A Factorial Experiment on the Scalability of Search-Based Software Testing

Arash Mehrmand
ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Robert Feldt, for his invaluable guidance and advices, without whom this thesis would not have been completed. I thank my best friend Nima Sharafeddin, for his support during the experiments. Many thanks go to Blekinge Institute of Technology for providing all the needed facilities for me to continue my studies. I also thank the providers of GEVA and JGAP for providing valuable open source packages. Last but not least, I would like to thank my family who supported me through out my stay in Sweden in any aspect, inspiration and understanding and especially my mother, and my cousin.
A Factorial Experiment on Scalability of Search-based Software Testing

Arash Mehrmand  
School of Computing  
Blekinge Institute of Technology  
Ronneby, Sweden  
arme08@student.bth.se

Abstract—Software testing is an expensive process, which is vital in the industry. Construction of the test-data in software testing requires the major cost and knowing which method to use in order to generate the test data is very important. This paper discusses the performance of search-based algorithms (preferably genetic algorithm) versus random testing, in software test-data generation. A factorial experiment is designed so that, we have more than one factor for each experiment we make. Although many researches have been done in the area of automated software testing, this research differs from all of them due to sample programs (SUTs) which are used. Since the program generation is automatic as well, Grammatical Evolution is used to guide the program generations. They are not goal based, but generated according to the grammar we provide, with different levels of complexity. Genetic algorithm is first applied to programs, then we apply random testing. Based on the results which come up, this paper recommends one method to use for software testing, if the SUT has the same conditions as we had in this study. SUTs are not like the sample programs, provided by other studies since they are generated using a grammar.

Keywords—Automated Software Testing; Search-based Software Testing; Genetic Algorithms; Random Testing; Grammatical Evolution;

I. INTRODUCTION

Software testing is one of the most critical and at the same time most time and cost consuming activity in software development process [6]. In order to cut this costs as much as possible we need to employ effective techniques to automate this process. What actually needs to be automated is generating the test data which is a demanding process [9]. Automation of generating test data helps to have fast, cheap and error free process [9].

If we can optimize the test cases which we have, then we saved a lot of time and money. Out of the optimization techniques, search based optimization is the one which has been applied to different areas of Software Engineering applications [9]. According to [9] “A recent survey of SBSE presented at ICSE07 as part of the “Future of Software Engineering” track.”

Software testing was not an exception in Software Engineering area and the term “Search Based Software Testing” proves it. The recent years shows a dramatic rise in interest in Search Based Testing (SBT) [9]. SBT has been applied to different testing areas such as structural, functional, non-functional, mutation, regression and so on [9].

There are many Search Based techniques and each of them uses a different algorithm to optimize the search and the results. The three popular ones which are in use for testing area are Neighborhood Search, Simulated annealing and Genetic algorithms and of course there are simple ones as well such as Random testing which does not use any optimization and search technique. [18]. According to [18] flexibility and power of these heuristic optimization techniques helps a lot in finding the optimal solutions in large and complex search spaces.

This study focuses on using the SBT techniques or as we said heuristic optimization techniques for software testing, when we have different programs regarding to complexity. We will apply the technique which is called Genetic algorithm to wide variety of programs to see the effectiveness of Genetic algorithm compared to random testing. The final results of this paper will help Software Engineers and especially Software testers to make sure which SBT technique is the most useful one from error finding point of view, when they deal with complex programs.

As the final result this study will show that GA in software testing is more effective than Random testing as we increase the complexity of software.

The structure of this paper is as following: Next section, section 2, presents the existing researches in this area. Section 3 explains the experimental approach chosen for this study and study’s design. Section 4 is a brief description of Software Under Test and the way we generate them in this study. Section 5 includes the results and discussion about them. And finally conclusion and analysis will come under section 6.

II. PREVIOUS RESEARCHES AND RELATED WORK

There are many researches which are done in software testing automation area and SBST. Some of them deal with software complexity as well but I could have not found any of them similar to this study as we deal with increasing of software complexity along our study. For example, authors in [13] discuss the experiments with test case generation with large and complex programs and they conclude that, there is a wide gap between the techniques based on GA
and those based on random testing. They have chosen some C programs and done the test on them.

Compared to what the authors tried to explain in [13], our study has a main difference and that is the use of many Java programs which are different in their size and complexity. Actually as we mentioned in introduction the purpose of this study is to test many different programs as we increase their complexity.

Many other studies tried to experiment software testing using genetic algorithms and compare the results with other techniques. Authors in [19] performed experiments on some small and a few complex test objects and they came up with this results: Particle Swarm Optimization overcomes GA in coverage of many code elements so not always GA is the best solution. Another paper also talks about GA based software testing and the main idea is to find problematic situations while testing programs with GA [1].

Alba and Chicano in their paper [2] describe how canonical GAs and evolutionary strategies can help in software testing. They used a benchmark of twelve test programs and applied both Evolutionary Strategies and GA and compared the results at the end.

However, many existing researches and studies (e.g. [15],[16],[11]) have used SBT specially GA in order to test the software and compare the results with other techniques.

III. EXPERIMENT APPROACH

The experimental approach, chosen for this study, is factorial experiment which compares GA and random testing in performance and efficiency. We call it factorial experiment because for both of our methods, we have some factors and levels which can be set at various values [3].

Factors are actually our coverage methods which could be statement coverage, and branch coverage while we have three levels for the complexity: low, medium, and high. Since we have to increase the complexity of our generated programs, gradually, there are some factors which we have to increase or decrease them, in order to play with the programs complexity. Factorial experiment is an experiment whose design consists of two or more factors and each factor has discrete possible values which we call them levels. In our case, one of the factors is the code complexity and the levels that we consider for this factor are low, medium and high. Code complexity is actually an independent variable in our experiment. The other factor in this study is the coverage method and it has two options, which are statement coverage and branch coverage. Below is a table which shows the factorial design of this experiment. You can see the factors which are used in rows and columns. For statement coverage, as well as branch coverage, we have source codes with low, medium and high complexity. For each coverage method, we test our codes with both GA and random testing and record the coverage percentages as you can see them as graphs in section 4.1.

These factors (variables) are explained in details in section 3.1 which is Experiment Design. After making all the experiments, we will have a sort of tables and graphs which shows the efficiency of GA compared to random testing; these results will come in section 3.2 which is Experiment Results.

IV. EXPERIMENT DESIGN

In order to design a fully automated process for our experiment (which is a must), we need to follow a process and go through them step by step. These steps are as follows:

1) Generate the SUTs with different size and complexity automatically using grammatical evolution [14].
2) Apply GA to the generated SUTs in order to test them.
3) Instrument, compile, and run the written unit test cases.
4) Apply random testing to SUTs.
5) Getting the coverage report which shows how many percentages we have covered our code using both GA and random testing so we can compare them.

Now all mentioned items above, will be described in order to clarify the experiment design.

1) Generate SUTs with GE: Grammatical Evolution (GE) is a kind of Genetic Programming based on grammar. GE combines basis and rules of molecular biology with representational power of formal grammars. GE has a unique flexibility due to its rich modularity, and makes it possible to use alternative search strategies, whether evolutionary, or other heuristic, be it stochastic or deterministic, and to radically change its behavior by only slight changes to the grammar supplied. One of the main advantages of GE, is the easiness of modifying the output structures by simply editing the plain text grammar. This advantage actually originates from the use of grammar to describe the structures that are generated by GE [14].

To use GE, for our SUTs generation, I used an open source software implementation of grammatical evolution in Java. Grammatical Evolution in Java (GEVA) is the name of the tool which provides the possibility to use GE with our supplied grammar, in order to generate evolved programs. However, for us, to use GEVA, there was a need to make some modifications and supply our own material to reach our goal which is generating Java programs with different complexities.

---

1 a platform-independent object-oriented programming language

---

<table>
<thead>
<tr>
<th>Level</th>
<th>Factor’s name</th>
<th>Complexity</th>
<th>Statement</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TST</td>
<td>Low</td>
<td>GA</td>
<td>Random</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Medium</td>
<td>75</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td></td>
<td>High</td>
<td>300</td>
<td>50</td>
</tr>
</tbody>
</table>

---

Table I

FACTORIAL DESIGN OF THE EXPERIMENT
First of all we need to define our problem, which is generating different Java programs. For this purpose, I wrote a Java code which automatically generates the header and footer of the programs and makes the contents of the body of the program, according to the BNF which we will provide. So next step is to write a BNF. Using Backus-Naur form (BNF), it is possible to provide the specification of a programming language structure, which is Java in this case. Since we have to have a subset of grammars, we should have our BNF according to what subsets of Java programming language we are going to generate. The genetic configuration of GEVA is also changed according to our need. Following is a part of our properties, used in GEVA:

- Population size: 200
- Generation: 10000
- Initial chromosome-size: 200
- Selection type: Roulette wheel [12].

