Parallel Grammatical Evolution for Circuit Optimization

Oldřich Kratochvíl , Pavel Ošmera, Ondřej Popelka

Abstract—**This paper describes a Parallel Grammatical Evolution (PGE) that can evolve complete circuits using a variable length of linear genome to govern the mapping of a Backus Naur Form grammar definition. In order to increase the efficiency of Grammatical Evolution (GE) the influence of backward processing and an influence of several fitness functions were tested. PGE with backward processing can also take advantage of progressive crossover and mutation operators. The algorithm is internally parallel and consists of three different interconnected populations. A new non-tree structure of GE was tested.**

Index Terms—**grammatical evolution, circuit optimization, parallel evolution**.

I. INTRODUCTION

Grammatical Evolution (GE) [1] can be considered as a form of grammar-based genetic programming (GP). In particular, Koza's genetic programming has enjoyed considerable popularity and widespread use. Unlike a Koza-style approach, there is no distinction made at this stage between what he describes as function (operator in this case) and terminals (variables). Koza originally employed Lisp as his target language. This distinction is more of an implementation detail than a design issue. Grammatical evolution can be used to generate programmes in any language, using Backus Naur Form (BNF). BNF grammars consist of terminals, which are items that can appear in the language, i.e. +, -, sin, log etc. and non-terminal, which can be expanded into one or more terminals and non-terminals. A non-terminal symbol is any symbol that can be rewritten to another string, and conversely a terminal symbol is one that cannot be rewritten.

The major strength of GE with respect to GP is its ability to generate multi-line functions in any language. Rather than representing the programs as parse tree, as in GP, a linear genome is used [1]-[3]. A genotype-phenotype mapping is employed such that each individual's variable length byte strings, contains the information to select production rules from a BNF grammar.

Manuscript received July 14, 2009. This work was supported in part by the Czech Ministry of Education under Grant No: MSM 00216305529 Intelligent Systems in Automation and by the grant agency GA ČR No: 102/09/1668: Control Algorithm Design by Means of Evolutionary Approach.

O. Kratochvíl is with the European Polytechnical Institute Kunovice, (e-mail: kratochvil@edukomplex.cz).
P. Ošmera is with the Brno University of Technology, Faculty of Mechanical Engineering, Brno, Czech Republic
(e-mail: osmera @fme.vutbr.cz).

O. Popelka is with the Institute Department of informatics, Mendel University, of Agriculture and Forestry, FBE, Brno, Czech Republic, (xpopelka@node.mendelu.cz).

The grammar allows the generation of programs, in an arbitrary language that are guaranteed to be syntactically correct. The user can tailor the grammar to produce solutions that are purely syntactically constrained, or they may incorporate domain knowledge by biasing the grammar to produce very specific form of sentences.

Because, GE mapping technique employs a BNF definition, the system is language independent, and theoretically can generate arbitrarily complex functions. There is quite an unusual approach in GEs, as it is possible for certain genes to be used two or more times if the wrapping operator is used. BNF is a notation that represents a language in the form of production rules. It is possible to generate programs using the Grammatical Swarm Optimization (GSO) technique [2] with a performance, which is similar to the GE. The relative simplicity, the small population sizes, and the complete absence of a crossover operator synonymous with program evolution in GP or GE are main advantages of GSO. In the grammatical evolution GE the different approach to the genotype and phenotype is used. GE evolves a sequence of rule numbers that are translated, using a predetermined grammar set, into a phenotypic tree.

Our approach uses a parallel structure of GE (PGE). A population is divided into several sub-populations that are arranged in the hierarchical structure [4]. Every sub-population has two separate parts: a "male" group and a "female" group. Every group uses quite a different type of selection. In the first group a classical type of GA selection is used. Only different individuals can be added to the second group. This strategy was inspired by harem system in Nature that solves problem of an adaptation of a complex organisms to microorganisms [5]. The biologically inspired strategy increases an inner adaptation of PGE. This analogy would lead us one step further, namely, to the belief that the combination of GE with 2 different selections that are simultaneously used can improve an adaptive behaviour of GE [5], [6], [7]. On the principle of two selections we can create a parallel GE with a hierarchical structure.

