, (4), pp. 282-311, 2003.

[11] Oltean, M., Grosan, C., Evolving Digital Circuits using Multi Expression Programming. *Proc. NASA/DoD Conference on Evolvable Hardware*, 24-26 June, Seattle, Edited by R. Zebulum (et. al), pages 87-90, IEEE Press, NJ, 2004.

[12] Oltean, M., Grosan, C., Evolving Evolutionary Algorithms using Multi Expression Programming, *The 7th European Conference on Artificial Life*, W. Banzhaf (et. al), (Editors), *LNCS* 2801, pp. 651-658, Springer-Verlag, Berlin, 2003.

[13] Oltean, M., Solving Even-Parity Problems using Multi Expression Programming, *Proceedings of the 5th International Workshop on Frontiers in Evolutionary Algorithms, The 7th Joint Conference on Information Sciences*, September 26-30, 2003, Research Triangle Park, North Carolina, Edited by Ken Chen (et. al), pp. 315-318, 2003.

[14] Walker, J. A. and Miller, J. F., Investigating the Performance of Module Acquisition in Cartesian Genetic Programming. *Proc. of the 2005 Genetic and Evolutionary Computation Conference*, Volume 2, Pages 1649-1656, Washington DC, USA, 25-29 June 2005. ACM Press, 2005

[15] Zeblum, R. S., Pacheco, M. A. and Vellasco, M. M., *Evolutionary Electronics: Automatic Design of Electronic Circuits and Systems by Genetic Algorithms*, CRC Press, 2001.