Providing GEVA, with the problem definition and BNF file, it can generate our desired programs. Because we want to increase the complexity of our programs, we need to run the generation, three different times. When the fitness function is set to the number of statements, we run the generation, once with 75 statements, once with 150 and finally with 300 statements. And when the fitness function is set to the number of branches in the code, we run the generation once with 25 branches, once with 50 branches and finally with 100 branches. However, the complexity levels could be extended to more than three levels, e.g., very low, low, medium, high, very high or we can have the same three levels with different numbers; the results and the effectiveness of GA over random search will not change though. So three levels which are used in this experiment are sufficient enough for us to conclude which method is more efficient.

2) GA: After generating the SUTs, we need to apply GA to our code so it will behave genetically during the testing process. The library which is used for this purpose, in this study, is called JGAP. JGAP is actually a Genetic Algorithm and Genetic Programming component which is provided as Java framework. Using JGAP it is possible to apply evolutionary principles to problem solutions since it provides us, the basic genetic mechanisms [10].

Using JGAP, first step is to plan the chromosomes, so we have to decide on type and quantity of genes that we are going to use. Each chromosome is made of genes, and in this study each gene is actually a test data. And depending on the size and number of input variables, we decide on number of genes. Second step is to implement fitness function. Although JGAP is designed to do almost all the evolutionary steps in different problems but it does not know, by itself, whether or not a potential solution is better than the other ones. Our fitness function, depending on which part of experiment we are, is the coverage percentage of the generated test data. For example if our complexity metric

is the number of statements, then the best chromosomes are those which provide better statement coverage on the code. Third step is to setup the configuration object of JGAP which means deciding on how big our population is, and all other GA configurations. In this experiment the population size is set as 200 and number of generations is set to 10000. Fourth and fifth steps are creating and evolving the population. And obviously after each generation, the best test data of each population are taken. However, the followings are the steps, GA basically, uses to test a software where \( P(t) \) is the population and \( t \) is the generation [12]:

\[
\text{Initialize } P(t) \\
\text{Evaluate } P(t) \\
\text{While the termination condition is not reached, do} \\
\text{select } P(t+1) \text{ from } P(t) \\
\text{recombine } P(t+1) \\
\text{evaluate } P(t+1) \\
t=t+1
\]

Evaluating the fitness function depends on which experiment we are running; so for statement coverage and branch coverage is different. Please refer to 3.1.3, where defining the fitness of the experiment is explained in more details. This was the whole strategy of GA application to our SUTs. For more information about GA and its mechanism you can refer to the Appendix.

3) Defining The Fitness Function: What genetic algorithm does is actually searching for the suitable test case, to kill a given mutant; for this purpose fitness function is used, which assigns a non-negative cost to each candidate input value. The more cost each input value has, the more appropriate it is, so we aim to maximize this value and therefore maximize the fitness function [5].

According to [5] there are three kinds of conditions which can be used as fitness function; in the other words, there are three kinds of costs which we can be maximized to maximize the fitness function. However, the combination of these costs can be used as well. Those conditions are called: reachability condition, necessity condition, and sufficiency condition. Reachability condition is actually the same issue as function minimization. In this case a numerical predicate is assigned to each branch. If the value of the branch predicate is equal to zero, this means that the condition has been satisfied and the fitness function is maximized. But in a case that two candidate inputs have a non-zero predicate value, then to compare them we need to pick the one which has the closer value to satisfy the common failed branch predicate. s The other two conditions are not used in our fitness function, this is why they are not defined.

I used the term which is called \textit{function minimization} [12]. Function minimization helps to find the desired input for each condition to be executed. Using this way, each condition in the code has its own function. Genetic search,
then, tries to find the input which minimizes the value of that specific function. For example, imagine we have this condition on line 300 of the program:

\[
\text{if } (x \geq z+10)
\]

and we aim to execute the true condition of this branch. Then the, the minimization function is defined as below:

\[
f(n) = \begin{cases} 
(z_{300} + 10) - x_{300} & \text{if } z_{300} + 10 \leq x_{300} \\
0 & \text{otherwise}
\end{cases}
\]

The above function says, if \( z_{300} + 10 \) is less than or equal to \( x_{300} \), then try to set the value of \( z_{300} \) and \( x_{300} \) so that the value of \( (z_{300} + 10) - x_{300} \) is as minimum as possible. Please notice that 300 here is the line number. We have to define the line number since the value of input parameters (\( x \) and \( z \)), change over time, as the programs run.

When the fitness function is statement coverage, we need to count the number of statements which are blocked between braces of correspondent branch. We use it as the weight and divide the value of [the correspondent] function with this weight. For branch coverage, we do not need this weight since we are looking for the covered branches not the statements.

Because our sample programs execute repeatedly over time, the input parameters change every time the program executed. The minimization function decides, how closed the input parameters are to the desired values which satisfy the conditions and we use GA in order to minimize the function; the fittest chromosomes the closer results to minimize the function.

4) Instrument, Compile, and Unit Testing: After the application of GA to our generated SUTs, we need to observe the coverage percentage that GA provides for each SUT so we can, later on, compare this percentage with random testing. However, this coverage depends on the fitness function which we define for our GA. For example, if the goal is statement coverage, then we need to count the total number of statements in each program, then count the number of statements which have been executed and divide them with total number of statements. This way, we can see how many statements, GA was able to cover. The rest of this section explains the process which we should follow to apply GA to our SUTs.

Instrumentation: First task is to instrument the code. Without having an instrumented code we are not able to compile our SUT and test it using GA. What instrumentation does actually it transforms each condition into an expression that is supposed to have the same value as the original condition [2]. Instrumentation of the code must be done carefully to avoid the change in the behavior of the program [2]. This part is also dependent on the fitness function. If we are generating the programs with the goal of number of statements, then we should instrument our code in such a way that all executable statements are possible to be counted. Or if we are generating programs with the goal of number of branches, then, the code should be instrumented in such a way that all executed branches are possible to be counted. For this purpose, a counter [variable] is used. Next to each statement we have this counter so when the statement is executed, the counter increments. The same story happens to branches also, when the goal is branch coverage.

Compile: After we instrumented the code we will be able to compile this code. Actually instrumentation prepares the code to be compiled. We need to compile the code to make sure it works properly and without any error. However, without compiling the code we can never test it.

Please notice that, in order to speed up the testing process, we only instrument and compile each program, once. This means Java Virtual Machine runs only once for each SUT to test.

Run the unit test case: For each SUT there is a written unit test case which needs to run.

Getting report: After running the test case we will have a report of our test. This report is generated according to our need; for example if we need the statement coverage of the code, then we have to set it up in way which gives us the number of executed statements divided by total number of statements. And of course, without an instrumented code is impossible to generate coverage report out of that code.

5) Random Testing: Random testing is the chosen approach in this study, to be compared with SBST. According to [7] random testing is actually the use of randomly generated test data which has some advantages such as easy and simple implementation and speed of execution, which means less run time. When running the test using random testing, we have to define a parameter; this parameter defines how many times we should generate test data using random testing. For example if we set the parameter to 100, then it generates 100 set of test cases. Out of these [e.g.] 100, we take the best and compare it with the fittest result of GA. However, in this experiment, this number is set to 100000.

Using random testing we need to define our input domain, so the numbers will be randomly generated form this domain. In this study this set starts from \(-1000000\) to \(+1000000\).

V. SUT Descriptions

As mentioned in section 3.1.1 SUTs are generated automatically using GE with the help of GEVA. It was also mentioned that we have to provide our own BNF, so that GEVA can understand what to generate. However, these programs are expected to have different complexities. Since we are running a factorial experiment we need to define levels of our complexity (which will be an axis of our graph at the end). We defined three levels for the complexity which are low, medium, and high. Depending on our fitness (test aim) we define these levels. For example if the aim is branch coverage, then low complexity means a code with the least
number of branches compared to medium and high. And for medium we increase the number of branches of each program and so on for high complexity. The programs are, of course, written in Java, so we should be careful with our BNF definitions, not to have syntax problems in our programs. Sample programs contain statements, assignments, expressions, if loops, while loops, for loops, and loops with multiple conditions; almost whatever a sample program should have. During the generation of programs, we need to provide GEVA, with a fitness function, for example if we are to generate programs with different number of statements, then the fitness function would be the number of statements in the code. This way, low complex software means, the one with less statements compared to medium and high. Number of statements is set to 75, for low complexity, 150 for medium, and 300 for high complexity. Same process will happen for generating programs with different number of branches. Number of branches is set to 25, for low complexity, 50 for medium, and 100 for high complexity. The genetic configuration of all the programs are the same, all of them have three input parameters and the header and the footer of the programs are same as well, however, they have totally different body which is generated with the help of GP (GEVA) according to the defined grammar in BNF.