II. PARALLEL GRAMMATICAL EVOLUTION

The PGE is based on the grammatical evolution GE [1], where BNF grammars consist of terminals and non-terminals. Terminals are items, which can appear in the language. Non-terminals can be expanded into one or more terminals and non-terminals. Grammar is represented by the tuple {N,T,P,S}, where N is the set of non-terminals, T the set of terminals, P a set of production rules which map the elements of N to T, and S is a start symbol which is a member of N. For example, the BNF is used for our problem below:

N = {expr, fnc}
T = {sin, cos, +, -, /, *, X, 1, 2, 3, 4, 5, 6, 7, 8, 9}

S = <expr>
and P can be represented as 4 production rules:

1. <expr> := <fnc><expr>
   <fnc><expr><expr>
   <fnc><num><expr>
   <var>
2. <fnc> := sin
   cos
   +
   *
   -
   U-
3. <var> := X
4. <num> := 0,1,2,3,4,5,6,7,8,9

The production rules and the number of choices associated with them are in Table 1. The symbol U- denotes an unary minus operation.

Table 1: The number of available choices for every production rule

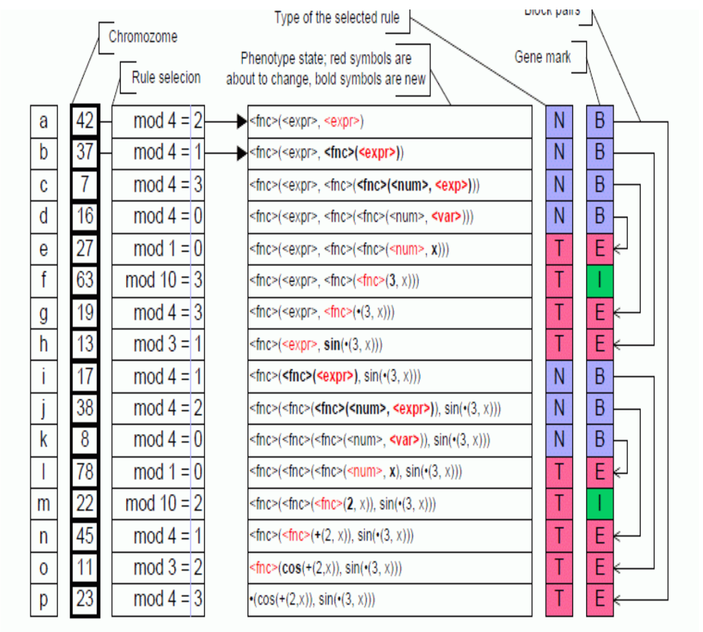
rule no	choices
1	4
2	6
3	1
4	10

There are notable differences when compared with [1]. We don't use two elements <pre_op> and <op>, but only one element <fnc> for all functions with n arguments. There are not rules for parentheses; they are substituted by a tree representation of the function. The element <num> and the rule <fnc><num><expr> were added to cover generating numbers. The rule <fnc><num><expr> is derived from the rule <fnc><expr><expr>. Using this approach we can generate the expressions more easily. For example when one argument is a number, then +(4,x) can be produced, which is equivalent to (4 + x) in an infix notation. The same result can be received if one of <expr> in the rule <fnc><expr><expr> is substituted with <var> and then with a number, but it would need more genes.
There are not any rules with parentheses because all information is included in the tree representation of an individual. Parentheses are automatically added during the creation of the text output.
If in the GE solution is not restricted anyhow, the search space is too large and can have infinite number of solutions. For example the function cos(2x), can be expressed as cos(x+x); cos(x+x+1-1); cos(x+x+x-x); cos(x+x+0+0+0...) etc. It is desired to limit the number of elements in the expression and the number of repetitions of the same terminals and non-terminals.

