EvoTest: Test Case Generation using Genetic Programming and Software Analysis

Arjan Seesing

Delft University of Technology

22nd June 2006

EvoTest: Test Case Generation using Genetic Programming and Software Analysis

Master's Thesis in Computer Science

Software Engineering group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Arjan Seesing
a.c.seesing@gmail.com

22nd June 2006

Author
Arjan Seesing
Title
EvoTest: Test Case Generation using Genetic Programming and Software Analysis
MSc presentation
July 11, 2006

Graduation Committee

prof. dr. ir. A. v Deursen (chair)	Delft University of Technology
ir. dr. H.G. Gross	Delft University of Technology
ir. dr. C. Witteveen	Delft University of Technology

Abstract

Testing in diverse software development paradigms is an ongoing problem in software engineering. Many techniques have been devised over the past decades to help software engineers create useful testing suites. In this thesis, the focus is on test case generation for object orientated software using genetic programming. The automatic creation of test data is still an open problem in object-oriented software testing, and many new techniques are being researched. For object orientated software, the automatic test data generation technique is not sufficient, because besides input data used for testing, it additionally has to produce the right sequences of method calls, and the right artefacts, to bring the object under test in the required state for testing.

Genetic algorithms have already been used to tackle typical testing problems with success, but the use of genetic programming applied to automatic test case generation is relatively new and promising. This master thesis shows how genetic algorithms combined with different types of software analysis can create new unit tests with a high amount of program coverage. Together with static analysis, the genetic algorithm is able to generate tests for more real world programs in a shorter amount of time. This new approach is implemented in a prototype tool called EvoTest. With this tool I demonstrate the coverage obtained for small programs and some larger real world programs.

Preface

In 2005, I started my research assignment. Hans-Gerhard Gross, my adviser and Arie van Deursen, my professor, were doing research in software testing, which was not really my calling. But interesting science happens where different fields come together. So the idea of combining software testing and genetic algorithms came to me. I always have been interested in machine learning techniques, such as neural networks, but I have never done anything substantial with genetic algorithms. But only during my internship at Georgia Institute of Technology, was I able to complete the project I started at TUDelft. They had the tools and expertise I needed to implement EvoTest.

Acknowledgements

Thank you Alex Orso, my adviser in Atlanta, for taking me under your wing during my time there. Without your support, EvoTest would still be only an idea in my head.

Thank you Jim Clause, my colleague during my internship at GATech, for your help on EvoTest, and especially the instrumentation. Without you, many bugs would still haunt me today.

Thank you Hans-Gerhard Gross, for reading and fixing this lengthy thesis. Not many would take so much time, marking all my spelling and punctuation mistakes and helping to make this the best work I made to date.

And finally, I want to thank the whole genetic algorithm community for the ideas and inspiration they gave me.

Arjan Seesing

Delft, The Netherlands
22nd June 2006

Contents

Chapter 1

Introduction

At the beginning of 2005 I read the paper of Tonella [44] on using a genetic algorithm to create unit tests. I recognized the technique he used as genetic programming (GP), because I had read about GP from in one of Koza's books [30]. I saw several approaches which could improve his techniques. For my research assignment, I started working on my own system which created test cases with several different approaches. But due to time constraints I was not able to finish it. I was allowed to finish the project later that year when I was working at Georgia Institute of Technology. This thesis describes the project I finished there, which resulted in a prototype tool called EvoTest.
This chapter will introduce several key concepts used in this thesis, such as a short introduction to testing and genetic algorithms. At the end it will formalize the problem statement and give an outline for the rest of this thesis.

1.1 Testing and Test Case Generation

Testing is an important field in software engineering. It is the process which wants to show the correctness, completeness, security and quality of developed computer software [1]. This is a very ambitious goal, because just the halting problem alone will prevent any kind of conclusive test for any arbitrarily software program. The halting problem will prevent the automatic detection of all different kinds of unwanted infinite loops.
So in practice, testing is only able to prove the presence of faults and not the absence of faults [20]. This is traditionally done by putting the software product through its paces. This is normally done by a tester who probes the software and evaluates the output by checking if it is up to specifications. This can be done on many levels and many stages of development. For example, potential users can evaluate the almost finished product, or small testing programs can invoke certain functions deep in the software and check their output directly, so called unit tests.

1.1.1 Regression Testing

This thesis was started with the assumption that there is a legacy software system which needs to be changed and that there is no test suite available or only a limited one. This is a problem because when programming code is changed, often new errors will be introduced, these new errors are called regressions [20, 39].
A regression is when the old system performed the task correctly, but because of a change the new system fails this task. These kinds of errors are caught by regressions tests. A form of a regression test

suite is a large suite of unit tests. The choice for the development team is, to create regression tests by hand before they start developing the new features, or starting immediately on the new features and taking the risk of introducing regressions.

Unit Tests

A regression test is normally a suite of unit tests, executed every time the system changes during development. A unit test is a procedure used to validate that a particular module of source code is working properly, such as a class or method in object oriented code. The idea about unit tests is to write test cases for all functions and methods so that whenever a change causes a regression, it can be quickly identified and fixed because the test identifies the faulty function.

1.1.2 Testing is Expensive

Because software is becoming more complex every year, these software systems need to be tested more thoroughly if the quality has to reach a certain standard. This comes at a cost, a great cost, often at least 50% of all costs are spent on testing [9]. Each part of the testing process which can be automated can reduce costs significantly. Testing software involves many steps and can be done in many different ways. There are unit tests, system tests, integration tests and regression tests, just to name a few. Automation can be done by automating the process of running the tests by special tools, for example every time a developer submits a new piece of code, or during the compile debug cycle. But automating the creation of the tests is a much more ambitious goal which this thesis will focus on. The automatic creation of tests is called test case generation.

1.2 Automatic Testing

There are many approaches to generating tests [11, 27], using different strategies. Most tools take the software which has to be tested as input and create a set of tests for that software as their output. It can, for example, detect states of the system and make tests for every state. Or it tries to generate a separate test for all the different paths through the code, thus testing every feature. Both techniques try to get the system in different states, so those states can be checked to be correct. The tool checks if the system is in the correct state when it should, and that it will act correct when it is in that state. State is essentially a snapshot of the measure of various conditions in the system. In general, the state of a program can be represented by a complete memory dump of the running program, combining the stack, heap and program counter. It can also be defined by sequence of instructions the program executed up to a certain point. It is clear that even a trivial program probably has too many different states to explore. To make testing of states possible, a selection of all possible states has to be made. This is similar to the case when a human is testing a computer program. He or she will test several representative cases, boundary cases and exceptional cases in the assumption that when those states are tested and found correct all states will be correct. When the automatic testing tool made its selection of states it is going to test, it also needs to be able to bring the program in those states before it can test them. Bringing the application in the correct state is basically a search problem.

1.2.1 Testing is a Search Problem

To be able to test a state, the system has to be brought into that state. This can be a complicated and a time consuming process. Checking if a state is correct is not a search problem, but bringing a system

in a certain state is, it can be modelled as a classic search problem. Search problems like the Traveling Salesperson Problem (TSP) [17] are very well known and can be attacked from different angles. TSP is relatively easy, because it is very well structured problem. It is easy to enumerate all possible sequences of cities the salesperson can travel. Searching for a specific state in a computer program is a very different. Instead of selecting cities, one must select inputs and check the corresponding state. Even with many cities, the number of possible inputs will dwarf the number of combinations of cities. Also because the number of different types of inputs are large, from mouse clicks, keys entered on the keyboard or calling functions with many different values in different combinations up to unlimited times.

1.2.2 Search Heuristics

Besides complete enumeration of the the entire search space, there are two different types of non deterministic search heuristics, local and global search techniques. Local search techniques take into account the situation at one part of the search space and global techniques try to cover multiple points at the same time.
Hill climbing is the best known local search technique. All the local points around the current point in the search space are evaluated, and the best point becomes the next current point. These steps are repeated until the top of the hill is found. Several different non determinism can be added to avoid local optima, resulting in many different algorithms like [18, 28, 40]:

- TABU search
- Simulated annealing
- Random walk

A well known set of global search algorithms are all the variants of genetic or evolutionary algorithms. These I will discuss in some more detail, because genetic algorithms are used extensively by EvoTest.

1.2.3 Evolutionary Algorithms

Evolutionary algorithms are algorithms inspired by mechanisms from biological evolution, such as reproduction, mutation, recombination, natural selection and survival of the fittest individual [29, 19, 21, 14]. Individuals in the algorithm are possible solutions and a fitness function determines how fit each individual is. Repeatedly applying the above mechanisms to a group of individuals, normally called the population, will make the population evolve toward the solution. Evolutionary algorithms have been used in fields as diverse as engineering, art [24], biology [12], economics [10], operations research [3], robotics [13], game theory [33], physics, chemistry [4], and software engineering [38].
These kinds of algorithms have been used many times in software testing, mainly for test data generation, also called evolutionary testing (ET) [2, 7, 15, 16, 34, 35, 36, 44, 45].

1.3 Problems of Evolutionary Testing

A computer program can be seen as a mathematical function with inputs and corresponding outputs. But a big difference between a mathematical function and a computer program is the presence of state in computer programs. States can cause a variety of problems for evolutionary testing [36], because test goals involving states can be dependent on the entire history of input to the test object, as well

as just the current input. This is present in all kinds of computer programming, but is even more prevalent in object oriented testing.

1.3.1 State in Object Oriented Code

In object oriented code, the real world is often modeled onto objects. An object represents an entity in the real or an artificial world, such as a person, place or a list of car parts. They combine data and programming code. The target programs in this thesis are Java programs, so here follows a quick overview of the way Java implements objects.

A class is the type of an object and it specifies how the object is defined. It can contain fields, methods and constructors, these are called its members. The fields contain data and can be of any type, which can be a class or a primitive value such as integer. The fields of an object represent its state.

Methods are pieces of code which have access to the fields in a class and act as functions. They can change the state of an object. Constructors are a special type of method. They create a new instance of an object and normally initialize all its fields. They can not be invoked on an already created object.

All members can be private or public (a third option is protected, which falls between public and private). The difference between the two is that public members can be accessed from outside the class. This is important for testing, because public members can be invoked and checked by testing code. When fields are private, it means that the different states of an object can not be seen by tests directly.

The state of an object often affects its behaviour. It is important to test all the different behaviours of classes. In object oriented programming it is desirable to hide as much information of the implementation of a class for the outside. One reason to do this, is to support the ability to radically change the class without breaking the whole application. This tends to happen quite often when other classes are written to be dependent on a particular implementation of an other class. It will be harder for an automatic test generator to create tests which only calls public methods, but still tests all the private methods.

1.4 Problem Statement

Testing takes a lot of effort but is needed to ensure high quality software. Automation of software testing can create significant savings on the costs of software production, even if it only automates a small part of the whole process. One part of testing, is creating unit test cases to ensure the correct operation of existing code. When legacy code has to be changed, there are not always regression test suites at hand to ensure that old functionality does not get compromised. The goal of this thesis is to partly automate the creation of regression test suites.

A regression test suite consists of at least a set of unit test cases and other tests. A unit test consists of calls to the class under test and one or more tests if the call was executed correctly. An adequate unit test suite achieves a certain coverage over the code under test. Parts which are not covered are not executed and thus not tested. It is my goal to create unit tests which achieve maximum branch coverage on the class under test.

The unit test consists of a part which executes the code under test and a part which checks the result, which is called the oracle. I do not try to automate the second part of the unit test, but only the calling of the code under test. This does not seem like a hard problem, but when formal specifications are not available, achieving high coverage is very complicated. This is also hard because state in objects

is often invisible and there is not always a direct relation between the state and the visible methods which operate on the relevant state.
For example, a list is implemented in a class which uses a private field which contains an array to store the elements in the list. The only three methods available are *add(Object)*, *remove(Object)* and *contains(Object)*. When the array is full, it has to be copied to a larger array. A test case which has to test this behaviour can not see how the list is implemented and has no way to query the object for this behaviour. It is unknow how many objects it has to pass to the list before this behaviour is triggered, while a tester with knowledge of the implementation knows it is one more than the initial size of the array, but what is that size?

1.5 Outline

The main part of this work is reserved for explaining the working of EvoTest, the prototypical tool I created for generating unit tests for object oriented systems. Chapter 2 discusses the relevant subjects related to EvoTest in some more detail. The design of the first version of EvoTest will be detailed in chapter 3. Because the performance of this first straightforward implementation was less than expected, several techniques used to improve EvoTest are documented in chapter 4. Chapter 5 will show some results from several experiments and the conclusion. That chapter will end with several techniques which could be evaluated as future work. The appendix describes InsectJ, an all purpose instrumentation tool which was also developed by me and is used by EvoTest for its interaction with Java programs.

Chapter 2

Related Work

This chapter explains the main techniques used by EvoTest and some alternative techniques used by other test case generation tools. Section 2.1 explains in some detail how genetic algorithms and genetic programming works. EvoTest also uses a form of genetic programming. Testing and automatic test case generation is the topic of section 2.2. Several techniques of test case generation will be explained. The goal of the tests created by EvoTest is based on coverage. What coverage is and what type EvoTest strives for is explained in section 2.3.

2.1 Genetic Algorithms

EvoTest uses Genetic Algorithms (GA) [6, 19, 30, 29] to generate test cases. This section explains what GA's are and how they work. GA's are search algorithms based on the mechanics of natural selection and natural genetics. GA's are inspired by the way nature evolves species using natural selection of the fittest individuals. In nature, populations of individuals breed among themselves, thereby implicitly searching for fitter individuals which are better adapted to their environment. The simplest GA's use minimally three operators on their population.

- selection
- crossover (= recombination)
- mutation

Selection generates the new population from the old one, thus starting a new generation. A selection mechanism is used to select the new individuals for the new population. Crossover combines two fit individuals to get even better individuals and mutation alters individuals in small ways to introduce new good traits. The basic algorithm is outlined in algorithm 1. These techniques were first used to simulate genetics with computers, but it didn't take very long before scientists started using GA's to solve many other problems.

2.1.1 Operation of a Genetic Algorithm

The possible solutions to a problem are represented by individuals. Usually the solution is encoded in the individual as a string of characters, traditionally as a binary string of 0s and 1s, but many different encodings are used in many problem domains. These encodings are called chromosomes and the blocks of information which decode to specific traits of an individual are usually called genes.

Algorithm 1 Basic algorithm for a genetic algorithm

```
initialize population
evaluate each individual evaluate~population
while (!end condition)
{
  select randomly fit individuals to reproduce
  crossover pairs of individuals
  mutate individuals
  evaluate each individual
}
```

The algorithm starts by creating an initial population, a population is used so that the search starts at many different starting points which samples the search space better than other techniques. This initial population can be totally random, or reasonably good starting chromosomes can be created using some other process, such as a greedy algorithm or manual seeding a limited number of individuals.
Each individual is evaluated in each generation to determine its fitness value. This is done by applying a fitness function to each individual. This fitness value is later used to select the better individuals from the population for reproduction. The fitness function is important, because it determines the relative fitness between individuals. For example, when the problem is to evolve a chess playing program, the program which beats the most opponents is given a better fitness value. The fitness value is also important because a value which has a too strong bias toward the best individuals will let those individuals dominate future generations, which has a detrimental effect on the diversity. This leads to premature convergence to a sub optimal solution. Some form of fitness value scaling can be employed to decrease the bias. A possibility is not to use the actual fitness values, but use those to rank all the individuals in the population and use the rank of each one for selection.
The next step is to create the next generation, which is done by applying the genetic operators: selection, crossover and mutation. A pair of individuals is selected for breeding, whereby selection is biased toward individuals which have a better fitness. Individuals with a lower fitness score do have a chance to reproduce, otherwise the number of possible solutions converge too quickly to a sub-optimal solution. There are a number of possible selection methods, these are only two of many more documented:

- roulette wheel selection. Each individual gets a chance to be selected, directly related to its fitness value.
- tournament selection. A number of individuals is randomly chosen to participate in a tournament. Only the fittest individual in the tournament is selected.

After selection, the crossover operation is applied to all, or a limited number of randomly selected individuals. When two individuals are selected to breed, but crossover is not applied, they are just copied over to the next population. Otherwise, the outcome of crossover are two new individuals, whereby they inherit a part of the parents' genetic material. The chromosomes of the parents are mixed in some way during crossover, most of the time by swapping a portion of the data between the parents. More complicated crossover operators are used by different chromosome structures and for different problem areas. This process is repeated with different parent individuals until the next generation has enough individuals. After crossover, the mutation operator is applied to a randomly selected subset of the population. Most algorithms only apply mutation with a very small probability

of 0.1 or less. Mutation randomly alters a small bit of the data structure of the chromosome. In classic genetic algorithms some bits in the string are altered. But many different mutation operators can be devised for different chromosome encodings.
Finally, the new generation is complete and the process continues with another generation and this is repeated until an individual is found that is good enough, or some other stop condition is reached. This can be a maximum number of generations or a maximum of process time. For more information see one of many books on the topic of Genetic Algorithms, for example [6, 14, 19, 21, 29, 30].

