Delft University of Technology
Software Engineering Research Group
Technical Report Series

A Genetic Programming Approach to Automated Test Generation for Object Oriented Software

Hans-Gerhard Gross, Arjan Seesing

Report TUD-SERG-2006-017

TUD-SERG-2006-017

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software

Arjan Seesing and Hans-Gerhard Gross

Delft University of Technology
EWI – Software Engineering Laboratory
Mekelweg 4, 2628 CD Delft, The Netherlands
a.c.seesing@gmail.com; h.g.gross@tudelft.nl

Abstract. In this article we propose a new method for creating test software for object-oriented systems using a genetic programming approach. We believe this approach is advantageous over the more established search-based test-case generation approaches because the test software is represented and altered as a fully functional computer program. Genetic programming (GP) uses a tree-shaped data structure which is more directly comparable and suitable for being mapped instantly to abstract syntax trees commonly used in computer languages and compilers. These structures can be manipulated and executed directly, bypassing intricate and error prone conversion procedures between different representations. In addition, tree structures make more operations possible, which keep the structure and semantics of the evolving test software better intact during program evolution, compared to linear structures. This speeds up the evolutionary program generation process because the loss of evolved structures due to mutations and crossover is prevented more effectively.

Keywords: Search-based Testing, Test Automation, Object-Oriented Programming

1 Introduction

Testing is the most widely used and accepted technique for verification and validation of software systems. It is applied to measure to which extent a software system is conforming to its original requirements specifaction and to demonstrate its correct operation [11]. Testing is a search problem that involves the identification of a limited number of good tests out of a shear, nearly unlimited number of possible test scenarios. "Good tests" are those runtime scenarios that are likely to uncover failures, or demonstrate correctness of the system under test (SUT). Identifiying good test cases typically follows predefined testing criteria, such as code coverage criteria [3]. This is based on the assumption that only the execution of a distinct feature, or its coverage, can reveal failures that are associated with this feature.

Because the primary activities of testing, test case identification and design, are typical search problems, they can be tackled by typical search heuristics. One of the most important search heuristics for software testing is known to be random testing. This is also one of the most commonly used testing strategies in industry today. Recently, also more advanced heuristic search techniques have been applied to software testing. These are based on evolutionary algorithms, and they are also slowly making their ways into industry [1, 2] since their performance in finding test cases was found to be at least as good as random testing, but usually much better [8, 22] The group of these testing techniques is referred to as evolutionary testing (ET) according to Wegener and Grochtmann [26]. ET is an automatic test case generation technique based on the application of evolution strategies [17], genetic algorithms [5, 10], genetic programming [13], or simmulated annealing [24]. ET searches for optimal test parameter combinations that satisfy a predefined test criterion. This test criterion is represented through a "cost function" that measures how well each of the automatically generated optimization parameters satisfies the given test criterion. For a test, various test criteria a perceivable, according to the goal of the test, such as how well a test case covers a piece of code, in the case of structural testing [12, 16], or how well a test case violates a (safety) requirement [23], for example.

Evolutionary testing has inititially only been applied to traditional procedural software. Here, ET is used to generate input parameter combinations for test cases automatically that achieve, i.e., high coverage, if the test target relates to some code coverage criterion. However, recently, also object-oriented software testing based on evolutionary testing has been tackled by researchers [6, 7, 21]. The two main differences are

- that object technology is inherently based on states which are not readily visible outside of an object's encapsulating hull,
- and that an object test, as the basic unit of testing, can incorporate an arbitrary number of operation invocations.

An object's internal state depends on any previously performed operation invocations, the so-called invocation history [9, 7], including input parameter settings. Hence, object testing involves not only the generation of suitable input parameter combinations for a single procedure under test, but, additionally, the generation of suitable test invocation sequences of various operations of an object, plus the generation of their respective input parameter combinations. As a consequence, in object testing, we have to deal with a number of test artifacts, such as the sequence or combination of operation invocations, the input parameter combinations for the tested operations, the sequence or combination of operation invocations that bring the object into an "interesting" initial testing state, including constructor invocation, and the input parameter combinations for the previously mentioned state setting operations. When applied to objects, ET must therefore generate optimization paramter values that correspond to a constructor, with input parameters, a number of operations, including input parameters, that bring the object into a distinct state, and a number of operations, including input parameter values, for the actual tested functionality.