III. BACKWARD PROCESSING OF THE GE

The chromosome is represented by a set of integers filled with random values in the initial population. Gene values are used during chromosome translation to decide which terminal or nonterminal to pick from the set. When selecting a production rule there are four possibilities, we use gene_value mod 4 to select a rule. However the list of variables has only one member (variable X) and gene_value mod 1 always returns 0. A gene is always read; no matter if a decision is to be made, this approach makes some genes in the chromosome somehow redundant. Values of such genes can be randomly created, but genes must be present.
The Fig. 1 shows the genotype-phenotype translation scheme. The individual body is shown as a linear structure, but in fact it is stored as a one-way tree (child objects have no links to parent objects). In the diagram we use abbreviated notations for nonterminal symbols: f - <fnc>, e - <expr>, n - <num>, v - <var>.

IV. PROCESSING THE GRAMMAR


Type of the selected rule
Block pairs
Chromozome
Gene mark
Phenotype state; red symbols are about to change, bold symbols are new
Rule selecion

a	42	mod 4 = 2	<fnc>(<expr>, <expr>)	N	B
b	37	mod 4 = 1	<fnc>(<expr>, <fnc>(<expr>))	N	B
c	7	mod 4 = 3	<fnc>(<expr>, <fnc>(<fnc>(<num>, <exp>)))	N	B
d	16	mod 4 = 0	<fnc>(<expr>, <fnc>(<fnc>(<num>, <var>)))	N	B
e	27	mod 1 = 0	<fnc>(<expr>, <fnc>(<fnc>(<num>, x)))	T	E
f	63	mod 10 = 3	<fnc>(<expr>, <fnc>(<fnc>(3, x)))	T	I
g	19	mod 4 = 3	<fnc>(<expr>, <fnc>(•(3, x)))	T	E
h	13	mod 3 = 1	<fnc>(<expr>, sin(•(3, x)))	T	E
i	17	mod 4 = 1	<fnc>(<fnc>(<expr>), sin(•(3, x)))	N	B
j	38	mod 4 = 2	<fnc>(<fnc>(<fnc>(<num>, <expr>)), sin(•(3, x)))	N	B
k	8	mod 4 = 0	<fnc>(<fnc>(<fnc>(<num>, <var>)), sin(•(3, x)))	N	B
l	78	mod 1 = 0	<fnc>(<fnc>(<fnc>(<num>, x), sin(•(3, x)))	T	E
m	22	mod 10 = 2	<fnc>(<fnc>(<fnc>(2, x)), sin(•(3, x)))	T	I
n	45	mod 4 = 1	<fnc>(<fnc>(+(2, x)), sin(•(3, x)))	T	E
o	11	mod 3 = 2	<fnc>(cos(+(2,x)), sin(•(3, x)))	T	E
p	23	mod 4 = 3	•(cos(+(2,x)), sin(•(3, x)))	T	E

Fig. 1 Relations between genotype and phenotype in the GE with backward processing [6]

The processing of the production rules is done backwards – from the end to the beginning of the rule (Fig. 2). Then production rule <fnc><expr1><expr2> is processed as <expr2><expr1><fnc>. We use <expr1> and <expr2> at this point to denote which expression will be the first argument of <fnc>.

