BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

Automating Test Driven Development with Grammatical Evolution

Jan Svoboda*

Abstract
Test driven development is a widely used process of creating software products with automated tests. In this process, developers first write tests based on given specifications and then proceed to write as little production code as possible that passes the tests. This work focuses on automating the second part using grammatical evolution i.e. a technique that generates programs in arbitrary language based on the specified cost function. The proposed system is able to create simple functions in an imperative programming language based on unit tests. Generated code contains common constructs such as command sequences, conditional branches, and loops. The system is demonstrated on evolving the `array_filter` function from tests describing its inputs and outputs. This approach could shift the role of developers, whose work would no longer involve thinking about implementation details. Instead, they would focus on formalizing high-level specifications.

Keywords: Grammatical Evolution — Test Driven Development — Artificial Intelligence

Supplementary Material: Code on GitHub

*xsvobo0s@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Automated testing plays an important role in quality assurance of software products. Its purpose is to automatically verify that each version behaves according to specifications. That leads to less time spent on manual testing and lower number of bugs, which can significantly reduce the cost of development and maintenance of software.

One approach to creating software with automated tests is the so-called test driven development. This process starts with the developer formalizing new requirements, for example, a change in current behavior or a new feature. This is usually done by creating tests in a testing framework of the programming language that the production code is written in.

The next step is implementing the requirements, which starts with the developer compiling and running the tests. In the beginning, this usually results in a number of compilation errors, because no production was written at this point. The programmer then solves all issues step-by-step by writing the code, while repeatedly running the test suite, until it compiles and all tests pass.

If we automate the implementation by offloading it to computers, software developers will have more time to create better specifications, further improving the software quality.

We focus on doing so with the use of grammatical evolution. Its goal is to find a computer program that passes all tests and behaves according to the specification. It is shown that assertions on expected values and values actually returned by tested programs can be used for determining the fitness of each solution.

The system was tested on generating the implementation of the `array_filter` function. That requires use of a loop, sequence of commands, and condition – the building blocks of imperative languages. Inputs of the system are a set of tests describing inputs and correct outputs of the function and a subset of grammar of the PHP language. Analysis of the experiments will be based on the number of generations needed to

evolve correct programs, the amount of time it took and code quality of the result.

This paper introduces general principles of grammatical evolution in Section 1. Existing libraries and research on automating test driven development with grammatical evolution are mentioned in Section 2.

Section 3 and 4 are dedicated to the overview of the proposed grammatical evolution library Gram and its application for automating unit tests in PHP. Experiments are presented and analyzed in Section 6. Results of this work and suggestions for future improvements are summed up in the last section of the paper.

2. Grammatical evolution

2.1 Introduction to grammatical evolution

Grammatical evolution is an evolutionary computation technique that combines genetic programming with constraints-based search, where the constraints are formulated via a context-free grammar. It can evolve complete programs in an arbitrary language using a variable-length integer vector [1]. The binary genotype determines which production rules of grammar are used to create the resulting program.

2.2 The genotype mapping process

The most commonly used notation of formal grammars in grammatical evolution is Backus-Naur form (see example in Listing 1).

```
(A)   <expr> ::= <expr> <op> <expr>      (0)
               | (<expr> <op> <expr>)    (1)
               | <var>                   (2)

(B)   <op> ::= +                         (0)
             | -                         (1)
             | *                         (2)
             | /                         (3)

(C)   <var> ::= x                        (0)
              | 1                        (1)
```

Listing 1. Grammar in Backus-Naur form

The genotype is used for mapping the start symbol of grammar onto terminals by using genes selecting an appropriate production rule with the following mapping function:

$$r = g \bmod R \tag{1}$$

where r is the chosen production rule, g is the value of current gene and R is the number of all possible rules for current non-terminal.

Consider non-terminal B from Listing 1, which has 4 possible rules. If the current gene has a value of 7, then the genotype maps to rule 3: `<op> ::= "/"`.