Object-oriented evolutionary testing can actually be regarded as search-based test software programming. In other words, the problem is suitable to be solved through genetic programming techniques. Genetic programming can be seen as a specialization of a genetic algorithm, and it is particularly aimed at evolving software programs according to the rules of simulated natural evolution. Genetic programming is well suited for test program generation for testing object-oriented code because the symbols used by GP are restricted only to operation invocations and input parameter types that are required and used by the SUT. This is quite in contrast to the generation of arbitrary functional code which is based on the full alphabet of the programming language under consideration. So, test code generation through genetic programming is much less complex.

In this paper, we introduce and explore an evolutionary testing approach for object oriented code that is based on the application of genetic programming. In the next section, Section 2, we will introduce evolutionary testing and explain briefly how it is typically applied under the procedural programming paradigm. Section 3 looks at related work, and it introduces two different approaches to apply genetic algorithms to the testing of object-oriented code. In Section 4 we will introduce our approach that applies genetic programming to the generation of test software for object oriented code. Here, we explain the details of the algorithm used and show a few very small examples of how to use genetic programming for the purpose under consideration. Section 5 shows some resutls from experiments that we have carried out, and Section 6 presents out conclusions and gives an outlook on future research.

2 Automated Software Testing and Evolutionary Algorithms

Traditionally, a human tester develops the test scenarios and writes the test code for an SUT manually. Ideally, a testing tool should generate the entire test code automatically, but this is very difficult to achieve, so that only parts of the testing process can be automated. The process of test automation can be subdivided into three main activities:

- Generation of test scenarios according to testing criteria, also referred to as software test data generation.
- Generation of a test oracle out of the SUT's specification.
- Combination of both, test secnarios and oracle, into executable test cases.

The first activity can be automated with relative ease, and this is also what most commercial testing tools are capable to do, although, usually, existing tools apply crude heuristics to find test scenarios. The automation of the second activity is usually much more daunting in practice, due to poor quality, or low formality, of the SUT's requirements specification available. Without formalization of the specification, it is nearly impossible to automate the generation of the oracle. The last step simply involves the creation of an arbiter that compares the observation from the SUT's execution with the expected observation from the oracle, and

decides whether the test passes or fails. This poses no particular difficulty on automation, once the oracle problem has been solved.

The work outlined in this paper and the related work presented, concentrate on the first testing activity, the automated generation of the test scenarios. An efficient way to do this is with a random generator. Random testing can be used to create a volume of test scenarios, but it is not specifically obeying any test coverage critiera. Test tools based on random testing, generate test scenarios and simply measure and illustrate the coverage of the SUT. They cannot generate test scenarios that are "guided" by the coverage.

Evolutionary Testing for Procedural Code

More advanced search heuristics such as evolutionary algorithms (EA) can be used to specifically look for test scenarios that cover certain branches of a program. This class of algorithms is loosely related to the mechanisms of natural evolution, and they are based on reproduction, evaluation and selection. The following pseudo code represents a standard genetic algorithm that can be used for testing; P, P1, P2, and P3 represent populations of feasible test scenarios:

```
initialize_random (P);
fitness_function (P);
while not stopping_criterion do begin
  P1 = selection (P);
  P2 = recombination (P1);
  P3 = mutation (P2);
  fitness_function (P3);
  P  = merge_populations (P3,P);
end-while
```

Each set of parameters, the so-called individual, is represented by a different binary string, the so-called chromosome, within a population. Each chromosome represents the input parameter values for the execution of the SUT. The GA starts with a random initial population of chromosomes. Selection chooses the chromosomes to be recombined and mutated out of this initial population. Recombination reproduces the selected individuals and exchanges their information (pair-wise) in order to produce new individuals. This information exchange is called crossover. Mutation introduces a small change to each newly created individual. The resulting individuals (P3 in the pseudo-code) are then evaluated through the fitness function. This transfers the information encoded in the chromosome, the so-called genotype, into an execution of the SUT, the so-called phenotype. The fitness function measures how well the chromosome satisfies the test criterion. In our case this is the coverage of a program branch. The implementation of the fitness function follows earlier standards in evolutionary testing, described in other articles, i.e., [12, 14, 15]. For the next generation, the old and the new populations are merged, thereby retaining the best individuals. The process of selection, reproduction and evaluation is referred to as one generation, and these steps are repeated until the stopping criterion is satisfied, e.g., a

predefined number of generations, or the satisfaction of the test criterion. Fitter individuals, represented by their chromosomes, that come closer to covering the current target are favored in the recombination and selection process, so that in subsequent generations, the population will comprise fitter individuals that are more likely to satisfy the test criterion. In order for such a search process to obtain full branch coverage, every branch must be successively selected as target and solved through an individual search process. The application of evolutionary algorithms to such structural testing problems has been demonstrated in practice, i.e. [1, 12, 15, 16].