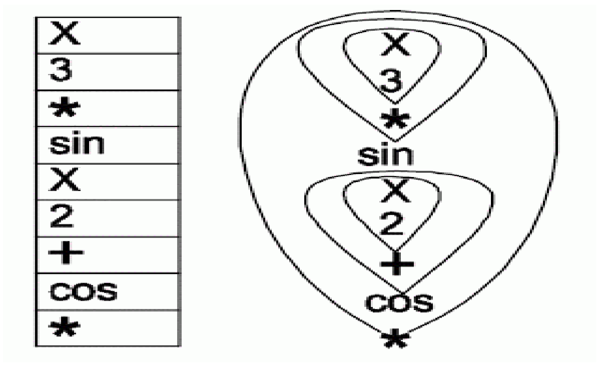


Fig. 2 Proposed backward notation of a function tree structure

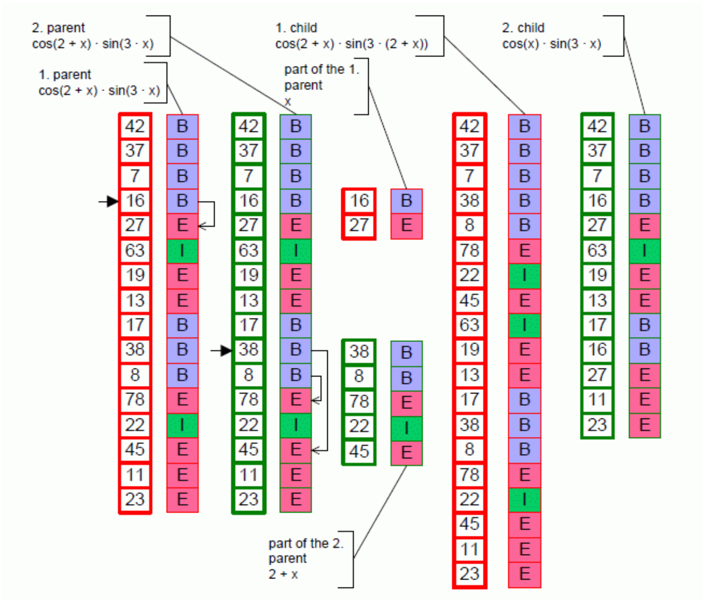


Fig. 3. Crossover in GE with backward processing [6]

The main difference between <fnc> and <expr> nonterminals is in the number of real objects they produce in the individual's body. Nonterminal <fnc> always generates one and only one terminal; on the contrary <expr> generates an unknown number of nonterminal and terminal symbols. If the phenotype is represented as a tree structure then a product of the <fnc> nonterminal is the parent object for handling all objects generated by <expr> nonterminals contained in the same rule. Therefore the rule <fnc><expr1><expr2> can be represented as a tree (Fig. 4).

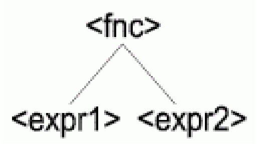


Fig. 4 Production rule shown as a tree

To select a production rule (selection of a tree structure) only one gene is needed. To process the selected rule a number of n genes are needed and finally to select a specific nonterminal symbol again one gene is needed. If the processing is done backwards the first processed terminals are leafs of the tree and the last processed terminal in a rule is the root of a subtree. The very last terminal is the root of the whole tree. Note that in a forward processing (<fnc><expr1><expr2>) the first processed gene codes the rule, the second gene codes the root of the subtree and the last are leafs.

When using the forward processing and coding of the rules described in [1] it's not possible to easily recover the tree structure from genotype. This is caused with <expr> nonterminals using an unknown number of successive genes. The last processed terminal being just a leaf of the tree. The proposed backward processing is shown in Fig. 1.

4.1 Phenotype to genotype projection

Using the proposed backward processing system the translation to a phenotype subtree has a certain scheme. It begins with a production rule (selecting the type of the subtree) and ends with the root of the subtree (in our case with a function). In the genotype this means that one gene used to select a production rule is followed by n genes with different contexts which are followed by one gene used to translate <fnc>. Therefore a gene coding a production rule forms a pair with a gene coding terminal symbol for <fnc> (root of the rule). Those genes can be marked when processing the individual. This is an example of a simple marking system:

BB – Begin block (a gene coding a production rule)
IB – Inside block
EB – End block (a gene coding a root of a subtree)

The EB and BB marks are pair marks and in the chromosome they define a block (Fig. 1G). Such blocks can be nested but they don't overlap (the same way as parentheses). The IB mark is not a pair mark, but it is always contained in a block (IB marks are presently generated by <num> nonterminals). Given a BB gene a corresponding EB gene can be found using a simple LIFO method.

