GRAMMATICAL EVOLUTION FOR HARDWARE OPTIMIZATION

Pavel Ošmera[1] , Ondřej Popelka[2],

[1]Institute of Automation and Computer Science, Brno University of Technology, Faculty of Mechanical Engineering, Brno, Czech Republic (osmera@fme.vutbr.cz).
[2]Institute Department of informatics, Mendel University,of Agriculture and Foresty, FBE, Brno, Czech Republic, (xpopelka@node.mendelu.cz).

Abstract. This paper describes a Parallel Grammatical Evolution (PGE) that can evolve complete circuits using a variable length linear genome to govern the mapping of a Backus Naur Form grammar definition. To increase the efficiency of Grammatical Evolution (GE) the influence of backward processing was tested and an influence of several fitness functions. 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.

Keywords: circuit optimization, parallel grammatical evolution, backward processing, parallel evolution

1 Introduction

Grammatical Evolution (GE) [1] can be considered 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 programs 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 representing 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. 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 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. Grammatical evolution was one of the first approaches to distinguish between the genotype and phenotype. 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 [6]. 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. To the second group only different individuals can be added. This strategy was inspired by Nature that solves problem of adaptation complex organisms to microorganisms. 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 behavior of GE [4] - [8]. On the principle of two selections we can create a parallel GE with a hierarchical structure.

2 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 figure Fig. 1 shows the genotype-phenotype translation scheme [6]. Body of the individual 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>.

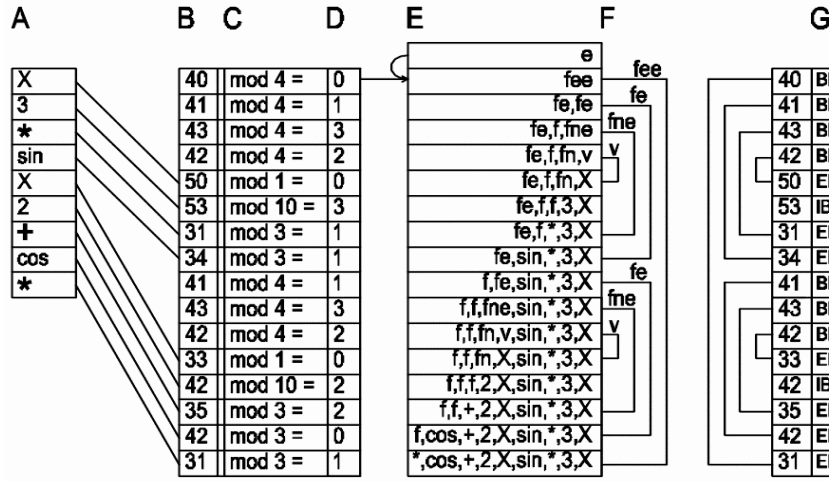

Fig. 1. Relations between genotype (column B) and phenotype (column A)

The column description in Fig. 1:

A. Objects of the individual's body (resulting trigonometric function),

B. Genes used to translate the chromosome into the phenotype,
C. Modulo operation, divisor is the number of possible choices determined by the gene context,
D. Result of the modulo operation,
E. State of the individual's body after processing a gene on the corresponding line,
F. Blocks in the chromosome and corresponding production rules,
G. Block marks added to the chromosome.

Since operation modulo takes two operands, the resulting number is influenced by gene value and by gene context (Fig. 1C = see Fig. 1 column C). Gene context is the number of choices, determined by the currently used list (rules, functions, variables). Therefore genes with same values might give different results of modulo operation depending on what object they code. On the other hand one terminal symbol can be coded by many different gene values as long as the result of modulo operation is the same (31 mod 3) = (34 mod 3) = 1. In the example (Fig. 1A) given the variables set has only one member X. Therefore, modulo divider is always 1 and the result is always 0, a gene which codes a variable is redundant in that context (Fig. 1D). If the system runs out of genes during phenotype-genotype translation then the chromosome is wrapped and genes at the beginning are reused.