The previously described simple representation of input parameter values in a chromosome, is not sufficient in object-oriented software testing. Here, in addition to the input parameter values, the search process also needs to include any arbitrary number and sequence of operation invocations on the object, and any internal state settings, as described earlier. This turns the fixed-length, simple chromosome of the procedural paradigm into an arbitrary-length, complex chromosome for the object-oriented paradigm. Especially, the fact that initial state settings of an object are part of a test scenario [9], makes the implementation of the automatic test generation process more difficult.

Recent publications on evolutionary testing of object-oriented systems have proposed encodings to deal with this additional dimension. The first alternative is to encode the operation invocation sequences as chromosome and come up with new recombination and mutation strategies [21]. The second alternative is to use a standard binary encoding of the chromosome, so that standard GA tools can be used, and devise a specialized so-called "genotype–phenotype transfer function" that maps the chromosome representation to a test scenario [25]. These are briefly laid out in the following section, before we go on to propose a third way of organizing the chromosome as a tree structure, in Section 4, so that we can apply a standard genetic programming technique.

3 Evolutionary Testing for Objects

3.1 Object-Specific Chromosome Encoding

One way to deal with the enhanced complexity of objects in evolutionary testing is to enrich the chromosome with representations that are capable to deal with these more complex entities (Tonella, [21]). This method adds structure to the chromosome during evolution, that can be mapped directly to an executing program. Tonella proposes the following grammar:

```
<chromosome> ::= <actions> @ <values>
<actions>    ::= <action> {: <actions>}?
<action>     ::= $id = constructor ( {<parameters>}? )
               | $id = CLASS # NULL
               | $id . method ( {<parameters>}? )
<parameters> ::= <parameter> {, <paraemters>}?
<parameter>  ::= builtin-type {<generator>}?
```

```
                | $id
<generator>  ::= [ low ; up ]
                | [ genclass ]
<values>     ::= <value> {, <value>}?
<value>      ::= integer
                | real
                | boolean
                | string
```

The "@" separates the chromosome into two parts. The first part contains the sequence of operation invocations, including constructor and method invocations, each separated by ":", and the second part represents the input values that these operations take, each separated by ",". Such a sequence of operation invocations plus parameter values represents a test scenario. An <action> can represent either a new object (indicated as `$id`), or a call to a method on an object identified by `$id`. Parameters of operation invocations (<parameters>) can represent built-in types such as `int`, `real`, `boolean`, and `string`, or chromosome variables (`$id`). The generation operator (<generator>) produces the values for the input parameters. It can generate random numbers in the range between `low` and `up`, or it can use an external class to have a value produced. The grammar proposed permits invalid chromosomes, so additional rules must be imposed for "well-formedness":

- chromosome variables cannot be used before they are assigned.
- built-in types in the first part require a corresponding input value in the second part of the chromosome.
- methods used in the chromosome must be visible for the used classes.

Because the genetic algorithm performs on chromosomes with this particular organization, the standard binary crossover and mutation operators may not be applied. Tonella proposes specific operators that lend their ideas from genetic programming. Mutation can change values or operations (constructors and methods). A value can be mutated through change to a randomly generated value of the same type. A constructor can be mutated through random change to another constructor. Redundant input values are then dropped, missing ones generated. A new method may be inserted by a mutation including the respective input paramter values for the method. A method may also be removed through a mutation including all its input values.

Crossover between two chromosomes works in a similar way, although it usually involves various of the previously described measures at the same time. Two chromosomes are cut at a randomly determined location (at an <action>-delimiter), in the case of a simple one-point crossover, and their respective tails are swapped and rejoined. Redundant constructors must be removed, as well as needless input values, and, finally, conflicting variable names must be changed.