2.1.2 Properties of GA's

Genetic Algorithms have two very nice properties. They are trivially parallel and extremely robust. The first property stems from the fact that large populations can be evaluated and mutated parallel with each other. Each individual or pair of individuals can be processed at the same time by different computers or processor cores. This will become even more important in the future when clock speeds will not increase anymore, but more central processing cores will be used inside single computers. Even now, distributed systems are very popular and they are extremely suited for computing these kind of algorithms.
The second property of GA's, their robustness, stems from the fact that there is no direct link between the problem and the way it is solved, this is also a weakness. Mutations and crossovers are used to search the search space, but they are not directed by the current problem, they are random. This means that even though it is not known what path should be taken to the solution, the GA can find it because it will automatically zoom in to the solution. This is also a weakness, because it will also explore area's which are fruitless.

2.1.3 Genetic Programming

Genetic programming [30] (GP) is a variant of standard genetic algorithms and they share many characteristics, like the evolutionary operators of selection, reproduction and mutation. The biggest difference is the problem domain, GP searches for computer programs which perform some user defined task. The fitness is computed by evaluating how well the programs performed the task. The definition of GP used in this paper is a GA which has these extra properties:

- The evolving individuals are executable programs
- The individual has to be executed to be evaluated

Another common difference between the two is the structure of the chromosome. The chromosome is normally organised as a tree-structure, but classic linear structures were also successfully used by Tonella [44]. Although Tonella does not mention the word genetic programming, he does perform GP when the definition above is applied because the evolving unit tests are programs which have to be executed.
Normally GP uses a tree structure because they are more similar to the abstract syntax trees used by compilers and many computer programming languages. The fitness value of an individual, which is a computer program, is derived by executing it and determining its performance in some respect. If the goal of the GP is to evolve a program which performs a specific task, the fitness of each individual depends on how close its behaviour was compared with the behavior specified by the requested task. Programs which fail to execute, because of memory or time constraints; for example due to an infinite loop; have their fitness values lowered or set to a fixed low value.

The last couple of years, GP has undergone a series of breakthroughs when it has been used to create solutions which equalled human inventions in area's like game theory, sorting, searching and quantum computing [31]. The running time of GP algorithms is typically orders of magnitude longer than for normal GA's, but because of the increase of computing power of newer computers and the relative ease at which these population based algorithms can be executed on multiple computers at the same time, for example on clusters, they are nowadays successful in solving relatively hard problems.

2.2 Testing and Automatic Test Case Generation

The field of testing is extremely large, but this thesis only covers unit tests. Unit tests are generally relatively small tests which target only small units of programming code. In the past and in this work the goal is to make it possible to generate those unit tests completely automatically.
There has been a significant amount of work in automatic test case generation that attempts to increase the amount of observed behavior. The most common way of assessing the amount of exercised behavior is to calculate the coverage of the test suite and then create tests that add new coverage to the existing suite. See section 2.3 for more information on coverage. It is possible to classify existing approaches in three main categories:

1. Random approaches
2. Approaches using symbolic execution
3. Directed search approaches

2.2.1 Random Test Case Generation

Random approaches to test case generation such as JCrasher [11], randomly choose values from the domain of potential inputs. There are three main problems with a random approach. First, there are many inputs that have the same observable behavior. Consider an array class, a random approach may generate several test cases that add a single value to the array. Each of these values may be different but the observable behavior is the same: an array containing one element. If the testing criteria is a coverage criteria, these test cases are redundant because they do not increase the coverage. The second problem with a random approach is that it may be very difficult to randomly select inputs that exercise a conditional. An if-statement that checks for a single value is nearly impossible to generate input for in a reasonable amount of time. The third problem with a random approach is that generated test cases for object oriented programs are usually shallow, which means that they sequence only a limited number of method calls after each other. The result is that shallow test cases only create a limited set of object state.

2.2.2 Approaches Using Symbolic Execution

Symbolic execution based approaches to test case generation address the redundant test case problem of random testing [27]. In symbolic execution, concrete values are replaced by symbolic variables. The program is then symbolically executed. As each conditional in the program is encountered it is added to a list of path constraints. Path constraints are a conjunction of the conditionals that need to be true for a specific path to be exercised. Once the path constraints are constructed, an equation solver attempts to find concrete values that make each path condition true. Unfortunately, the usefulness of symbolic execution is limited by the power of the constraint solver. Currently, only linear integer

equations can be solved. Tools like CUTE [27] attempt to overcome this limitation by substituting concrete values when path constraints become too complex. However, it is not yet clear how well this technique works for object oriented languages such as Java.

2.2.3 Directed Search Techniques

Generating test cases can be seen as a search. Random methods are undirected search techniques, because the outcome is not used to direct the next search attempt. When the outcome is used, it is a directed search technique. Local and global search techniques can be used for test case generation. The well know technique is the use of genetic algorithms (GA's). Approaches based on GA's have had some success creating unit tests for procedural and object oriented languages [44]. Genetic algorithms are a type of random search based algorithm that is based on the mechanics of natural selection. They are traditionally implemented as a simulation in which a population of candidate solutions evolves toward better solutions. When used for testing, GA's create unit tests by adding, removing, and modifying methods found in the public interface of a class. While a fitness function directs the search to the solution. This approach works well for smaller programs and is fairly robust unlike symbolic execution. However, the search space of the problem grows exponentially as the number of classes and methods interacting with the class under test grow. As the size of the search space increases, it becomes harder for the GA to evolve good solutions in a limited amount of time. For sufficiently large programs it is impossible for the genetic algorithm to evolve enough good solutions. See section 2.1 for more information on GA's.

2.3 Coverage

EvoTest has as its goal to create tests with complete branch coverage. This section explains the term coverage and in some more detail how branch coverage works.
Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested. It is a form of testing that looks at the code directly and as such comes under the heading of white box testing. Coverage criteria can be used to test the quality of a test suite. There are a number of different ways of measuring code coverage, the main ones being [9]:

- **Statement Coverage** Has each line of the source code been executed and tested?
- **Condition Coverage** Has each evaluation point (such as a true/false decision) been executed and tested?
- **Path Coverage** Has every possible route through a given part of the code been executed and tested?

Code coverage is ultimately expressed as a percentage, as in "We have tested 67% of the code". The meaning of this depends on what form(s) of code coverage have been used, because 67% path coverage is more comprehensive than 67% statement coverage. The goal is to achieve 100% coverage with tests on the test subject.
Instead of statement or path coverage, we will use branch coverage, which is similar to condition coverage. It needs to be clear how we define a branch and how the corresponding coverage is calculated.

2.3.1 Branches

Branches in programming code are needed to make decisions and to execute a certain program sequence multiple times. A branch changes the point of execution, it either jumps forward or back in the program. There are two types of branches, unconditional and conditional branches. The former always jumps to the new execution point and the latter only does when a certain condition holds. When a branch is 'taken', it means the condition held and the jump was made, when it was not taken, the next instruction will be executed. Every conditional branch thus has two possibilities.

2.3.2 Branch Coverage

Branch coverage is the percentage of total number branches that have been executed when the test is run.

$$C = \frac{t + n}{2 \times b} \tag{2.1}$$

Branch coverage (*C*) is calculated using equation 2.1, by counting all the branches which were taken (*t*), adding all the branches which were not taken (*n*) and dividing the total by the total number of branches (*b*) times two. When a test achieves 50 percent branch coverage, this means that of all branches half were taken and not taken, or that all branches were only taken or not taken.

2.4 Summary

This chapter covered the basics of genetic algorithms (GA) and its derivative, genetic programming (GP), in section 2.1. It is unclear which is the generalisation of the other, but I assume that GP is a special form of GA. GA's and GP's have much in common, both represent individuals in a population through a special data structure, called the chromosome. These structures are mutated and evaluated, which includes the execution of the chromosome as a program, in the case of GP.
The second section in this chapter gave a short overview about several techniques used for automated test case generation. I divided the area in three approaches, random test case generation, generation using symbolic execution and test case generation using directed search techniques. Genetic algorithms fall in the last category and are the focus of this thesis.
The last section covered the topic of coverage, with a focus on branch coverage. Complete branch coverage is the goal of the test cases generated by EvoTest, the prototype tool developed for this thesis. EvoTest and the algorithms used by EvoTest, will be the focus of the next chapter.

Chapter 3

EvoTest - Genetic Algorithm Design

This chapter describes the main algorithm of EvoTest. EvoTest is the tool I developed over the course of a year, at first at the Technical University Delft and later at the Georgia Institute of Technology with Jim Clause and Alex Orso.
The goal of EvoTest is to be able to evolve simple Java test cases in the Java language which achieve a certain kind of coverage criteria. In the course of the project it was decided to use branch coverage because it was relatively easy to calculate for Java programs, and still a pretty strong criteria. To generate these test cases it uses genetic programming, which is a type of genetic algorithm.
Before we go in to many details I will give a general overview of the algorithm used by EvoTest.

3.1 General Overview

EvoTest uses a genetic algorithm as its core. It can be seen as genetic programming, because it actually evolves and executes programs, the test cases. It has all the normal ingredients also seen in normal genetic algorithms. Individuals are represented by a certain genome, which are created and mutated. New generations are selected through evaluation of the individuals and this cycle continues until a perfect individual is found or the maximum number of generations is reached.
Normally there is one goal for a genetic algorithm or a set of goals which have to be found, but in this case, the goal is too broad to be attacked at once. It is not practical to evolve a whole set of test cases at once. That is why each coverage target becomes a goal for the GA, and the GA switches between targets when a target is reached or when it cannot be found within a certain number of iterations. When every target is tried several times, or actually reached, the algorithm stops and shows which targets have been covered. It generates a compilable Java program which contains a method for each test case which has been generated. When this program is compiled and run, it executes the target program with the same amount of coverage as found by the genetic algorithm. This is considered the result of EvoTest. Figure 3.1 shows the algorithm in abstract form.
EvoTest uses a tree structure to represent a test case, section 3.2 describes in detail how the individuals are represented. The following section 3.3 describes how the first population is created. This is needed to have a starting point. Mutation and crossover are the two methods to change and thus improve an individual. The various mutations and how crossover works on the individuals is described in section 3.4. In genetic algorithms the selection is done by a fitness function which tests how fit each individual is. A fitter individual has a better chance of reproducing and surviving to the next generation. The evaluation is done by EvoTest by executing the test case and tracking which branches are covered and which are not. This information is then used to calculate the fitness compared to the

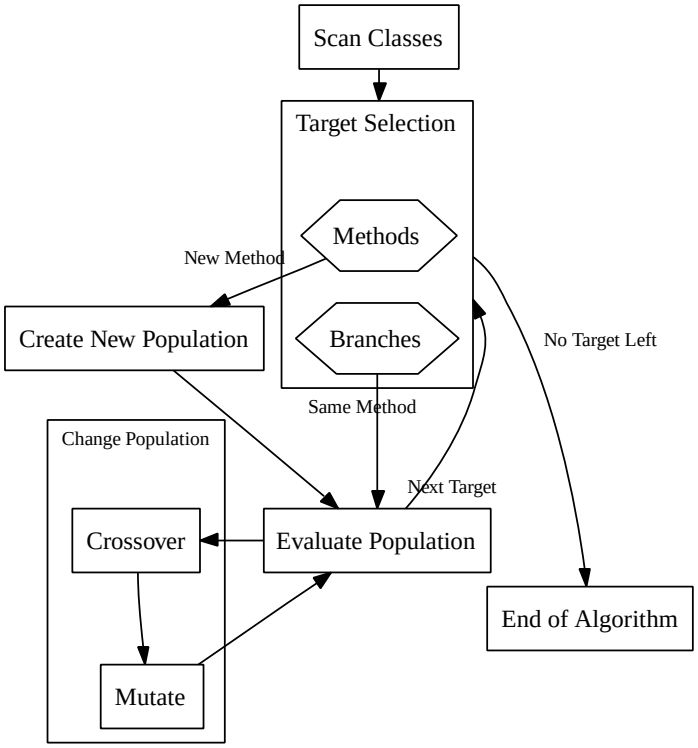

Figure 3.1: Overview of the genetic algorithm used by EvoTest

current target. This chapter ends with a small section which describes various other parameters used by the GA and a short evaluation of the algorithm so far.

3.2 Test Case Representation

The individuals have to represent a test case in the Java programming language, which is an object oriented language. It has to be able to represent the usual language constructs such as static and non static methods. A test case usually consist of a sequence of functions calls, which can be static or non static method calls and constructors. If those calls have return values, these can be stored in variables and those can later be reused as inputs for other calls.
The internal structure is also an important design decision because certain representations allow efficient modification of individuals. Instead of using the typical GA representation using a linear array, for example a string of 1's and 0's I use a tree based representation, which is the typical GP representation. Koza [29] developed the use of trees for genetic programming, which are very similar to the abstract syntax trees (AST) in compiler technology. An example is shown in Figure 3.3.
In the representation, the root node is a special node which represents the test case and contains an array of children which contains operations like method calls. Besides the root node, which is unique, the tree is build by nodes. There are different types of nodes, one for every kind of operation EvoTest supports in a test case. Every type of node is a meta node. The full list of meta-nodes is shown in Table 3.1.
The tree structure consists of nodes and all those nodes have in common that they return a Java type,

Node name	Description	Children
L-Variable	Variable definition	yes
R-Variable	Reference to a previously defined L-Variable	no
Constant	A concrete value (int, double, String, etc)	no
Constructor	Creates an Object	yes/no
Method	Calls a method	yes/no
Field Assignment	Alters the state of one field in an object or class	yes/no
Field Read	Reads a field in an object or class	yes/no
Method Under Test	Parent of all other nodes and unique in the tree	yes
Array	Creates array with objects	yes
Null	Defines the null keyword	no

Table 3.1: Meta node list

```
List var1 = new List();
var1.add(new Object());
var1.add(null);
var.size();
```

Figure 3.2: The equivalent Java code of the test case in figure 3.3

in other words they are of a certain Java type. This type is the type of object the node places on the stack when it is executed. This can be *void* or any other Java type. For example, the type of a Method node would be the method's return type and for a Constructor node, the type of the object which gets created. Only the Method Under Test node does not have a type.
Some nodes have children. These children are other nodes which need to be executed before the parent node can be executed. They normally generate the arguments for a method or constructor, but also the value stored in a field, which needs one child node to generate that value.

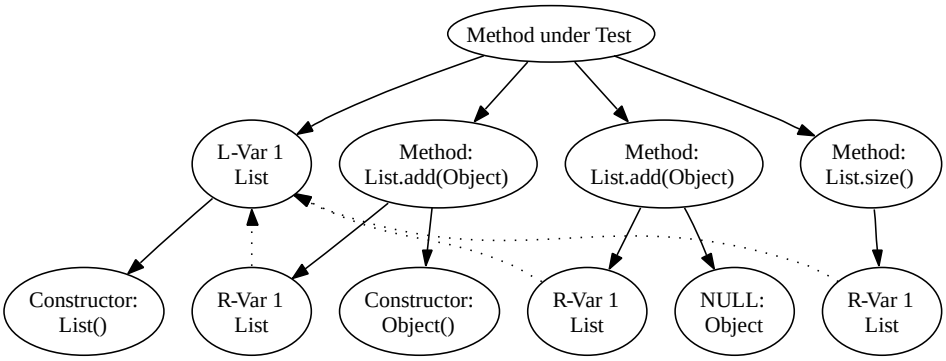

Figure 3.3: Intermediate Representation of a Test Case, the dotted lines are variable references

3.2.1 Variable Nodes

EvoTest stores return values of method calls which do not return void. It does this to make it possible to reuse those objects for other method calls. This reuse can be done once or multiple times. Sometimes

this is needed because sequences of method calls on certain objects might be needed to bring an object in the correct state to reach a certain part of the code. Or it is only possible to acquire an object of the correct type by having it returned by a call to another method.
EvoTest uses special variable nodes to store those return values. There are two variable node types, the L-Variable node and R-Variable node. The L-Variable node defines the variable and the R-Variable references it. Figure 3.3 shows an individual which uses those variables.

3.2.2 Private and Protected Members

Normally it is not possible in Java to call or use private members of a class. But it is possible to access them using various techniques. It is possible to call private and protected methods outside the class, by directly generating the correct byte code for it or changing the visibility of the private and protected members to public using the byte code instrumenter (see Appendix A). It is also possible to access those hidden members using the reflection mechanism provided by Java. The reflection library in Java makes it possible to inspect and use all classes and their members loaded by the virtual machine.
Even though it is possible, the techniques described are not used by EvoTest, because one of the goals is to generate normal test cases, and test cases which access private or protected members are not usable in normal testing suites. When EvoTest is forced to call a hidden method, it performs a simple call graph analysis on that class. This is done to determine which public methods call the private of protected method. Any of the methods found can be used to call the otherwise inaccessible method.

3.3 Creation of the Initial Population

At the start of the algorithm, a new population has to be created. For this, there are two methods, random creation and creation using traces. A trace is a detailed log of the execution of a program. Traces are generated from runs of the program while it is instrumented by a tool which builds those. A run can be an execution of the program by a user or the execution of another existing test suite. Traces can be used to extract small sequences of method calls. These sequences can be used to create an initial population for the GA. When there is no trace available, because the program is not yet executable, the random creation method can be used. The random method, randomly selects some initial input values and only adds extra methods to the initial test case if it is necessary to be able to call the target method. When multiple possibilities exist, it randomly selects one.
The system selects a method which needs to be fully covered. This is the *target* method and thus the *current* method. Normally this method will be the method under test (MUT) in the test cases which will be generated. The last executed method by the test case should be this MUT.
If the target method is private or protected, any of the methods directly or indirectly calling the target method can be used instead. The list of public methods calling the target method becomes the *current* methods list. This list of methods is the input for the test case builders. They will take a method from this list and generate a test case for it.