A block of chromosome enclosed in a BB-EB gene pair then codes a subtree of the phenotype. Such block is fully autonomous and can be exchanged with any other block or it can serve as completely new individual.

Only BB genes code the tree of individual's body, while EB and IB genes code the terminal symbols in the resulting phenotype. The BB genes code the structure of the individual, changing their values can cause change of the applied production rule. Therefore change (e.g. by mutation) in the value of a structural gene may trigger change of context of many, or all following genes.

This simple marking system introduces a phenotype feedback to phenotype; however it doesn't affect the universality of the algorithm. It's not dependent on the used terminal or nonterminal symbols; it only requires the result to be a tree structure. Using this system it's possible to introduce a progressive crossover and mutation.

4.2 Crossover

When using grammatical evolution the resulting phenotype coded by one gene depends on the value of the gene and on its context. If a chromosome is crossed at random point, it is very probable that the context of the genes in second part will change. This way crossover causes destruction of the phenotype, because the newly added parts code different phenotype than in the original individual.

This behaviour can be eliminated using a block marking system. Crossover is then performed as an exchange of blocks. The crossover is made always in an even number of genes, where the odd gene must be BB gene and even must be EB gene. Starting BB gene is presently chosen randomly; the first gene is excluded because it encapsulates (together with the last used gene) the whole individual.

The operation takes two parent chromosomes and the result is always two child chromosomes. It is also possible to combine the same individuals, while the resulting child chromosomes can be entirely different.

Given the parents:
1) cos(x + 2) + sin(x * 3)
2) cos(x + 2) + sin(x * 3)
The operation can produce children:
3) cos(sin(x * 3) + 2) + sin(x * 3)
4) cos(x + 2) + x

This crossover method works similar to direct combining of phenotype trees, however this method works purely on the chromosome. Therefore phenotype and genotype are still separated. The result is a chromosome, which will generate an individual with a structure combined from its parents. This way we receive the encoding of an individual without backward analysis of his phenotype. To perform a crossover the phenotype has to be evaluated (to mark the genes), but it is neither used nor know in the crossover operation (also it doesn't have to exist).

4.3 Mutation

Mutation can be divided into mutation of structural (BB) genes and mutation of other genes. Mutation of one structural gene can affect other genes by changing their context therefore structural mutation amount should be very low. On the other hand the amount of mutation of other genes can be set very high and it can speed up searching an approximate solution.

Given an individual:
sin(2 + x) + cos(3 * x)
and using only mutation of non-structural genes, it is possible to get:
cos(5 – x) * sin(1 * x)

Therefore the structure doesn't change, but we can get a lot of new combinations of terminal symbols. The divided mutation allows using the benefits of high mutation while eliminating the risk of damaging the structure of an individual.

4.4 Population model

The system uses three populations forming a simple tree structure (Fig. 5). There is a Master population and two slave populations, which simulate different genders. The links among the populations lead only one way - from bottom to top.

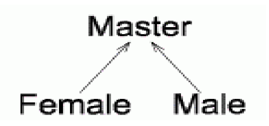


Fig. 5 The population model

4.5 Female population

When a new individual is to be inserted in a population a check is preformed whether it should be inserted. If a same or similar individual already exists in the population then the new individual is not inserted. In a female population every genotype and phenotype occurs only once. The population maintains a very high diversity; therefore the mutation operation is not applied to this population. Removing the individuals is based on two criterions. The first criterion is the age of an individual - length of stay in the population. The second criterion is the fitness of an individual. Using the second criterion a maximum population size is maintained. Parents are chosen using the tournament system selection.