3.2 Object-Specific Genotype–Phenotype Transfer

An alternative way to apply evolutionary testing to the more complicated requirements of object technology, is to maintain a binary or numerical chromosome that can be handled by any standard genetic algorithm, and then provide rules, or a grammar, to map the binary representation into a test scenario. Each test program may be represented as a sequence of statements, and each statement consists of an object, an operation, constructor or method, and some parameters (Wappler and Lammermann, [25]). The mapping of the chromosome to test scenarios can be determined by sequentially reading the chromosome and turning it into operation invocations according to rules. Two genes can be assigned for operation invocations, one for the target object, and one that denotes the operation to be invoked on that object. Because operation invocations take varying numbers of input parameters, input values must be accomodated by a variable number of genes. The genes are then mapped into a real test scenario, a phenotype, according to the production rules of a grammar:

```
test_program ::= {statement;}+
statement    ::= [return_value]{constr_call|method_call}
return_value ::= class_name instance_name =
constr_call  ::= new class_name (parameters)
method_call  ::= {class_name|instance_name}.method_name(parameters)
paramters    ::= [parameter {, parameter}*]
parameter    ::= basic_type_value|instance_name|NULL
```

In this system `[]` represents an option, `{|}` alternatives, `{}+` at least one repetition, and `{}*` arbitrary repetitions. Because these rules allow the generation of erroneous test scenarios, the fitness function assigns a degree of failure to the decoding This failure is part of the fitness, so that such "defective genetic material" is eventually evading from the population.

The decoding from the chromosome into a real test scenario is performed through specific functions, fully described in [25]. Methods are numbered in a series, and each number of one gene in the chromosome corresponds to a specific number of a method. Input parameters are represented by one gene in the chromosome, and they can map to concrete values and objects.

4 Proposed Genetic Programming Approach

Genetic programming (GP) is a specicalization of a genetic algorithm that is particularly aimed at evolving computer programs based on the principles of natural evolution [13]. The chromosomes in genetic programming represent hierarchically structured computer programs made up of arithmetic operations mathematical functions, boolean and conditional operations, and terminal symbols, such as types, numbers, and strings. The genotype-phenotype mapping of GP is much more natural for the domain of test program generation compared with a standard genetic algorithm. The fact that GP is based on hierarchically

organized trees requires specialized genetic operators for recombination and mutation [13].

Recombination takes sub-trees from previously selected parent individuals and swaps them in order to reorganize them into new individuals (trees). The chromosomes are always cut and reassembled at nodes, and not wihtin nodes of the tree representing the computer program. The mutation operator introduces random changes in the tree by selecting a node of the tree randomly, deleting everyting beyond that node, or adding a randomly generated subtree, or changing leaves of the tree randomly. These are all standard GP operators according to [13].

4.1 GP Chromosome

Table 1 lists the basic classes of representations that are used in our proposed genetic programming approach.

Table 1. Function set for the GP to chose from

Node Name	Description	Can have Children	Can be Terminal
L-Variable	Variable definition	yes	no
R-Variable	Reference to an L-Variable	no	yes
Constant	A primitive value (int, double, ...)	no	yes
Constructor	Creates an object	yes	yes
Method	Calls an object's method	yes	no
Field Assignment	State change of an object	yes	no
SUT	Subject under test	yes	no
Array	Creates an array of objects	yes	no
NULL	Keyword, implemented in the constructor	no	yes

The types to be used by GP are arbitrary, because every single object that is created, represents a type. Every operation refers to an object, and thus a type, plus some input parameters, including their individual types. These must be created by the GP process and added as leaves to the node in the tree that represents the operation. Each operation maps to a subtree of the entire GP hierarchy, including constructors and input values for the required (sub-)objects. Apart from arbitrary object types, we also have to allow basic or primitive types, such as boolean, integer, real, and the like. These are primarily used in order to denote input and return values.

