We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

2

Automatic Generation of Programs

Ondřej Popelka and Jiří Štastný
Mendel University in Brno
Czech Republic

1. Introduction

Automatic generation of program is definitely an alluring problem. Over the years many approaches emerged, which try to smooth away parts of programmers' work. One approach already widely used today is colloquially known as *code generation* (or code generators). This approach includes many methods and tools, therefore many different terms are used to describe this concept. The very basic tools are included in various available Integrated Development Environments (IDE). These include templates, automatic code completion, macros and other tools. On a higher level, code generation is performed by tools, which create program source code from metadata or data. Again, there are thousands of such tools available both commercial and open source. Generally available are programs for generating source code from relational or object database schema, object or class diagrams, test cases, XML schema, XSD schema, design patterns or various formalized descriptions of the problem domain.

These tools mainly focus on the generation of a template or skeleton for an application or application module, which is then filled with actual algorithms by a programmer. The great advantage of such tools is that they lower the amount of tedious, repetitive and boring (thus error-prone) work. Commonly the output is some form of data access layer (or data access objects) or object relational mapping (ORM) or some kind of skeleton for an application - for example interface for creating, reading, updating and deleting objects in database (CRUD operations). Further, this approach leads to generative programming domain, which includes concepts such as *aspect-oriented programming* (Gunter & Mitchell, 1994), *generic programming*, *meta-programming* etc. (Czarnecki & Eisenecker, 2000). These concepts are now available for general use – for example the AspectJ extension to Java programming language is considered stable since at least 2003 (Ladad, 2009). However, they are not still mainstream form of programming according to TIOBE Index (TIOBE, 2010).

A completely different approach to the problem is an actual generation of algorithms of the program. This is a more complex then *code generation* as described above, since it involves actual creation of algorithms and procedures. This requires either extremely complex tools or artificial intelligence. The former can be probably represented by two most successful (albeit completely different) projects – Lyee project (Poli, 2002) and Specware project (Smith, 1999). Unfortunately, the Lyee project was terminated in 2004 and the latest version of Specware is from 2007.

As mentioned above, another option is to leverage artificial intelligence methods (particularly evolutionary algorithms) and use them to create *code evolution*. We use the term

code evolution as an opposite concept to *code generation* (as described in previous paragraphs) and later we will describe how these two concepts can be coupled. When using code generation, we let the programmer specify program metadata and automatically generate skeleton for his application, which he then fills with actual algorithms. When using code evolution, we let the programmer specify sample inputs and outputs of the program and automatically generate the actual algorithms fulfilling the requirements. We aim to create a tool which will aid human programmers by generating working algorithms (not optimal algorithms) in programming language of their choice.

In this chapter, we describe evolutionary methods usable for code evolution and results of some experiments with these. Since most of the methods used are based on genetic algorithms, we will first briefly describe this area of artificial intelligence. Then we will move on to the actual algorithms for automatic generation of programs. Furthermore, we will describe how these results can be beneficial to mainstream programming techniques.

2. Methods used for automatic generation of programs

2.1 Genetic algorithms

Genetic algorithms (GA) are a large group of evolutionary algorithms inspired by evolutionary mechanisms of live nature. Evolutionary algorithms are non-deterministic algorithms suitable for solving very complex problems by transforming them into *state space* and searching for optimum state. Although they originate from modelling of natural process, most evolutionary algorithms do not copy the natural processes precisely.

The basic concept of genetic algorithms is based on *natural selection process* and is very generic, leaving space for many different approaches and implementations. The domain of GA is in solving multidimensional optimisation problems, for which analytical solutions are unknown (or extremely complex) and efficient numerical methods are unavailable or their initial conditions are unknown. A genetic algorithm uses three genetic operators – *reproduction*, *crossover* and *mutation* (Goldberg, 2002). Many differences can be observed in the strategy of the parent selection, the form of genes, the realization of crossover operator, the replacement scheme, etc. A basic *steady-state genetic algorithm* involves the following steps.

Initialization. In each step, a genetic algorithm contains a number of solutions (individuals) in one or more populations. Each solution is represented by genome (or chromosome). Initialization creates a starting population and sets all bits of all chromosomes to an initial (usually random) value.

Crossover. The crossover is the main procedure to ensure progress of the genetic algorithm. The crossover operator should be implemented so that by combining several existing chromosomes a new chromosome is created, which is expected to be a better solution to the problem.

Mutation. Mutation operator involves a random distortion of random chromosomes; the purpose of this operation is to overcome the tendency of genetic algorithm in reaching the local optimum instead of global optimum. Simple mutation is implemented so that each gene in each chromosome can be randomly changed with a certain very small probability.

Finalization. The population cycle is repeated until a termination condition is satisfied. There are two basic finalization variations: maximal number of iterations and the quality of the best solution. Since the latter condition may never be satisfied both conditions are usually used.

The critical operation of genetic algorithm is crossover which requires that it is possible to determine what a "better solution" is. This is determined by a *fitness function* (criterion function or objective function). The fitness function is the key feature of genetic algorithm, since the genetic algorithm performs the minimization of this function. The fitness function is actually the transformation of the problem being solved into a state space which is searched using genetic algorithm (Mitchell, 1999).

2.2 Genetic programming

The first successful experiments with automatic generation of algorithms were using Genetic Programming method (Koza, 1992). Genetic programming (GP) is a considerably modified genetic algorithm and is now considered a field on its own. GP itself has proven that evolutionary algorithms are definitely capable of solving complex problems such as automatic generation of programs. However, a number of practical issues were discovered. These later lead to extending GP with (usually context-free) grammars to make this method more suitable to generate program source code (Wong & Leung, 1995) and (Patterson & Livesey, 1997).

Problem number one is the overwhelming complexity of automatic generation of a program code. The most straightforward approach is to split the code into subroutines (functions or methods) the same way as human programmers do. In genetic programming this problem is generally being solved using *Automatically Defined Functions* (ADF) extension to GP. When using automatically defined function each program is split into definitions of one or more functions, an expression and result producing branch. There are several methods to create ADFs, from manual user definition to automatic evolution. Widely recognized approaches include generating ADFs using genetic programing (Koza, 1994), genetic algorithms (Ahluwalia & Bull, 1998), logic grammars (Wong & Leung, 1995) or gene expression programming (Ferreira, 2006a).

Second very difficult problem is actually creating syntactically and semantically correct programs. In genetic programming, the program code itself is represented using a *concrete syntax tree* (*parse tree*). An important feature of GP is that all genetic operations are applied to the tree itself, since GP algorithms generally lack any sort of genome. This leads to problems when applying the crossover or mutation operators since it is possible to create a syntactically invalid structure and since it limits evolutionary variability. A classic example of the former is exchanging (within crossover operation) a function with two parameters for a function with one parameter and vice versa – part of the tree is either missing or superfluous. The latter problem is circumvented using very large initial populations which contain all necessary prime building blocks. In subsequent populations these building blocks are only combined into correct structure (Ferreira, 2006a).

Despite these problems, the achievements of genetic programming are very respectable; as of year 2003 there are 36 human-competitive results known (Koza et al, 2003). These results include various successful specialized algorithms or circuit topologies. However we would like to concentrate on a more mainstream problems and programming languages. Our goal are not algorithms competitive to humans, rather we focus on creating algorithms which are just working. We are also targeting mainstream programming languages.