4.5 Fitness function

Around the searched function there is defined an equidistant area of a given size. Fitness of an individual's phenotype is computed as the number of points inside this area divided by the number of all checked points (a value in <0,1>). This fitness function forms a strong selection pressure; therefore the system finds an approximate solution very quickly.

4.6 Logical function XOR as test function

Input values are two integer numbers a and b; a, b 2< 0, 1 >. Output number c is the value of logical function XOR. Training data is a set of triples (a, b, c):

P = {(0, 0, 0); (0, 1, 1); (1, 0, 1); (1, 1,0)}.

Thus the training set represents the truth table of the XOR function. The function can be expressed using _, ^, ¬ functions:

a + b = (a ^ ¬b) _ (¬a ^ b) = (a _ b) ^ (¬a _ ¬b) = (a _ b) ^ ¬(a ^ b)

The grammar was simplified so that it does not contain conditional statement and numeric constants, on the other hand three new terminals were added to generate functions _, ^, ¬. Thus the grammar generates representations of the XOR functions using other logical functions.

```
function xxor($a,$b) {
$result = "no_value";
$result = ($result) | (((~$b & ($a & ($a & ~$b))) & $a) |
(~$a & $b));
return $result;
```

Number of generations: 53

V. CIRCUIT OPTIMIZATION

The objective is to generate the structure of a combinatorial logic circuit performing as full binary adder. Binary adder can be represented with the following equations:

$$s_i = y_i \oplus x_i \oplus c_{i-1} \tag{1}$$

$$c_i = x_i \cdot y_i + x_i \cdot c_{i-1} + y_i \cdot c_{i-1} \tag{2}$$

The circuit has three inputs x_i, y_i, c_{i-1} and two output variables s_i, c_i, where s_i is the actual sum result, c_i is carry bit, x_i, y_i are the actual binary inputs and c_{i-1} is carry bit from previous addition. The truth table of binary adder has 16 output values, where equations (1) and (2) each define eight of them.

5.1 Methods

The parallel grammatical evolution with backward processing was used to solve the problem. The core of the method is a genetic algorithm extended with several supporting algorithms. The main extension added to the genetic algorithm is a translation layer inserted between the chromosome and the actual solution which is formed by a processor of context-free grammar. The main advantage of such extension is the ability to create generic tree structures and retrieve them in reusable format. Grammatical evolution with backward processing can also take advantage of progressive crossover and mutation operators. The system is internally parallel as it consists of three different interconnected populations.

<fnc> ::= AND |	<gate> ::= <fnc> <gate> <gate> |
NAND |	<fnc> <gate> |
OR |	<input>
NOR |	
XOR |	<input> ::= x_i |
NOT	y_i |
<dummy> ::= <gate> <gate>	c_{i-1}

Tab. 1 Production rules

The production rules that are shown in Tab.1 are the most generic ones, allowing any syntactically correct solution to be generated. This can however be adjusted in case we would like to use specific sets of gates. For example we can define a rule so that first input of a gate is connected to OR gate and the other is connected to AND gate.

5.2 Problems specific to combinatorial logic

Although grammatical evolution was successfully used to solve many different problems there are several challenging aspects when applied to generating logic circuits. The first issue is that there is more than one output variable, that is the generated structure is supposed to be parallel. The number of output variables alone is not a big issue, since we can use a dummy function to encapsulate results of both output variables into one vector output. This defers the problem to the fitness evaluation module. However this does not solve the core of the problem of generating parallel structure. As mentioned above; a rewriting grammar uses a set of non-terminal symbols, which are being rewritten into terminal symbols. Terminal symbols represent the actual blocks of a logic circuit. All non-terminals need to be translated into terminals before the individual is usable; hence once a non-terminal is translated it has to be removed from the individuals' body. Terminals in the body are no further processed.