I defined a logic in the BNF, so that if there is any problem with infinite loops, it will exit the loop after certain number of iterations. The idea is called loop-exit [20], [4]. This process is explained more in section 6, which is problems and limitations.

For each factor and level, we generate ten sample programs which means sixty samples in total. For example if the factor is statement coverage, we generate ten programs with the purpose of statement coverage and low complexity, ten for medium and ten for high.

A. Experiment Results

After testing the generated SUTs, I came up with the results. As you can see in the following plots, GA outperforms the random testing in almost, all the experiments which I have done. As mentioned before, the whole experiment includes testing 60 programs; 30 (10 with low, 10 with medium, and 10 with high complexity) with statement coverage purpose and 30 (10 with low, 10 with medium, and 10 with high complexity) with branch coverage purpose. So the fitness function was different for each purpose. The generated programs are also different since they are generated with two different fitness function. The used programs for statement coverage purpose, are generated with number of statements purpose and the ones which are used for branch coverage purpose, are generated with number of branches purpose. Please note, the following plots include 10 programs each, which are randomly chosen from multiple runs that I had.
As mentioned above, these plots represent the result of 10 chosen programs, but to see how efficient GA is, please refer to the table above, where it shows the average percentage of coverage with both GA and random search.

In Table II, for statement coverage, from low to high, t-test shows that means are not significantly different from the Selected Confidence Level.

Note: desired confidence level is set to 90

And in Table III, for branch coverage, from low to high, t-test shows that means are significantly different from the Selected Confidence Level.

Note: desired confidence level is set to 90

VI. DISCUSSION

As it is obvious from the graphs in section 4.1 GA has outperformed random testing in almost all cases and in a few cases they have same coverage percentage. With all complexity levels in this study, GA has better coverage percentage compared to random testing.

The design of this experiment was factorial so that we can assure in different situations, with different complexity levels and different source codes, random testing can never outperform GA from the coverage point of view. This means random testing could be a faster solution in many cases but since it does not cover the code as GA does, GA is the preferred solution according to our experiment.

There exist some issues which need to be discussed here, some of them could be serious problems though.

Our SUTs here are generated using grammatical evolution. As mentioned in previous sections, for grammatical evolution to generate programs, we have to provide it with

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Test Search Technique</th>
<th>GA</th>
<th>Random</th>
<th>Actual CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>GA</td>
<td>75</td>
<td>95.87</td>
<td>12.13</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>GA</td>
<td>150</td>
<td>96.12</td>
<td>8.74</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>GA</td>
<td>300</td>
<td>95.39</td>
<td>5.63</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Test Search Technique</th>
<th>GA</th>
<th>Random</th>
<th>Actual CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>GA</td>
<td>25</td>
<td>87.27</td>
<td>6.98</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>GA</td>
<td>50</td>
<td>79.33</td>
<td>6.99</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>GA</td>
<td>100</td>
<td>76.57</td>
<td>8.25</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table II

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Test Search Technique</th>
<th>GA</th>
<th>Random</th>
<th>Actual CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Branch</td>
<td>25</td>
<td>87.27</td>
<td>6.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>Branch</td>
<td>50</td>
<td>79.33</td>
<td>6.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>Branch</td>
<td>100</td>
<td>76.57</td>
<td>8.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table III
a BNF. Unfortunately it is impossible to define semantics (logic) in BNF file which means there is very low chance of controlling what you are generating. During these experiments, I had many programs which had same statements, or same loops or even same blocks. Since we follow GA in our grammatical evolution, whenever it finds a fit chromosome, it keeps using it to generate the new population since it is the fittest. What I mean by chromosome here is actually the statements inside the code. For example if the purpose is number of statements, then if GEVA finds a block a statement which satisfies the fitness function, then it sees no reason not to use it any more. So at the end we will have a code which contain some repetition. Repetition can cause better percentage by random testing since we have many same blocks of codes.

Big numbers, which are made throughout the process of multiplication, addition and subtraction, are out of range of integer and double variable. These out of range variables can cause the code not to be tested completely since when the test faces with this kind of number, stops running. We did not define divisions in our BNF since it can cause many problems e.g. division by zero.

Another important issue is the infinite loops in the code. And as I mentioned before, I controlled them using exit-loop which means having maximum iteration and break it after the value of maximum iterations have been reached. However, having breaks, even if we assign maximum iteration to a large number, causes the code to not get tested completely [20], [4].

Conditions in the loops are very important during testing, since if we can satisfy the condition and go through the block of that condition, we can execute more statements. There are two kind of conditions which can cause problems. First one is the condition which is not reachable at all and second kind is a very easy condition which both GA and random testing can cover it easily.

As an example, have a look at the following code, when GA can never cover it; and of course when GA cannot cover it, random testing cannot do it either.

```
while ( ( z + 6 7 2 ) <= ( z - 6 7 5 4 ) )
AND

while ( ( y <= ( y + 5 3 7 ) ) && ( z >= ( z - 6 9 ) ) )
```

This problem is solved in our BNF which means we have the minimum number of easy and unreachable conditions so we do not face serious problems during our experiment.

Have a look at the source code below, which is a small piece from a source code.

```java
if(((x+41168)<(z+4))&&((y+641)!=(x-3619))
&&((x+36321)==(x+1691)))
{
x=(y-((z+x)-x));
}
else
{
x++;
}
```

Let’s say we want to test this code with the purpose of statement coverage. First loop will never run, no matter if GA is in use or random testing because the last condition of this loop makes it false all the time. $(x + 36321) = (x + 1691)$ will never become true and it make the whole loop not to run. So the statement block of this loop will never be reached. On the other hand, next loop which is $if((z + 25) <= (y + 96131))$ is so easy that both GA and random testing can cover it and of course the statement block of this loop will be reached with either GA or random testing. Now imagine a source code with hundreds of lines and filled with many of mentioned loops and blocks. In these cases, it does not really matter which method we are using to test our programs since we get the same coverage.

Using GE and BNF to generate programs has some difficulties and limitations which make it very hard to generate a piece of code without infinite loops, hard but possible to cover conditions. And testing the programs which are generated using GE has its own limitations since you have limit the number of iterations of your loops, and eliminate the nested loops because of exit-loop problem. The authors in [21] do not recommend using nested loops; instead they suggest another way which is called LoopN. In LoopN the use of nested loops is forbidden. They think loop does not do much harm if it is not nested. However limiting the code, and not having the nested loops can decrease the overall complexity, which was not our aim but there was no other way than eliminating them. Without eliminating the nested loops it is very hard to instrument the code, and it requires very powerful package which can satisfy our requirements: not running the Java Virtual Machine everyday, instrument in a way which is usable to test. So what is recommended as the most useful way, is to have some sort of semantics in the BNF to limit the branch conditions and then we can focus on nested loops and instrumentation. The author in [17] says semantics is everything that cannot be described in BNF and he also thinks that semantics, compared to syntax, is much more difficult to describe. Generally, there are some theoretical ideas about writing semantics in BNF but there is no practical implementation. For example M. D. Derk in [8] talks about a theory about defining relational semantics to be designed and used alongside BNF. According to [8] a complete definition (syntax and semantics) to define a grammar should have the following elements:

1) BNF syntax definition
2) Predicates: indicate conditions that must be true in order for the semantics operations to be carried out.
3) Response: is the resulting actions to be taken by the computer system for the correct execution of the statement.
4) Error conditions and responses: resulting actions which will be taken by computer system in case of being in an error condition.
5) Remarks (optional)

However, these elements cannot be defined inside the BNF and as I mentioned above, they should be designed alongside BNF; there is no practical implementation for them though. It is possible to limit your BNF and control some parts such as having limited number of conditions, and limited number of digits, but this does not help in our case since we need to define semantics in our BNF to control the logical parts such as not having infinite loops. So, after all this discussion about BNF, it seems generating programs using GE is not a good choice since it has to be provided with a grammar (either BNF or EBNF) unless we can bring semantics in our grammars.