Because of the late binding principle in object technolgy, not all types are known to the GP system a-priori. The SUT is used as the starting point of the GP system. It indexes all its operations, that it has to test and which it can use to change its state. All the classes it references in the signatures of its operations are also indexed, plus all subsequent classes used by these. This indexing is

performed recursively until all classes needed for the test case are loaded and known to the GP system. Abstract parameter types, or interfaces, and classes that extend or implement these must be added manually to the index of the GP system. That is, only if they are not referenced by some other already existing and indexed class. Figure 1 shows an example tree-shaped representation of

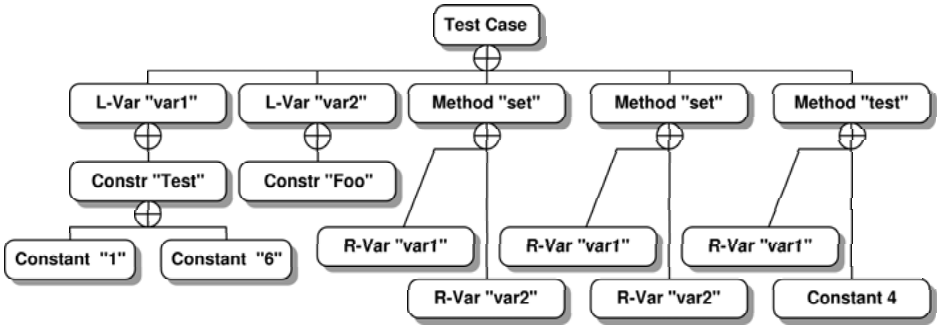

Fig. 1. Example tree-shaped representation of a GP-chromosome; the anchor symbol indicates containment

a GP-chromosome that translates into the following Java testing code snippet (moving from left to right).

```
var1 = new Test(1,6);
var2 = new Foo();
var1.set(var2);
var1.set(var2);
var1.test();
```

There are two types of variables in programming languages, L(eft)-type and R(ight)-type variables. L-type variables define and initialize variables, and R-type variables reference them. The compiler will issue an error, when a variable is used as an R-type, before it has been used (initialized) as an L-type. R-types are terminal, and the L-types require one "child-node".

4.2 Object Reflection

In order for GP to work properly and generate valid testing code, it needs rules, on the basis of which it can recombine existing nodes and generate new nodes. In traditional genetic programming, the grammar usually comprises all constructs of the programming language used [13].

Test code is usually straight-forward, and all it needs to do is to invoke a certain sequence of operation invocations with parameter values, including the creation and initialization of the variables used. The rules are restricted to the operations of the SUT plus the objects and return types that it uses in these

operations. In Java, these can even be generated at runtime through Java's built-in reflection mechanism [4]. This information is then stored in a repository of basic symbols which represents the rules that the genetic programming algorithm can use to generate test programs. Earlier, we referred to this repository as the GP index.

The hierarchical structure of testing code is typcially flat, like the one displayed in Fig. 1. This flat hierarchy is due to the fact that testing has a more sequential nature, by calling one operation after another, leading to a single path through the test program. This is different from "normal functional code" that is usually made up of conditional executions, leading to various paths through the program. Every operation invoked is attached as subtree close to the root node of the entire chromosome tree. Extensive hierarchical structure is only exhibited if operation invocations require objects as input parameters, although this can be circumvented by imposing flat hierarchies. This is described below.

4.3 Detailed Genetic Operators

Initial Population. Two different methods are used in order to create the first population: random population, or a population based on execution traces. The first method selects initial operation invocations including their input values randomly. The second method applies existing knowledge from executing the SUT. Here, we can form an initial population from already known typical usage scenarios of the SUT. This leads to an initial population that can already cover many of the SUT's runtime paths for typical usage profiles. This method improves the performance of the test generation considerably.

Mutation. GP requires a separate mutation operator for each individual basic building block, each of which may be subject to mutation according to a predefined mutation rate. We can distinguish three types of mutation operators, one that creates a new building block, one that changes an existing one, and one that deletes a building block. We have devised these three operators for each of the functions in Table 1.

A constructor can be created, deleted or changed to a different constructor. Creation or deletion implies that their respective sub-trees, comprising input parameters, are created or deleted. The same principles that apply to constructors apply also to other operations, the normal methods of an object. They can be added or removed, or their input values can be changed. Constructors and normal methods are different only in the way that we need at least one constructor in order to create the object, and the constructor must always be invoked before any other operation.

Some methods take objects as arguments. These objects need to be created through a constructor and, maybe, also their operations need to be invoked. This principle may be applied recursively, depending on the operations required, thus potentially leading to constructor compostions of arbitrary depth. We have decided not to permit the generation of such hierarchies, and move the composite

constructor sub-tree up towards the root node. This is illustrated in Fig. 2. The object constructor can then be moved to a position in the tree where it is executed before the object reference is used as input value. It is important to note that there is no reason for restricting the composition depth other than controllability of the experiments. It makes it easier to understand what the GP-algorithm is doing and to control and assess its behavior.