3.3.1 Random Test Case Builder

The random test case builder builds a test case around the current method. It will recursively generate methods, constructors and other nodes until the test case can be compiled. Besides the root of the tree and current method node, all other nodes are random generated. This random generation is also used by the mutators discussed in section 3.4.1. This is done by a recursive algorithm together with special

data generators. Data generators are simple classes which create a random String or primitive value. In the next subsection and in section 4.1.3, in the next chapter, these are explained a more throughly.

3.3.2 Recursive Test Case Creation

EvoTest scans the class under test and all referenced classes to create a repository which contains all the methods, fields and constructors of all the relevant classes. These are indexed on the return values and the public fields they contain. The input for the algorithm is then a type. This means that it has to create a tree which has that type as return value. It will retrieve all the constructors and methods which produce this type and will pick one of those in a random manner. Besides those nodes, there is always a probability it will insert a *null* node, this probability value can be set in a configuration file. When a method or constructor contains arguments, the algorithm is recursively called again for each argument until the tree is complete. When the type is primitive or can be generated by a constant generator, like a *String* or *Double*, this generator is called and a constant node is generated. Those generators are very simple and randomly create or mutate primitive values and strings.
A depth counter is used to prevent infinite recursion. When the maximum depth is almost reached it will try to use constructors only, and it will prefer the ones with the least amount of arguments. This is done by giving the constructors a complexity rating based on the number and kind of arguments. When the depth is reached it will only use constructors or static methods without any arguments or if those are not available it will create null nodes.

3.4 Modifications

Evolution of the population of individuals is done by two processes working together. Selection of the fitter individuals and modification of those selected individuals. The process of modification of individuals is done by two types of modification.

1. Mutation, a small modification in an individual
2. Crossover, the recombination of two individuals into one or two new ones

Evolutionary algorithms can use various different mutations and crossovers to alter the populations they are evolving. Mutations introduce diversity in the population by adding new features to an individual, which can possibly raise or lower the fitness of an individual. Crossover attempts to combine the good characteristics of two individuals into one to create an even better solution.
There are various mutations and types of crossover possible because of the complex structure used for the individuals. Those mutations and crossovers are described in the next subsections.

3.4.1 Mutation

Each individual has a certain probability to be mutated by a specific mutation operator. In GP, there are generally two types of mutation. Mutation that alters something existing and mutations which introduce something new. These are the mutations we tried:

1. Constructor Mutation, replaces a constructor call with another call with the same return value
2. Method Introduction, introduces a new method call on a local variable
3. Method Deletion, removes a method and its subtree

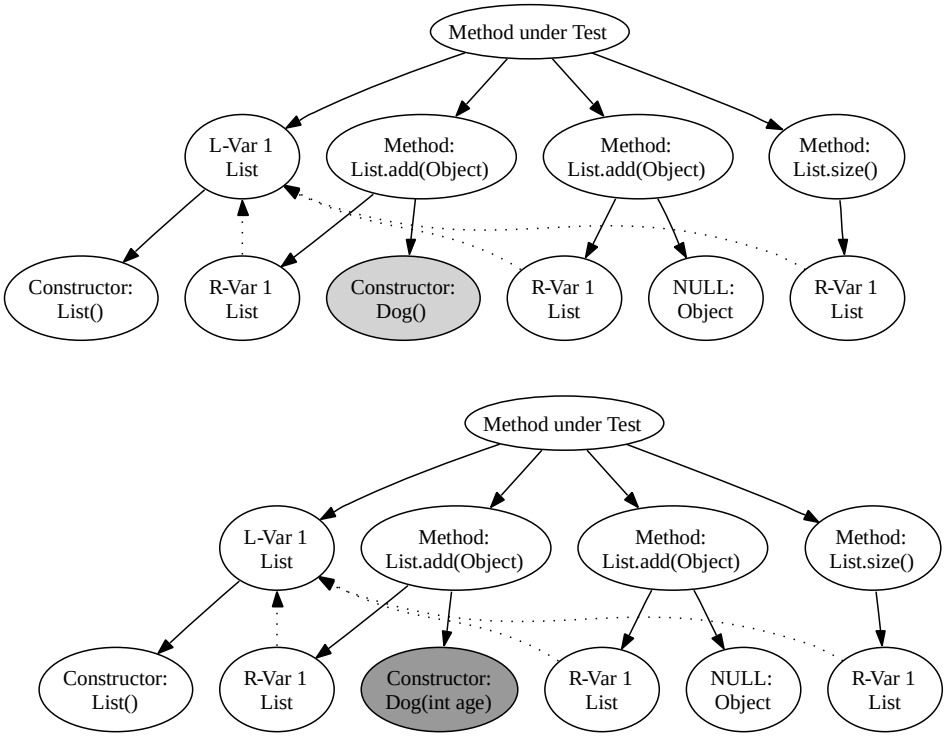

Figure 3.4: Constructor mutation example, a constructor is mutated to a different constructor

4. Variable Introduction, stores the result of a method or constructor in a variable

5. Constant Mutation, mutates a random constant to another value

Constructor Mutation

Out of all the null and constructor nodes, one is selected and mutated. A node can be mutated to a different constructor of the same type or sub type or it can be mutated to a null node. When a new constructor has arguments, the children nodes will be recursively generated.
Figure 3.4 shows the mutation of a constructor to a different constructor. It is also possible that a constructor is replaced by a null node or a null node is replaced by a constructor node.

Method Deletion

Methods are different from the constructors, mostly because they are the main way of altering the (hidden) state of an object. They are not needed for the initial creation of a test case, so there are different ways for them to be added to a test case. There are two different mutators, one that adds a method, and one that deletes a method randomly. The delete method mutator selects randomly a method from the individual and removes it including its subtree. When there are no methods to delete, the algorithm aborts.

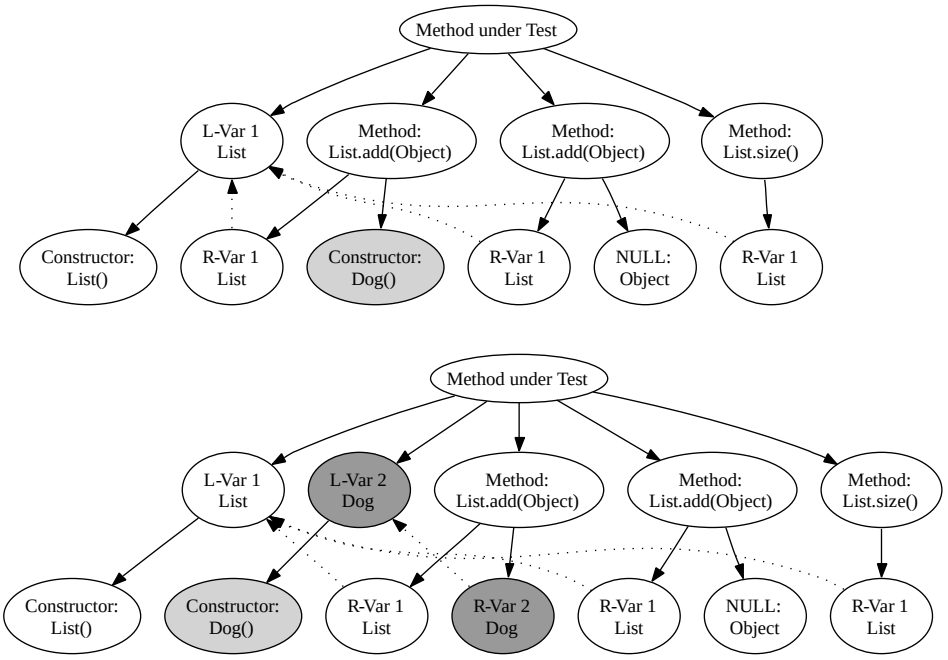

Figure 3.5: Variable introduction example

Method Introduction

Methods need to be invoked on an object. When one needs to be added, an object needs to be selected where it can be called on. One of the variables in a test case is selected randomly. The variable holds an object of a specific class, and one of that class its methods is added.

Variable Introduction

Methods can only be called on existing objects, if an object is not stored in a variable, it is not persistent and its state cannot be changed. When arguments are created, the objects for them are not stored in a variable. Their state can only be changed if they are stored in a variable and altered by methods. To make this possible, a mutator which introduces variables is needed. This mutator randomly selects a node which returns an object from the test case and promotes it to a variable. An L- and R-variable pair is created. The L-variable is inserted before the old location of the selected node. The selected node and its subtree is made a child of the L-variable, because it will initialize the new variable. The corresponding R-variable is then added to the old location of the transplanted node. Figure 3.5 shows the introduction of a variable which holds an object of type *"Dog"*.

3.4.2 Crossover

One of the differences between normal genetic algorithms and genetic programming is how crossover is performed on individuals. In nature, the chromosomes have certain spots where they are likely to break and where they will be reconnected. Two chromosomes break at the same spot and the halves are swapped between them. The computer algorithm works in the same way but there are various variations possible. Just like in nature, one or more random points can be selected in the chromosome,

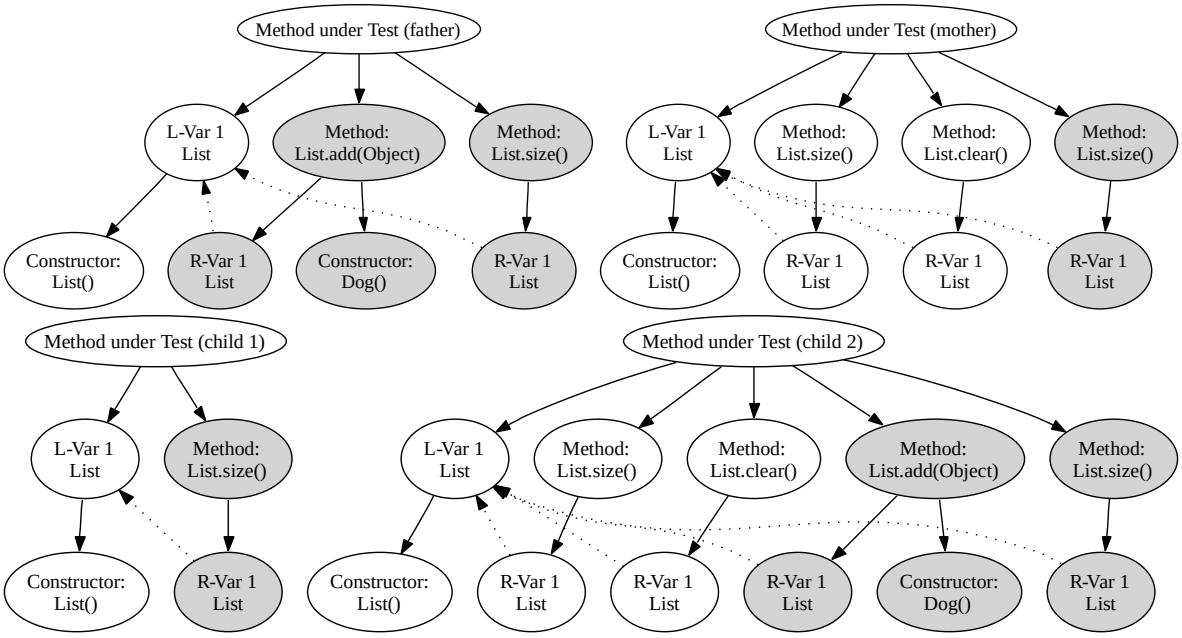

Figure 3.6: Point crossover on trees, two halves of the chromosomes are swapped

which will become the spots where the individual's data will be split up. Paolo Tonella [44] uses one point crossover to combine individuals.
Normaly in genetic programming this system doesn't work because the individuals are not linear like chromosomes in a GA, but have a tree structure. Instead of picking two random points in the chromosomes, two nodes in the tree are selected. Those and each in their subtrees is swapped when they are compatible. Whether they are compatible or not is dependent on the types of the nodes in the trees, and it differs between applications.

3.4.3 One Point Crossover

When the variable creation is dominating variable removal, most objects are stored in variables. This results in sequentially organised flat trees. The whole tree is very similar to a sequential chromosome because the root node gets a large number of children. One point crossover is then possible on the chromosome and it is very easy to implement. It is called one point crossover because only one point is selected in each chromosome where it is broken, two points or more are also possible.
Figure 3.6 shows two very flat chromosomes with their children after one point crossover has been performed. The methods all change the state of the same object stored in a variable. Two halves of the chromosomes are selected and combined to create two new individuals.

3.4.4 Tree Crossover

The default crossover in GP is sub-tree crossover. This operator selects two sub-trees from the two parents and swaps them. The sub-trees need to be compatible, otherwise they cannot be transplanted between the two parent chromosomes. In classic GP, two trees are compatible if the types of the two root nodes of the subtrees are compatible and the references in the subtrees to global structures

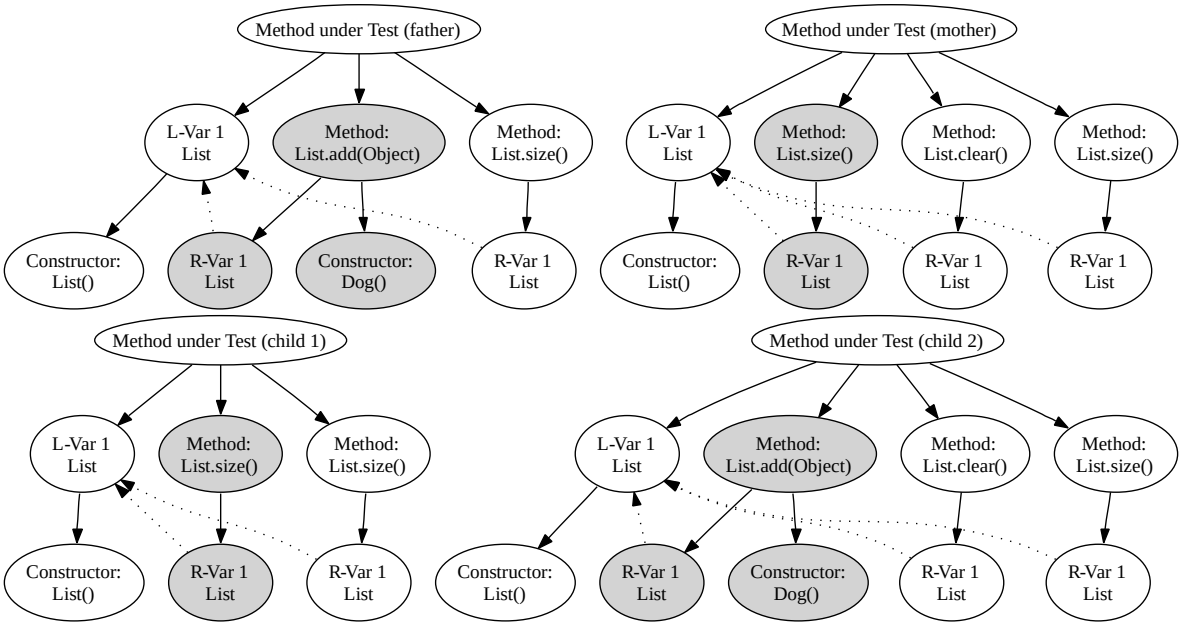

Figure 3.7: Sub-tree crossover, two subtrees are selected from both parents and swapped

are valid in the individuals. Those references in classic GP point to subprograms and variables. In our situation, we only have variables. When a subtree uses a variable and it is transplanted to a chromosome where the variable does not exist, the chromosome breaks and the test case is illegal. That is why usually all the references to variables need to be equal. I relax this requirement, because it limits the possibility of crossover to such a degree that it can only happen between very related chromosomes. Now, only the root nodes of the subtrees have to be equal. Because using crossover on two incompatible chromosomes breaks the references to variables, a new step is introduced which is explained in the next subsection. An example showing a sub-tree crossover is depicted in figure 3.7.
If two nodes are selected and swapped between their chromosomes, their types need to be the same. When only one of the nodes is transplanted in the other, the type requirement can be relaxed. A type is compatible if its a *subclass* of the given type that it is going to replace. If two parents produce only one child, it is possible to use this relaxed requirement. The individual receiving the subtree of the other is called the mother and the one giving the subtree the father. This is because the amount of genetic material given to its offspring is smaller (only one subtree) while the mother gives the rest of the chromosome. This version of tree crossover is implemented in EvoTest.

3.4.5 Mutation Fixer

This operator is normally absent in genetic algorithms, but some process equivalent to it happens in nature. In human cells, free radicals and other energetic particles can damage genetic material. When that is not fixed, mutations can build up which would cause diseases like cancer. This is much like normal mutations which occur during reproduction, but because these mutations are not hereditary, they are useless and serve no purpose. Computer algorithms do not have any free radicals, but several of the previously discussed mutations can cause genetic defects.
In GP tree-crossover, the global references, like variables, in each subtree need to be compatible. In

this case, this means that variable references must exist in both individuals and they have to be defined before the reference is made. We do not check this because it would severely limit the possibilities of crossover between non related individuals, to such an extent that in many cases crossover is not possible.
It thus happens that variables are 'scrambled' in individuals. The mutation fixer finds and repairs those errors. A simple algorithm scans through the individual and repairs it locally when it encounters an error. It keeps a list of all declared variables and adds each variable declaration when it encounters one. When it comes across a reference it checks the list of declared variables for a matching declaration. When none is found, it takes the following steps to repair the problem:

1. Find in the list of already declared variables a compatible variable, with the same type or sub type. Reference the found variable instead of the the old variable which did not have a declaration.