Rather than having the problem with BNF, instrumentation itself, is a big issue. This is mainly because, the more details we need to instrument, the less speed we have. For example in case of nested loops; instrumenting nested loops needs us to keep track of each loop (inner and outer), each branch, and each condition more precisely.

VII. CONCLUSION AND FURTHER WORK

In this paper, I have reported on results from a factorial experiment, where the performance GA is compared to random testing in automation of software testing. In this experiment, even the process of generating SUTs were automated, which means we used GE to generate our sample Java programs. To my knowledge, the presented results we had are not yet reported in the software test data generation area, because of having automatically generated programs, different levels of complexity and different kinds of coverage at the same time. The followings are my observations from the experiments and the taken results:

- Generating programs using GE, can limit the source code because of mentioned problems about BNF, loops, conditions etc. Of course GE is effective when we are looking for a solution of a problem. The studies which show the flexibility of GE in problem solving include: grammars have been used to represent a diverse array of structures including binary strings, code in various programming languages (e.g., C, Scheme, Slang, Postscript), music, financial trading rules, 3D surfaces, and even grammars themselves [14].
- GA can outperform random testing and it did actually in this study and with our automatically generated SUTs. This is precisely proven in our factorial experiment. This factorial experiment with different levels of code complexity and two methods of code coverage, proves that in almost, all situations GA has better performance versus random testing.
- Random testing could be, however, a faster solution but it is not recommended since cost is somehow more important than time. This means more bugs that we find in our programs, more cost we save; this is what happens in real world.
- Number of statements, is not good metrics for software complexity adding extra statement does not really make the code complex. What makes the codes really complex, is the condition. Having more complex and executable conditions can help to make the source codes more complex.

This study can be extended by applying well known software complexities i.e. cyclomatic complexity; since adding more statements and more branches did not affect the code complexity as expected.

Another area which is possible to work on, is to use GE with Extended BNF (EBNF). Using EBNF, it is possible to define not only the syntax, but also some additional control elements such as sequence, choice, option, and repetition. It does not solve the semantics problem though.

If it is possible to develop a fast coverage tool (open source preferably) which includes the instrumentation as well, then many problems are solved in this area. They could be even more useful if they are open source since being open source can speed up the testing and getting reports by grouping the required classes as one unit package and compile the desire tool with only desired features.

REFERENCES


Chapter 1

Software Complexity Measures and Metrics

In order to talk about software complexity we first need to define what software complexity means. It is widely believed that there is no single dimension description of complexity[19]. However, when this term is applied to software, there exist some definitions which differ from each other.

By reviewing the literature and summarizing several definitions we can better understand the alternatives, even if they differ.

IEEE [1] defines software complexity as the degree to which a system or component has a design or implementation that is difficult to understand and verify. Basili [4] defines complexity as a measure of resources expended by a system while interacting with a piece of software to perform a given task. This interacting system could be either computer or programmer. In case of computer, complexity is execution time and storage required to perform the computation; and in case of programmer, complexity is defined by difficulty of performing tasks such as coding, testing, debugging or modifying the software [40]. Some other researchers such as Cruits [12] refine the software complexity term into two different views: algorithmic and psychological. Algorithmic defines the run time performance of an algorithm and psychological view deals with performance of programmers trying to understand or modify a code module [19].

Since in this thesis we are looking for the meaning of complexity from source code point of view, we will focus on those definitions which are more related to source code and control flow graph of a program.

After giving the definition of software complexity using different points of view, in next section different aspects and metrics of software complexity will be discussed.

1.1 Software Complexity Aspects and Metrics

Complexity of a software could be discussed using different aspects and also measured using different metrics. There are different existing metrics, although many of them are old, which help to measure the complexity of software. They vary in simplicity of measurement. Some of them are very simple such as LOC.
or lines of codes and as we study more about them we can find the more complicated ones as well.

According to [23] there is a classification in categories of existing measures for software complexity. One is micro and other one is macro measures for software complexity. Micro is based on the source code of the program and usually depends on program size, flow graph and module interfaces. And macro deals with the overall structure of the software.

Two of most famous software complexity metrics according to a literature survey are McCabe’s cyclomatic complexity and Halstead’s Software Science. Since these two metrics are the most popular ones I will define them in more details and then more metrics will be introduced.

One of the most well known complexity metrics is the one which is created by Halstead and his colleagues and it is called Halstead software science. The Halstead measures are functions of number of operators and operands in the program [40]. According to [23] Halstead’s focus is actually on operators which means defined operations in programming language that may change or move the operands, and on operands which are variables, constants, and addresses in a software component.

Major components of Halstead software science are:
- n1: the number of unique operators
- n2: the number of unique operands
- N1: total number of operators
- N2: total number of operands

Where the volume of program is: 
\[ V = (N1 + N2) \log_2 (n1 + n2) \]
And program difficulty:
\[ D = n1XN2/2n^2 \]

On the other hand, Cyclomatic number is another complexity metric and it is developed by Mc’Cabe and that is why this metric is called Mc’Cabe Cyclomatic complexity. McCabe considers the program as a directed graph in which the edges are lines of control flow and nodes are straight line segments of code [40]. Cyclomatic number, in a well structured module, is one plus number of branching statements in the module.

Cyclomatic number could be defined according to the following formula as well:
\[ C(z) = e(z) - n(z) + 2 \]
Where n(z): number of nodes and e(z): number of edges.

Cyclomatic number has some weaknesses and criticisms which some of them are mentioned below [25]:

- The failure to count ELSE Branches
- Decisions have equal weights regardless of depth or nesting
- Inconsistent behavior when measuring modularized software
- It has a strong relationship with LOC and this is enough not to count it as a general metric

When comparing Halstead with Mc’Cabe we conclude that Halstead’s software science calculates the numbers of operators and operands but does not consider the internal structures of software components; while McCabe’s cyclomatic measure does not consider I/O of software systems [37].
1.2. SOFTWARE COMPLEXITY AND CONTROL FLOW GRAPH

We now know two of most popular and traditional complexity metrics but since both of them treat a program as a single body of code, Henry and Kafura developed a measure which is sensitive to the structural decomposition of the program into procedures and functions. Henry and Kafura measure is depends on procedure size and the flow of information into procedures (fan-in) and out of procedures (fan-out) [40]. According to this measure, complexity of a procedure is defined as:

$$Complexity = length X (fan - in X fan - out)^2$$

There are many other complexity measures which have been developed and so far we just mentioned three of them. However for the rest I will not go into details and mentioning their name and maybe a few words about each one would be sufficient.

- Statement count: Probably the oldest and most intuitively obvious notion of complexity and with advantage of simplicity [10].
- Oviedo’s data flow
- MEBOW: defining the scope weights for branches [34]
- NPATH: Measure of execution path complexity [34]
- Function points
- The physical size: LOC
- Path Complexity: sum of node complexity of each node of the path.

1.2 Software Complexity and Control Flow Graph

In order to use some of the mentioned metrics in last section we need to have a graph of our program which is called Control Flow Graph (CFG).

Control flow graph is developed to use in the modeling of systems and usually simple systems. In CFGs the behavior of each system component is specified by control flow graph and the interaction between components are specified by message passing across directed channels interconnecting model components [13].

In control flow, different instructions of a program are executed in an order. There are some statements which do not affect the sequential flow and they are called functional statements and they do not fall in sequential flow [34].

Some examples of metrics which are based on control flow graph:

- Mc’Cabe Cyclomatic complexity
- “Path Count metric” is based on number of distinct paths in a control graph.
- “Knot count” metric is based on the number of intersections of control flow in a program’s text, not necessarily the graph though [41].
CHAPTER 1. SOFTWARE COMPLEXITY MEASURES AND METRICS

1.3 Metrics Evaluation

For a metric to be accepted as a valid metric, there are some properties which should be satisfied. We use these properties when we want to propose a new metric as well [37].

Property 1: When applying a metric to a program, the metric should not make all the programs have the same complexity.

Property 2&3: A metric should be neither too lessen nor too fine.

Property 4: Implementation of a function might have an impact on its complexity.

Property 5: A component of a program is always simpler than the whole program.

Property 6: When two programs have same complexity, they do not need to have same complexity after being concatenated with the third program.

Property 7: Alteration of statements may affect their complexity.

Property 8: Renaming a program does not affect its complexity.

Property 9: When two subprograms interact with each other, their complexity might be increased.