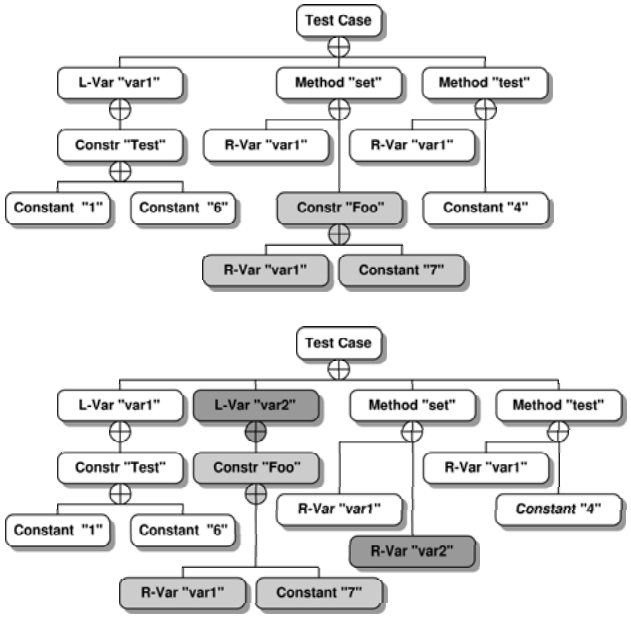

Fig. 2. Tree flattening activity

Crossover. In GP, unlike genetic algorithms, the crossover is only applied at nodes in the chromosome tree, and not at leaves. The nodes for crossover are determined randomly for each of the two participating individuals, or through a search for distinct nodes. If the two crossover nodes are compatible, the crossover operator simply exchanges the entire subtrees. Two subtrees are compatible, if the types of the two root nodes of the candidate sub-trees are the same, and a search through the tree can actually determine feasible nodes of the same type. Compatibility is always given at the root node level of the entire chromosome trees. The simplest crossover is performed at the method level, thus exchanging entire methods including input parameters. Input parameter nodes can also be exchanged, given that they have the same types, and constant values can be swapped between individuals. Figure 3 illustrates an exchange at the root level, swapping entire sub-trees of methods.

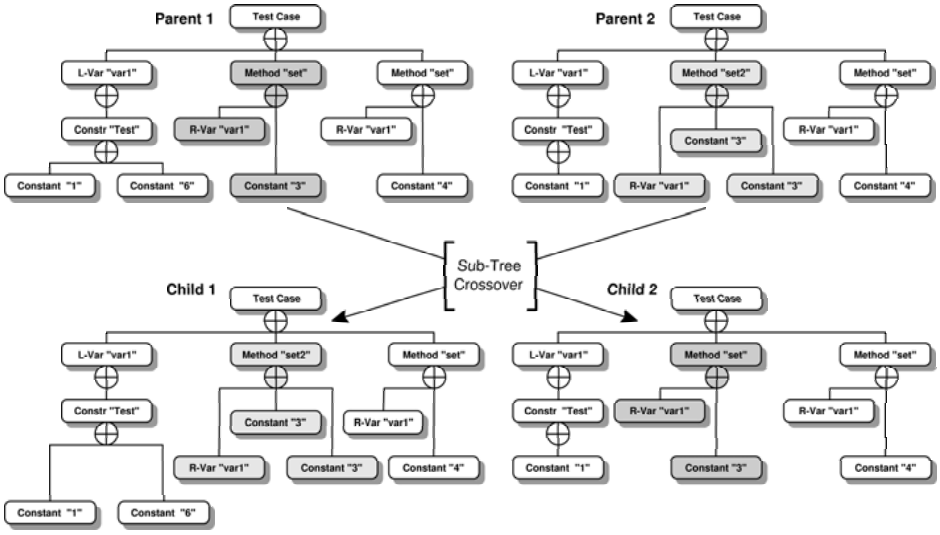

Fig. 3. Tree-based crossover of Genetic Programming

Crossover and mutation can generate chromosomes of arbitrary length over time, simply by adding more and more sub-trees. This is not desirable, so overgrowth of the chromosome must be regulated through the introduction of a penalty on the overall fitness for larger individuals. This turns our approach into a multi-objective evolutionary algorithm, although, here, size of the test case is the second optimization objective, thus putting selective pressure on the generation of short test scenarios.