2. If (1) fails, because there is no matching declaration, scan the individual for a node with a matching type for the variable which is needed. If a node is found, create a new variable declaration (see Variable Introduction in 3.4.1 for details).

3. If (2) fails, create a new variable declaration, including a new initialization.

3.5 Selection

An important part of the genetic algorithm is the selection method. After each generation, the fitness of each individual is calculated, so that fitter individuals survive and the less fit individuals die out. The selection mechanism directs the search through the search space, similar to how hill climbing algorithms climb the hill of the search space. The question which need to be answered is, what does the system select for and how does it calculate a fitness value for that selection?
In white box software testing, there are many different coverage criteria to test on. Ranging from statement coverage to full path coverage. How harder it is to fulfil a criteria, the better a piece of software is tested, but how more impractical it becomes. Full path coverage is completely impractical, because it is impossible to fulfil, also more stricter coverage criteria have a larger percentage of infeasible parts. This is because most coverage criteria are not possible to fulfill completely due to infeasible paths in the system under test. I wanted to use acyclic path coverage as the coverage criteria to use, which is just slightly less strict then path coverage. But this criteria is bounded, unlike path coverage, and relatively easy to calculate. Although I chose acyclic path coverage. the prototype system uses branch coverage, because it was even easier to implement. The prototype is designed so that it is possible to switch to other coverage criteria relatively easy.

3.5.1 Target Selection

If full branch coverage is the goal of the prototype, each individual branch is a target for the genetic algorithm. Normally, a genetic algorithm has only one target, which can come in different forms, but when there are many separate targets it becomes very different. One separate evolutionary run for every branch would be very inefficient. All the branches are a target, and one is the current selected target. The fitness value is calculated toward this current target. All the other targets are checked if they are covered by an individual. If this is the case, they get stored in a list of covered targets with the individual that covers that target. This individual can later be retrieved for inclusion in a test suite.

Branch Selection

We already defined what we think is a branch in subsection 2.3.1 but here is a short refresher with Java byte code in mind. A branch is a location in the byte code where a conditional jump is used. A normal branch has two directions, *true* and *false*, and a switch has N directions, which ranges from 0 to N-1.
The class under test is scanned for branches. For every branch which is not a switch, two targets are created and for every switch there is a target for each case. Because EvoTest scans the byte code and not Java source code, not all the branches relate directly to a branch in the Java code. For example, some boolean logic is implemented using branches in the byte code. The reverse is also true, if-statements which use final boolean values are compiled away. In this case it is a good thing, because it would have been an infeasible branch if it was not optimised away. Because the final boolean value can not be changed, because it is final, the branch which depends on it will always be taken the same way. The other possibility is infeasible.

3.5.2 Execution

To calculate the fitness value compared to a branch target, an individual has to be evaluated, or in other words, executed. The individual is stored as a tree in memory, consisting out of Java objects. This tree can be executed in several different ways. Execution comprises two stages, converting the tree to something which can be executed, also called compilation and the execution itself. It should be clear, that when an individual is executed only once, that the compilation time can become the dominant factor in the overall evaluation time. When many thousands of evaluations are necessary, this can become a problem.

Create Java Source

The most obvious approach of executing a generated individual is to transform the individual to the equivalent Java source code and use a Java compiler to generate a class file. This class file can then be executed inside the current VM, or in a separate VM. The obvious problem with this arrangement is the extreme overhead introduced by the compiler. These are the steps involved:

1. Convert the individual to Java source
2. Write the source to disk
3. Compile the source to a class file using the compiler:
    a. Start a new VM
    b. Read the source file
    c. Parse and check the source
    d. Write class file to disk
4. Read the compiled individual from disk
5. Execute the individual

Reflection

It is possible to execute the individual directly instead of changing the representation using a compiler. This can be done using the *reflection* library in Java, which can execute methods and constructors on objects. Each node in the tree can be executed, and they can execute the children they have. The results can be passed through the tree using some data object. This type of execution is very similar to normal Genetic Programming and emulation. This method is very efficient if an individual is only executed once because no translation steps are needed and there is no disk IO. Although there is no translation step, the execution step is not as efficient as a compiled individual because the reflection calls incur a lot of overhead compared to a normal method call.

Byte Code Generation

More similar to compiling source code and executing the result but with less overhead is direct byte code generation. Byte code is an assembler like language which is executed by the Java VM and the result of the Java compiler. It is quite straight forward to generate byte code directly from the individual, similar how Java source code would be generated. But the byte code can be directly loaded in the VM using a class loader and in this way it does not generate any IO and circumvents the Java compiler and all its overhead.

State Reset

Any static fields which get modified by the executing test case need to be reset in between executions of individuals. If this is not done, the same test case can execute differently. This is not common, but can happen, for example with the singleton design pattern. Only the first test case will execute the branch to store the value, all subsequent tests will take the branch the other direction.

The first method to reset a class is to reload it. To reload a class, it first needs to be unloaded. Unloading of a class happens when it gets garbage collected. It can only be garbage collected when the class loader that loaded it gets collected and the garbage collection is unreliable. So this is not a very practical approach.

The best solution was to use InsectJ (see Appendix A, to modify the class so it can be reset to its initial state. This is done by copying the byte codes from the class initializer to a new method and executing that method using reflection when the class needs to be reset. There are two caveats. The instructions which initialize *final* fields are not allowed in a normal method, but they are not needed, so those have to be removed. And, some classes do not have a class initializer, so they have to be skipped. More details can be read in Appendix A.3.2.

3.5.3 Evaluation

Executing the individual is not enough. The system needs to register which targets get executed and if none got executed, how close the execution was to the desired one. An execution which comes very close gets a high fitness value and otherwise a low one. The function which generates this value is called the fitness function. The input to this function is the execution of the individual together with a target, and the output is a value between 0 inclusive and positive infinite, where 0 corresponds to the ideal individual which reached the target.

Branch Tracing

The fitness function takes the program execution as input, and there are several useful representations of that execution. Someone could manually add instructions to the system under test which execute some code which prints something to the screen or a file when the execution reaches a certain point. This is very time consuming and error prone. The best approach was to instrument the class under test on the branch level and let it create a branch trace. A branch profile would have been just as useful, but a trace is more general and can later be used for more strict coverage criteria.
Using InsectJ, which is described in appendix A, a trace of all the branches which get executed is created during the execution of the system under test. When the system is loaded by EvoTest, it is instrumented with probes. A probe is a small piece of code which can record certain values from the stack, such as variables and other state without modifying the method in which they are inserted. These probes get inserted at method entry and exit points and at conditional branches in the methods and constructors. While InsectJ instruments the class, it also creates the target list. The created branch trace is not one single array of branches but a collection of small traces. One for every invocation of each method or constructor. The following is done for each three different events:

Method entry A new branch trace list is made, which stores the name of the method and its signature, and this list is made the current trace. If there already was a current trace, this trace is put on a stack.

Method exit The current trace is finished and stored in a list of finished traces. If the stack of traces is not empty, the top one is removed and becomes the new current trace.

Branch A new branch trace element is added to the current trace. This branch element stores the following values:

1. Branch ID
2. Line number
3. The direction, (true of false, or which case in a switch statement)
4. The distance, a value which determines how close the branch was to taking the other direction.

So the result of one execution is a collection of traces which contain zero or more branch elements.

3.5.4 Fitness Function

The fitness function uses a target branch and the branch trace from an individual to create a fitness value. When the target branch is in the trace and the correct direction is taken, the test case covers a target. This means the fitness value is zero and an ideal individual is found. If this is not the case there are several other options:

1. The target branch is in the trace but it was not the correct direction.
2. The target branch was not in the trace, but the correct method was invoked.
3. The target branch was not in the trace, and the correct method was not invoked.
    a. Because one of the methods calling the method did not call it.
    b. Because an exception was thrown before the target was reached.

Distance Value

A trace from option 1 is the closest to the optimal solution, only the branch has to be taken differently. Take the example where a branch is taken when an input value is equal to a constant:

```
public void test(int a){
    if(a == 10)
        // target
}
```

Here it is clear the correct solution would be to make the input value *a* equal to 10. If it is not, it is clear that a value close to 10 is more fit. The difference between the actual value and the target value is the *distance value*. When the branch is reached, this is enough information to create a useful fitness function, which together with mutations is able to solve the problem. In the other cases, this is not enough, for this we use the approximation value.

Approximation Value

In option 2, the branch was not in the trace. Now it is possible to determine which path should have been taken to the correct branch. A fitness value could then be constructed from how close the path was to the target path. Every branch which decides if the target branch will be reached or not is a critical branch. The fitness value will be the number of critical branches missed to the target branch. That number is calculated by creating a dependency graph for the method. This graph is turned to a tree to make the calculation easier to perform. This is done by removing the back edges which occur due to loops, this does not influence the computation. When a trace has to be evaluated, it is not known which branch is the closest to the target branch, so for every branch it is calculated how close it is to the target.

Normalization

If the approximation value is larger than 0, the distance value is still relevant, because it can be used to direct the search to a lower approximation value. The combined distance and approximation value becomes the fitness value. For this, the distance value has to be normalized to a value between 0 inclusive and 1 exclusive. Equation 3.1 represents the normalization function used.

$$F = a + atan(\frac{d}{50}) \times \frac{2}{\pi} \tag{3.1}$$

F is the fitness function, *a* is the approximation value and *d* is the distance value.

Exceptions

The current implementation does not handle option 3a and 3b optimally. When the correct method is not invoked or when there is an exception before the target is reached, the worst fitness value is assigned which is positive infinity. It is possible to create a call graph of all the methods combined with dependency analysis to create a better dependency graph to calculate a more precise approximation value, but this simple implementation is already quite effective for most situations we tested. More complex programs which utilize exceptions more often are harder to cover by EvoTest without this addition to the algorithm.

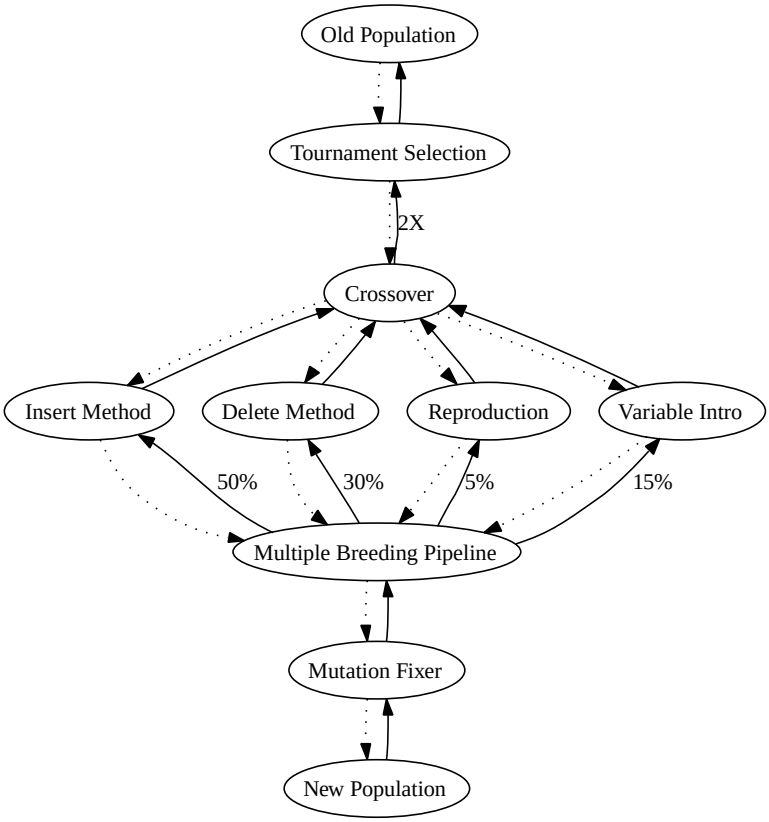

Figure 3.8: The breeding pipeline used by EvoTest.

3.6 Configuration of the Genetic Algorithm

Now that all the building blocks of the GA are constructed, they have to be fit together so they work as one algorithm. Because there are multiple types of mutations, there are multiple ways to let those work together. Mutations can be set in a serial order or in a parallel manner, whereby only one mutation can be done at a time. The order of the different operators working on an individual while it travels from the old to the new population used by EvoTest is shown in figure 3.8. This is called the breeding pipeline. For every individual in the new population, a call is made to the bottom stage of the pipeline. Each stage calls its upper stage one or more times for the individuals it needs and operates on those and returns those to its caller. Normal lines are method calls and dotted lines are the returns with the individuals. The node "Multiple Breeding Pipeline" randomly calls one of its upper stages, where the percentages specify the distribution of those calls. It is important that each stage is implemented in an efficient manner, because they will be called many times.

3.7 Early Evaluation of EvoTest

There are several new elements in the approach EvoTest takes compared with previous work:

1. Using genetic programming with a complex language, Java
2. The use of static analysis on the test subject to help the algorithm

3. The use of collected traces to create the initial population

Despite these new features, I still encountered inefficient behaviour of the genetic algorithm during several test runs during development and I saw several areas of improvement. A fully working version of this early EvoTest was never completed, this explains the lack of a complete thorough evaluation. In this section I will try to highlight those problems. There are three major areas which give problems. The most important one, are the presence of methods which do not affect the current state. The second one is an inefficiency which was encountered while executing test cases, and the last one is related to the generation of test data. These problems are elaborated in the following three sub-sections.

3.7.1 Executing Useless Methods

When inspecting the test cases generated by EvoTest, there are certain problems visible. While testing class A with two methods it would always call useless methods in between the other two defined methods. Methods like *Object.toString()* and *Object.hashCode()*. When *String* values were involved, all the methods on *String* would be added to the test cases. All these methods do not change the state of any of the objects and were thus redundant. Sometimes test cases would explode in size when more and more useless objects were added to the search space. An object would take a *String* object as an argument. A method with a *Locale* argument would be called on the *String* object and more methods would be called on that object and so on.
Methods which do not change the state of the heap are called pure methods. This prevented efficient test case generation for classes which had too many pure methods, even more so when they took arguments of complex types. A method of removing pure methods had to be found, otherwise EvoTest would be useless in any but the most trivial cases.

3.7.2 Complex Constants

Data generation is done through random creation of new values and mutation of existing values in test cases. This is a big problem when the selection mechanism does not direct the mutation to the correct values fast enough or at all. This is the case when specific values are needed, then it can take a long time before these are found, especially in the case of strings, because of the infinite size of the search space and because the system does not know how close it gets to the target string.
This problem could be partly solved using symbolic execution with a good constraint solver, which could find the values used. The next chapter will explain the easier approach used by EvoTest.

3.7.3 Inefficient Test Case Evaluation

The first version of EvoTest used reflection to evaluate the generated test cases. This is relatively inefficient when many test cases are structurally the same but only differ in the values to the methods and constructors. It would be more efficient to compile the test case in that situation which would result in EvoTest being able to execute the test cases much quicker.

3.8 Summary

This chapter explains the largest part of the work I have performed for my master thesis. It explains in detail the workings of the algorithm I designed for EvoTest. After a short overview section, it is explained in section 3.2, how the chromosomes are designed. I decided to use a tree like structure, which is the most common structure in genetic programming algorithms.

Section 3.3 explains what methods EvoTest uses to create an initial population which can be evolved. At random, a minimal test case is created, which can execute the method under test with valid data.
Section 3.4 covers the topic of chromosome modifications through mutation and crossover. There are many types of mutation and crossover possible. Crossover is a method of combining two individuals in one or two new ones, with the goal of combining the good aspects of the two. Mutation makes small changes in the chromosome of one individual.
The description of the algorithm ends with section 3.5. In this section the focus is on the fitness calculation of the evolved test cases. This is done by instrumenting the system under test to record the coverage achieved when a test case is executed. There are several alternatives for executing the generated test cases and three of them are explored, generating source code, using reflection and directly generating java byte code. The section ends with the method used by EvoTest to convert the trace of branch coverage information, recorded using instrumentation, to a fitness value which can be used to select the best individuals from the population.
Because most aspects of the GA can be tweaked using various properties and its building blocks can be configured in certain ways, section 3.6 explains how the mutations and the crossover operators are combined in EvoTest.
Because the version EvoTest described in this chapter never got in a completely working state before it was modified to integrate the more advanced features described in the next chapter, it was never fully evaluated. The limited evaluation, which was done during development, and the conclusions derived from it, are discussed in section 3.7 of this chapter. The following chapter describes the changes done to the algorithm to make it more efficient and usable.

Chapter 4

Improving EvoTest

The early evaluation showed a significant number of possibilities of improving EvoTest. This chapter will explain all the optimisations added to EvoTest. Every optimisation takes it a bit further from a pure genetic algorithm. These improvements touch every aspect of the GA, the generation of an initial population and the overall architecture of the algorithm.