The processing of the production rules is done backwards – from the end of the rule to the beginning. E.g. 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>.

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.

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 then 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 grammar 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. 1E.

3 Hardware 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:

$$s_i = y_i \oplus x_i \oplus c_{i-1} \quad (1)$$

$$c_i = x_i \cdot y_i + x_i \cdot c_{i-1} + y_i \cdot c_{i-1} \quad (2)$$

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.

3.1 Methods

To solve the problem parallel grammatical evolution with backward processing was used. 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.

3.2 Grammatical evolution

Same as in genetic algorithms the process is started by generating random chromosomes which are represented by vector of integers. From the initial set of chromosomes the initial set of individuals is created using the abstraction layer. In grammatical evolution genes in the chromosome do not represent elements or parameters of the solution; they represent rules necessary to generate the solution. The grammar used is a subset of context-free rewriting grammar and is defined as follows:

$$\Pi = \{\text{fnc, gate, input, dummy}\} \quad (3)$$

$$\Sigma = \{\text{AND, OR, NOT, NAND, NOR, XOR}\} \quad (4)$$

$$S = \text{dummy} \quad (5)$$

Where Π is set of non-terminals, Σ is set of terminals, S is initial non-terminal and P is table of production rules (Fig. 2.). Each non-terminal is translated using a set of rules until there are no non-terminals left. The rewriting process is started with the initial non-terminal (S). This is a placeholder function called dummy, which does nothing but encapsulates the two outputs of the structure (s_i, c_i). The purpose of this is to simplify the implementation details of the algorithm and to adhere to the definition of a grammar. The initial non-terminal is therefore rewritten into two <gate> non-terminals, each of them representing one output variable. Both variables have to be further processed using the appropriate rules. Whenever there is more then one applicable rule, it is necessary to choose only one. Using modulo function the values of genes are used to choose one possible rule which is applied to the processed non-terminal. This process continues until there are no more non-terminals left. The result of this process is a string representing the actual solution. A sample solution is a string representing the binary adder function, similar to equations (1) and (2). When the translation process is finished it is possible to evaluate fitness of each individual and continue with the population cycle of generic genetic algorithm.

The production rules shown on figure 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 one input of a gate is connected to OR gate and the other is connected to AND gate.

<fnc> ::= AND |
NAND |
OR |
NOR |
XOR |
NOT

<gate> ::= <fnc> <gate> <gate> |
<fnc> <gate> |
<input>

<input> ::= x_i |
y_i |
c_{i-1}

<dummy> ::= <gate> <gate>

Fig. 2. Production rules

3.3 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 then 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 (see equation (5)) 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.

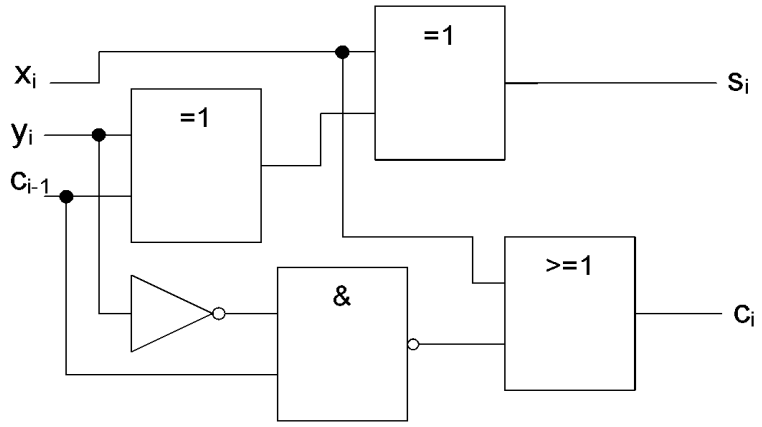

Fig. 3. Example of a found solution

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 overcame and it would be possible to generate truly parallel structures.