2.3 The algorithm of grammatical evolution

Every run of grammatical evolution begins with creating the first generation of individuals which is seeded either randomly or heuristically. The genotype of each individual is mapped to a string representation of a program in the chosen language.

The program is then evaluated and by comparing its outputs with the desired outputs and its fitness score is assigned. If no suitable solutions were found, the current generation is reproduced using the crossover and mutation operators. This process is repeated until a viable solution is found (see Algorithm 1).

```
Result: individual meeting requirements
individuals = initialize();
forall individual ∈ individuals do
    program = map(individual);
    results = evaluate(program);
    individual.fitness = calc_fitness(results);
end
while not good_enough(best(individuals)) do
    individuals = reproduce(individuals);
    forall individual ∈ individuals do
        program = map(individual);
        results = evaluate(program);
        individual.fitness = calc_fitness(results);
    end
end
result = best(individuals);
```

Algorithm 1: Grammatical evolution

3. Existing solutions

Evolutionary algorithms have been successfully used in combination with automated testing. Researchers are focused either on evolving test cases for existing software or on co-evolving both the production code and tests.

3.1 Evolving tests for existing software

While evolving tests for existing software can have a positive effect on code coverage [2], the research does not demonstrate its usability in real-world scenarios. It was shown that the evolutionary algorithm is able to generate test-cases that the current production code passes [3]. However, it is not clear how well the new tests reflect the real specifications. In fact, they may be misleading and make existing bugs in the code appear as features.

3.2 Co-evolving production code and tests

Co-evolving production code along with automated tests appears more pragmatic. A. Arcuri and X. Yao

were the first to propose a system that successfully evolved a program implementation based solely on tests [4] [5]. Their implementation used a considerably large number of testing data sets. Every five generations, the testing data were swapped to prevent overfitting.

Although changing the testing data every few generations is generally a powerful technique, automated tests in existing real-world code-bases usually use the same dataset for each run. This work therefore focuses on evolving software using the standard testing approach with static data thorough all generations.

4. New grammatical evolution library

4.1 Overview of the library

This section presents the proposed grammatical evolution software Gram that I have created as a project accompanying my bachelor thesis.

Gram is a C++ library designed with performance and extensibility in mind. The original goal is to build a system that solves symbolic regression. This paper shows another possible application of evolutionary algorithms in a more common real-life scenario.

4.2 Implemented algorithms

The initialization algorithm generates random genotypes, as existing research [6] shows no clear benefit of using heuristic methods such as ramped-half-and-half initialization.

The mapping algorithm uses wrapping when the genotype cannot be mapped to a complete program. This happens when the algorithm reaches the end of a genotype and non-terminals are still present in the program. In this situation, the genotype is read again from the beginning and the mapping process can continue. If the genotype cannot be mapped even after several wrapping events, the individual is assigned a very high fitness, which reduces its probability of reproducing.

Reproduction uses the tournament selection algorithm. The algorithm randomly picks a number of individuals from the current generation and chooses the best one for reproduction. After it has chosen two distinct individuals, they are combined by one-point or two-point crossover.

The crossover operators take a part of the genotype of the first individual and substitute it for a piece of the genotype of the second individual. Genes of individuals in the new generation can also undergo a random mutation with user-defined probability.

4.3 Included tools

The library comes with a parser of grammars in Backus-Naur form and a tool for communication with the command line.

Gram itself does not provide algorithms for evaluating generated programs and calculating their fitness, as they are dependent on the target language that is chosen by the user.

The whole code-base is covered by automated tests, can be compiled by both GCC and Clang and makes use of portable CMake build system.

5. Evolving programs in PHP using automated tests

5.1 The target language

The target language of choice for this work is PHP with its testing framework PhpUnit. PHP is suitable because of its weak typing and implicit type conversion. While not always desirable in real-world systems, these properties make the language more forgiving to semantic type-related errors that usually occur in code generated by grammatical evolution systems.