4.1 Two Stage Algorithm

In section 3.7.3, we discussed that it is most efficient if a compiled individual can be executed multiple times, so that the overhead of the compilation is spread over multiple fast executions. But in principle it is only necessary to evaluate and thus execute an individual once. To make this more efficient, the architecture of EvoTest has been changed in what I call a *Two Stage* genetic algorithm. The idea behind this is to split the evolution of the structure of the test case and the data of the test case. This is equivalent to the evolution of animal species, where there is a pure genetic component and a learning environment, which build on each other. To make this possible, the constants are removed from a test case to create *Parameterized Test Cases*.

Normal test case:

```
public void test(){
    TestObject o = new TestObject(5);
    o.test(6);
}
```

A parameterized test case:

```
public void test1(int i1, int i2){
    TestObject o = new TestObject(i1);
    o.test(i2);
}
```

Figure 4.1: The difference between a normal test case and a parameterized test case.

4.1.1 Parameterized Test Cases

The structure of the individual or test case is the sequence method and constructor calls and the data are all the constants used in a test case. Instead of compiling all the constants directly in the test, they

are replaced by variables so they can be given as parameters during the execution of the test case. This way, a test case can be compiled once, and many variants of one structural individual can be executed in sequence. These I call parameterized test cases, which are slightly different from the parameterized unit tests introduced in [43], because EvoTest only uses those for primitive types and strings. An example of a normal test case and the parameterized version is shown in figure 4.1.

4.1.2 Second Genetic Algorithm

The introduction of parameterized test cases splits the evolution of the structure and the data in two stages. This is a burden because two algorithms now work together but it is also an opportunity because the best algorithm for each stage can be used. The first algorithm is the main genetic algorithm described in detail in chapter 3, but with some small changes. The second algorithm can be anything that generates primitive data. This enables future work to extend this basic algorithm with more advanced techniques, see the section Future Work in chapter 5 for some possibilities.

Changes in the Original GA

Only small changes are needed in the old genetic algorithm:

- Constant nodes are different
- Generation of code is different
- Evaluation involves a second algorithm

While before, constant nodes had an actual value in the tree and there was a mutator which changed those values, now the constant nodes are 'empty'. They only reference one of the primitive types or types which can be generated by the data generators (see section 4.1.3).
The constant nodes do not have an actual value in the tree structure. An individual exists as a structure and a data array and many individuals share the same structure. The structure evolves in the normal way, but the evaluation of the structure consists of executing it many times with different data in the execution. The fitness value of the best execution of one structure will be the fitness value for that structure. If there is no data in a test case, for example there are only method calls and constructors calls without any primitive values, it is executed just once. An overview of the new algorithm is shown in figure 4.2.
There are multiple ways how the data can be chosen for each structural individual, or how the data search space can be explored. This is the domain of the classic evolutionary testing and many techniques can be used, such as:

- Random search
- Hill climbing
- Evolutionary algorithms
- Symbolic execution

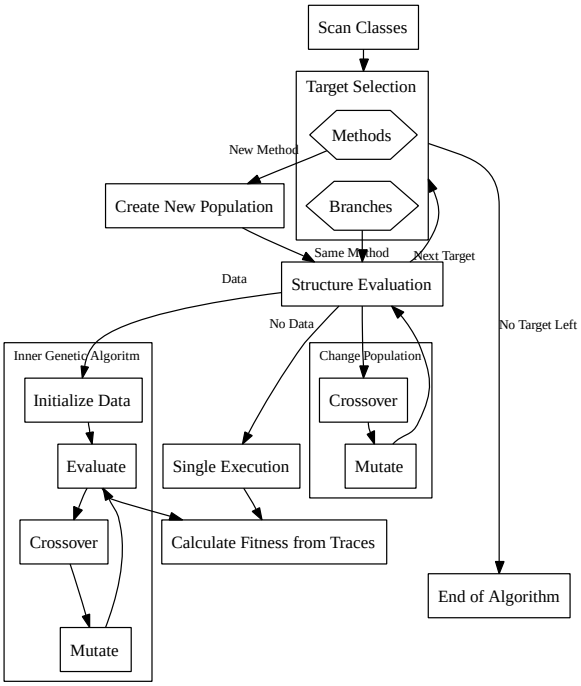

Figure 4.2: Overview of the two stage algorithm used by EvoTest

Random Search

Random search is the simplest method. Random values would be chosen for the constants and evaluated. The best execution would result in the fitness value for the structural individual. Random search works pretty well in many cases when the differences between fitness values of the different random values are not great or when the optimal values are easy to find. But because the search is undirected, it does not give any guarantees for finding a good solution.

Hill Climbing

Hill Climbing is a local search method. Local search methods explore the area around one initial candidate solution. The next candidate solution is the best solution of all the candidate solutions in the area of the last one. This strategy is very effective for search spaces which are smooth and do not have suboptimal plateaus where the hill climbing algorithm would get stuck.

Evolutionary Algorithms

The most logical approach in light of the total project is to also use an evolutionary algorithm, for example a genetic algorithm, for the generation of the data. The current version of EvoTest uses a pretty standard genetic algorithm as described by Goldberg [19]. Other evolutionary algorithms can also be used, for example evolution strategies [32], which perhaps perform better than the standard genetic algorithm used at the moment.

Symbolic Execution

Instead of searching for the correct data, it could be deduced by a satisfiability solver [42]. This method is called symbolic execution. The code of the test case is executed symbolically, which means all values are not real, but symbolic. All paths in the code are symbolically executed at the same time. When a branch is encountered, both paths are evaluated, but a different constraint on the symbolic values is added to each path. The combined constraints for the path which needs to be executed to reach the target need is now an algebraic expression. A constraint solver can solve this expression, the resulting solution will specify the values of the involved variables needed to execute the path. These values complete the test case. See also section 5.8.2.

4.1.3 Improved Test Data Generation

The current implementation of the second stage of the genetic algorithm uses a simple genetic algorithm approach. When a structural test case generated by the main GA has any constant nodes it will start a new GA run to evaluate it. When there are no constant nodes it does not need any data generation and it can evaluate the test case by just executing it once. When there are constant nodes, it will evaluate the test case multiple times in this second GA.
The test data is represented as an array of objects. Each element in the array corresponds to a parameter for the test case and can be of any type unlike other linear GA's where the elements of the array are all one type, for example booleans or integers. Although the types can be very different, the array sizes of the individuals all have the same size and have the same types in the same order.
This array representation allows easy mutation by changing a single value and simple crossovers by switching halves of two other data arrays. The only complication is keeping track of the type of each element in the array. When generating values for mutation we use a generator class that can create values of the appropriate type.

Mutation and Data Generators

The constants in our test cases can be of any type where one can write a generator for. It is easy to add generators of multiple types. When a generator exists for a data type, it will not be created by indirect constructor or method calls in a test case but will be encoded by a constant. So far, we have implemented generators for all the Java primitive types and their wrappers and String objects. Possible other types would be Date objects and Collections.
A generator should support two functions. A generate function, which creates a new instance of the constant and a mutate function, which will mutate a given instance. For example, the String generator can create the String *AB*, and later it can mutate it by appending the C character making the new String *ABC*. This is similar for numeric types, which can be increased or decreased when they mutate.

Data Generation Sources

The generators in EvoTest do not always randomly generate their values, but get their values from several different sources.

1. Purely randomly generated values
2. Constant values extracted from the class files of the system
3. Constant values used in a captured execution of the system

4. Previous generated values

EvoTest can be configured to change the ratio between these sources when they are available. Random generated values are always available and this is the easiest source of values. Constant values extracted from actual executions and the class files are obvious sources of good values. Values used as constants in class files are probably special values which have a meaning in some context. They could be, e.g., boundary conditions. The GA will only have to find the context but already has the value, speeding up the algorithm. Values are extracted using InsectJ described in appendix A

The ability to use values from class files and captured executions is important for values which are used in comparisons, but sometimes the comparison can not drive the evolution. This is the case with Strings. Methods which operate on a certain sequence of characters in a string argument they receive, it will be pure luck for the GA to find the correct sequence. When the sequence is not found, it will not drive the GA to that sequence, while numeric values will direct the GA to the correct value.

Many pieces of code have special checks for equal data, the well known *equals(Object)* method. Methods compare stored values in the fields of the object with values received as arguments. The chance that the same value will be generated twice in a row, once when building the object and once when calling the method, is very small. To help the system recreate this particular case more often, a small list of previous generated values is stored in memory, and every so often, it will pick a value from this list.

4.2 Purity Analysis

Another technique to improve the GA in EvoTest is the use of some form purity analysis. Purity analysis checks for each methods what fields they potentially modify. The idea of applying this technique came very early as a simple improvement to EvoTest.

When a certain path to the code is dependant on the value of a field in an object, that field has to be set to the correct value. This is most often done by invoking methods on the object which change that field. The GA randomly adds method calls to the test case, it also calls which do not change the state of the system. For example, the methods *Object.toString()* and *Object.hashcode()* will never change the state of the system and can be ignored by the GA. These methods are pure. There are several different definitions of purity, the one which is broadest and still correct to the intuitive notion has the most benefit to the GA.

The purity analysis described by Alexandru Salcianu and Martin Rinard in [41] was integrated in to EvoTest. It uses comprehensive pointer analysis to ignore temporary objects and reduce the number of false positives found by other methods. But tests revealed that this analysis still made many mistakes and declared several methods not pure while they obviously were pure. For this reason it is possible to declare pure methods using an *annotation*. The user of EvoTest adds the *Pure* annotation to a method it thinks will not change the state of the system and should not be added to the test cases.

EvoTest can be configured to use one or both systems to define purity for methods. The static analysis is performed before hand and the results are stored in a file. When EvoTest scans the class files under test and indexes the methods, the pure methods read from the file or annotated with the Pure annotation, are ignored. Only the remaining methods in the index can be selected for use in test cases. Thus reducing the search space size.

4.3 Trace Test Case Builder

As an improvement over the Random Test Case Builder there is a test case builder which uses information from previous executions of the system under test. A previous execution can be a test suite or a actual run of the system. The system is instrumented with extra code which records which methods get executed with which data and this information is stored in a trace. These traces are then used by the Trace Test Case Builder to create an initial population for the Genetic Algorithm with in theory a more realistic coverage of the class under test. This optimisation should make it easier for the system to achieve coverage in parts of the system which are easy to recreate by a manual execution, but hard for the genetic algorithm to find, mostly because of specific data in the system or particular complicated method sequences.

4.4 Summary

EvoTest in its first incarnation, described in chapter 3, was not able to do what it was designed for. There were two issues I wanted to tackle with an improved version. The first, was the ability to reuse the compiled version of a test case, so it can be executed many times with different data sets. This is mostly an efficiency improvement, but it has many implications for the rest of the algorithm. It separates the algorithm in two parts, which can both be tackled in different ways. This opens the door to combining a GA with several different techniques, such as symbolic execution. I call the result of this separation the two stage algorithm and it is explained in section 4.1.
The other concern was the complications caused by pure methods. Pure methods do not change the state of the object they belong to or any other object. The genetic algorithm assumes that all methods potentially change some state. This has the effect of needlessly increasing the number of possible test cases which need to be explored. This can be solved by applying purity analysis, which can statically analyse which methods are pure and which are not. This is discussed in section 4.2. It is also possible to mark pure methods with a special annotation. Pure methods will then be ignored by the algorithm when it generates test cases, speeding everything up and preventing the use of classes which do not affect the state.
The next chapter will focus on the amount of speedup the purity analysis brings and how the various parameters change the time used and coverage reached.

Chapter 5

Evaluation

EvoTest was designed to be able to generate test cases for many types of software and not only data structures or other niches of programming code. To demonstrate the ability of EvoTest to handle different kinds of programming code I ran various experiments on different test subjects in section 5.3. Still, certain types of code are hard to test with EvoTest. Pieces of code that write to disk or influence the outside of the Java VM need special care. The same applies to random testing. When the evolutionary aspects of EvoTest are removed, it reverts to random testing. I also ran some experiments with EvoTest in its random testing setting in section 5.4, to compare the two test case generation methods.

5.1 Test Subjects

I ran EvoTest in different configurations on different classes, shown in table 5.1. The table also shows the number of branches EvoTest tried to cover and how many it actually was able to cover and in how many seconds it did that. As can be seen most classes get a relatively high coverage in a reasonable time. The results shown in the table were the highest coverage EvoTest got in several runs with different configurations of the GA. The three most important configuration settings are the maximum number of generations, the size of the population and the a number I call the data generation ratio. Random testing also gets some good results, but not as good as EvoTest. It is quite a lot quicker though. This is due to the extra steps EvoTest has to perform.

NanoXML NanoXML is a small open source XML parsing package. Most experiments were run on the *XMLElement* class which contains a lot of small methods. XMLElement is basically a container method for elements in a XML document. So most methods only have a hand full of branches. The

Subject		**EvoTest**		**Random**	
Name	**Branches**	**Coverage (%)**	**Time (s)**	**Coverage (%)**	**Time (s)**
NanoXML XMLElement	121	90.08	369	80.17	101
HashMap	50	94	180	72	43
BitSet	124	100	495	86.29	133
TreeMap	39	92.31	13	46.15	29
StringTokenizer	5	100	5	100	2

Table 5.1: Test Subjects for EvoTest, and the best results found with EvoTest compared with random testing

most complicated methods are the *toString* and *equals* methods which contain many checks. Most methods take strings as arguments, but most are not used in any computations but only stored in fields of the class and later retrieved, these are variations of getter and setter methods. EvoTest generates strings at random and the class only performed equals branches on strings. This means that the data in the test case generation is not extremely important, I decided to generate a normal amount of test data for every evolved structural individual as described in section 4.1.

Collection Classes Many other papers, like [42] use the Java collection classes as classes which contain explicit state. Most often the state is only controlled through the sequence of method calls and not the data used in the method calls. For example, the code which triggers the doubling of an array inside the *ArrayList* class gets executed when the *add* method is called a certain number of times and it does not matter what is added to the class. Certain classes do depend on the data used, examples are the sorting collection classes, such as *TreeMap* and *TreeSet*. *BitSet* which uses an array of longs to represent an arbitrarily length array of bits. Classes which only depend on the number and sequence of method calls and not the values of the arguments used in those calls used a minimal amount of test data generation. This means that EvoTest only generated one set of data for the execution of the generated test cases for the classes *HashMap* and *ArrayList*, reducing EvoTest to a one stage algorithm. With the other classes several settings were tested, from the minimal amount to twice the default setting.

5.2 Testing Setup

All test were run on my personal home machine which has an Athlon XP central processing unit which runs at 2.15 GHz and has 1 gigabyte of main memory with the latest version of EvoTest. The Java virtual machine (VM) used in all experiments is version 1.6.0 from Sun [37], which is still in beta but did not cause any problems. The VM was given 512 megabytes of memory for the heap which was enough for all the experiments. All experiments were run from within the Eclipse 3.2 [22] Integrated Development Environment running in Windows XP, but initial development of EvoTest was performed on various GNU/Linux machines.

Most settings of the genetic algorithm were kept the same between all experiments. We did not use elitism, which means the best individuals in a population were not guaranteed to survive to the next population. Our mutation rates were set to these values:

- 70% Method Introduction
- 15% Method Removal
- 15% Variable Introduction

I chose for a low percentage of method removal because methods which are not needed can be removed from a test case by a later stage. This refinement of test cases is not done by EvoTest. Method introduction is the most important mutation and is chosen to dominate the other mutations. New individuals created only contain a call to the method under test, extra methods are needed to change the state of the class under test. To add these quickly, a high method introduction mutation rate was chosen. When individuals are created with a random selection of additional methods to the class under test, this rate can be lowered and the method removal mutation rate can be increased.

Variable introduction does not change the behaviour of the test case, but only facilitates other mutations. The ratio of 15% is enough to promote almost all objects to a variable in several generations. These new variables are then used to add methods to the objects used as arguments to other calls.

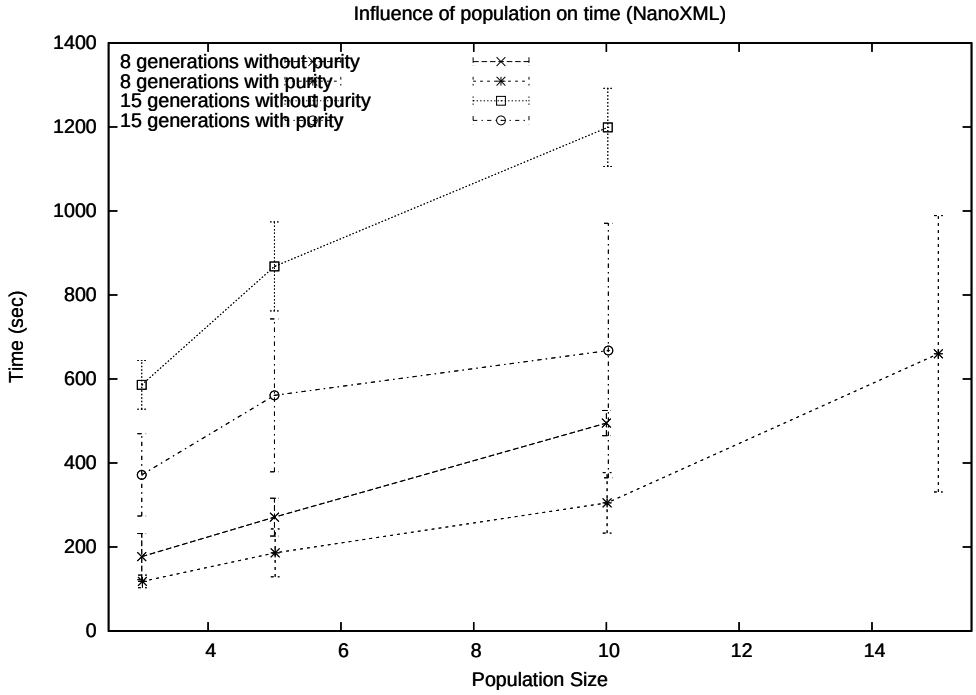

Figure 5.1: Increasing the population size causes a linear time increase

5.3 Results

I ran many experiments to evaluate the performance of EvoTest. Most were run on the XMLElement class of NanoXML. I mostly varied the maximum number of generations it was allowed to spend on one target, and I varied the number of individuals in the population. In the collection class experiments, I also varied some other settings, like the amount of generations and the population size of the data generation part of the full genetic algorithm which is controlled by the data/gen ratio variable.
First I will explore some of the more obvious properties of EvoTest. What are the effects of the following properties of the algorithm on the time used and the coverage achieved?