3.4 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). It is important to note that the arbitrary chromosome

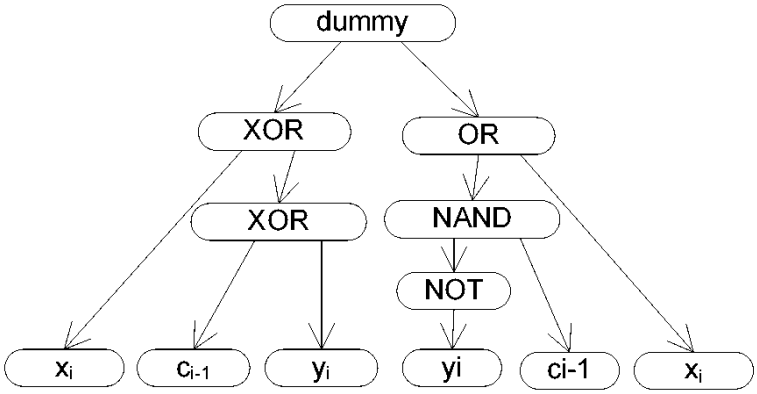

Fig. 4. Tree representation of the solution on figure 3

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 by replacing scalar fitness value with a vector of fitness which 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.

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 variables 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 then the actual number of gates. Figure 5 shows the tree representation of a solution, the complexity is defined as number of nodes in the tree.

4 Results

The first approach was to simply use both the sum of matches and complexity of both tree root branches (Fig. 5). A fitness value for n-th individual is then defined as:

$$F_n = \left(\sum_{j=1}^{8} MS_j^n + \sum_{j=1}^{8} MC_j^n, C^n \right) \qquad (6)$$

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.

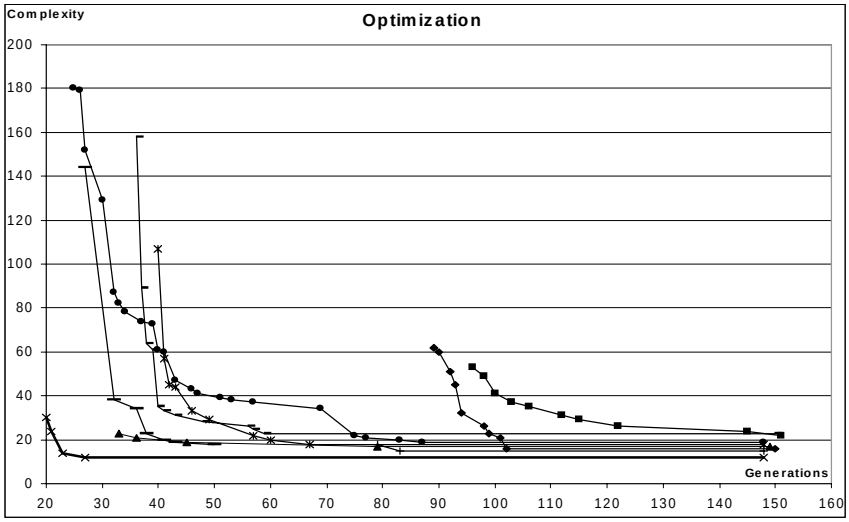

Fig. 5. Optimization part of the process

To avoid premature convergence the complexity was compared only when all output values matched the truth table. Also the number of matches has to be maximized whereas the complexity of the solution is being minimized. Therefore the comparison routine can be written as:

```
Compare(F[n][0], F[n+1][0])

if Equals and F[n][0] = Max then

        Compare(F[n+1][1], F[n][1])
```

The algorithm is based on a $\lambda + 1$ evolution strategy, which means that newly generated individuals (either with mutation or crossover) are added to the existing population. Maximum lifetime of an individual is approximately 20 generations. The whole searching process can be split into two parts – matching and optimizing. The objective of the former is to find an acceptable solution which yields correct results. The later is focused on optimizing this solution to lowest possible number of elements. When the computation is started only matching is in progress since the second element of a fitness value (F[n][1]) is being ignored. As soon as a fully matching solution is found it is started to be optimized. It is important to note that both the matching and optimization run parallel to each other. A graph of the optimization process is shown on figure 4. The starting point of the curve is the first solution which matched all output values; the curve then shows the complexity of the solution as it is being optimized. Some of the runs go through rapid changes in complexity; these are caused by replacing the best individual with an entirely new solution. We consider this as a negative, since there is no telling when or whether this will happen.