However, according to [34], the attempts to develop measures for complexity so far have not yet resulted in one which is completely satisfying. The reason is we still do not know how various factors such as control flow, data flow, modularity, size, etc. which have contribution to software complexity, can be quantified and actually unified into a single measure. We should instead concentrate on a single aspect such as control flow to have the possibility of doing more precise analysis and develop measures which can measure that aspect of complexity perfectly.

1.4 What Metric We Used in This Study

With all the explanations, in previous sections, many of mentioned metrics are very hard or almost impossible in this study due to the shortage of time we have. So what we used is the number of statements in each program and also number of branches, which of course increase the complexity of code.
Chapter 2

Search-Based Software Engineering and Its relation with Software Testing

2.1 Introduction

In order to make sure about the quality of the software, there exist different techniques such as static analysis, code reviews, formal specifications, refinement, proof and testing. Between all these techniques and the others which are probably not mentioned here, testing is still the primary method especially in industry [39].

However, software testing or we better say the process of test data selection is very expensive and time consuming [8]. This is mainly because this process is manual. Automation of this process helps a lot to reduce time and money which is usually spent to generate test data [17]. According to [39] we need to use automated test data generation because of its two advantages: increase the test quality and reduce costs.

Definition: Test data generation in software testing is defined as the process of finding program input data which satisfy selected testing criterion [21].

One of the most used and effective techniques to make the automated process is to use optimization techniques associated with Search-Based Software Engineering (SBSE) [17].

In Search-Based Software Engineering, search based optimization is applied in software engineering areas such as requirements engineering, project planning, quality assessment and many other areas [17]. As the author in [17] mentioned in his paper, use of search based optimization in test case generation is a good proof about the recent explosion in SBSE.

While generating test cases and testing a software we usually face with a very large input space and with manual process it is very hard to choose those inputs which fits in our purpose. What makes SBST attractive is, we can define our test goal as a fitness function and by changing the input space and fitness function we can adopt this approach to various test case generation problems [17].
In test case generation we usually deal with locating good test data for a unique test criterion. So the need of using evolutionary algorithm is obvious and the application of evolutionary algorithm in software testing is called search-based test case generation [32].

According to the research which is done by the authors in [3] fifty percent of the whole software development cost, if not more, goes to testing part thus a logical way to decrease this cost is to automate the test case generation. One of the strategies which has attracted great interest is the application of metaheuristic search algorithms and this is mainly because test case generation problems can often be re-defined as optimization or search problems. Metaheuristic search (MHS) algorithms are those algorithms which are used to find optimal solutions in [search] problems that have large complex search spaces [3].

There is an obvious relationship between MHS algorithms and test case generation since we can simulate the process of test case generation as search or optimization process. For example for a SUT, there could exist hundreds of thousands of generated test cases and out of them we only need those test cases which satisfy the coverage criteria at a reasonable cost [3].

Most of the MHS techniques are based on the theory of natural evolution. These techniques are proven to be very productive and efficient in solving hard optimization techniques [2]. These techniques are based on a set of initial and experimental individuals called population. These individuals need to be attached to a quality value and this is the job of what we call fitness function. Usually EA families, manipulate this population or the initial pool of individuals. What manipulation means here is to produce one or two new children and mutate their contents by applying a selection of fittest individuals, recombination of slices of two individuals [2]. Fitness function plays a primary role since the quality of new children is based on fitness function and according to it, fittest individuals will be selected. However, since this thesis is not going into details of all types of SBST, in coming sections I will introduce some SBST techniques which are more popular and give an explanation about how they work.

Among various MHS techniques some of them are more in use in software testing field. The ones that we are going to describe in next sections are:

- Neighborhood search
- Simulated annealing
- Genetic algorithms

2.2 Neighborhood search

Many optimization problems have many more locally optimal solutions than globally optimal solutions. The idea of local optima needs the introduction of a neighborhood [39]. An informal definition according to Nigel J. Tracy in [39] is: "this is a set of solutions which are close in some respect to a given solution.", but a formal definition could be: for each solution \( x \in X \) there is a set of neighboring solutions, \( \mathcal{N}(x) \subseteq X \). We call it neighborhood of \( x \).

An example or a simple form which might be familiar to many people is hill-climbing. In hill-climbing method an initial solution is selected randomly
from a set of possible solutions. Then, the neighborhood of this initial solution is intensively searched to find the locally optimal solution. The neighborhood of this locally optimal solution is then searched to find the next locally optimal solutions. The search stops when there is no new locally optimal solution [39].

However, obviously, as you can see neighborhood search works well when we need to find locally optimal solutions not global ones and this is clearly counted as a disadvantage since this method fails to give good approximations to the global optimum [39].

2.3 Simulated annealing

The modified version of neighborhood search which tries to overcome the problem of accepting minor and secondary solutions in a controlled way is called simulated annealing [39]. According to [39] back in 1983, Kirkpatrick suggested that a form of simulated annealing could be used to solve complex optimization problems. Since simulated annealing’s applicability comes from thermodynamics and statistical mechanics, it can solve many combinatorial optimization problems [15].

First step is to select an initial solution which is usually selected randomly, an initial temperature and number of solutions to try at this temperature. The initial solution has a cost which is calculated as well. The search begins to work and it repeats until the stopping criterion is reached; this usually happens when the search has no more improvement in cost for a number of iterations. The search selects specified number of solutions for each iteration and examines them within the neighborhood of the current solution. Better solution are all accepted but a worse solution could be accepted probabilistically using an exponential acceptance criterion based on the temperature. In this case worse solutions are accepted freely when the temperature is high, and less freely when the temperature is reduced [39].

So as mentioned above, the main advantage of this technique is to solve the problem of neighborhood search in accepting inferior solutions and improve the search results in solving complex optimization problems.

2.4 Genetic algorithms

Since the focus of this thesis was using genetic algorithms (GA) and apply GA to the sample programs (which we call them software under test or SUT), in this section I will explain GA and go into details of how it works.

2.4.1 Introduction to Genetic Algorithms

According to [39] Holland et al in the 1960s and 1970s first came up with the idea of genetic algorithms. Since they tried to model the natural genetic evolutionary process, this method is called “genetic algorithms”. Darwin’s principal of the survival of fittest and process of natural genetic are the basis of GAs; that is why we can call them computer model of biological evolution [36].

Compared to the natural genetic evolutionary process, GAs simulate the evolution of individual structures using three main operators which are: selection, mutation and reproduction. Each individual is called a chromosome and in
the beginning we have an initial population of these individuals (chromosomes) and GA maintain them according to genetic rules of selection and operators such as mutation and crossover. Fitness value here plays a vital role since it is a guide for reproduction because each individual receives a measure according to its fitness [36, 7].

There are three steps which after all of them are done we call them an iteration or generation. These three steps are: (i) evaluate all the individuals in the population using their fitness, (ii) perform the operations (crossover, reproduction, and mutation) and create a new population with the individuals that have high fitness value, (iii) throw away the old population and iterate using a new population. After these three steps we will have a generation.

There are five stages which together make the fundamental mechanism of GA [26]:

1. Generate the initial population (randomly).
2. Find and select the chromosomes that have best quality compared to fitness value.
3. Recombine selected chromosomes using the operators (crossover and mutation)
4. Inject offsprings into the population.
5. If the stop criterion is reached, return the chromosomes with the best fitness. Otherwise go to step 2.

Figure 2.1: Genetic Algorithms Overview
2.4. GENETIC ALGORITHMS

According to [26] there are some basic elements which need to be explained in order to clearly understand the GA mechanism.

First element is called representation or we better say genetic representation for potential solutions to the problem. This representation is actually modeling of chromosomes into data structures. Second element is the chromosome which is usually encoded as a bit string and we call it candidate data solution to the problem. Chromosome also is made of smaller element which is called allele. If we imagine 1-dimension binary representation then every allele is 0 or 1 and a chromosome is a sequence of these 0 and 1’s [26].

Initialization is the operator which randomly creates the initial population of the chromosomes [26].

Next operator is selection which determines the way of selecting chromosomes and the number of offsprings that each chromosome will create [26].

The crossover operator defines how an offspring will be generated from two parents. Crossover is actually a method for sharing information between two chromosomes. Crossover is the most important feature of GA [26].

Next operator is called mutation. Mutation operator alters one or more genetic cells or allele of the chromosome to increase the structural variability [36, 26]. Mutation works on one chromosome at a time and usually takes place after crossover [26].