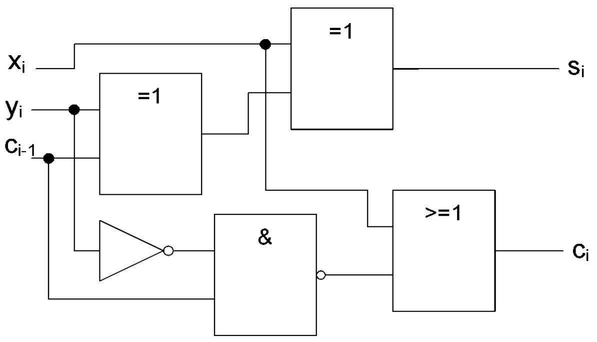


Fig. 6 Example of a found solution

These principles mean that it is only possible to generate tree structures. Logic circuits cannot however be represented as a simple tree. A signal from s_i output can use the same gates as the output signal of c_i. At this point we have chosen to accept this limitation and test overall performance of grammatical evolution applied to this problem. This means that the generated solutions can never reach the optimal number of building blocks – since it is not possible to reuse existing blocks. For the binary adder circuit it should be possible to reach the optimal time-delay, although with a more complicated circuit. Also it is necessary to note that many of possible optimal solutions are ruled out simply because reusing of gates is prohibited; this makes finding the optimal time-delay solution more difficult. However we are confident that this limitation of the algorithm can be overcome and it would be possible to generate truly parallel structures.

5.3 Fitness function

There are several options how to compute fitness and compare different hypotheses produced by genetic algorithm. As the main criterion we choose the number of matches against the input truth-table of combinations. However this criterion alone is insufficient, since there are only 16 output values, there are also only 16 values of fitness. The number of possible solutions either correct or incorrect is only limited by arbitrary size of the search space, which can be adjusted by the length of the chromosome (in the experiments set so that the effective maximum of elements in the structure is approximately 70).

A fitness value for n-th individual is then defined as:

$$F_n = \left(\sum_{j=1}^{8} MS_j^n, \sum_{j=1}^{8} MC_j^n, CS^n, CC^n\right) \tag{3}$$

where *MS* = 1 if the j-th output value of variable s_i matches the desired value in truth table and similarly *MC* = 1 if the value of variable c_i is matched. *C* is the count of nodes in the generated structure. *CS* is the count of nodes in the tree branch responsible for computing output of variable s_i, and *CS* is the complexity of the c_i branch

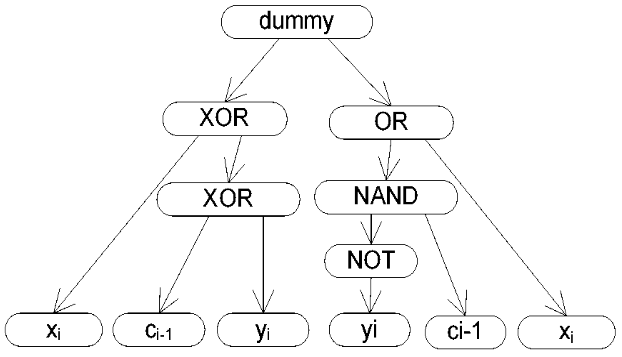


Fig. 7 Tree representation of the solution on figure 3

It is important to note that the arbitrary chromosome size does not limit the solution size reliably, using the crossover operator the algorithm can bypass the limitation and generate solutions with up to approximately 1000 elements. Therefore the fitness consisting of only 16 values is inappropriate since it does not value lower complexity of the solution. The simplest solution – to compute fitness value as a weighted sum of matches and complexity of an individual didn't fit our needs and led to premature convergence. This problem was solved replacing scalar fitness value by a vector of fitness. It allows evaluating individuals with a finer granularity then number of matched values alone. To compare the vector fitness values a hierarchical set of rules was used.