- Population size
- Maximum number of generations
- The ratio between data and structural evolution (data/gen ratio)
- The use of purity information

Also some more practical benefits of purity information will be shown. At last I will explore the nature of the two stage design of EvoTest and how that affects the speed expressed in evaluations per second.
The experiments without the use of purity information generally took more time and sometimes failed because of side effects by invoked code, such as the creation of files on disk. This limited the number of test runs which resulted in less accurate data about average time and coverage numbers.

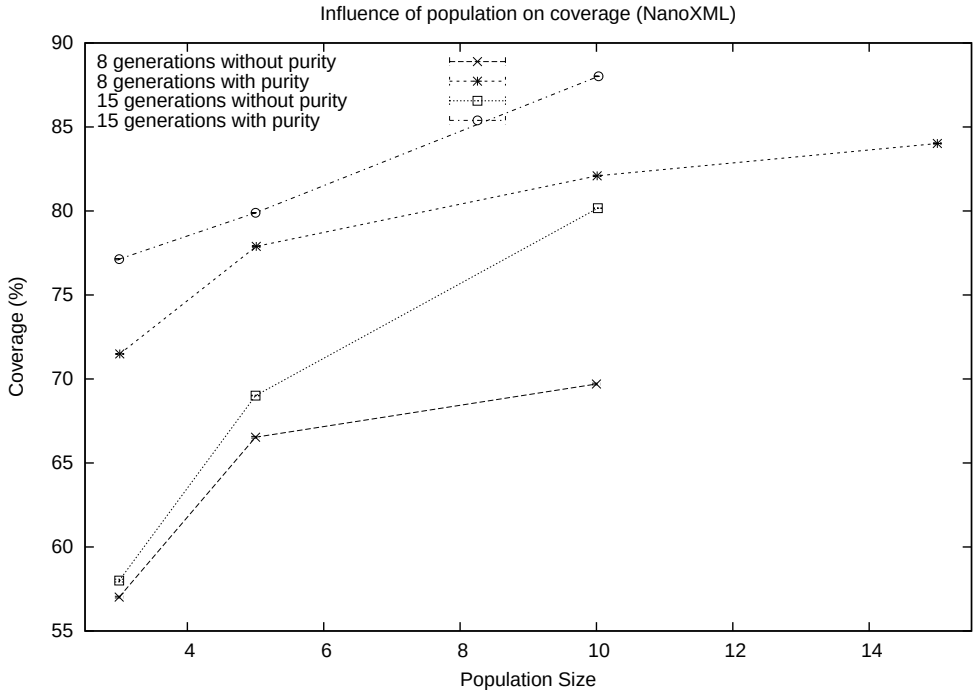

Figure 5.2: Increasing the population size causes a logarithmic coverage increase

5.3.1 Population Size and Time

When the number of generations is kept constant and the population size is increased the time used by EvoTest increase is constant, this can be seen in picture 5.1. The number of generations is kept constant but the population size is increased. Already it can be seen that using purity information has a positive effect on the time used by EvoTest.

The linear behaviour can be explained by how the algorithm works. When a target is covered by the algorithm it continues to the next target and the number of generations is reset to 0. When the maximum number of generations is reached, the next target is selected. Increasing the population size does not always help cover a target more quickly, but when it is not able to cover a target it does increase the time spend trying to cover it, causing the linear time increase.

5.3.2 Population Size and Coverage

The previous subsection showed that when the population size is increased, the time used is also increased, but what is the effect on the coverage achieved by EvoTest? It can seen in figure 5.1, that when the number of generations is kept constant and the population size is increased, that the coverage also increases. It does not increase in a linear fashion like the time, but more like a logarithmic trend toward the maximum coverage achieved. The maximum would be when EvoTest is run with an infinite amount of time. If this was possible and EvoTest would not skip any possible states, all branches which are not infeasible would be covered. In practise, only branches which are relatively easy to cover will be covered, but even then, the generated test cases can become quite complex.

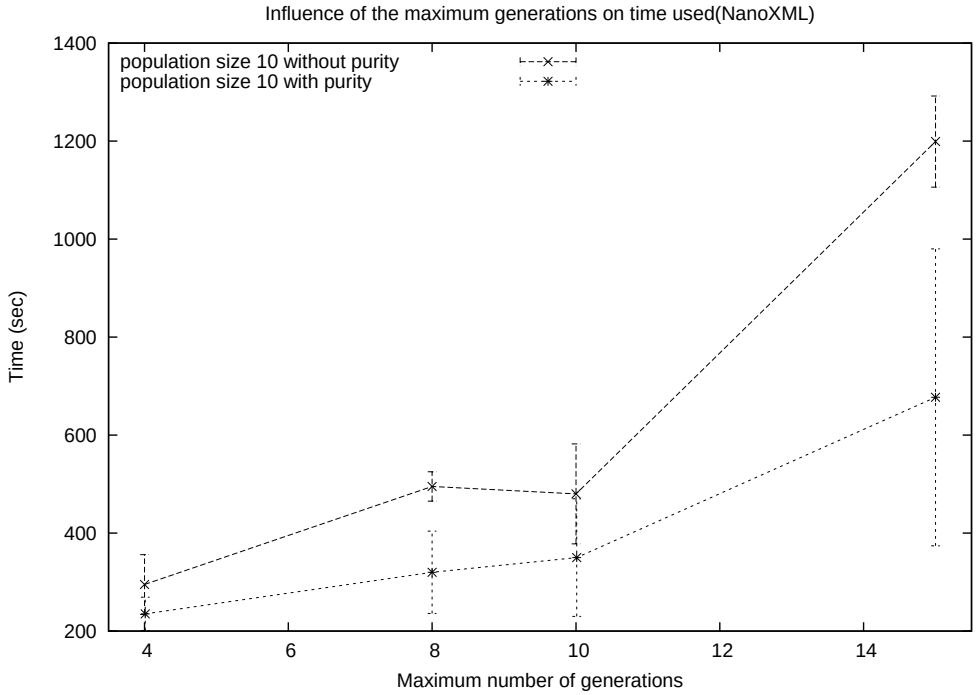

Figure 5.3: Increasing the maximum number of generations causes a super linear time increase

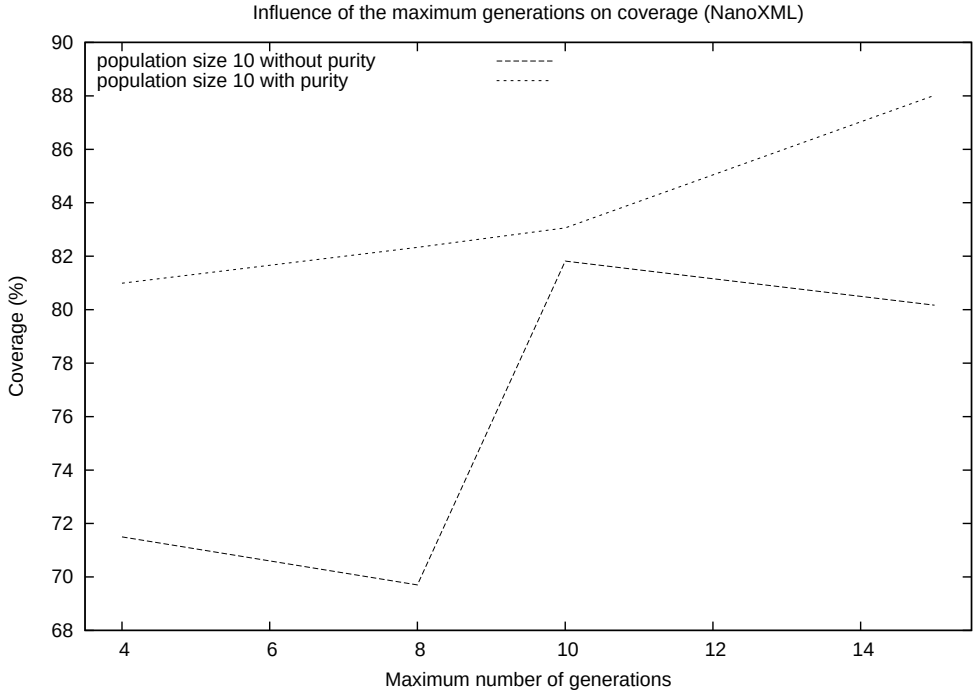

Figure 5.4: Increasing the maximum number of generations causes an increase in coverage

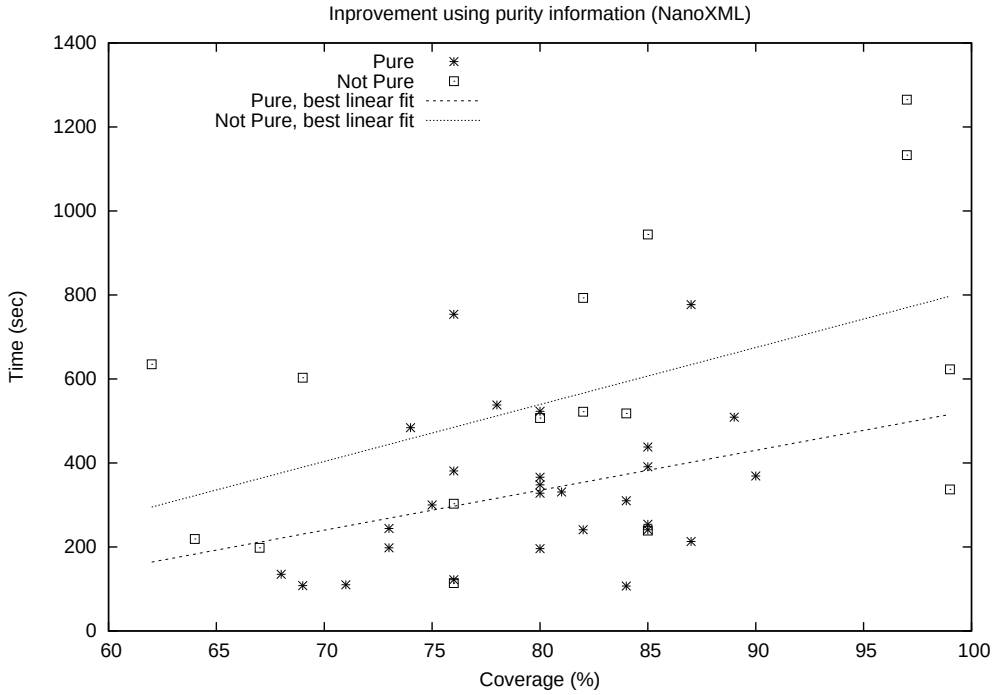

Figure 5.5: Using purity information improves coverage for a given time

5.3.3 Maximum Generations and Time

When the maximum number of generations are increased the time used on a normal run of EvoTest increases. This increase appears to be linear, but it is not very clear if this is really the case. This can be seen in picture 5.3. If each execution of a generated test case takes the same amount of time the time increase would probably be linear. But when a population is allowed to evolve for a longer time, the size of the test cases tend to grow because the settings used by EvoTest has an emphasis on growth. Larger test cases tend to take a longer time to execute, which causes a super linear increase of the time spend in EvoTest.

5.3.4 Maximum Generations and Coverage

Increasing the number of generations generally helps the coverage of the class under test. But it is not clear what kind of relation there is. I believe there is a logarithmic correlation just like the population size and coverage in subsection 5.3.2. But I can not prove it with these results. The results are displayed in figure 5.4.

5.3.5 Purity Information

In the previous subsections it can be seen that increasing the population size and the maximum number of generations increases the coverage of the test subject. But it is very clear that the use of purity information is very important as well. In all figures it can be seen that the performance is greatly increased when purity information is used. In figure 5.5 all the measurements are plotted in one graph. It is very clear that using purity information almost doubles the coverage/time performance of

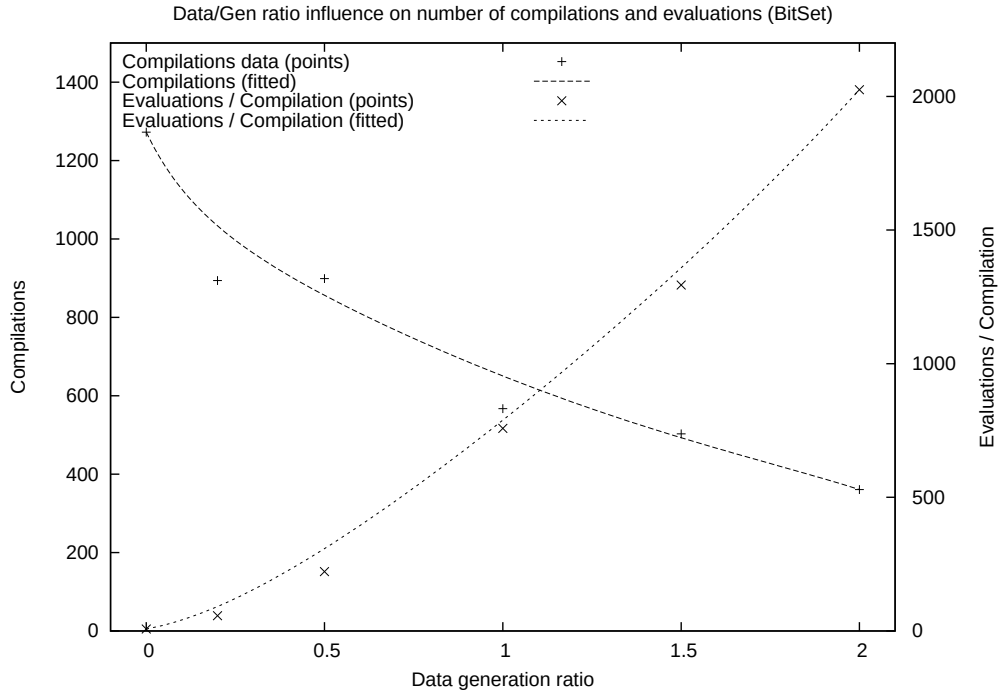

Figure 5.6: Increasing the data/gen ratio increases the number of evaluations for each evolved individual

EvoTest.

Using purity information also has another less obvious performance increase which is not only apparent in the coverage reached and time used. While developing EvoTest it became clear that not using purity information for normal classes caused the evolutionary process to run out of control. When a class used the String class for one of its arguments, EvoTest would use that class and would assume it has state. But it did not know the fact that String objects are immutable. Trying to change the state of strings, it would add many calls to the string objects. The result was even slower execution, but also introducing other classes, like the *Locale* class. Something similar happened when no purity information was used on XMLElement with many generations. Sometimes it would pull in many other classes, including classes which would write files to the disk and slowing everything down even more and it almost crashed the operating system due to the number of files created. This is one of the reasons why the non purity experiments are not very exhaustive.

5.3.6 Two Stage Evaluation

Besides evaluating what influence the number of generations, the population size and the use of purity information has, we also want to know what the effect is of the two stage algorithm on the results. For this, I selected one relatively easy problem which I knew benefited from the multiple executions of a single test case with different data and was not to large. This way it could scale to higher numbers of executions of a single test case without running out of time. I used the BitSet class. The number of generations and the population size were kept constant on 15 and 5 respectively and I increased the data/gen ratio number from 0 to 2. Besides time used and coverage achieved, I recorded how many times a test case was evolved and compiled and how many times one was executed with different

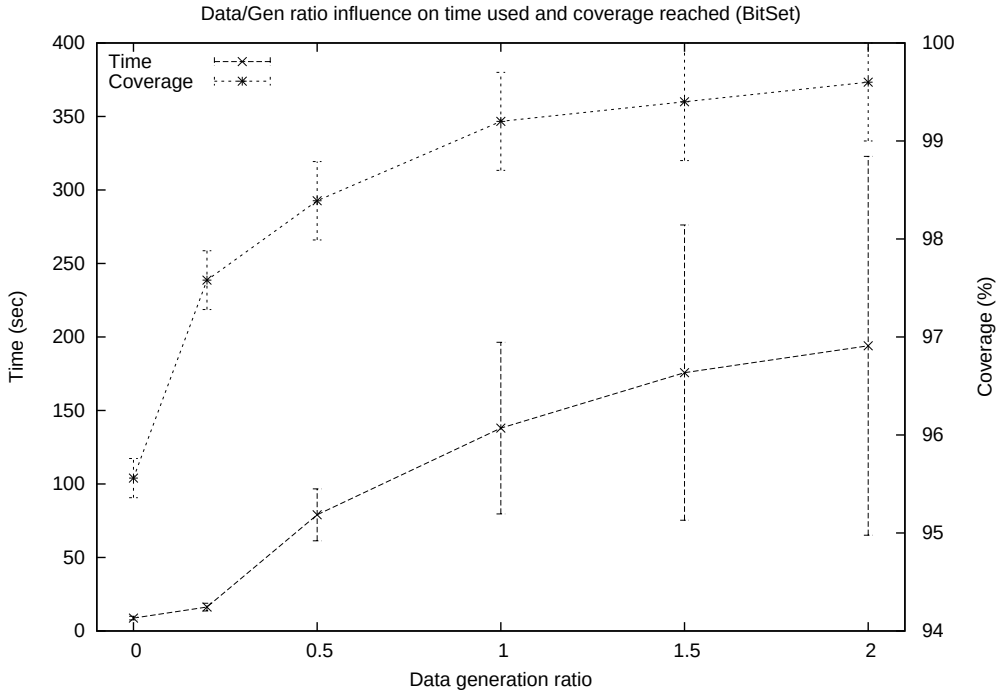

Figure 5.7: Increasing the data/gen ratio increases time spend and coverage reached

data. It can be seen in figure 5.6 that when the data/gen ratio is increased, the number of evaluations for each compilation is increased. This is to be expected, because when this ratio is increased, the genetic algorithm which generates the data for each parameterized test case has its population size and maximum number of generations increased. A ratio of zero results in 7 evaluations for each compiled individual on average while one compiled individual is evaluated 2000 times on average with a ratio of two.