2.4.2 Application of Genetic Algorithms

GAs have been applied to many different fields and have been used in many different applications as an adaptive search method, e.g. by Grefenstette et al. [1985], Alen and Lesser [1989], and Jog et al. [1989]. GAs also have been applied to computer software models to solve optimization problems. Some engineering projects had also applied GAs to solve their problems such as optimization of a gas pipeline control. GAs are applied to classical problem of the Prisoner’s Dilemma problem studied by Mhlenbein [1991], Fujiki and Dickinson [1987] and Wilson [1987]. Traveling Salesman Problem which is a famous combinatorial optimization problem, has been solved using GAs. The authors in [16] applied GA to a n job single machine scheduling problem and they came up with interesting results and the authors in [30] applied GA to generation of school timetables. And as we can see many different examples, GAs are applied in many fields to solve optimization problems especially when the search space is quite large.

2.4.3 Overview of Genetic Algorithms

Whenever we face with problems with large number of variables, GAs can present a robust non-linear search technique [36]. Although GAs might be slow compared to few other search techniques but there is no doubt in better quality of solution and therefore reduction in cost specially in software testing industry. GAs are being used to solve a variety of problems and they are becoming more powerful everyday specially in machine learning and function optimization. According to [36] the technique of applying recombination operators, which are crossover and mutation, is the primary power of GAs. GAs work from a population of points not from a single point like the other search techniques (hill climbing). In many search techniques such as hill climbing, we move carefully from one point in the decision space to the next but in GAs we move from a
population at the same time and climb parallelly many peaks therefore the risk of finding a local optimum is very low.

In the list below, there are four main characteristics of GAs [36]:

1. Focuses and works on chromosomes which are above average fitness.
2. Exploits information about large number of values and processing a small population at the same time.
3. Does not let the search to stop at a local optimum.
4. Generate new solutions with improved performance using the old knowledge from old population.
Chapter 3

Applying GAs to Software Testing

3.1 Introduction to Software Testing

Software testing as one of the primary activities in software development life cycle is intended to validate whether the software behaves as intended and identify potential activities [6]. Among all verification and validation techniques, testing the program is said to be the most effective one since it identifies more bugs and errors than the other techniques. Earlier studies and researches in this area, estimated that fifty percent, or even more, of total cost of development goes to testing and a recent survey in the United States proves the high economic impacts of poor software testing substructure. According to [6] software testing is a very broad term, which includes a wide range of different activities from unit testing to acceptance testing. Because software testing provides careful analysis during the software execution, it gives us realistic feedbacks about the software behavior and this gives the testing, an advantage which makes it the best technique, compared to the other analysis techniques [6].

The term “test criterion” in software testing, determines the aim of testing. Test criterion includes a set of rules which show when sufficient testing for a specified test criterion has been accomplished; these rules will define three main objectives: (i) aspects of SUT (program under exercise) (ii) criteria for terminating the test and (iii) the information that must show test’s completeness. A test criterion is ideal when it is both reliable and valid [39]. Another term which contains very important information is test case. Test cases could be generated both manually and automatically and they are the basis of software testing. According to [31] a good test case is not the one which shows the program works correctly, but is the one which has a high chance of detecting undiscovered faults.

One classification of software testing which is based on source of information to measure the adequacy, divides software testing categories into two general categories [33]:

**Structural (white-box) testing:** Structural or white-box testing considers the unit under test as a white box, which means, test is designed based
on the internal structure of the program and with complete knowledge of implementation design. So it requires programming skills because the tester has to identify all the paths in the software.

**Functional (black-box) testing:** Functional or black-box testing has nothing to do with the internal structure of the code and treats the software as a black box which should have certain functionalities. Black box tests are designed using specification information in hand, so the knowledge of internal structure of program is not available.

Obviously, in this thesis, white box testing has been used due to the structural test design. White-box testing, itself is also categorized into four types:

- Path-based
- Data-flow-based
- Fault-based
- Error-based

Between these four categories, the definition of *path-based testing* matches our study in this thesis. Since path-based testing is a category of structural testing, test design is based on fundamental control structure of the program. The control structure of the program is represented as control-flow graph [39]. Control-flow graphs were made as a model representation language for discrete event simulation and they are aimed to make information explicit to enable the development of simulation execution algorithms for parallel and distributed computers [11]. Authors in [24] defined control-flow graph as a directed graph a graph made of nodes and edges, where each node corresponds to a block of sequential code and each edge corresponds to possible flow of control between the blocks.

Other categories of white-box testing are out scope for this thesis so they are not discussed anymore here.

The term *automatic software testing* or *automatic test data generation* is what we focus on, in our study. Next section talks about automatic software testing.

### 3.2 Introduction to Automated Software Testing

As explained above, in previous section, test cases are primary elements of software testing since without them, it is not possible to test the programs. Korel in [22] defines test data generation as process of identifying program input data which satisfy selected testing criterion. These test data (test cases) can be generated manually or automatically. Automatic generation of test data helps to cut down the cost of developing the software because this process is very labor-intensive and expensive [22]. The authors in [9] believe cost reduction is not the only advantage of AST but it also increases the software reliability.

In AST, when there is a case which the goal is to find test inputs to satisfy a given set of adequacy test criteria, there are many existing ways but the most common ones are: random test data generation, symbolic or path oriented test data generation, and dynamic test data generation [27].
Random test data generation: as it is obvious from the name, this technique works by generating inputs by random until the useful input is reached. The problem with random test data generation is, it does not work properly in cases where there are complex programs or complex adequacy criteria and the reason is in such cases, number of adequate inputs may be very small and since this technique works randomly, the chance to select those small number of inputs from total number of inputs is very low. This technique is usually outperformed by other techniques, even in small and non-complex programs [27]. However, random testing is often used as the benchmark for evaluation with other [search] techniques, mainly because it is supposed to be the lowest acceptance rate [14].

Symbolic test data generation: consists of assigning symbolic values to variables in order to find out the mathematical design of the program and to have a picture of mathematical characterization of the program [27]. This means instead of using actual values, variable substitution is used [14]. Symbolic execution of a program encounters some problems; for example, it might need infinite amount of time due to dependability of iterations on non constant expressions. Another problem could arise when data is referenced indirectly [27]. There are disadvantages as well; for example it needs plenty of computer resources. This might happen in transformation of expressions. Also use of symbolic execution needs a symbolic evaluator of some particular languages, which takes a lot of time [14].

Dynamic test data generation: dynamic methods themselves, are divided into different categories but in general when we talk about dynamic test data generation we mean executing the program and exercising the information which are only available during run time. Miller and Spooner [1976] discussed that numeric optimization problem is the best way test data generation in structural testing [39]. The idea in dynamic test data generation is: collected data during execution can be used to satisfy the test criteria, in case that some test requirements are not satisfied yet [27]. Dynamic test data generation can be applied to wide range of programs because of its advantage. The advantage is, we should only be able to extract the needed information to calculate the objective function and to find out whether the path of execution leads to the point where the objective function is evaluated [27]. Authors in [38] believe that test-data generation is best formulated as numerical maximization problem and this is why dynamic methods are useful. Executing the dynamic test generation is possible through the [search] optimization techniques.

3.3 GA and Software Testing

As explained in section 2.4, genetic algorithms were made based on the natural model of evolution. They work basically with simulation of chromosomes (individuals) structures by using three main operators: selection, mutation, and
reproduction. GAs have been applied to many fields in order to solve optimization problems and because of their success, now a days, they are very popular. In this section we talk about how GAs can be applied to and help software testing industry.

In dynamic test data generation we come to the point where we need minimizing function. The paragraph below makes it clear what I mean by minimizing function.

Usually the test is aimed at discovering as many faults as possible, using a set of powerful tests. One problem which arises here is, it is almost impossible to predict the number of faults which will be discovered by a set of test cases and its reason is the unclear definition which we have about the term “fault”. So having some sort of standards, which help to decide when a program has been tested good enough, is very useful. These standards in software testing are called test adequacy. Now that, the test adequacy exists, next step is to generate test data which are good compared to test adequacy, and here automatic test data generation helps a lot since this job is very hard and time consuming by hand. High number of test adequacy types, leads to another problem, which can be solved using heuristic algorithms (minimizing function) and genetic algorithms is one of the most useful ones [27].

We talked about test adequacy criteria and how it works in software testing. In many cases, the code (program) need to be executed, due to specific test adequacy criteria. The term coverage analysis is one method of defining test adequacy criteria. A simple example could be "All statements of the program should be executed at least once", which is called statement coverage. There are similar coverage criteria like branch coverage [27]. However, GAs are shown to give more coverage than all other methods in many experiences [27, 28].