In our experiments, we could observe that although semantically flawed individuals usually have low fitness, they might not be far from a significantly better solution, which makes them potentially useful in future generations.

5.2 Infrastructure for program evaluation

Given that most of the algorithms required to implement grammatical evolution are part of the Gram library, applications making use of it only need to define the configurable parameters and implement algorithms for evaluation of individuals and fitness calculation.

The generated program is stored to a dedicated file on the disk. The PhpUnit tests are run through the command line tool and load the to-be-evaluated program. The program is then repeatedly called with sets of predefined input parameters. The testing framework then compares the real output of the program with the expected values. If not the same, they are saved in an XML log file.

After running all tests, the log file is loaded by the grammatical evolution runner. The serialized PHP literals are converted to native C++ objects and the system can calculate the fitness of the generated individual program.

5.3 Calculating fitness from test assertions

Fitness f of an evolved program p is computed with the following equation:

$$f(p) = \sum_{i=1}^{t} dist(e_i, a_i) \tag{2}$$

where t is the number of assertions in all tests, e_i is the expected value and a_i is the actual value the program returned. Determining distance *dist* of two variables in one assertion differs based on their type.

For numeric types, this function is used:

$$dist(e,a) = abs(e-a) \tag{3}$$

For strings, Levenshtein distance appears to work very well:

$$dist(e,a) = lev(e,a) \tag{4}$$

Distance of two booleans is calculated with the following function:

$$dist(e,a) = \begin{cases} 0, & \text{if } e = a, \\ C, & otherwise, \end{cases} \tag{5}$$

where C is a predefined constant.

For arrays, the following equation is used:

$$dist(e,a) = \sum_{i=1}^{n} dist(e_i, a_i) \tag{6}$$

where n is the number of items in arrays. In the case of arrays with a different number of items, the shorter array is normalized by padding with `null` values.

6. Experiment

6.1 Description of desired program

The goal of our experiment is to generate the `array_filter` function based on hand-written automated tests. The generated function must pass all tests for the experiment to be considered successful.

The first parameter of this function is an array of elements to be filtered. The second parameter is a lambda function that takes one argument – an element of the array – and decides whether it should be kept in the array or not. `array_filter` returns an array of elements that meet the filter requirements (see Listing 2). The generated function should make use of it to satisfy our specification (tests).

```
function array_filter(
    array $input,
    callable $filter
) : array;
```

Listing 2. Signature of the `array_filter` function

Table 1. Data used in tests of `array_filter`

Input	Correct output
`[]`	`[]`
`[-10, -5, -3, -1]`	`[]`
`[-10, -1, 3, 5]`	`[3, 5]`
`[1, 20, 42]`	`[1, 20, 42]`

Table 2. Experiment parameters

Initialization	Random
Genotype length	40 integers
Population size	200 individuals
Selection algorithm	Tournament
Tournament size	5 individuals
Crossover operator	One-point
Mutation operator	Integer-level
Crossover probability	1.00
Mutation probability	0.15
Fitness calculation	See Subsection 5.3
Success predicate	Fitness is 0

6.2 Parameters of grammatical evolution

The automated tests contain `assertEquals` calls that check if the tested function returned the expected result.

For our test, we created a filter that passes only positive numbers, as can be seen in Listing 3.

```
function ($element) {
    return $element > 0;
}
```

Listing 3. The `$filter` lambda function used in tests

The set of input values and correct output values is in Table 1.

The grammar we used defines a small subset of the PHP language. It contains rules that allow for generating functions, creating sequences of commands, loops, conditional statements, array initializations, pushing new elements to an array, working with lambda functions and returning values.

It also tries to compensate for the lack of type semantics in grammatical evolution by separating variable types into different rules, which appears to reduce the number of semantically incorrect programs.

The grammar does not contain rules for generating new identifiers – both `array_filter` parameters are hard-coded in the grammar along with few temporary variable names. With this measure, the system does not spend time on evolving new identifiers. It instead focuses of the basic control structures (commands, loops, conditions) that are far more important for our experiments.