2.3 Grammatical evolution

The development of Grammatical Evolution (GE) algorithm (O'Neill & Ryan, 2003) can be considered a major breakthrough when solving both problems mentioned in the previous

paragraph. This algorithm directly uses a generative context-free grammar (CFG) to generate structures in an arbitrary language defined by that grammar. A genetic algorithm is used to direct the structure generation. The usage of a context-free grammar to generate a solution ensures that a solution is always syntactically correct. It also enables to precisely and flexibly define the form of a solution without the need to alter the algorithm implementation.

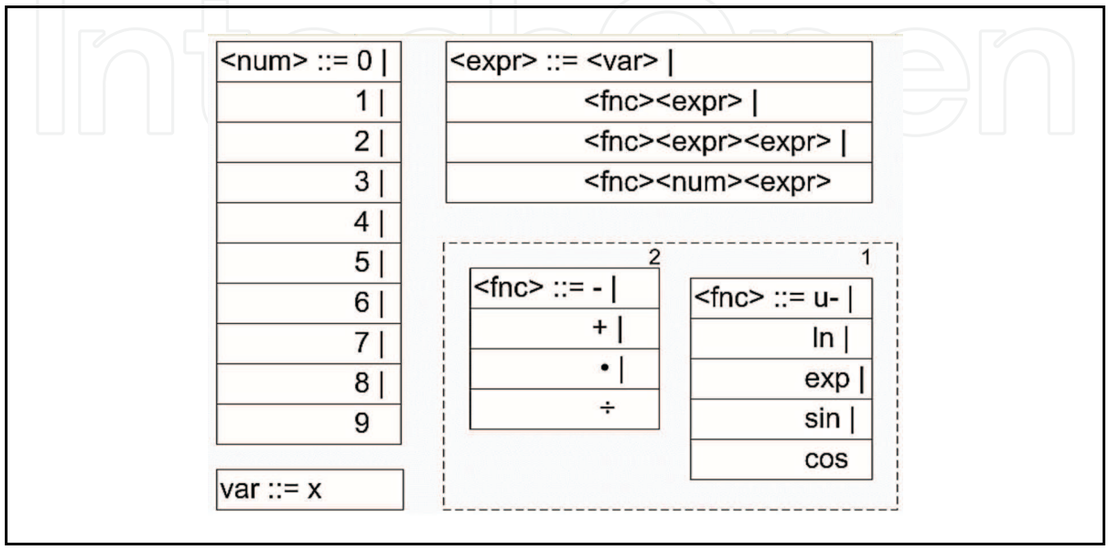


Fig. 1. Production rules of grammar for generating arithmetic expressions

In grammatical evolution each individual in the population is represented by a sequence of rules of a defined (context-free) grammar. The particular solution is then generated by translating the chromosome to a sequence of rules which are then applied in specified order. A context-free grammar G is defined as a tuple G = (Π,Σ,P,S) where Π is set of non-terminals, Σ is set of terminals, S is initial non-terminal and P is table of production rules.

The non-terminals are items, which appear in the individuals' body (the solution) only before or during the translation. After the translation is finished all non-terminals are translated to terminals. Terminals are all symbols which may appear in the generated language, thus they represent the solution. Start symbol is one non-terminal from the non-terminals set, which is used to initialize the translation process. Production rules define the laws under which non-terminals are translated to terminals. Production rules are key part of the grammar definition as they actually define the structure of the generated solution (O'Neill & Ryan, 2003).

We will demonstrate the principle of grammatical evolution and the backward processing algorithm on generating algebraic expressions. The grammar we can use to generate arithmetic expressions is defined by equations (1) – (3); for brevity, the production rules are shown separately in BNF notation on Figure 1 (Ošmera & Popelka, 2006).

$$\Pi = \{expr, fnc, num, var\} \tag{1}$$

$$\Sigma = \{\sin, \cos, +, -, \div, \cdot, x, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \tag{2}$$

$$S = expr \tag{3}$$

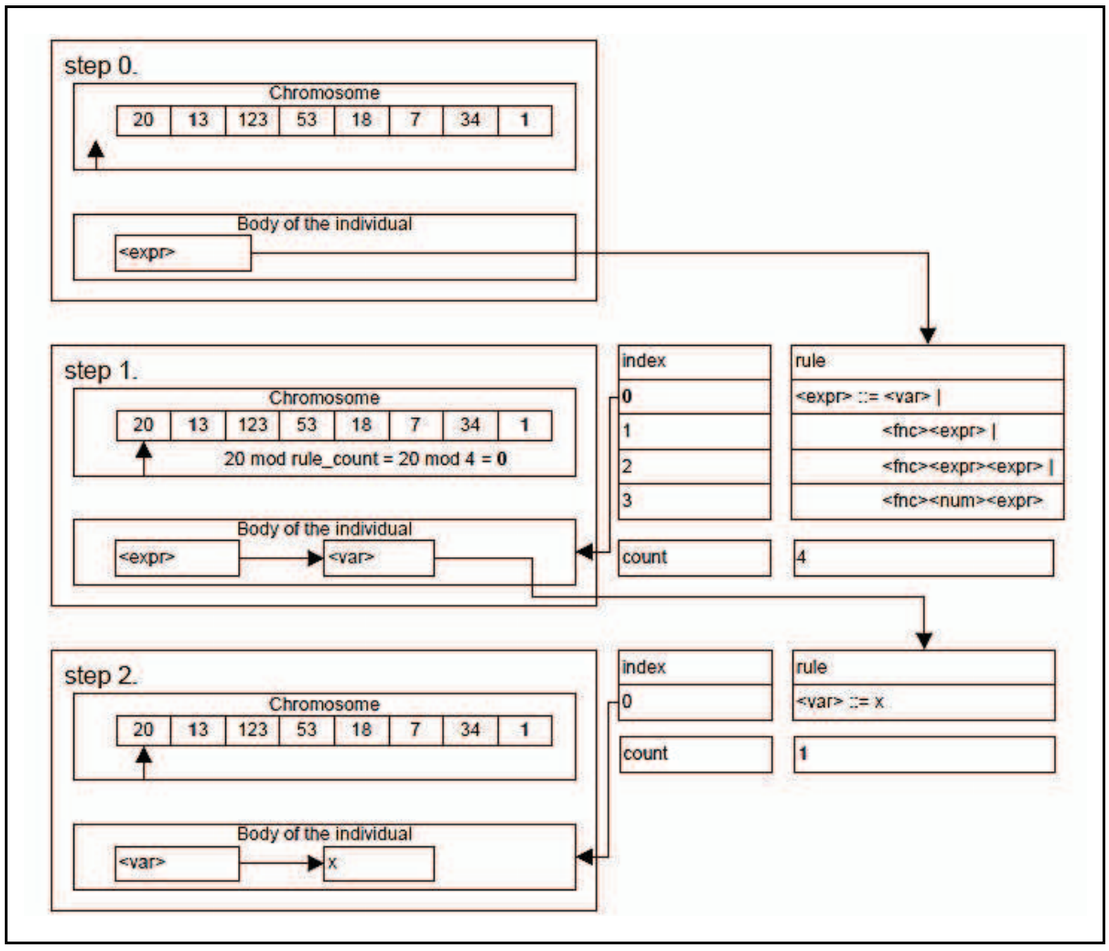


Fig. 2. Process of the translation of the genotype to a solution (phenotype)

The beginning of the process of the translation is shown on Figure 2. At the beginning we have a chromosome which consists of randomly generated integers and a non-terminal <expr> (expression). Then all rules which can rewrite this non-terminal are selected and rule is chosen using modulo operation and current gene value. Non-terminal <expr> is rewritten to non-terminal <var> (variable). Second step shows that if only one rule is available for rewriting the non-terminal, it is not necessary to read a gene and the rule is applied immediately. This illustrates how the genome (chromosome) can control the generation of solutions. This process is repeated for every solution until no non-terminals are left in its' body. Then each solution can be evaluated and a genetic algorithm population cycle can start and determine best solutions and create new chromosomes.