We defined fitness function (see section 2.1) as what is said to be our test goal. GAs use fitness function to select the fittest individuals. In software testing these individuals are test inputs; by looking at their value and compare to the fitness function, fit individuals will be selected. But, in order to be able to calculate the fitness function, the SUT should be instrumented. Instrumentation of code means each condition in the code must be transformed into an expression which is supposed to have the same value as the original condition [2]. After instrumenting the code, we are able to compile our code for testing. And finally, we can run our unit test case when we have our compile code.

When GAs are applied to software testing, they generate first set of chromosomes randomly which are in fact test elements. Then they calculate the fitness for each individual and then based on their fitness value, they perform mutation and crossover. This process will not stop until all individuals gain the maximum fitness. This is how, it reduces time and cost, because there is no need of human interference since all the mentioned process is automatic, from initial population to last generation [35].

Authors in [5] mention the advantages of using automated software testing using genetic algorithms.

• Thesis is no human biases which can influence the test suites. Thus, automated software testing [using GA] should produce stronger test cases.

• For very long sequence testing, there is a need for large scale test-banks which requires some form of automated generation.
3.3. GA AND SOFTWARE TESTING

- To form new cases, the test case breeding process recombines highly fit building blocks, this is why genetic algorithms seem to be appropriate. Because genetic algorithms, using crossover and mutation, construct large population of related test cases. So not only the repetition of same test case over and over but also entering many slight variations to the mix.

- Using automated techniques, it is possible to generate test cases that can focus on particular areas, either before or during the long sequence tests.
Chapter 4

Description of the manual iteration of the thesis

This chapter contains the information about how the iterations of the thesis are designed and in what way they have been followed.

Since the main focus of the thesis was to design and implement the automated software testing using metaheuristic search algorithms and show the results as the complexity increases, we decided to divide the experiment into two main parts.

1. Use written source codes, from low to high complexity, and apply GA to them so that we can prove as the complexity of programs increase, GA will be more efficient [than random testing] to test the programs. This iteration is called manual iteration since we are not generating the SUTs and we are using the ones which are made by hand.

2. This iteration does the same thing as first iteration. It differs from the first one in SUTs, since in this iteration SUTs are generated automatically from low to high complexity. First, programs are generated automatically then they go under the test which means GA will be applied to them.

4.1 Manual Iteration

This section gives a complete description about the first iteration of the thesis which is called manual iteration. To remove the confusion, it is necessary to mention manual here refers only to generation of SUTs not the test itself. When we talk about test we mean automatic test.

In the manual iteration of the experiment, I chose three different programs with different complexities, which means number of statements and the total cyclomatic complexity, and test them with both GA and random testing. Next subsection gives a brief explanation about the sample programs I used in the manual iteration.
4.1.1 SUTs description

In this experiment I needed to have programs with different complexities in order to have comparable results since the goal of this experiment is to compare the coverage that GA gives us on programs with different complexity. First SUT is called GCD or greatest common divisor. As its name says, this program aims to find the greatest common divisor of two integers. These two integers will be generated automatically. GCD is considered to be our least complex program in this experiment. Second SUT is called Triangle Area. This application accepts 3 values as input and then calculates the area of a triangle which is created by those 3 values. This program was more complex than than GCD and it is considered to be our second complex program. Last and most complex program is called Matrix Inverse. This application accepts a 2 x 2 matrix as input and returns the inverse of this matrix. From every point of view Matrix Inverse is our most complex program.

Note: the input values of these programs are all integer and double.

4.1.2 Manual Iteration Results

This section describes the results which I got from testing three different programs with different complexities. These complexities differ from statements, branches, LOC, and total cyclomatic complexity of the programs. As you can see, table 1 contains the results which we have from tests. Please note coverage in this table means statement coverage which was our fitness value while testing. Simplest coverage metric is the ratio between the covered objectives and total number of objectives [2] which in our case it is statement coverage so the ratio between the covered statements and the total number of statements.

<table>
<thead>
<tr>
<th>Program</th>
<th>non-c LOC $^a$</th>
<th>T C $^b$</th>
<th>C (GA) $^c$</th>
<th>C(Rnd) $^d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCD</td>
<td>15</td>
<td>3</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Triangle</td>
<td>18</td>
<td>4</td>
<td>80</td>
<td>75</td>
</tr>
<tr>
<td>Inverse Matrix</td>
<td>60</td>
<td>18</td>
<td>100</td>
<td>85</td>
</tr>
</tbody>
</table>

Table 4.1: Test Results

$^a$non-commented line of code  
$^b$Total Complexity  
$^c$Coverage with GA  
$^d$Coverage with random testing
Chapter 5

Libraries used: GEVA, Clover and JGAP

To execute the experiments in this study, I needed to use three Java libraries. These libraries helped a lot in running the experiments. The important areas which I used these libraries are: automatic program (SUT) generation, instrumenting the code, generating test report, and application of GA to SUTs. In coming sections, each of these libraries are explained in more details.

5.1 Clover

Clover is a tool which is designed to measure code coverage and is provided in form IDE plug-in. What code coverage means here is, the percentage of code which is covered by automated test. As it is mentioned above, clover is a tool for code coverage measurement; measuring code coverage means discovering which statements have been executed through a test run, and which ones have not been executed [18].

Nowadays, unit testing plays a very important role since it can improve the quality and predictability of our software releases. So it is vital to know how well our unit tests actually test the code, how many tests are enough and if we need more tests. Code coverage measurement tries to find an answer for these questions. Through the release cycles, there is always the possibility of adding a piece of code or functionality and to make sure that they meet the standards which are put in place when the project was first released, coverage measurement plays an important role [18].

There are three broad approaches to code coverage measurement [18].

Source code instrumentation adds instrumentation statements to the source code and then compiles the code with normal compile tool.

Intermediate code instrumentation adds new byte codes to instrument the compiled class files and generates new instrumented class.

Runtime information collection when the code executes, this approach collects information form runtime environment.
CHAPTER 5. LIBRARIES USED: GEV A, CLOVER AND JGAP

The one that Clover uses is source code instrumentation mainly because of its advantage. It produces the most accurate coverage measurement for the least runtime performance overhead. Clover measures three basic types of coverage analysis which are:

- **Statement**: statement coverage measures whether each statement is executed or not.
- **Branch**: branch coverage measures which branches are followed in control flow graph.
- **Method**: method coverage measures if a method was entered at all during execution.

Note: due to the low speed of instrumentation and report generation, Clover is only used in the manual iteration of this thesis. The reason of low speed is mostly because for each fitness function call, Clover needs to run the Java Virtual Machine and it takes a long time for each fitness to calculate the value.

5.2 JGAP [20]

To apply genetic algorithms in our experiment, there was a need to use JGAP. Using JGAP, it is possible to apply GA to a project. JGAP is actually a GA and GP component which is provided as a Java framework. JGAP has already default values for mutation, crossover and all other values which are used in GA. JGAP provides basic genetic mechanisms that can be easily used to apply evolutionary principles to problem solutions.

Since there is a need for defining a fitness function for JGAP, What I used as fitness function, is actually the value which the coverage report gives us. The value which we get from the report is between 0 and 1; the closer we are to 1, the better fitness we have. So the chromosomes which give good coverage and therefore good fitness, will be chosen for the next generation.

We should be careful with the fitness value calculation of each tool since some for some of them zero means bad fitness and for some of them the closer we are to zero the better fitness we have. For JGAP, the more we are closer to one, the better fitness we have but for GEVA it is vice versa.

5.3 GEVA [29]

GEVA stands for Grammatical Evolution and is developed at UCD’s Natural Computing Research Application Group. GEVA is an open source implementation of Grammatical Evolution (GE) and is released under GNU GPL v3.0, that supplies a search engine framework, a simple GUI and genotype-phenotype mapper of GE [29].

What GEVA does in our experiment, is actually generating Java programs with different complexities. When we want to solve a problem using GE, we first need to define a suitable BNF (Backus Naur Form). This BNF definition, actually describes the output language which is going to be produced by the system. BNF is used to express the the grammar of a language in form of production rules [29]. Since in this thesis, we need to generate programs (SUTs)
with different complexities, we need to define the variables in BNF such as the number of statements, number of branches, number and type of variables etc.