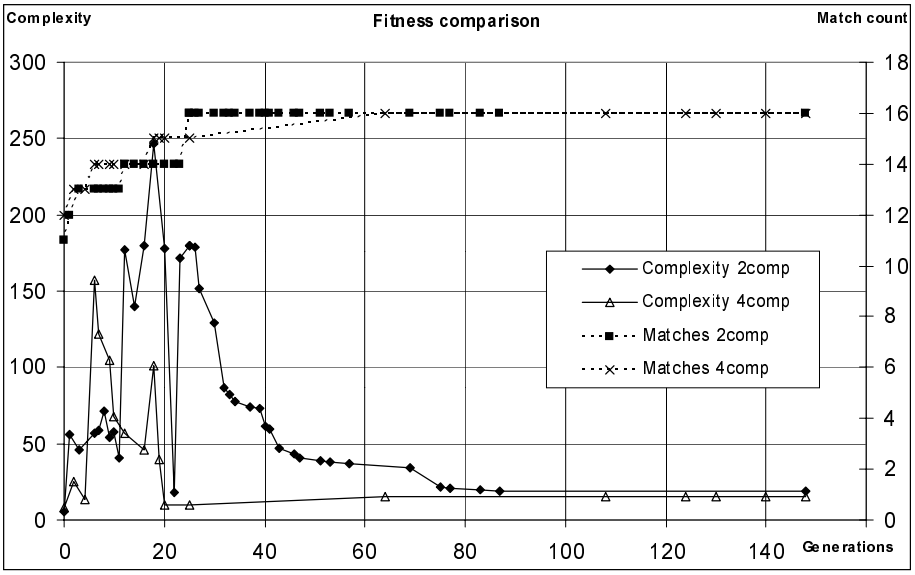

Fig. 6. Comparison of 2 runs with different fitness function

At this point it is possible to take advantage of the fact, that the two output variables are computed using completely separate logic gates. It is possible to further split the fitness value:

$$F_n = \left(\sum_{j=1}^{8} MS_j^n, \sum_{j=1}^{8} MC_j^n, CS^n, CC^n\right) \quad (7)$$

Where *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. A single fitness value is now built-up from four components. There are a vast number of options on how to compare such fitness value. We used the following rules:

- if both individuals have both branches complete then compare by sum of complexity
- if one individual has more complete branches then it is better
- if both individuals have the same branch complete then compare by number of matches of the other branch
- if both individuals have the same branch complete and same number of matches then reverse compare by complexity of that branch
- if both individuals have different branch complete then compare by number of matches