Other non-terminals used in this grammar can be <fnc> (function) and <num> (number). Here we consider standard arithmetic operators as functions, the rules on Figure 1 are divided by the number of arguments for a function ("u-" stands for unary minus).

3. Two-level grammatical evolution

In the previous section, we have described original grammatical evolution algorithm. We have further developed the original grammatical evolution algorithm by extending it with

Backward Processing algorithm (Ošmera, Popelka & Pivoňka, 2006). The backward processing algorithm just uses different order of processing the rules of the context free grammar than the original GE algorithm. Although the change might seem subtle, the consequences are very important. When using the original algorithm, the rules are read left-to-right and so is the body of the individual scanned left-to-right for untranslated non-terminals.

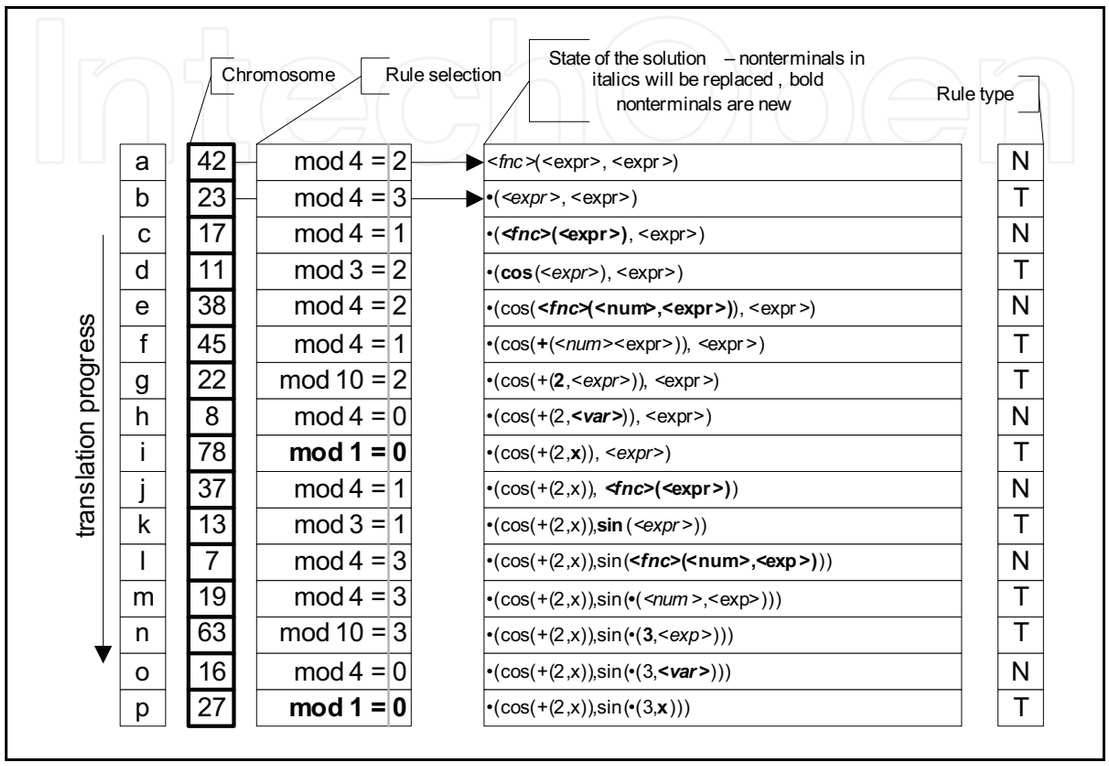


Fig. 3. Translation process of an expression specified by equation (4)

3.1 Backward processing algorithm

The whole process of translating a sample chromosome into an expression (equation 4) is shown on figure 3 []. Rule counts and rule numbers correspond to figure 1, indexes of the rules are zero-based. Rule selected in step a) of the translation is therefore the third rule in table.

$$\cos(2+x)\cdot\sin(3\cdot x) \tag{4}$$

The backward processing algorithm scans the solution string for non-terminals in right-to-left direction. Figure 4 shows the translation process when this mode is used. Note that the genes in the chromosome are the same; they just have been rearranged in order to create same solution, so that the difference between both algorithms can be demonstrated. Figure 4 now contains two additional columns with *rule type* and *gene mark*.

Rule types are determined according to what non-terminals they translate. We define a *T-terminal* as a terminal which can be translated *only* to terminals. By analogy N-terminal is a terminal which can be *translated* only to non-terminals. T-rules (N-rules) are all rules translating a given T-nonterminal (N-nonterminal). Mixed rules (or non-terminals) are not

allowed. Given the production rules shown on Figure 1, the only N-nonterminal is <expr>, non-terminals <fnc>, <var> and <num> are all T-nonterminals (Ošmera, Popelka & Pivoňka, 2006).

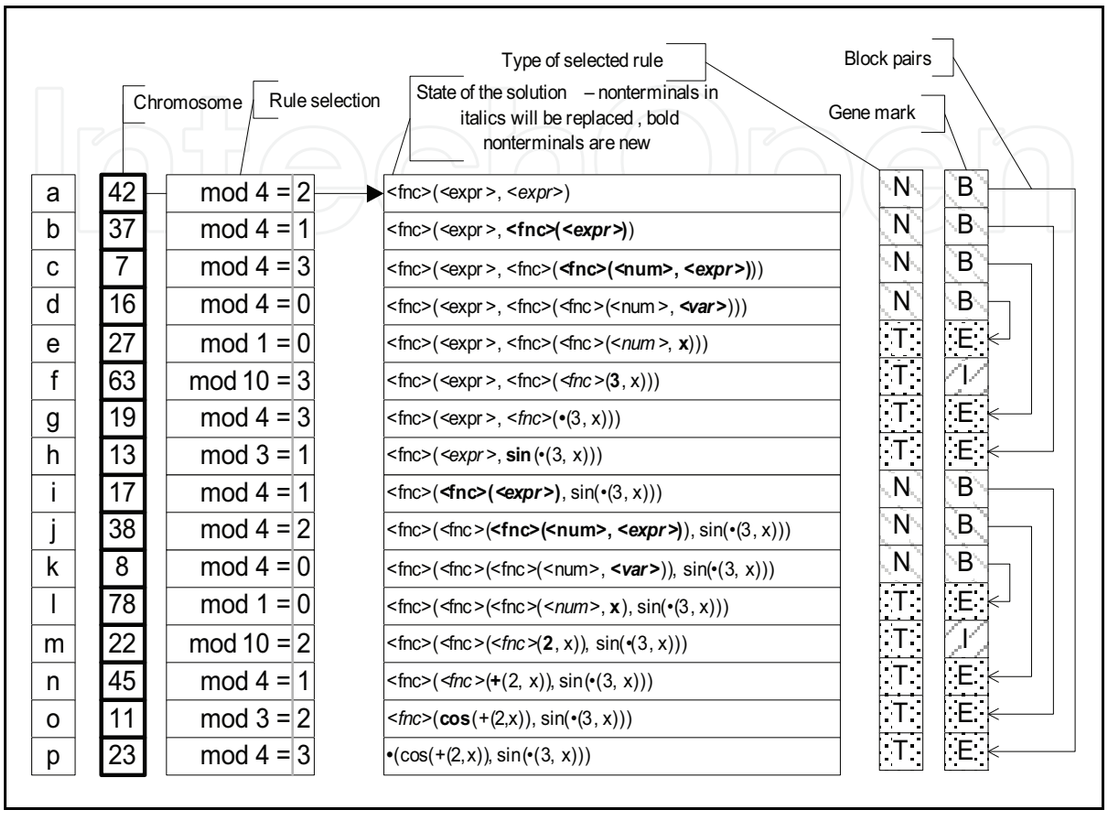


Fig. 4. Translation of an expression (equation (4)) using the backward processing algorithm