4.4 Genotype/Phenotype Transfer and Program Execution.

We are using coverage metrics to indicate an individual's fitness [12, 16]. There are to approaches to execute an individual and obtain coverage information. The first one generates the test program code, for example as Java source or byte code, after which it will compile and execute it. The second one uses the reflection mechanism in Java. This allows us to skip the creation, compilation and class loading steps present in the first approach. For example, the method node has as its children a node which creates an object to call a method on (if it is not a static method), and nodes to create the arguments it might need. Although, reflection calls are much slower than the normal Java calls, using reflection compensates for the additional overhead of creation, compilation, and loading of a normal Java class.

5 Experiments

In order to demonstrate the applicability of our proposed test case generation technique based on GP, we have applied it to 5 test programs, XMLElement, an

XML parsing package including a number of classes, and from the Java collection classes, HashMap, BitSet, TreeMap, and TreeTokenizer. All test were executed on a 2.1 GHz Athlon XP under Java 1.6 [20]. Mutation was set to 70% method introduction, 15% method removal, and 15% variable introduction. The results displayed in Table 2 demonstrate the advantage of the GP approach over a traditional random testing strategy. Only for the smallest SUT, StringTokenizer, the random testing technique could generate the same high coverage (100%) as our GP approach. For the other examples, the GP approach achieves much higher test coverage. The columns time in seconds, Time(s), give an indication of how much more processing time is required for the GP, which is a much more complex algorithm, compared to the random generation. A more thorough discussion of these experiments can be found in [19].

Table 2. Tested SUTs, comparison between GP-based testing and random testing.

SUT		GP testing		Random testing	
Name	Branches	Coverage(%)	Time(s)	Coverage(%)	Time(s)
BitSet	124	100	495	86	133
XMLElement	121	90	369	80	101
HashMap	50	94	180	72	43
TreeMap	39	92	13	46	29
StringTokenizer	5	100	5	100	2

6 Conclusions and Future Work

The purpose of this paper is the proposition of an genetic programming approach to generate test software for object-oriented systems automatically. We have not applied the techniques presented to extended problems, so experiments that we have performed may only be regarded as a proof of concept and an initial step towards a more extensive application. The main improvement, or advantage, of our proposed method over the other two described approaches [21, 25] is that the test software is already represented and altered as a fully functional computer program. This means that the experience gained in genetic programming can be utilized to create these test cases. Genetic programming proposes many more different types of mutations and more robust cross-over algorithms which are designed to keep the structures they alter semantically correct, preventing loss of evolved structures. Genetic programming uses tree structures which are more similar to the abstract syntax trees used in computer programs. This leads to more powerful programs, because certain tests are impossible to create in linear programs, and to a simplification of the execution of generated test cases. Genetic programming is specifically geared toward program generation, and this makes its application to test software generation so straight-forward.

Test software generation for object-oriented Java code as it is introduced in this article, through its very nature, is much easier to perform than the GP-

based generation of abritrary functional software. This is because the alphabet of the GP system is not the entire alphabet of the programming languge under consideration, but merely the methods and input parameters of the object under test. And these can even be retrieved automatically. At least, this is the case for a modern object-oriented development environment like Java.

Although, the number of different operation types is quite limited, large classes which contain many methods will lead to huge hierarchical trees. This increases the search space drastically that the genetic programming algorithm has to work its way through. Future work will be geared toward limitation of size and complexity of the search space as much as possible. This can already be done manually, by skipping methods which do not alter the state of their object, so-called pure methods [19]. An extension to our automatic testing system may detect such methods and remove them from the GP alphabet.

The performance of genetic algorithms is not only influenced by their internal data structures and their alleged operators, but even more so by an efficient fitness function. The fitness function used for the experiments is only a very crude implementation of the standard fitness function proposed for coverage-based evolutionary testing. There is definitely still leeway for improvement. Also, the various mutation parameters need to be adjusted to achieve optimal performance. This work may be seen as an initial step towards object-oriented test program generation based on genetic programming.