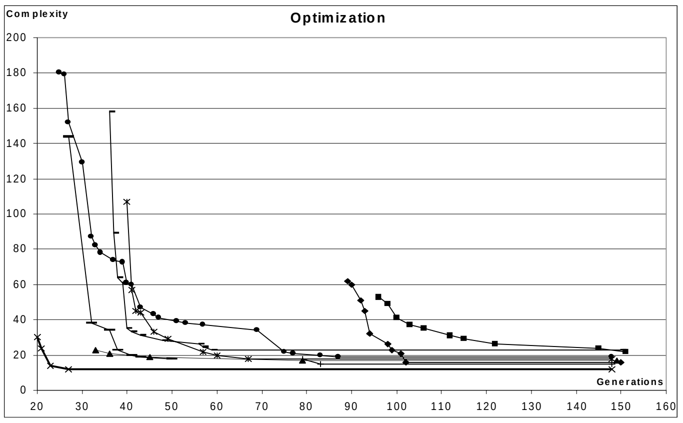


Fig. 8 Two step optimization process

Once the algorithm is adapted to vector fitness a vast number of definitions of fitness arise. As the qualitative characteristics the number of matches and solution complexity were chosen. Another possible choice would be the maximum time-delay of the circuit. In our case where the output variable paths are not interconnected the time-delay is correlated with complexity of each path and thus makes no difference to convergence of the algorithm. The choice of complexity above time-delay was therefore driven only by implementation. Complexity of a solution is defined as number of terminals in the string representation of the structure, this is slightly higher than the actual number of gates. Figure 7 shows the tree representation of a solution shown on Fig.6. The complexity is defined as number of nodes in the tree. The first approach was to simply use both the sum of matches and complexity of both tree root branches.

VI. CONCLUSION

PGE has proved successful for circuit optimization. Parallel GEs with hierarchical structure can increase the efficiency and robustness of systems, and thus they can track better optimal parameters in a changing environment. From the experimental session it can be concluded that modified standard GEs with only two sub-populations can create PGE much better than classical versions of GEs.

The parallel grammatical evolution can be used for the automatic generation of circuit structures. We are far from supposing that all difficulties are removed but first results with PGEs are very promising.

Although we are at early stages of experiments it seems that it is possible to use parallel grammatical evolution with backward processing to generate combinatorial logic circuits. The grammatical algorithm can be outperformed with algorithms, which are designed specifically for this purpose.

ACKNOWLEDGMENTS

This work has been supported by Czech Ministry of Education No: MSM 00216305529 Intelligent Systems in Automation and GA ČR No: 102/09/1668: Control Algorithm Design by Means of Evolutionary Approach

REFERENCES

[1] O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language Kluwer Academic Publishers 2003.
[2] O'Neill, M., Brabazon, A., Adley C.: The Automatic Generation of Programs for Classification Problems with Grammatical Swarm, Proceedings of CEC 2004, Portland, Oregon (2004) 104 – 110
[3] Piaseczny, W., Suzuki. H., Sawai, H.: Chemical Genetic Programming – Evolution of Amino Acid Rewriting Rules Used for Genotype-Phenotype Translation, Proceedings of CEC 2004, Portland, Oregon (2004) 1639 - 1646.
[4] Ošmera, P., Šimoník, I, Roupec, J.: Multilevel distributed genetic algorithms. In Proceedings of the International Conference IEE/IEEE on Genetic Algorithms, Sheffield (1995) 505–510.
[5] Li Z., Halang W. A., Chen G.: Integration of Fuzzy Logic and Chaos Theory; paragraph: Osmera P.: Evolution of Complexity, Springer, 2006 (ISBN: 3-540-26899-5) 527 – 578.
[6] Ošmera, P.,Panáček, T., Pivoňka, P.: Parallel Grammatical Evolution, Proceedings of WCECS 2007, San Francisco,24-26 October 2007, USA, 897-902
[7] RUKOVANSKÝ, I. Evolution of Complex Systems. 8th Joint Conference on Information Sciences. Salt Lake City, Utah, USA. July 21-25, 2005.