Now that we are able to determine type of the rule used, we can define *gene marks*. In step c) at figure 4 a <expr> non-terminal is translated into a <fnc>(<num>, <expr>) expression. This is further translated until step g), where it becomes $3 \cdot x$. In other words – in step c) we knew that the solution will contain a function with two arguments; in step g) we realized that it is multiplication with arguments 3 and x. The important feature of backward processing algorithm that all genes which define this sub-expression including all its' parameters are in a single uninterrupted block of genes. To explicitly mark this block we use *Block marking algorithm* which marks:

- all genes used to select N-rule with mark B (Begin)
- all genes used to select T-rule except the last one with mark I (Inside)
- all genes used to select last T-rule of currently processed rule with mark E (End).

The B and E marks determine begin and end of *logical blocks* generated by the grammar. This works independent of the structure generated provided that the grammar consists only of N-nonterminals and T-nonterminals. These logical blocks can then be exchanged the same way as in genetic programming (figure 5) (Francone et al, 1999).

Compared to genetic programing, all the genetic algorithm operations are still performed on the genome (chromosome) and not on the actual solution. This solves the second problem described in section 2.2 – the generation of syntactically incorrect solutions. Also the

problem of lowered variability is solved since we can always insert or remove genes in case we need to remove or add parts of the solution. This algorithm also solves analogical problems existing in standard grammatical evolution (O'Neill et al, 2001).

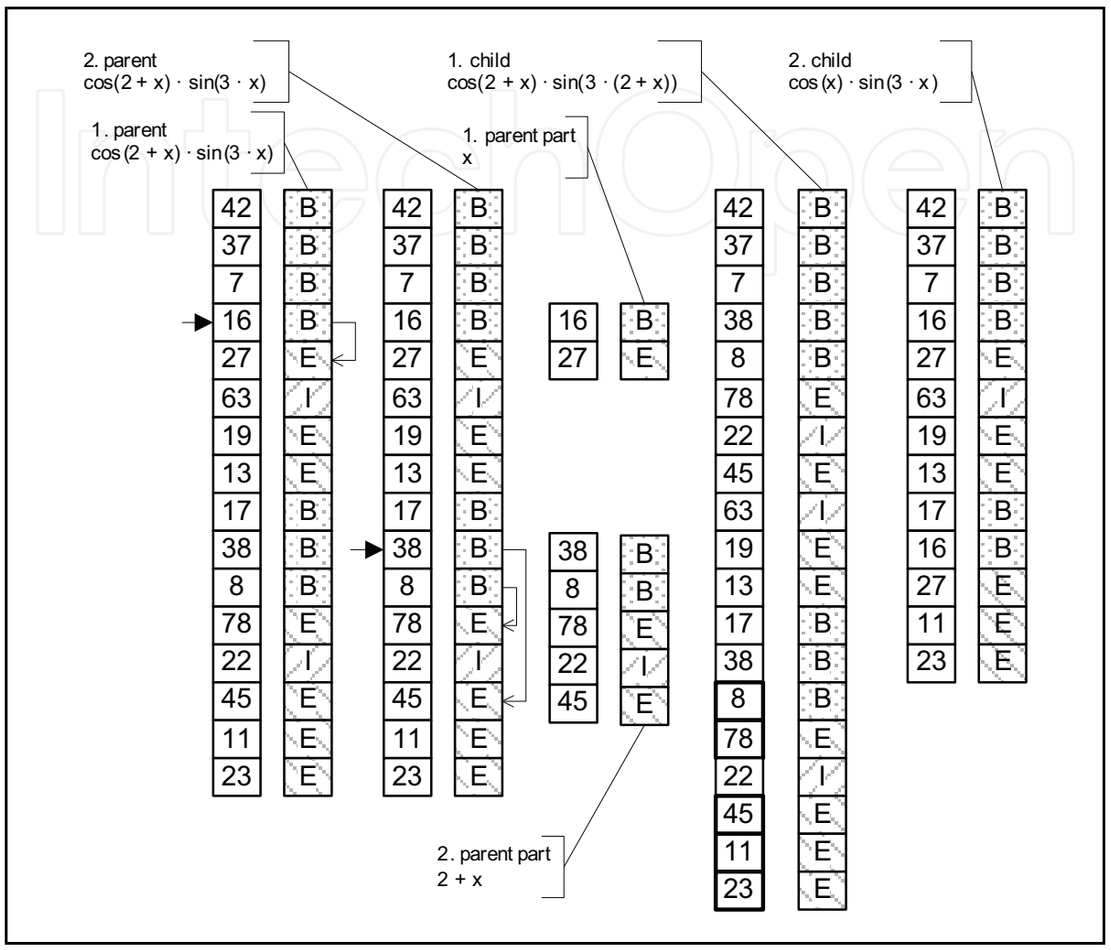


Fig. 5. Example of crossing over two chromosomes with marked genes

The *backward processing algorithm* of *two-level grammatical evolution* provides same results as original grammatical evolution. However in the underlying genetic algorithm, the genes that are involved in processing a single rule of grammar are grouped together. This grouping results in greater stability of solutions during crossover and mutation operations and better performance (Ošmera & Popelka, 2006). An alternative to this algorithm is *Gene expression programming* method (Cândida Ferreira, 2006b) which solves the same problem but is quite limited in the form of grammar which can be used.

3.2 Second level generation in two-level grammatical evolution

Furthermore, we modified grammatical evolution to separate structure generation and parameters optimization (Popelka, 2007). This is motivated by poor performance of grammatical evolution when optimizing parameters, especially real numbers (Dempsey et al., 2007). With this approach, we use grammatical evolution to generate complex structures. Instead of immediately generating the resulting string (as defined by the grammar), we store

the parse tree of the structure and use it in second level of optimization. For this second level of optimization, a Differential evolution algorithm (Price, 1999) is used. This greatly improves the performance of GE, especially when real numbers are required (Popelka & Šťastný, 2007)