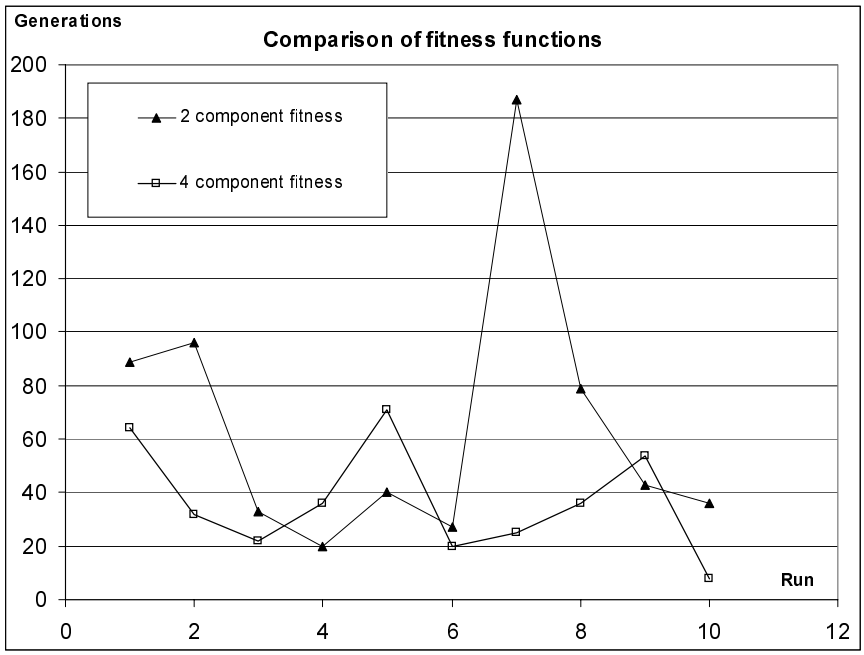

Fig. 7. Comparison of performance of different fitness functions

Figure 6 shows the sample comparison of two runs, one with summed fitness consisting of 2 components (2comp) and one with split fitness consisting of 4 components (4comp). The graph shows both the course of matches and complexity. When using summed fitness the complexity of solution increases uncontrollably and also functionally equivalent but less complex solutions are being forgotten. This can be seen on the graph around the 20th generation where the complexity of the best solution suddenly increased and then dropped by more then 200 elements and then raised again without any change in the number of matches. When using the split fitness values the complexity of the best individuals grows up only when followed with an increase of matched values. Therefore the already found solutions are not being discarded and are optimized more thoroughly. This makes the optimization process more steady and reliable (Fig. 7). However there is still a good chance of a new entirely new solution appearing in the middle of optimization (individuals with less then max matches are still compared only by number of matches).

8 Discussion

Figure 6 shows the performance comparison of using summed fitness and split fitness value. The graph shows the number of generations necessary to find first acceptable solution. Using the split fitness value the average number of generations was 36.8 compared to 65 using summed fitness. With the size of population of approximately 150 individuals and three populations the 36 generations roughly correspond to 17000 tested hypotheses. The summed fitness value means that no optimizations regarding complexity are made until a fully matching individual is found, thus somehow vastes computation time. Contrary to expectations split fitness value does not seem to lead to premature convergence. Howerver this can be confirmed only after the problem with generating parallel structures is solved and the algorithm is enabled to generate optimal solutions.

9 Conclusions

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. In terms of pure computational complexity the generic grammatical algorithm would be probably always outperformed by algorithms designed specifically for this purpose. Since in grammatical evolution there is some computational and memory overhead associated with processing the rules. This however also creates an advantage for grammatical evolution, which is the possibility to very specific substructures according to the definition of the grammar.

ACKNOWLEDGMENTS

This work has been supported by the Ministry of education; Grant No: MSM21630529.

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, pp. 104 – 110 (2004)
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, pp. 1639 - 1646 (2004)
4. Ošmera, P., Roupec, J.: Limited Lifetime Genetic Algorithms in Comparison with Sexual Reproduction Based GAs, Proceedings of MENDEL'2000, Brno, Czech Republic, pp. 118 – 126 (2000)
5. Li Z., Halang W. A., Chen G.:"Integration of Fuzzy Logic and Chaos Theory"; paragraph: Osmera P.: Evolution of Complexity, Springer, (ISBN: 3-540-26899-5), pp. 527 – 578. (2006)
6. Ošmera P., Popelka O., Pivoňka P.: Parallel Grammatical Evolution with Backward Processing, ICARCV 2006, 9th International Conference on Control, Automation, Robotics and Vision, Singapore, pp. 1889-1894 (2006)
7. Ošmera P., Popelka O : The Automatic Generation of Programs with Parallel Grammatical Evolution, , 13th Zittau Fuzzy Colloquium, Zittau, Germany, pp. 332-339 (2006)
8. Popelka O :Two-level Optimization using Parallel Grammatical Evolution and Differential Evolution, Proceedings of MENDEL'2007, Prague, Czech Republic, pp. 88 – 92 (2007)