The consequence of the increased number of evaluations is the increase of time and the increase of the coverage achieved, this can be seen in figure 5.7. While BitSet did not get fully covered in any of the experiments when the data/gen ratio was on 0.5 it did get covered 3 out of 4 times when the ratio was 2.0. Because EvoTest becomes more efficient when it executes one test case multiple times the time increase is less than would normally be expected. Besides spending relatively less time evolving the structural test cases, which takes quite some time, there are two other effects which decrease the time used. The first one is a practical one. The Java virtual machine used optimizes the code when it is executed several times, this is called a Hot Spot compiler. When a test case is executed more times than a predefined number of times it will be optimized by the VM. I believe this helps keep the time down a little bit but a larger reduction in expected time comes from the fact that in the case of BitSet less structural individuals are needed. A factor 3 reduction in the number of compiled test cases can be observed when the ratio is increased from 0 to 3. In the case of BitSet several branches are very hard to cover but they can be found by changing data values only.

Making it possible to configure the algorithm to dedicate more or less resources on data generation compared to test case generation proves useful when it is known what is more important in a certain situation.

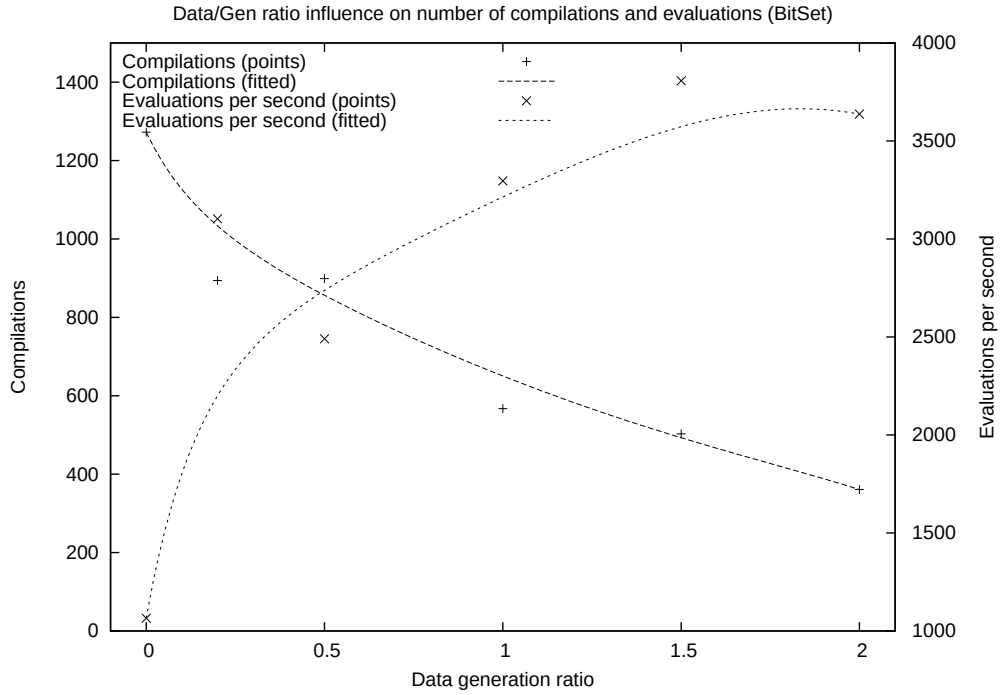

Figure 5.8: Increasing the data/gen ratio lowers the number of evolved individuals/compilations and increases the number of evaluations per second

5.4 Random Testing

All the previous evaluation sections concentrated only on EvoTest, but how does it stack up to random testing? In short, it fares rather well. I modified EvoTest to perform random testing by removing the evolution aspect and running with just one population which is evaluated once and does not evolve. There was only one other modification I had to make. By default, EvoTest creates new random individuals by creating just one call to the method under test and nothing else. Without evolution, EvoTest was not able to change the test case. This limited the coverage reached by EvoTest in random testing mode, giving the normal EvoTest an unfair advantage. To remedy the situation, the random individual creation code was changed to add random methods. The number of methods added was derived from a geometric distribution with a probability of 0.5. This added on average two other methods to a test case before it was evaluated. This improved the random testing with NanoXML, from only 40% coverage before the change, to a more realistic 80% coverage, after the modification.

In figure 5.9, I compare two samples of executions, EvoTest in normal mode and in random mode. The two subjects, NanoXML and BitSet, show very different behaviours, but the general trend is that random testing is faster and EvoTest reaches a higher coverage. With NanoXML, EvoTest needs a lot more time to get high coverage than random testing, but gets a little higher coverage in the long run. The BitSet experiment shows a much more pronounced advantage for EvoTest. In this case EvoTest is just as fast as random testing, but reaches a much higher coverage, up to 100%.

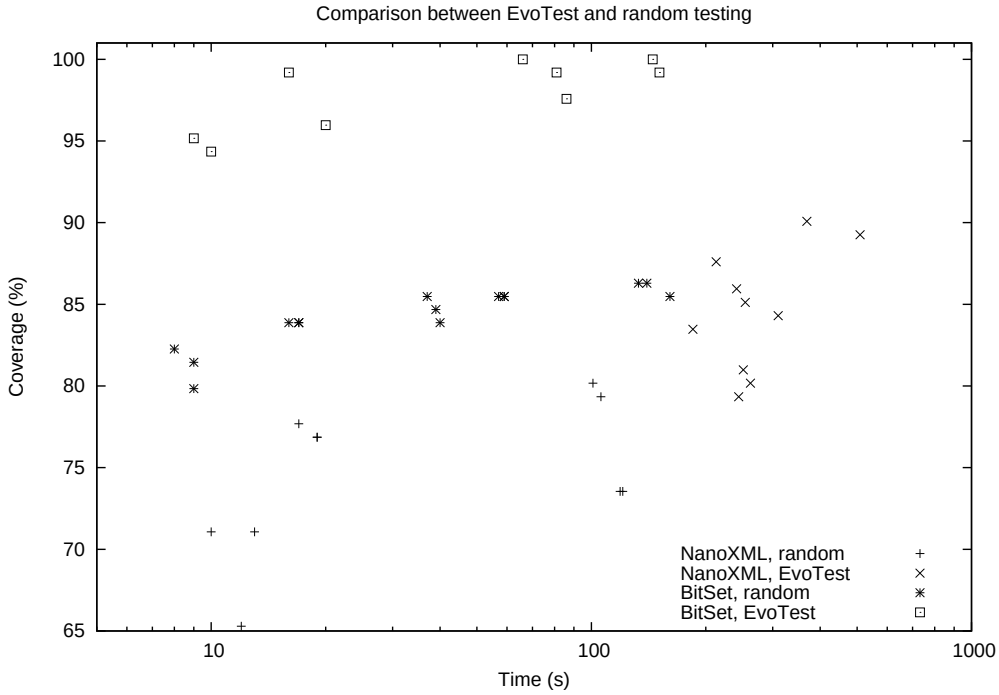

Figure 5.9: Random testing is sometimes much faster than EvoTest, it reaches its peak performance quicker, but does not improve much when given a lot more time

5.5 Output

An evaluation is not complete without an example of the type of test cases EvoTest generates. The individuals evolved by EvoTest can be translated to actual Java test cases in several different manners. I decided not to create standard JUnit test cases but make a small executable program. This was also done for the purity analysis.

Purity analysis needs a well defined entry point and a complete java program where all the relevant methods are reachable from that entry point. For the libraries used we did not want to write programs which used all the methods, an easy solution was to run EvoTest without purity information on a test subject and use the created test cases as input for the purity analysis.

Each evolved structural individual is transformed to a static method and each data set is converted in a call to its method. All these calls are put in the main method which can be invoked by executing the class file to execute all test cases. Figure 5.10 shows some of the generated test cases for the class BitSet.

It is very obvious that these are not human created test cases and it would be useful if they were more natural. Removing the full class names and using import statements would help, as would be more natural variable names. A separate clean up step could perform those changes, including removing useless statements, for example copying a variable to a new variable when it is not needed. The algorithm does not try to create the least amount of test cases, which can result in a large set of test cases which, in theory, can be reduced to a smaller set.

```
public class TestBitSet {
   public static void test0(int a,int b,boolean c,int d)throws Exception{
        eval.java.collections.BitSet var217 = new eval.java.collections.BitSet(a);
        var217.set(b,c);
        var217.nextSetBit(d);
   }
   public static void test1(int a,int b,int c)throws Exception{
        eval.java.collections.BitSet var862 = new eval.java.collections.BitSet(a);
        var862.flip(b,c);
   }
   public static void test2(int a,int b,int c,int d,int e)throws Exception{
        eval.java.collections.BitSet var653 = new eval.java.collections.BitSet(a);
        var653.set(b,c);
        var653.clear(d,e);
        var653.or(new eval.java.collections.BitSet());
   }
   public static void test3(int a,int b)throws Exception{
        eval.java.collections.BitSet var775 = new eval.java.collections.BitSet();
        var775.set(a);
        var775.clear(b);
   }
   public static void test41(int a,int b)throws Exception{
        eval.java.collections.BitSet var43 = new eval.java.collections.BitSet();
        eval.java.collections.BitSet var476 = new eval.java.collections.BitSet();
        int var566 = a;
        var476.set(b,var566);
        var476.andNot(new eval.java.collections.BitSet());
        var43.and(new eval.java.collections.BitSet(var566));
        var43.or(var476);
   }
   public static void main(String[] args) throws Exception{
        /*
         * Covers these targets:
         * [BitSet.checkInvariants()V brid:6 dir:true linenr: 96]
         */
        test1(140, 140, 2780);
        /*
         * Covers these targets:
         * [BitSet.clear(II)V brid:30 dir:true linenr: 385]
         */
        test2(3420, 2155, 3490, 56, 4549);
        /*
         * Covers these targets:
         * [BitSet.clear(II)V brid:33 dir:true linenr: 405]
         */
        test2(3420, 2155, 3490, 56, 4549);
        /*
         * Covers these targets:
         * [BitSet.set(I)V brid:21 dir:true linenr: 258]
         */
        test3(4420, 2436);
        /*
         * Covers these targets:
         * [BitSet.or(Leval/java/collections/BitSet;)V brid:76 dir:false linenr: 763]
         */
        test41(4004, 2383);}}
```

Figure 5.10: Output for the class BitSet, showing 5 out of 57 test cases

5.6 Summary

EvoTest is able to generate test cases with a high branch coverage. It is able to do this in a reasonable amount of time for the subjects I used in the experiments. Giving EvoTest more time, in the form of a bigger population or more generations, raises the average number of branches the generated test cases cover. Even more important is the evaluation of the effects of purity information on the algorithm in subsection 5.3.5. In all the subsections before that one, it can already be seen that the use of purity information always improves the time used and the coverage reached by EvoTest. This trend is consistent over all the experiments. Subsection 5.3.6 evaluates the ability to shift more time from and to the generation of test data, compared to the evolution of the structure of the generated test cases, using the data/gen ratio parameter. It is shown that certain classes, such as BitSet, greatly benefit from a high data/gen ratio, which means that more time is spend on generating more test data, while others are less affected.
Section 5.4 compared EvoTest with random testing. It is shown that, although EvoTest is slower in some cases, it is able to outperform random testing in coverage reached by the generated test cases. In section 5.5, I give a sample of the type of test cases EvoTest generates.

5.7 Conclusion

The results shown in this chapter are a good indication that EvoTest is capable of creating unit tests in a reasonable amount of time with an adequate testing criteria. A direct comparison with other test generation methods, besides random testing, is not practical because of several factors. EvoTest can not be directly compared with other systems which use a different testing criteria. The generated test cases will differ too much and it is not always clear what testing criteria is used. But even when they are the same, performance can not always be directly compared because of different computer and software systems used. This is not the case with random testing because I was able to revert EvoTest to a random testing mode.
In the end, I tried to show how and how well EvoTest works on its own merits, without directly comparing it to any other test case generators. These results are encouraging and support the goal of being able to generate unit tests for real world programs.

5.8 Future Work

5.8.1 Dependency Analysis

If the purity analysis described in section 4.2 is taken further, it determine which fields an impure method modifies. This information can be incorporated in the mutation step of the GA. To use this information, the fitness function also needs to be adapted to do some more analysis. When the fitness value is calculated, the fields which influence the last critical branch missed toward the target need to be determined. This can be done in advance using static dependency analysis.
An attempt to implement this using an existing back wards slicer failed because of the immaturity of the slicer. The results of the slicer on a branch are all the fields, arguments and constants influencing the branch, directly or indirectly, possibly ranked by influence. This information would then be stored in the individual for the mutation phase in the transition to the next generation. Only the methods which change the fields which affect the branch should be added, or deleted and all the other methods

should stay the same. See subsection 3.4.1 for more information about those mutations. This should reduce the search space and increase the chance in a favourable mutation.
Which arguments affect a branch can also be used, but at the data generation stage of the GA, see subsection 4.1.3. All the arguments which do not affect the branch can be left alone. This can greatly reduce the search space size.

5.8.2 Symbolic Execution

The optimisations described in the previous sections, transform the genetic algorithm to a pseudo genetic algorithm, which is part GA and part pure analysis. The next step is symbolic execution, which in theory could remove the search for correct data altogether and replace it by a logical deduction. Subsection 4.1.2 explains how symbolic execution would fit in the current implementation.
The use of symbolic execution to find the optimal solution at this stage would be the most direct approach. Symbolic execution together with constraint solving, can be used to directly deduce the values of the constants needed to reach the target. Symbolic execution only works on actual code, so a test case would be generated by the GA and in the data generation phase the whole test case would be symbolically executed. The purity and dependence analysis could still be used to help the GA but would not be needed by the symbolic execution.

Drawbacks

The drawback of symbolic execution is that it is not general enough and can fail to solve all problems. Although the limitations differ between different constraint solvers, this is mostly the case in the presence of floating point numbers, non linear constraints and API method calls. In these cases the constraint solver can not solve the constraints and gives up. GA's never give up and always return the most optimal solution they can find.
But there are ways to combine the symbolic execution with search methods, like GA's. This can be done by letting symbolic execution solve a part of the problem. We can let it execute the created test cases and let it solve the constraints to reach a particular path. If it is able to solve those constrains, the algorithm is done for this target. If it is unsuccessful, a GA or a different search algorithm can take over.

Data Search Space Reduction

When the constraint solver is only able to partly solve the constraints, those constraints can then be used to reduce the search space of search algorithm, for example the GA used by EvoTest. There are several approaches to do this. The best one would be to constrain the data generation to certain ranges. For example, if the solver is not able to deduce an exact value for an argument to reach a certain path, but able to say that it should be positive, it halved the search space for the GA left to search.

Appendix A

Instrumentation - InsectJ

All the interaction of EvoTest with existing Java classes is done by a tool called InsectJ. InsectJ was developed concurrently with EvoTest as a general Java class file instrumentation tool. InsectJ is mainly responsible for collecting the coverage information of a test case execution for the evaluation of the test case. Besides that it is used for some other less critical functions also performed by EvoTest:

- Branch coverage information for evaluation
- Resetting of static fields in classes
- Extracting constants from classes
- Simple call graph creation
- Tracing actual executions

This chapter will explain how InsectJ works in general and together with EvoTest. Mainly because InsectJ is so tightly integrated with EvoTest and is responsible for a lot of functions performed by EvoTest.

A.1 Introduction

EvoTest needs to inspect and instrument Java programs. It needs to collect static and dynamic information from real executions and from generated test cases for their evaluation. In the case of Java programs, there are two common approaches for collecting dynamic information. Either using instrumentation, which can be added by hand or automatically, or by using the debugging facilities of the runtime system (e.g., the Java Virtual Machine Profiling or Debugging Interface [25, 26]).
In the case of EvoTest manual instrumentation is out of the question and the runtime does not give enough information of the execution. InsectJ is generic framework that enables the collection of various kinds of runtime information, such as data on the execution of various code entities (e.g., branches and paths) and constructs (e.g., assignments). And it also lets users define how to process the collected information, which is ideal in the case of EvoTest which needs to process the information directly.