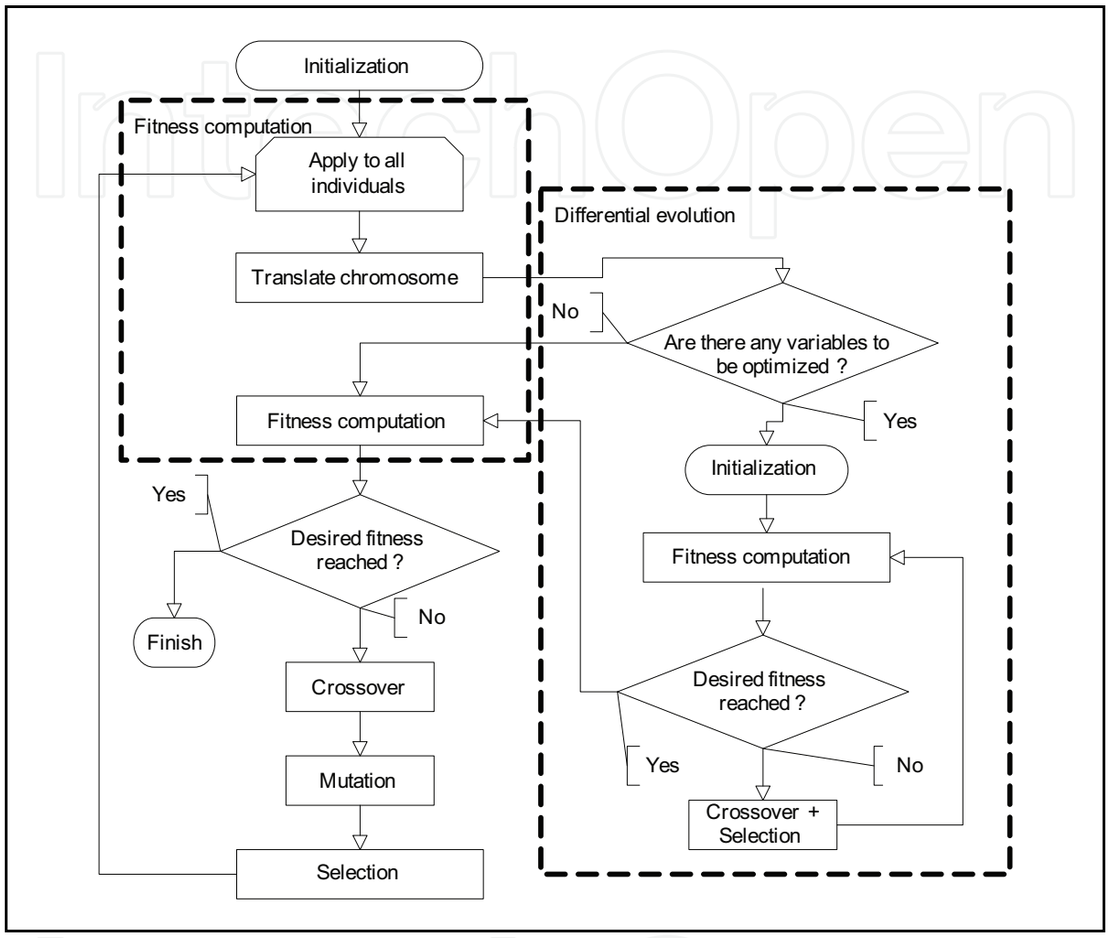


Fig. 6. Flowchart of two-level grammatical evolution

The first level of the optimization is performed using grammatical evolution. According to the grammar, the output can be a function containing variables (x in our case); and instead of directly generating numbers using the <num> nonterminal we add several symbolic constants (a, b, c) into to grammar. The solution expression cannot be evaluated and assigned a fitness value since the values of symbolic constants are unknown. In order to evaluate the generated function a secondary optimization has to be performed to find values for constants. Input for the second-level of optimization is the function with symbolic constants which is transformed to a vector of variables. These variables are optimized using the differential evolution and the output is a vector of optimal values for symbolic constants for a given solution. Technically in each grammatical evolution cycle there are hundreds of differential evolution cycles executed. These optimize numeric parameters of each generated individual (Popelka, 2007). Figure 6 shows the schematic flowchart of the two-level grammatical evolution.

3.3 Deformation grammars

Apart from generating the solution we also need to be able to read and interpret the solutions (section 4.2). For this task a syntactic analysis is used. Syntactic analysis is a process which decides if the string belongs to a language generated by a given grammar, this can be used for example for object recognition (Šťastný & Minařík, 2006). It is possible to use:

- *Regular grammar* – Deterministic finite state automaton is sufficient to analyse regular grammar. This automaton is usually very simple in hardware and software realization.
- *Context-free grammar* – To analyse context-free grammar a nondeterministic finite state automaton with stack is generally required.
- *Context grammar* – "Useful and sensible" syntactic analysis can be done with context-free grammar with controlled re-writing.

There are two basic methods of syntactic analysis:

- *Bottom-up parsing* – We begin from analysed string to initial symbol. The analysis begins with empty stack. In case of successful acceptance only initial symbol remains in the stack, e.g. Cocke-Younger-Kasami algorithm (Kasami, 1965), which grants that the time of analysis is proportional to third power of string length;
- *Top-down parsing* – We begin from initial symbol and we are trying to generate analysed string. String generated so far is saved in the stack. Every time a terminal symbol appears on the top of the stack, it is compared to actual input symbol of the analysed string. If symbols are identical, the terminal symbol is removed from the top of the stack. If not, the algorithm returns to a point where a different rule can be chosen (e.g. with help of backtracking). Example of top down parser is Earley's Parser (Aycock & Horspool, 2002), which executes all ways of analysis to combine gained partial results. The time of analysis is proportional to third power of string length; in case of unambiguous grammars the time is only quadratic. This algorithm was used in simulation environment.

When designing a syntactic analyser, it is useful to assume random influences, e.g. image deformation. This can be done in several ways. For example, the rules of given grammar can be created with rules, which generate alternative string, or for object recognition it is possible to use some of the methods for determination of distance between attribute description of images (string metric). Finally, *deformation grammars* can be used.

Methods for determination of distance between attribute descriptions of images (string metric) determine the distance between attribute descriptions of images, i.e. the distance between strings which correspond to the unknown object and the object class patterns. Further, determined distances are analysed and the recognized object belongs to the class from which the string has the shortest distance. Specific methods (Levenshtein distance Ld(s, t), Needleman-Wunsch method) can be used to determine the distance between attribute descriptions of image (Gusfield, 1997).

Results of these methods are mentioned e.g. in (Minařík, Šťastný & Popelka, 2008). If the parameters of these methods are correctly set, these methods provide good rate of successful identified objects with excellent classification speed. However, false object recognition or non-recognized objects can occur.

From the previous paragraphs it is clear that recognition of non-deformed objects with structural method is without problems, it offers excellent speed and 100% classification rate. However, recognition of randomly deformed objects is nearly impossible. If we conduct syntactic analysis of a string which describes a structural deformed object, it will apparently

not be classified into a given class because of its structural deformation. Further, there are some methods which use structural description and are capable of recognizing randomly deformed objects with good rate of classification and speed.

The solution to improve the rate of classification is to enhance the original grammar with rules which describe errors – *deformation rules*, which cover up every possible random deformation of object. Then the task is changed to finding a non-deformed string, which distance from analysed string is minimal. Compared to the previous method, this is more informed method because it uses all available knowledge about the classification targets – it uses grammar. Original grammar may be regular or context-free, enhanced grammar is always context-free and also ambiguous, so the syntactic analysis, according to the enhanced grammar, will be more complex.

Enhanced deformation grammar is designed to reliably generate all possible deformations of strings (objects) which can occur. Input is context-free or regular grammar G = (VN, VT, P, S). Output of the processing is *enhanced deformation grammar* G′ = (VN′, VT′, P′, S′), where P′ is set of weighted rules. The generation process can be described using the following steps:

Step1:

$$V_N^{'} = V_N \cup \{S'\} \cup \{E_B | \ b \in V_T\} \tag{5}$$

$$V_T \subseteq V_T^{'} \tag{6}$$

Step 2:

If holds:

$$A \to \alpha_0 b_1 \alpha_1 b_2 ... \alpha_{m-1} b_m \alpha_m;\ m \geq 0;\ \alpha_1 \in V_N^{'} \wedge b_i \in V_T^{'};\ i = 1,2,...,m;\ l = 0,1,...,m \tag{7}$$

Then add new rule into P' with weight 0:

$$A \to \alpha_0 E_{b1} \alpha_1 E_{b2} ... \alpha_{m-1} E_{bm} \alpha_m \tag{8}$$

Step 3:

Into P'add the rules in table 1 with weight according to chosen metric. In this example Levenshtein distance is used. In the table header L is Levenshtein distance, w is weighted Levenshtein distance and W is weighted metric.