For parameter configuration see Table 2.

Table 3. Stat measured in 100 runs

	Generations	Time [s]
Average	20.9	89.9
Median	14.5	71.0
Max	104.0	364.0
Min	0.0	8.0

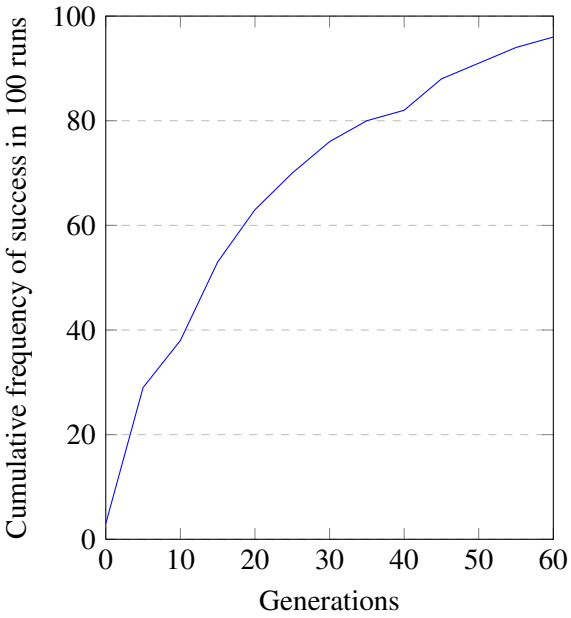

Figure 1. Cumulative frequency of success in 100 runs in generating `array_filter` function

6.3 Analysis of results

In this experiment, we repeated 100 independent runs of the system with the same parameters and the system was able to generate a viable program in all of them.

The number of generations necessary to complete varied quite a bit across the runs: in some cases the desired program was generated right in the first population by the random initializer while some runs required over 90 populations. For more measurements, see Table 3. For cumulative frequency of success across all runs, see Figure 1.

The result of majority of the runs looks like Listing 4. In some cases, the solutions contain some extra code that does not affect the fitness. That includes multiple array initializations after line 3 or additional code after the `return` statement on line 9 (dead code).

```
 1 <?php
 2 function array_filter($input, $filter) {
 3     $output = [];
 4     foreach ($input as $item) {
 5         if ($filter($item)) {
 6             $output[] = $item;
 7         }
 8     }
 9     return $output;
10 }
```

Listing 4. Evolved `array_filter` function

7. Conclusion

This work introduced a new grammatical evolution library. It implements all necessary algorithms, is easily extensible and can be used in various use-cases.

We used it on the problem of automating the test driven development. The system was able to generate complete program using only automated tests and a small subset of the target language – PHP.

The process of generating a program took on average 90 seconds, required 21 generations and produced code with only a small amount of dead code.

This system can be further improved by using static analysis tools to modify and reduce the grammar in run-time, rather than beforehand.

Acknowledgements

I'd like to thank my supervisor prof. Lukáš Sekanina for his valuable advice and friendly approach.

References

[1] M. O'Neil and C. Ryan. *Grammatical Evolution*. Kluwer Academic Publishers, 2003.

[2] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 2006.

[3] N. Gupta and M. Rohil. Using genetic algorithm for unit testing of object oriented software. *International Journal of Simulation Systems, Science & Technology*, 10(3), 2009.

[4] A. Arcuri and X. Yao. Coevolving programs and unit tests from their specification. In *Proceedings of ASE-2007: The 22th IEEE Conference on Automated Software Engineering*, 2007.

[5] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of 2008 IEEE Congress on Evolutionary Computation*, 2008.

[6] M. Fenton, J. McDermott, D. Fagan, S. Forstenlechner, M. O'Neill, and E. Hemberg. Ponyge2: Grammatical evolution in python. *ArXiv e-prints*, 2017.