Below, is the BNF that I used as the grammar input for genetic evolution.

```bnf
<code> ::= <block>|<code><block>|<code><code>
<block> ::= <stmt>|<block><stmt>|<loopstatement>|<block><loopstatement>
<expr> ::= (var <oper> <expr>) | (expr <oper> var) | (var <oper> var) |
(var<oper<const>) | (expr<oper(expr))
<stmt> ::= "Instrument.increaseLast_MoveNext();\nAssign\n"<stmt><stmt>
<loopstatement> ::= "Instrument.increaseLast_MoveNext();\nInstrument.broadcast(x,y,z,a,b,c,d,e,f,g);\n"<whileloop> |
"Instrument.increaseLast_MoveNext();\nInstrument.broadcast(x,y,z,a,b,c,d,e,f,g);\n"<ifloop> |
"Instrument.increaseLast_MoveNext();\n"<forloop>
<whileloop> ::= "while" "(" <condition> ")" 
"\n" "{\niter++;\n" <stmt
"if(iter>=maxiter)\n"nbreak;\n"Instrument.setLoop();\n"Instrument.endLoop();\n"niter=0;\n
<forloop> ::= <forloopx>|<forloopy>|<forloopz>|<forloopa>|<forloopb>
<forloopa> ::= <forloopx>|<forloopy>|<forloopz>|<forloopa>|<forloopb>
<forloopz> ::= <forloopa>
<conditionx> ::= "x = "<expr>|x++|x--
<assignmentx> ::= "x = "<expr>|x++|x--
<conditiony> ::= "y = "<expr>|y++|y--
<assignmenty> ::= "y = "<expr>|y++|y--
<conditionz> ::= "z = "<expr>|z++|z--
<assignmentz> ::= "z = "<expr>|z++|z--
```

```
CHAPTER 5. LIBRARIES USED: GEVA, CLOVER AND JGAP

<forloopb> ::= "for" "(" <assignmentb> ",Instrument.broadcast(x,y,z,a,b,c,d,e,f,g);" <conditionb> ");" "\n\n\niter++;
" <stmt>

"if(iter>=maxiter)\n{\nInstrument.setLoop();\n}\nInstrument.endLoop();\niter=0;\n"

<forloopc> ::= "for" "(" <assignmentc> ",Instrument.broadcast(x,y,z,a,b,c,d,e,f,g);" <conditionc> ");" "\n\n\niter++;
" <stmt>

"if(iter>=maxiter)\n{\nInstrument.setLoop();\n}\nInstrument.endLoop();\niter=0;\n"

<forloopd> ::= "for" "(" <assignmentd> ",Instrument.broadcast(x,y,z,a,b,c,d,e,f,g);" <conditiond> ");" "\n\n\niter++;
" <stmt>

"if(iter>=maxiter)\n{\nInstrument.setLoop();\n}\nInstrument.endLoop();\niter=0;\n"

<forloope> ::= "for" "(" <assignmente> ",Instrument.broadcast(x,y,z,a,b,c,d,e,f,g);" <conditione> ");" "\n\n\niter++;
" <stmt>

"if(iter>=maxiter)\n{\nInstrument.setLoop();\n}\nInstrument.endLoop();\niter=0;\n"

<forloopf> ::= "for" "(" <assignmentf> ",Instrument.broadcast(x,y,z,a,b,c,d,e,f,g);" <conditionf> ");" "\n\n\niter++;
" <stmt>

"if(iter>=maxiter)\n{\nInstrument.setLoop();\n}\nInstrument.endLoop();\niter=0;\n"

<forloopg> ::= "for" "(" <assignmentg> ",Instrument.broadcast(x,y,z,a,b,c,d,e,f,g);" <conditiong> ");" "\n\n\niter++;
" <stmt>

"if(iter>=maxiter)\n{\nInstrument.setLoop();\n}\nInstrument.endLoop();\niter=0;\n"

<assignments> ::= "a = "<expr>|a++|a--
<assignmentb> ::= "b = "<expr>|b++|b--
<assignmentc> ::= "c = "<expr>|c++|c--
<assignmentd> ::= "d = "<expr>|d++|d--
<assignmente> ::= "e = "<expr>|e++|e--
<assignmentf> ::= "f = "<expr>|f++|f--
<assignmentg> ::= "g = "<expr>|g++|g--
<doloop> ::= "do {<stmt> } while "<condition> ";"

<conditionx> ::= "<expr>\n" <opera> "(" <notx> <oper> <const> ")"
<conditiony> ::= "<expr>\n" <opera> "(" <noty> <oper> <const> ")"
<conditionz> ::= "<expr>\n" <opera> "(" <notz> <oper> <const> ")"
<conditiona> ::= "<expr>\n" <opera> "(" <nota> <oper> <const> ")"
<conditionb> ::= "<expr>\n" <opera> "(" <notb> <oper> <const> ")"
5.3. GEVA [29]

| "("b<oper><const>")"<opera>"("<notb><oper><const>")"
| "("c<oper><const>")"<opera>"("<notc><oper><const>")"
| "("d<oper><const>")"<opera>"("<notd><oper><const>")"
| "("e<oper><const>")"<opera>"("<note><oper><const>")"
| "("f<oper><const>")"<opera>"("<notf><oper><const>")"
| "("g<oper><const>")"<opera>"("<notg><oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>")"

<condition> ::= <oper><const>"("<not<oper><const>")"
| <oper><const>"("<not<oper><const>
One of the most attractive advantages of using GE is, simply by editing the plain text, we can change the output structures [29]. What plain text means here, is actually the contents of BNF file which are easy to edit.

The software has two main components which are:

1. GUI
2. GEVA

GUI helps the beginners to explore the software by providing a simple interface. GEVA, for more advanced user, provides more advanced and unlimited suite of features and is accessible through the command line [29].

As you can see in figure 5.1, when we work with GEVA through the GUI, it is easier to set the configurations. I can say, the most three important boxes are properties, fitness function and grammar. For the properties, there is a properties file which we need to configure it in a way that works for us. It contains all the genetic terms. In the fitness function we need to choose our own fitness function which we wrote. The purpose of our fitness function could be generating programs with many statements, or with many branches etc. And finally the grammar contains the grammar which GEVA follows to evolve the programs; it actually contains the BNF of the problem. Using these three libraries (tools), we succeeded to make our experiment. As mentioned in previous sections, these libraries helped a lot to:

1. Generate programs automatically.
2. Instrument the source code.
3. Apply GA to SUT, in order to test them.
4. Getting the test report (coverage report).
Figure 5.1: GA vs. Random for medium complexity
CHAPTER 5. LIBRARIES USED: GEVA, CLOVER AND JGAP
Chapter 6

A Piece of Generated Sample Codes

In this chapter, you can see a piece of code which is extracted from one of the generated source codes.

Please notice, here I didn’t put the header of the source code. The header was constant for all the generated programs. Here is the header which you can see in the beginning of all programs:

```java
public class code {
public static void complex(double x,double y,double z){
int maxiter=1000;
int iter=0;
```

As you can see, all the programs have a public class which is called code. There are three input parameters x, y, and z. The integer variable, maxiter which is set to 1000 here defines how many times a loop could be run. So more than 1000 times, it will exit automatically. The integer variable, iter which is set to 0 is used as a counter which increments, each time a loop runs.

Following code is a sample of a body, from one of the generated source codes.

```java
Instrument.increaseLast_MoveNext();
Instrument.broadcast(x,y,z);
for(z=(y-8);(z-6212)>(y+9);y=(z-(z+(y-647))))
{
iter++;
Instrument.increaseLast_MoveNext();
y++;
if(iter>=maxiter)
{
break;
}
Instrument.setLoop();
}
Instrument.endLoop();
iter=0;
Instrument.increaseLast_MoveNext();
```
Instrument.broadcast(x,y,z);
for(x--;z<(x+49384);x=(z+z))
{
  iter++;
  Instrument.increaseLast_MoveNext();
  z=(x-y);
  Instrument.increaseLast_MoveNext();
  z--;
  Instrument.increaseLast_MoveNext();
  z=(y+z);
  Instrument.increaseLast_MoveNext();
  x=(z+969);
  Instrument.increaseLast_MoveNext();
  x++;
  if(iter>=maxiter)
  {
    break;
  }
  Instrument.setLoop();
}
Instrument.endLoop();

The expression, Instrument.increaseLast_MoveNext() is used to count the number of statements. Instrument.broadcast(x, y, z) is called before each loop to broadcast the value of x, y, and z. This is mainly because every time we enter a loop we have different values, set to x, y, and z; and because we need these values for function minimization we have to get them each time we enter a loop. Instrument.setLoop() says that we are inside a loop and Instrument.endLoop() defines this loop is finished so that the statements inside loops are not counted more than once.
Bibliography