Rule	L	w	W	Rule for
$S' \to S$	0	0	0	-
$S' \to Sa$	1	w_I	$I'(a)$	$a \in V_T^{'}$
$E_a \to a$	0	0	0	$a \in V_T$
$E_a \to b$	1	w_S	$S(a,b)$	$a \in V_T, b \in V_T^{'}, a \neq b$
$E_a \to \delta$	1	w_D	$D(a)$	$a \in V_T$
$E_a \to bE_a$	1	w_I	$I(a,b)$	$a \in V_T, b \in V_T^{'}$

Table 1. Rules of enhanced deformation grammar

These types of rules are called deformation rules. Syntactic analyser with error correction works with enhanced deformation grammar. This analyser seeks out such deformation of

input string, which is linked with the smallest sum of weight of deformation rules. G' is ambiguous grammar, i.e. its syntactic analysis is more complicated. A modified Earley parser can be used for syntactic analyses with error correction. Moreover, this parser accumulates appropriate weight of rules which were used in deformed string derivation according to the grammar G'.

3.4 Modified Early algorithm

Modified Early parser accumulates weights of rules during the process of analysis so that the deformation grammar is correctly analysed (Minařík, Šťastný & Popelka, 2008). The input of the algorithms is enhanced deformation grammar G' and input string w.

$$w = b_1 b_2 ... b\_m \tag{9}$$

Output of the algorithm is lists $I_0, I_1, ... I_m$ for string w (equation 9) and distance d of input string from a template string defined by the grammar.

Step 1 of the algorithm – create list I_0 . For every rule $S' \to \alpha \in P'$ add into I_0 field:

$$[S' \to \cdot\alpha, 0, x] \tag{10}$$

Execute until it is possible to add fields into I_0 . If

$$[A \to \cdot B\beta, 0, y] \tag{11}$$

is in I_0 field then add

$$B \xrightarrow{Z} \gamma \text{field} [B \to \cdot\gamma, 0, z] \tag{12}$$

into I_0 .

Step 2: Repeat for $j = 1, 2, ..., m$ the following sub-steps A – C:

a. for every field in I_{j-1} in form of $[B \to \alpha \cdot a\beta, i, x]$ such that $a = b_j$, add the field

$$[B \to \alpha a \cdot \beta, i, x] \tag{13}$$

into I_j . Then execute sub-steps B and C until no more fields can be added into I_j .

b. If field $[A \to \alpha\cdot, i, x]$ is in I_j and field $[B \to \beta \cdot A\gamma, k, y]$ in I_j , then
   a. If exists a field in form of $[B \to \beta A \cdot \gamma, k, z]$ in I_j , and then if x+y < z, do replace the value z with value $x + y$ in this field
   b. If such field does not exist, then add new field $[B \to \beta A \cdot \gamma,\ k,\ x + y]$

c. For every field in form of $[A \to \alpha \cdot B\beta, i, x]$ in I_j do add a field $[B \to \cdot\gamma, j, z]$ for every rule

$$B \xrightarrow{Z} \gamma \tag{14}$$

Step 3: If the field

$$[S' \to \alpha\cdot, 0, x] \tag{15}$$

is in I_m , then string w is accepted with distance weight x. String w (or its derivation tree) is obtained by omitting all deformation rules from derivation of string w.

Designed deformation grammar reliably generates all possible variants of randomly deformed object or string. It enables to use some of the basic methods of syntactic analysis for randomly deformed objects. Compared to methods for computing the distance between attribute descriptions of objects it is more computationally complex. Its effectiveness depends on effectiveness of the used parser or its implementation respectively. This parses is significantly are more complex than the implementation of methods for simple distance measurement between attribute descriptions (such as Levenshtein distance).
However, if it is used correctly, it does not produce false object recognition, which is the greatest advantage of this method. It is only necessary to choose proper length of words describing recognized objects. If the length of words is too short, excessive deformation (by applying only a few deformation rules) may occur, which can lead to occurrence of description of completely different object. If the length is sufficient (approximately 20% of deformed symbols in words longer than 10 symbols), this method gives correct result and false object recognition will not occur at all.
Although deformed grammars were developed mainly for object recognition (where an object is represented by a string of primitives), it has a wider use. The main feature is that it can somehow adapt to new strings and it can be an answer to the problem described in section 4.2.

4. Experiments

The goal of automatic generation of programs is to create a valid source code of a program, which will solve a given problem. Each individual of a genetic algorithm is therefore one variant the program. Evaluation of an individual involves compilation (and building) of the source code, running the program and inputting the test values. Fitness function then compares the actual results of the running program with learning data and returns the fitness value. It is obvious that the evaluation of fitness becomes very time intensive operation. For the tests we have chosen the PHP language for several reasons. Firstly it is an interpreted language which greatly simplifies the evaluation of a program since compiling and building can be skipped. Secondly a PHP code can be interpreted easily as a string using either command line or library API call, which simplified implementation of the fitness function into our system. Last but not least, PHP is a very popular language with many tools available for programmers.

4.1 Generating simple functions

When testing the two-level grammatical evolution algorithm we stared with very simple functions and a very limited grammar:

```
<statement> ::= <begin><statement><statement> |
          <if><condition><statement> |
          <function><expression><expression> |
          <assign><var><expression>
<expression> ::= <function><expression> |
           <const> |
           <var> |
           <function><expression><expression>
<condition> ::= <operator><expression><expression>
<operator> ::= < | > | != | == | >= | <=
```

<var> ::= $a | $b | $result
<const> ::= 0 | 1| -1
<function> ::= + | - | * | /
<begin> ::= {}
<if> ::= if {}
<assign> ::= =

This grammar represents a very limited subset of the PHP language grammar (Salsi, 2007) or (Zend, 2010). To further simplify the task, the actual generated source code was only a body of a function. Before the body of the function, a header is inserted, which defines the function name, number and names of its arguments. After the function body, the return command is inserted. After the complete function definition, a few function calls with learning data are inserted. The whole product is then passed to PHP interpreter and the text result is compared with expected results according to given learning data.

The simplest experiment was to generate function to compute absolute value of a number (without using the abs() function). The input for this function is one integer number; output is absolute value of that number. The following set of training patterns was used:

P = {(−3, 3); (43, 43); (3, 3); (123, 123); (−345, 345); (−8, 8); (−11, 11); (0, 0)}.

Fitness function is implemented so that for each pattern it assigns points according to achieved result (result is assigned, result is number, result is not negative, result is equal to training value). Sum of the points then represents the achieved fitness. Following are two selected examples of generated functions:

```
function absge($a) {
        $result = null;
        $result = $a;
        if (($a) <= (((-(-((-($result)) + ((-($a)) - (1))))) - (-1)) - (0))) {
                $result = -($result);
        }
        return $result;
}
function absge($a) {
        $result = null;
        $result =  -($a);
        if ((-($result)) >= (1)) {
                $result =  $a;
        };
return $result;
}
```

While the result looks unintelligible, it must be noted that this piece of source code is correct algorithm. The last line and first two lines are the mandatory header which was added automatically for the fitness evaluation. Apart from that it has not been processed, it is also important to note that it was generated in all 20 runs from only eight sample values in average of 47.6 population cycles (population size was 300 individuals).