References

1. A. Baresel et al. Structural and Functional Sequence Test of Dynamic and State-Based Software with Evolutionary Algorithms. In: Proc. of the Genetic and Evolutionary Computation Conference, Chicago, Illinois, USA, July 2003.
2. O. Buehler and J. Wegener. Evolutionary Functional Testing of an Automated Parking System. In: Intl Conf. on Computer, Communication and Control Technologies: CCCT '03 and the 9th. Intl Conf. on Information Systems Analysis and Synthesis: ISAS '03, Orlando, Florida, July 31, August 1-2, 2003.
3. B. Beizer. Software Testing Techniques. Van Nostrand Reinhold, 1990.
4. I.R. Forman and N. Forman. Java Reflection in Action. Manning, October 2004.
5. D.E. Goldberg. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading, 1989.
6. H.-G. Gross and N. Mayer. Evolutionary Testing in Component-Based Real-Time System Construction. In: Proc. of the Genetic and Evolutionary Computation Conference, New York, July 8–14, 2002.
7. H.-G. Gross and N. Mayer. Search-Based Execution Time Verification in Object-Oriented and Component-Based Real-Time System Development. In: *P*roc. of the 8th IEEE Intl. Workshop on Object-Oriented Real-Time Dependable Systems, Guadalajara, Mexico, January 15-17, 2003.
8. H.-G. Gross. An Evaluation of Dynamic, Optimisation-based Worst-case Execution Time Analysis. Proceedings of the International Conference on Information Technology: Prospects and Challenges in the 21st Century, Kathmandu, Nepal, May 2003.
9. H.-G. Gross. Component-Based Software Testing with UML. Springer, Heidelberg, 2005.

10. J. Holland. Adaptation in Natural and Artificial Systems. MIT Press, Cabmridge, MA, 1975.
11. IEEE. Standard Glossary of Software Engineering Terminology. Volume IEEE Std. 610.12-1990. IEEE, 1999.
12. B. Jones et al. Automatic Structural Testing Using Genetic Algorithms. Software Engineering Journal, 11(5), 1996.
13. D. Koza. Genetic Programming, On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, 1992.
14. P. McMinn. Search-Based Software Test Data Generation: A Survey. Software Testing, Verification and Reliability, 14(2), pp. 105-156, 2004.
15. C. Michael, G. McGraw and M. Schatz. Generating Software Test Data by Evolution. IEEE Transaction on Software Engineering, 27(12), December 2001.
16. R. Pargas et al. Test data generation using genetic algorithms. Software Testing, Verification & Reliability, 9(4), 1999.
17. H.P. Schwefel and R. Männer. Parallel Problem Solving from Nature. Springer, Berlin, 1990.
18. A. Seesing. An Overview of Automatic Object-Oriented Test Case Generation using Genetic Programming. Internal Report, Research Assignment, EWI – Delft University of Technology, April 2005.
19. A. Seesing. EvoTest: Test Case Generation using Genetic Programming and Software Analysis. Msc Thesis, Delft University of Technology, June, 2006.
20. Sun Microsystems. Mustang (Java 6.0 beta). https://mustang.dev.java.net.
21. P. Tonella. Evolutionary Testing of Classes. In: Proc. of the 2004 ACM SIGSOFT Intl. Symposium on Software Testing and Analysis, pp. 119–128, Boston, July 11–14, 2004.
22. N. Tracey, J. Clarke, and K. Mander. The way forward in unifying dynamic test case generation: The optimisation-based approach. In Proc. of the IFIP Intl Workshop of Dependable Computing, South Africa, January 1998.
23. N. Tracey et al. Integrating Safety Analysis with Automatic Test-Data Generation for Software Safety Verification. In: Proc. of 17th International System Safety Conference, August 1999.
24. P. von Laarhoven and E. Aarts. Simmulatd Annealing: Theory and Applications. Mathematics and its Applications. Kluwer, Dordrecht, 1987.
25. S. Wappler and F. Lammermann. Using Evolutionary Algorithms for the Unit Testing of Object-Oriented Software. In: Proc. of the Genetic and Evolutionary Computation Conference, Washington D.C., June 25–29, 2005.
26. J. Wegener and M. Grochtmann. Verifying timing constraints by means of evolutionary testing. Real-Time Systems, 3(15), 1998.

TUD-SERG-2006-017
ISSN 1872-5392