Instrumentable entity	Information available
Method entry	current object and argument objects
Method exit	return or exception object
Before method call	target and parameters objects
After method return	return or exception object
Field read	field and owner object
Field write	old, new and owner object
Basic block	*none*
Before or after a branch	type and distance

Table A.1: Instrumentable entities and information available for each entity.

A.2 Approach

The goal was to provide an extensible and configurable framework for gathering information from an executing program. Examples of this type of information include coverage, profiling, and data values from specific points in a program's execution. Further, we would like our framework to provide the information that it gathers in a generic manner. Such a capability lets users easily build tools and experimental infrastructure using the framework.
The framework consists of three parts: (1) a common instrumentation system; (2) a library of different instrumentation probes; and (3) monitor interfaces which collect and process the information from the probes. The monitor classes which implement the monitor interfaces are to be created by the user collecting the information.

A.2.1 Library of Probes

InsectJ includes a predefined set of probes. A probe is typically associated with a program construct (e.g., a method call or the catching of an exception). Code constructs that can be instrumented with probes are referred to as *instrumentable entities*. For each instrumentable entity, the framework can provide various different types of information associated with that entity. For example, in the case of a method call, it can report the target object and all of the parameters passed to the called method. Table A.1 shows a partial list of instrumentable entities, along with the information available for each entity.
Instrumentable entities are similar in spirit to join points in AspectJ [5], but are different in several respects. It is much easier in InsectJ to add instrumentable entities by creating a new *probe inserter* and it supports entities which can not be monitored by aspects. For example, there are no join point counterparts for entities such as basic blocks, predicates, and branches.

A.3 Implementation

InsectJ is implemented for the as a set of three plug-ins for the Eclipse platform:

1. Core plug-in: basic instrumentation framework
2. Probes plug-in: a collection of probe inserters
3. GUI plug-in: an user interface for InsectJ

The first plug-in is called the *Core plug-in*. It implements the basic functionality needed for instrumentation. Processing the configuration file, loading the classes and it contains a library of functions for instrumenting byte code. When used in Eclipse, the Core plug-in exposes an extension point for Probe Inserter plug-ins to extend. The *Probes plug-in* contains most of the probe inserters developed with InsectJ. Each probe inserter is responsible for instrumenting a code construct. The third plug-in, the *GUI plug-in*, provides a GUI for use in Eclipse, that makes it easier to use the framework. Wizards make it easy to define monitoring tasks, which parts of a system needs to be instrumented and the creation of new monitors to process the data created by the instrumentation.

A.3.1 Dynamic Instrumentation

InsectJ uses the Byte-Code Engineering Library [8] to rewrite the byte code. There are two ways of instrumenting a Java program, either as processing the class files once or by processing them just before they are loaded using the instrumentation facilities available in Java 5.0 [23]. Right before a class is loaded, InsectJ checks whether the class must be instrumented and, if so, invokes the appropriate probe inserters. Each probe inserter instruments some or all of the class's methods by inserting calls to the appropriate monitor in correspondence of the instrumentable entity for which the probe inserter is defined. For example, the probe inserter for method entries adds instrumentation before the first statement in each instrumented method. The kind of instrumentation added depends on the entity being instrumented and on the information that the user needs to collect. For example, the instrumentation inserted at method entries differs depending on whether the user is interested in collecting information on the parameters passed to the method or not.

On-line, or dynamic, instrumentation has two main advantages. First, it is totally dynamic and, thus, more flexible. Second, it does not require to keep several copies of the program. One problem with on-line instrumentation, though, is that users must pay the cost of instrumenting the code for each execution. To eliminate this problem, it is also possible to do off-line, or static, instrumentation.

Figure A.1 illustrates, using a class diagram, a partial design of our infrastructure and how probe inserters are defined. This design allows for extensibility that goes beyond the definition of new monitors. In fact, in case users need to add an instrumentable entity that is not yet included in our set, they can extend the extension point defined by InsectJ, as shown in the figure. More precisely, new probe inserters must extend the *AbstractProbeInserter* class, which provides methods to simplify the actual instrumentation. The new probe inserter must also define a new monitor interface, which must in turn extend *MonitorObject*. After the probe inserter has been defined, it behaves like any predefined probe inserter. The next subsection provides more details on several probes inserters used by EvoTest.

A.3.2 Probe Inserters

EvoTest uses a set of probe inserters from the InsectJ instrumentation suite. The details on how these work and how the data is processed will be described in this subsection. The part of EvoTest which implements the evaluation of the test cases are actually several probe inserters and a monitor working together. The main probe inserter is the branch probe inserter which instruments every conditional branch. Together with the method entry and exit probe inserters they insert calls to a monitor object which combines the information in branch traces. The branch probe inserter is extended so EvoTest knows how many branches are instrumented and which type they are. This is needed for targeting specific branches and knowing what the coverage is of a class. How the trace gets created can be read in subsection 3.5.3.

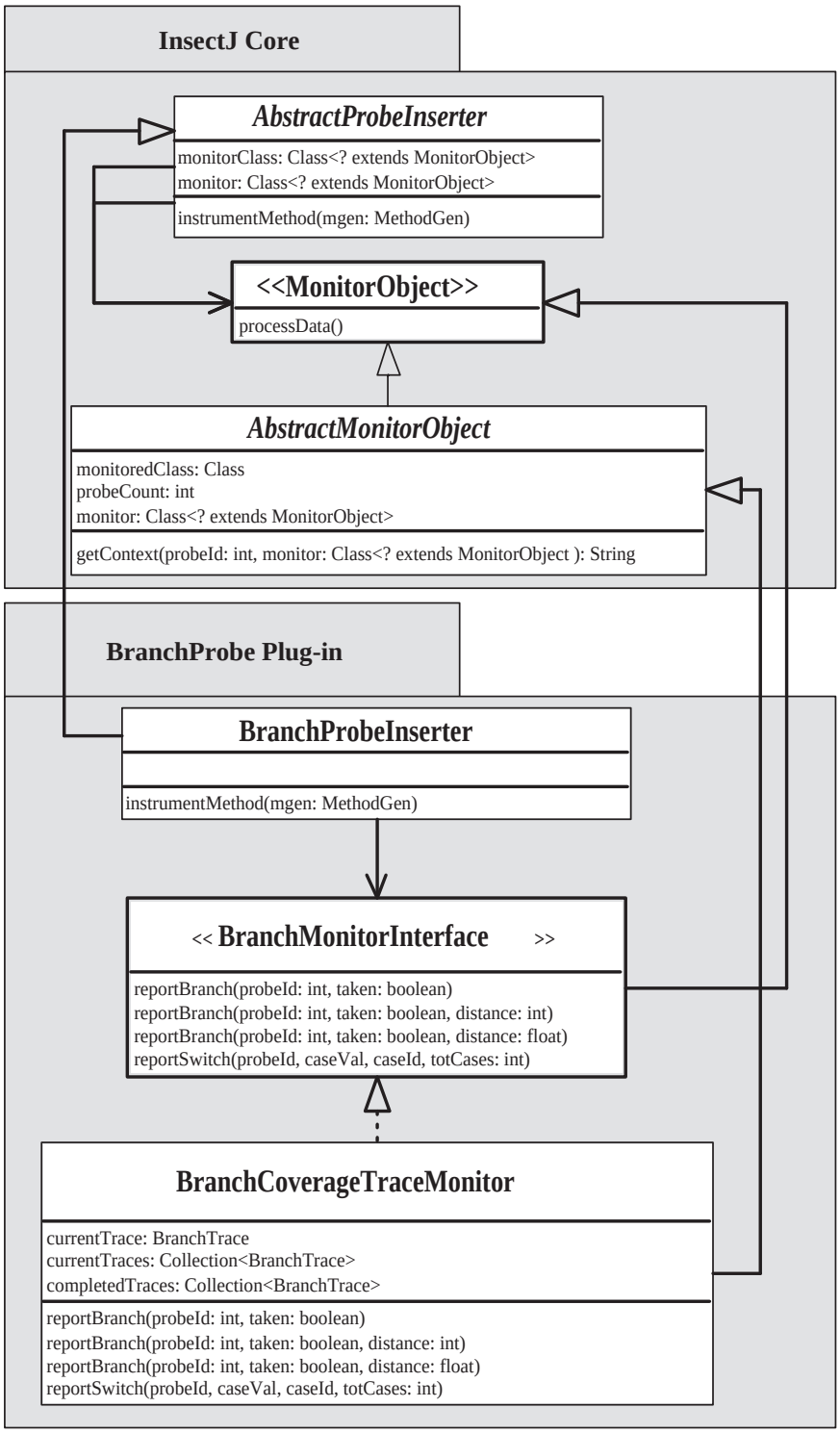

Figure A.1: Class diagram that shows how a specific probe inserter (for branches, in this case) extends the core plug-in.

Method Entry and Exit

The method entry probe inserter can record the *this* object and the arguments passed to it, while the method exit probe inserter can record the return or thrown exception object. But EvoTest only needs to know when a method is entered or exited and does not use the extra information available at those locations. This saves a quite some extra overhead, especially with many primitive arguments because those would need to be wrapped in objects and stored in an array. Method entry instrumentation is thus easy, method exit is a little more complicated because there can be many exits at the same time, while some are not very obvious.
A method can return through a return statement, or by an exception being thrown and not caught. We handle both cases by instrumenting every return statement and adding a catch block which will catch any Exception of the super interface *Throwable*.

Branches

The branch probe inserter is the most complex inserter I wrote, mostly because of the distance value I want to extract from branches where this is possible. When comparing two primitive types, for example integers, the distance value is the difference between those values. When one value is compared to 0, then the distance value is the size of the value. In theory this is very simple, but there are many possibilities in Java byte code. There are 18 different conditional branches in four groups:

- Two value integer compare ($= \neq \geq \leq < >$)
- Integer to zero compare ($= \neq \geq \leq < >$)
- Object compare ($= \neq \ isnull\ isnotnull$)
- Switch (table and lookup)

Comparing of other primitive types is done by 6 instructions which compare two doubles, floats or long values and return an integer value of -1, 0, or 1. This value is then compared with 0 to make the actual branch. The same strategy is used for the *instanceof* operator. When the branch probe inserter finds one of the integer compare to zero instructions, it checks for one of those instructions just before it.
For the branch monitor, there are 4 different types of branches, with different types of information:

1. A normal branch with no information (object compares)
2. An integer branch, where the distance is an integer
3. Any other primitive branch, where the distance is cast to a float
4. A switch, which records the value, the case-id and the total number of cases

For the three normal branches, a boolean is also passed, indicating if the branch is taken or not, because only the distance value is not sufficient for that when the value is 0. Case 1 is the easiest to instrument, where only the boolean value has to be calculated. This is done by instrumenting at two locations, the fall through of the branch and its jump location.
Case 2 is much harder because the distance value has to be calculated before the branch is taken, so instrumenting the targets is not possible as with case 1. The current implementation is not the most elegant or efficient, but correct. The values to be compared are stored in variables. The monitor object

is loaded on the stack. Now the values are loaded and subtracted to create the distance value. Then they are loaded again and the jump is copied with different targets, if will now push the boolean on the stack and make the call to the monitor. The inefficiency comes from duplicating the branch to calculate what the result will be of the branch and the use of variables. Future implementations can be more efficient.
The situation in case 3 is very similar to that of case 2, but even more complicated, because the values need to be stored in different variables because they are bigger and of different types. The results also need to be cast to float values.
The hardest was case 4, because it needed to retrieve the case-id besides the value which is on the stack. The default case has id 0 and the other cases start counting at 1. Those id values are only available at the targets of the switch instruction, but it is confusing to have instrumentation for one probe (the switch) on different locations (just before the switch and at every target). This was solved by creating a map object when the class is loaded, which maps the values to the case-ids. Before the switch is taken, the value is mapped to its case-id so the call to the monitor can be made.

Class Reset

It was already covered in subsection 3.5.2, before every test execution, the state needs to be reset. In objects that is done automatically, but static values need to be reset to their initial values. EvoTest uses the class reset probe inserter. Which is rather strange because there are no probes, and the instrumented class also does not call a monitor. But even then InsectJ is quite a good match.

Bibliography

[1] Ieee glossary of software engineering terminology, Mar 1990.

[2] H. Sthamer A. aresel and M. Schmidt. Fitness function design to improve evolutionary structural testing. Technical report, DaimlerCrystler AG, Research and Technology, Alt-Moabit 96a, 10559 Berlin, Germany, 2002.

[3] Abramson and Abela. A parallel genetic algorithm for solving the school timetabling problem. Technical report, Royal Melbourne Institute of Technology, 2476V, Melbourne 3001, Australia, april 1991.

[4] Jarmo T. Alander. An indexed bibliography of genetic algorithms in chemistry and physics.

[5] Aspectj project. `http://eclipse.org/aspectj/`.

[6] T. Back, F. Hoffmeister, and H. Schwefel. A survey of evolution strategies. pages 2–9, July 1991. `http://citeseer.ist.psu.edu/back91survey.html`.

[7] A. Baresel and H. Sthamer. Evolutionary testing of flag conditions. In *GECCO*, pages 2442–2454, 2003.

[8] Byte-code engineering library (bcel). `http://jakarta.apache.org/bcel/`.

[9] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[10] K. E. Boulding. What is evolutionary economics? *Journal of Evolutionary Economics*, 1:9–17, 1991.

[11] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, Sept. 2004.

[12] T. Dandekar and P. Argos. Folding the main chain of small proteins with the genetic algorithm. *Journal of Molecular Biology*, 236:844–861, 1994.

[13] Yuval Davidor. *Genetic Algorithms and Robotics, A Heuristic Strategy for Optimization*. World Scientific, 1991.

[14] L. Davis. *Handbook of Genetic Algorithms*. International Thomson Computer Press, 1996.

[15] B. Jones et al. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5), 1996.

[16] R. Pargas et al. Test data generation using genetic algorithms. *Software Testing, Verification & Reliability*, 9(4), 1999.

[17] R. Fulkerson G. B. Dantzig and S. M. Johnson. Solution of a large-scale traveling salesman problem. In *Operations Research 2*, pages 393–410, 1954.

[18] F. Glover and M. Laguna. *Tabu Search*. Kluwer, Norwell, MA, 1997.

[19] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[20] D. Hamlet and J. Voas. Faults on its sleeve: amplifying software reliability testing. In *ISSTA '93: Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, pages 89–98, New York, NY, USA, 1993. ACM Press.

[21] J. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1975.

[22] IBM. Eclipse integrated development environement, 2006. `http://www.eclipse.org`.

[23] Instrumentation api in java 5. `http://java.sun.com/j2se/1.5.0/docs/`.

[24] C. Johnson and J. R. Cardalda. Genetic algorithms in visual art and music special edition: Leonardo, 2002.

[25] Java Platform Debugger Architecture (JPDA). `http://java.sun.com/products/jpda/index.jsp`.

[26] Java Virtual Machine Profiler Interface (JVMPI). `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html`.

[27] D. Marinov K. Sen and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM Press.

[28] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[29] D. Koza. *Genetic Programming, On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[30] J. R. Koza. Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems. Technical report, Stanford University, Stanford, CA, USA, 1990.

[31] J. R. Koza. Human-competitive applications of genetic programming. pages 663–682, 2003.

[32] I. Song M. Dianati and M. Treiber. An introduction to genetic algorithms and evolution strategies. Technical report, University of Waterloo, Electrical and Computer Engineering, 200 Univ. Ave. West, Ontario, N2L 3G1, Canada, 2003.

[33] Maynard-Smith. Evolution and the theory of games. *CUP*, 1982.

[34] P. McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 2004.

[35] P. McMinn and M. Holcombe. Hybridizing evolutionary testing with the chaining approach. Technical report, 2003.

[36] P. McMinn and M. Holcombe. The state problem for evolutionary testing, 2003.

[37] Sun Microsystems. Mustang (java 6.0 beta), 2006. `https://mustang.dev.java.net`.

[38] Leo Rela. Evolutionary computing in search-based software engineering. Master Thesis, 2004. `http://www2.lut.fi/~rela/dtyo_Leo_Rela.pdf`.

[39] Gregg Rothermel and Mary Jean Harrold. A safe, efficient algorithm for regression test selection. In *ICSM*, pages 358–367, 1993.

[40] Pal Révész. *Random walk in random and non-random environments*. World Scientific Pub Co., 1990.

[41] A. Salcianu and M. Rinard. A combined pointer and purity analysis for java programs. Technical Report MIT-CSAIL-TR-949, Massachusetts Institute of Technology, 2004.

[42] W. Schulte T. Xie, D. Marinov and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pages 365–381, April 2005. http://www.csc.ncsu.edu/faculty/xie/publications/tacas05.pdf.

[43] N. Tillmann and W. Schulte. Parameterized unit tests with unit meister. *SIGSOFT Softw. Eng. Notes*, 30(5):241–244, 2005.

[44] P. Tonella. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 119–128, New York, NY, USA, 2004. ACM Press.

[45] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1053–1060, New York, NY, USA, 2005. ACM Press.