Another example is a classic function for comparing two integers. Input values are two integer numbers a and b. Output value is integer c, which meets the conditions $c > 0$, for $a > b$; $c = 0$, for $a = b$; $c < 0$, for $a < b$. Training data is a set of triples (a, b, c):

P = {(−3, 5,−1); (43, 0, 1); (8, 8, 0); (3, 4,−1); (−3,−4, 1);}

The values intentionally do not correspond to the usual implementation of this function: $c = a - b$. Also the fitness function checks only if c satisfies the conditions and not if the actual value is equal, thus the search space is open for many possible solutions. An example solution is:

```
function comparege($a, $b) {
        $result = null;
        if ((($a) - (($b) * (-((($result) / (1))))) <= ($result)) {{
                $result = 0;
                $result = $b;
        }}
        $result = ($b) - ($a);;
        $result = -($result);;
        return $result;
}
```

The environment was the same like in the first example; generation took 75.1 population cycles on average. Although these tests are quite successful, it is obvious, that this is not very practical.

For each simple automatically generated function a programmer would need to specify a very specific test, function header, function footer. Tests for genetic algorithms need to be specific in the values they return. A fitness function which would return just "yes" or "no" is insufficient in navigating the genetic algorithm in the state space – such function cannot be properly optimized. The exact granularity of the fitness function values is unknown, but as little as 5 values can be sufficient if they are evenly distributed (as shown in the first example in this section).

4.2 Generating classes and methods

To make this system described above practical, we had to use standardized tests and not custom made fitness functions. Also we wanted to use object oriented programming, because it is necessary to keep the code complexity very low. Therefore we need to stick with the paradigm of small simple "black box" objects. This is a necessity and sometimes an advantage. Such well-defined objects are more reliable, but it is a bit harder to maintain their connections (Büchi & Weck, 1999).

Writing class tests before the actual source code is already a generally recognized approach – *test-driven development*. In test-driven development, programmers start off by writing tests for the class they are going to create. Once the tests are written, the class is implemented and tested. If all tests pass, a coverage analysis is performed to check whether the tests do cover all the newly written source code (Beck, 2002). An example of a simple test using PHPUnit testing framework:

```
class BankAccountTest extends PHPUnit_Framework_TestCase {
        protected $ba;
        protected function setUp() {
                $this->ba = new BankAccount;
        }
        public function testBalanceIsInitiallyZero() {
                $this->assertEquals(0, $this->ba->getBalance());
        }
```

```
	public function testBalanceCannotBecomeNegative() {
		try {
			$this->ba->withdrawMoney(1);
		}
		catch (BankAccountException $e) {
			$this->assertEquals(0, $this->ba->getBalance());
			return;
		}
		$this->fail();
	}
	...
}
```

The advantage of modern unit testing framework is that it is possible to create class skeleton (template) from the test. From the above test, the following code can be easily generated:

```
class BankAccount {
	public function depositMoney() {}
	public function getBalance() {}
	public function withdrawMoney() {}
}
```

Now we can use a PHP parser to read the class skeleton and import it as a template grammar rule into grammatical evolution. This task is not as easy as it might seem. The class declaration is incomplete – it is missing function parameters and private members of the class.

Function parameters can be determined from the tests by static code analysis, provided that we refrain from variable function parameters. Function parameter completion can be solved by extending the PHPUnit framework. Private members completion is more problematic, since it should be always unknown to the unit test (per the black box principle). Currently we created grammar rule for grammatical evolution by hand. In future, however, we would like to use deformed grammar (as described in section 3.3) to derive initial rule for grammatical evolution. We use <class_declaration_statement> as starting symbol, then we can define first (and only) rewriting rule for that symbol as (in EBNF notation):

```
<class_declaration_statement> :==
	“class BankAccount {“ <class_variable_declarations>
		“public function depositMoney(“<variable_without_objects>”) {“
			<statement_list>
		“}
		public function getBalance() {“
			<statement_list>
		“}
		public function withdrawMoney(“<variable_without_objects>”) {“
			<statement_list
		“}
	}”
```

This way we obtain the class declaration generated by the unit test, plus space for private class members (only variables in this case) and function parameters. It is important to note that the grammar used to generate a functional class needs at least about 20 production rules

(compared to approximately 10 in the first example). This way we obtain grammar to generate example class *BankAccount*. This can now be fed to the unit test, which will return number of errors and failures.
This experiment was only half successful. We used the concrete grammar described above – that is grammar specifically designed to generate *BankAccount* class with all its' public methods. Within average of 65.6 generations (300 individuals in generation) we were able to create individuals without errors (using only initialized variables, without infinite loops, etc.). Then the algorithm remained in local minimum and failed to find a solution with functionally correct method bodies.
After some investigation, we are confident that the problem lies in the *return* statement of a function. We have analyzed hundreds of solution and found that the correct code is present, but is preceded with return statement which exits from the function. The solution is to use predefined function footer and completely stop using the return statement (as described in section 4.1). This however requires further refinement of the grammar, and again deformation grammars might be the answer. We are also confident that similar problems will occur with other control-flow statements.
We have also tested a very generic production rules, such as:

```
<class_declaration_statement> :== "class BankAccount {" {<class_statement>} "}"
<class_statement> :== <visibility_modifier> "function ("<parameter_list>"){"
                      <statement_list> "}"
                      | <visibility_modifier> <variable_without_objects> ";"
                      ...
```

When such generic rules were used, no solution without errors was found within 150 allowed generations. This was expected as the variability of solutions and state space complexity rises extremely quickly.

5. Conclusion

In this chapter, we have presented several methods and concepts suitable for *code evolution* a fully automated generation of working source code using evolutionary algorithms. In the above paragraphs, we described how code evolution could work together with code generation. Code generation tools can be used to create a skeleton or template for an application, while code evolution can fill in the actual algorithms. This way, the actual generated functions can be kept short enough, so that the code evolution is finished within reasonable time.
Our long term goal is to create a tool which would be capable of generating some code from unit tests. This can have two practical applications – creating application prototypes and crosschecking the tests. This former is the case where the code quality is not an issue. What matters most is that the code is created with as little effort (money) as possible. The latter is the case where a programmer would like to know what additional possible errors might arise from a class.
The method we focused on in this chapter is unique in that its' output is completely controlled by a context-free grammar. Therefore this method is very flexible and without any problems or modifications it can be used to generate programs in mainstream programing languages. We also tried to completely remove the fitness function of the genetic algorithm and replace it with standardized unit-tests. This can then be thought of as an extreme form of test-driven development.

6. Acknowledgement

This work was supported by the grants: MSM 6215648904/03 and IG1100791 Research design of Mendel Univeristy in Brno.

7. References

Ahluwalia, M. & Bull, L. (1998). Co-evolving functions in genetic programming: Dynamic ADF creation using GliB, Proceedings of *Evolutionary Programming VII - 7th International Conference, EP98 San Diego*. LNCS Volume 1447/1998, Springer, ISBN-13: 978-3540648918, USA

Aycock, J. & Horspool, R.N. (2002). Practical Early Parsing, *The Computer Journal*, Vol. 45, No. 6, British Computer Society, pp. 620-630, DOI: 45:6:620-630

Beck, K. (2002). Test Driven Development: By Example, Addison-Wesley Professional, 240 p., ISBN 978-0321146533, USA

Büchi, M., Weck, W. (1999). The Greybox Approach: When Blackbox Specifications Hide Too Much, Technical Report: TUCS-TR-297, Turku Centre for Computer Science, Finland

Cândida Ferreira (2006a). Automatically Defined Functions in Gene Expression Programming in *Genetic Systems Programming: Theory and Experiences, Studies in Computational Intelligence*, Vol. 13, pp. 21-56, Springer, USA

Cândida Ferreira (2006b). *Gene Expression Programming: Mathematical Modelling by an Artificial Intelligence* (Studies in Computational Intelligence), Springer, ISBN 978-3540327967, USA

Czarnecki, K. & Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley Professional, ISBN 978-0201309775, Canada

Dempsey, I., O'Neill, M. & Brabazon, A. (2007). Constant creation in grammatical evolution, *Innovative Computing and Applications*, Vol. 1, No.1, pp. 23–38

Francone, D. F, Conrads, M., Banzhaf, W. & Nordin, P. (1999). Homologous Crossover in Genetic Programming, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1021–1026. ISBN 1-55860-611-4, Orlando, USA

Goldberg, D. E. (2002). *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, 272 p. ISBN 1-4020-7098-5, Boston, USA

Gunter, C. A. & Mitchell, J. C. (1994). *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, The MIT Press, ISBN 978-0262071550, Cambridge, Massachusetts, USA

Gusfield, D. (1997). Gusfield, Dan (1997). *Algorithms on strings, trees, and sequences: computer science and computational biology*, Cambridge University Press. ISBN 0-521-58519-8. Cambridge, UK

Kasami, T. (1965). An efficient recognition and syntax-analysis algorithm for context-free languages. Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA, USA

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, The MIT Press, ISBN 978-0262111706, Cambridge, Massachusetts, USA

Koza, J. R. (1994). Gene Duplication to Enable Genetic Programming to Concurrently Evolve Both the Architecture and Work-Performing Steps of a Computer Program, *IJCAI-*

95 – Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, Vol. 1, pp. 734-740, Morgan Kaufmann, 20-25 August 1995, USA

Koza, J.R. et al (2003). Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Springer, 624 p., ISBN 978-1402074462, USA

Laddad, R. (2009). *Aspectj in Action: Enterprise AOP with Spring Applications*, Manning Publications, ISBN 978-1933988054, Greenwich, Connecticut, USA

Mitchell, M. (1999). *An Introduction to Genetic Algorithms,* MIT Press, 162 p. ISBN 0-262-63185-7, Cambridge MA, USA

Minařík, M., Šťastný, J. & Popelka, O. (2008). A Brief Introduction to Recognition of Deformed Objects, *Proceedings of International Conference on Soft Computing Applied in Computer and Economic Environment ICSC*, pp.191-198, ISBN 978-80-7314-134-9, Kunovice, Czech Republic

O'Neill, M. & Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*, Springer, ISBN 978-1402074448, Norwell, Massachusetts, USA

O'Neill, M., Ryan, C., Keijzer, M. & Cattolico, M. (2001). Crossover in Grammatical Evolution: The Search Continues, Proceedings of the European Conference on Genetic Programming (EuroGP), pp. 337–347, ISBN 3-540-41899-7, Lake Como, Italy

Ošmera P. & Popelka O. (2006). The Automatic Generation of Programs with Parallel Grammatical Evolution, *Proceedings of: 13th Zittau Fuzzy Colloquium*, Zittau, Germany, pp. 332-339

Ošmera P., Popelka O. & Pivoňka P. (2006). Parallel Grammatical Evolution with Backward Processing, *Proceedings of ICARCV 2006, 9th International Conference on Control, Automation, Robotics and Vision*, pp. 1889-1894, ISBN 978-1-4244-0341-7, Singapore, December 2006, IEEE Press, Singapore

Patterson, N. & Livesey, M. (1997). Evolving caching algorithms in C by genetic programming, Proceedings of Genetic Programming 1997, pp. 262-267, San Francisco, California, USA, Morgan Kaufmann

Poli, R. (2002). Automatic generation of programs: An overview of Lyee methodology, *Proceedings of 6th world multiconference on systemics, cybernetics and informatics*, vol. I, proceedings - information systems development I, pp. 506-511, Orlando, Florida, USA, July 2002

Popelka O. (2007). Two-level optimization using parallel grammatical evolution and differential evolution. *Proceedings of MENDEL 2007, International Conference on Soft Computing, Praha*, Czech Republic. pp. 88-92. ISBN 978-80-214-3473-8., August 2007

Popelka, O. & Šťastný, J. (2007). Generation of mathematic models for environmental data analysis. *Management si Inginerie Economica*. Vol. 6, No. 2A, 61-66. ISSN 1583-624X.

Price, K. (1999). An Introduction to Differential Evolution. In: New Ideas in Optimization. Corne D., Dorigo, M. & Glover, F. (ed.) McGraw-Hill, London (UK), 79–108, ISBN 007-709506-5.

Salsi, U. (2007). PHP 5.2.0 EBNF Syntax, online: http://www.icosaedro.it/articoli/php-syntax-ebnf.txt

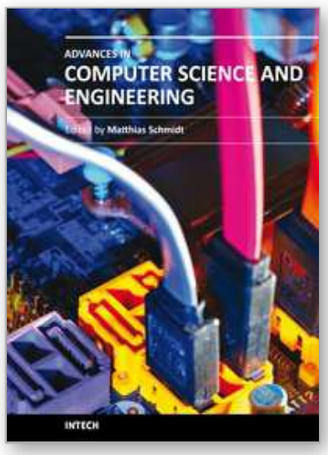
Smith, D. R. (1999). Mechanizing the development of software, In: *Nato Advanced Science Institutes Series*, Broy M. & Steinbruggen R. (Ed.), 251-292, IOS Press, ISBN 90-5199-459-1

Šťastný, J. & Minařík, M. (2006). Object Recognition by Means of New Algorithms, *Proceedings of International Conference on Soft Computing Applied in Computer and Economic Environment ICSC*, pp. 99-104, ISBN 80-7314-084-5, Kunovice, Czech Republic

TIOBE Software (2010). TIOBE Programming Community Index for June 2010, online: http://www.tiobe.com/index.php/content/paperinfo/tpci/

Wong M. L. & Leung K. S. (1995) Applying logic grammars to induce sub-functions in genetic programming, *Proceedings of 1995 IEEE International Conference on Evolutionary Computation (ICEC 95)*, pp. 737-740, ISBN 0-7803-2759-4, Perth, Australia, November 1995, IEEE Press

Zend Technologies (2010). Zend Engine – Zend Language Parser, online: http://svn.php.net/repository/php/php src/trunk/Zend/zend_language_parser.y



Advances in Computer Science and Engineering
Edited by Dr. Matthias Schmidt

ISBN 978-953-307-173-2
Hard cover, 462 pages
Publisher InTech
Published online 22, March, 2011
Published in print edition March, 2011

The book Advances in Computer Science and Engineering constitutes the revised selection of 23 chapters written by scientists and researchers from all over the world. The chapters cover topics in the scientific fields of Applied Computing Techniques, Innovations in Mechanical Engineering, Electrical Engineering and Applications and Advances in Applied Modeling.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Ondřej Popelka and Jiří Šťastný (2011). Automatic Generation of Programs, Advances in Computer Science and Engineering, Dr. Matthias Schmidt (Ed.), ISBN: 978-953-307-173-2, InTech, Available from: http://www.intechopen.com/books/advances-in-computer-science-and-engineering/automatic-generation-of-programs

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen