Comparing the effectiveness of automated test generation tools "EVOSUITE" and "Tpalus"

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF MINNESOTA
BY

Sai Charan Raj Chitirala

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. Andrew Brooks

July 2015

Acknowledgements

I would like to thank Dr. Andrew Brooks for his constant guidance, patience, enthusiasm and immense knowledge. He has continuously helped me throughout my research. I would also like to thank Dr. Gary Shute and Dr. Steve Trogdon for being a part of the thesis committee.

I would also like to express my sincere thanks to all the faculty members of the computer science department for their continuous support.

Lastly, I would like to thank my friends and family members for their love and constant support during my whole life and academic career.

Dedication

I would like to dedicate my thesis to my parents for their immense love and support at all the stages of my life.

Abstract

Automated testing has been evolving over the years and the main reason behind the growth of these tools is to reduce the manual effort in checking the correctness of any software. Writing test cases to check the correctness of software is very time consuming and requires a great deal of patience. A lot of time and effort used on writing manual test cases can be saved and in turn we can focus on improving the performance of the application. Statistics show that 50% of the total cost of software development is devoted to software testing, even more in the case of critical software. The growth of these automated test generation tools lead us to a big question of "How effective are these tools in checking the correctness of the application?" There are several challenges associated with developing automated test generation tools and currently there is no particular tool or metric to check the effectiveness of these automated test generation tools.

In my thesis, I aim to measure the effectiveness of two automated test generation tools. The two automated tools on which I have experimented on are Tpalus and EVOSUITE. Tpalus and EVOSUITE are capable of generating test cases for any program written in Java. They are specifically designed to work on Java. Several metrics have to be considered in measuring the effectiveness of a tool. I use the results obtained from these tools on several open source subjects to evaluate both the tools. The metrics that were chosen in comparing these tools include code coverage, mutation scores, and size of the test suite. Code coverage tells us how well the source code is covered by the test cases. A better test suite generally aims to cover most of the source code to consider each and every statement as a part of testing. A mutation score is an indication of the test suite detecting and killing mutants. In this case, a mutant is a new version of a program that is created by making a small syntactic change to the original program. The higher mutation score, the higher the number of mutants detected and killed. Results obtained during the experiment include branch coverage, line coverage, raw kill score and normalized kill score. These results help us to decide how effective these tools are when testing critical software.

Table of Contents

List of Tables

List of Figures

1. Introduction

In the present world of ever growing demand for software applications, there is often a need to find a way to check if the application being developed works on the inputs. We term this process as software testing. Testing is not only a process of finding the errors in the given application; it is also a way of measuring the effectiveness/correctness of the developed application. Software testing is considered to be one of the important phases of the Software Development Life Cycle.

The usual way of checking the correctness of the software is to write/generate test cases that check the software systematically and we also need test oracles, which are responsible for measuring the correctness of the obtained result. But given the amount of time being spent in testing rather than development, automated testing seems a very useful approach. There have been several automated test tools developed over the years, but there is a need to decide on which tool to use, the ones that are well suited for our use. This thesis aims to evaluate two state of the art automated testing tools for Java. Three major metrics are used: coverage scores, number of test cases and mutation scores.

Important terms covered in software testing with some brief explanation are as follows:

a. Test data: Test data are data, which have been specifically identified for use in testing a computer program.
b. Test case: Test case is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not.
c. Test oracle: The mechanism for determining whether a software program or system has passed or failed such a test is known as a test oracle.
d. Test suite: A set of test cases is called a test suite.
e. Test automation: Developing software for the automatic testing of a software product.

f. Coverage: Coverage of program or faults. The aim of coverage based testing is to 'cover' the program with test cases that satisfy some fixed coverage criteria. The aim of fault coverage criteria is to cover the most faults in a program.

g. Path: sequence of nodes and edges. If we start from an entry node and end at an exit node then we call that a complete path.

h. Branch predicate: is a condition in a node that may lead to either the true path or the false path being taken.

i. Path predicate: A path predicate is defined as the collection of branch predicates, which require to be true in order to traverse a path.

j. Feasible path: A path where there is valid input that executes path.

k. Infeasible path: A path where there is no valid input that executes the path.

l. Constraint: A constraint is an expression that specifies the semantics of an element, and it must always be true.

m. Constraint generator: is a program that can automatically generate the set of conditions or constraints in a path.

n. Constraint solver: is a program that provides value to the input variables of a path predicate such that it satisfies all the constraints of the path predicate at a time.

The next chapter (Chapter 2) gives the background information required to understand the thesis and it is followed by the experiment (Chapter3) conducted on these tools with results (Chapter 4) explaining how well the tools perform on the test subjects and how the scores of the two tools are compared in evaluation.

2. Background

2.1 Unit Testing

[23] A unit test is “a test, executed by the developer in a laboratory environment, that should demonstrate that the program meets the requirements set in the design specification.” Unit testing means testing the smallest module in the system. The primary goal of unit testing is to take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect. Each unit is tested separately before integrating them into modules to test the interfaces between modules. Unit testing has proven its value in that a large percentage of defects are identified during its use.

2.2 EVOSUITE

EVOSUITE is an automated test generation tool, which generates test cases with assertions [1] [7]. EVOSUITE suggests possible oracles by studying the program behavior. A description of how EVOSUITE works is explained in the next section.

We have talked about automatic test generation tools producing test cases, but there is a major concern in generating them. There needs to be a correct way of determining oracles i.e., a way to verify that the outputs of the test cases are the expected ones. There is a chance that faults can be detected, if they lead to a program crash. But not all faults lead to program crashes or deadlocks.

EVOSUITE effectively produces test cases which have high coverage and generate a minimal amount of assertions that covers the entire code. There are several steps that EVOSUITE goes through before generating the final test suite. It follows an evolutionary approach, which evolves the test suite generation, primarily focusing on a coverage criterion. Branch coverage is considered at the byte-code level. High-level branch statements in Java (e.g. predicates in loop conditions) are transformed into simpler statements similar to atomic “if ” statements in the byte-code. During test generation, EVOSUITE considers one top-level class at a time. The class and all its anonymous and member classes are instrumented at the byte-code level to keep track of called methods and branch distances during execution.

2.2.1 Whole Test Suite Generation

The approach taken by EVOSUITE to generate a whole test suite is called a search-based approach [2]. EVOSUITE goes through the following steps before generating the final test suite.

1. A set of random test suites are generated as an initial population.
2. A test suite is evolved using an evolutionary search (Genetic Algorithm) towards satisfying a chosen coverage criterion.
3. The best resulting test suite is minimized.

In terms of generating a test suite, coverage criterion is an important choice. EVOSUITE chooses branch coverage as the default coverage criterion. Mutation coverage is also a possible choice as a coverage criterion. In the process of the generating the best test suite, EVOSUITE uses a genetic algorithm.

2.2.2 Genetic Algorithm

The genetic algorithm [2] used in EVOSUITE is given below:

```
current population ← generate random population
repeat
    Z ← elite of current population
    while |Z| not equals |current population| do
    P1,P2 ← select two parents with rank selection
```

```
        if crossover probability then
            $O_1,O_2 \leftarrow$ crossover $P_1,P_2$
        else
            $O_1,O_2 \leftarrow P_1,P_2$
        mutate $O_1$ and $O_2$
        $f_p$ = min(fitness($P_1$),f itness($P_2$))
        $f_p$ = min(f itness($O_1$),f itness($O_2$))
        $l_p$ = length($P_1$) + length($P_2$)
        $l_o$ = length($O_1$) + length($O_2$)
        $T_B$ = best individual of current population
        if $f_o < f_p \vee (f_o = f_p \wedge l_o \leq l_p)$ then
            for O in { $O_1,O_2$ } do
                if length(O) $\leq 2 \times$ length($T_B$) then
                    $Z \leftarrow Z \cup \{O\}$
                else
                    $Z \leftarrow Z \cup \{P_1$ or $P_2\}$
        else
            $Z \leftarrow Z \cup \{P_1,P_2\}$
        current population $\leftarrow Z$
until solution found or maximum resources spent
```

Initially, the random test cases are used to initialize the first generation used in the algorithm. The next step in the process is evolution of the given population. Offspring are generated from parents. The process is repeated to find a better test suite from the initial test suite. This process includes the following steps:

a. Two parents (test suites) are selected with the highest rank (i.e. with highest coverage).

b. Two offspring are generated using the crossover technique and if the crossovers are better, then they are selected over the parents (if the crossovers have better coverage than the parents).
c. Selected test suites are mutated to improve the chosen criterion i.e. branch coverage.
d. A fitness function is used to select the best test suite. The fitness function is explained below.
e. The stopping criterion for test suite generation is easy: If the entity to be reached (e.g. a branch) is actually reached.
f. Consequently, the genetic algorithm iterates until a number of generations has been reached and returns the test suite with the highest fitness.

The fitness function is the most important aspect in test minimization/test optimization. It is responsible for minimizing the test suite and choosing the best test suite in the evolutionary approach. The fitness function estimates how close a test suite is to covering all branches of a program. Therefore it is important that each predicate has to be executed at least twice so that each branch can be taken (if and else case). The best test suite is the parent from which other offspring are created. A test suite with highest fitness value is chosen in the evolutionary approach.

The fitness of a test suite is calculate by the formula

$$|M| - |M_T| + \sum_{b_k \in B} d(b_k, T)$$

M is the total number of methods in the set of methods.
Mt is the total number of executed methods.
d is the branch distance, rules to calculate the branch distance are mentioned in [28] [29]
b_k is the branches in the program
T is the entire test suite

The next step in the process is to produce assertions for a test case. [16] A test case is run against the original program and against all mutants and the necessary information recorded. Observers are responsible for recording the values and they indicate the type of assertion (Primitive, Comparison, etc.). Traces generated by the observers are analyzed for differences between the runs on the original program and on the mutants. For each difference an assertion is added. Numbers of assertions are also minimized by tracing for each assertion, which mutation it kills, and then finding a subset for each test case that is sufficient to detect all mutations that can be detected with this test case.

2.3 Tpalus

Test generation techniques being used in most state of the art of the tools include: bounded-exhaustive, symbolic execution-based, and random approaches. The random approach is relatively easy to use and is scalable. The strategy in Tpalus is to consolidate both a static and dynamic investigation method.

Test generation in Tpalus happens in three stages: dynamic inference, static analysis and guided arbitrary test generation. In the dynamic inference stage, Tpalus takes a sample execution as an input and infers the call sequence model, i.e. it tries to interpret the call sequence for the code being tested. The static analysis first recognizes field access (read or write) data for every method and utilizes a tf-idf weighting plan to weight the reliance between every pair of method. The important feature of Tpalus is that it does not miss out methods not covered in the initial sample execution. This step is an important difference between Tpalus and other automated test generation tools. Finally, dynamically inferred model and statistically identified dependence information is used in the random test generator to create the test suite [11].

Now, I will describe the three phases of the tool in detail in the following sections:

2.3.1 Dynamic Analysis: Model Inference from Sample Executions

The process of dynamic analysis in the test generation tool is to produce a call sequence model, which is a rooted, directed, acyclic graph. A model is constructed for every class under test, where the edges represent the calls to the methods along with their arguments and each node in the graph represents a collection of object states. To summarize this, a model has nodes, which represent the state of the object when a method is called (a change in the state of the object). The call sequence model has many problems; this model might lead to illegal sequences, which leads to an illegal behavior. Tpalus enhances the existing model with two additional constraints. Direct state transition dependence constraints are related to how a method argument is created. Abstract object profile constraints are related to the value of a method argument's field.

2.3.1.1 Direct State Transition Dependence Constraint

This constraint makes sure that the call sequence model is not an illegal sequence. It does so by adding direct state transition edges from one node to another, which indicates that the first object state may be used as an argument when extending the object at the second state.

2.3.1.2 Abstract Object Profile Constraint

This is the second constraint that is used on the call sequence model. In the model every object is defined by the concrete state as a vector $v_i = (v_1,v_2...v_n)$, where v_i is the value of the object field. The aim of the abstract object profile constraint is to map the values for the object field with the help of the abstract function. The rules are as follows:

- A concrete numeric value v_i is mapped to the following abstract values, where $v_i > 0$, $v_i < 0$ and $v_i = 0$
- An array value v_i is mapped to any of the four abstract values, v_i = null, v_i = empty, v_i contains null and all other values.

- An object reference value of v_i may be, v_i = null and v_i not equals to null
- The values of Boolean and enumeration values are not abstracted.

For example, consider an object of a class named “obj1” and it has a method named method1 which has four arguments passed to it. When obj1 calls the method, it reaches a new state.
method1(string name, int id, string url, float num), when the object calls the method, it reaches to a new state v and it corresponds to an abstract state s=(not null, not null, not null, not null).

2.3.2 Static Analysis: Model Expansion with Dependence Analysis

As the name indicates this stage is an extension to the model that was generated during the dynamic analysis stage. This stage differentiates Tpalus from other automated testing tools. Many tools mainly employ the dynamic analysis stage, which has a lower chance of exploring new program states. The hypothesis behind the static analysis is that two methods are related if they read/write the same field, i.e. two methods are accessing the same method. This behavior leads to new program states and helps in building the model completely on the methods which were uncovered in Dynamic Analysis stage.

Two methods are related in two ways:

- Write-read relation: When one method writes the field and the other method reads the field, then it is termed a write-read relation.
- Read-read relation: When two methods read the same field, then it is termed a read-read relation.

Apart from the method relations, Tpalus comes up with another measurement, which concentrates on finding how tightly the methods are coupled. Tpalus uses the tf-idf metric to evaluate the importance of fields to each method [14]. tf-idf, termed frequency–inverse document frequency, is a metric which defines how important a word is to the document in which it is present. The dependence relevance W(m_k,m_j) between methods m_k and m_j is the sum of the tf-idf weights of all fields.

2.3.3 Guided Test Generation

This step comprises of two sequence generation algorithms. The first algorithm takes in four parameters. These include: time limit for test generation, set of methods for test generation, enhanced call sequence models and method dependence relations. It has two phases: the first phase for the methods covered in the sample execution and the second phase for all the uncovered methods.

The second algorithm extends the sequence using the static and dynamic dependence information. Given a tested method *m()*, the algorithm uses the constraints recorded in the dynamic model to find the sequences for its parameters. Then, the algorithm uses the method dependence information to find the dependent method *m_dep()* and find the parameter sequences for it. Finally, both the sequences are concatenated to get the final sequence.

Combining the dynamic analysis, static analysis stage along with the guided test generation stage, Tpalus makes use of the ASM bytecode analysis framework. Use of the ASM bytecode work allows Tpalus to build the execution trace. Even the method dependence is recorded using the bytecode. Tpalus initially takes in a sample execution and time limit (maximum time spent on each class) from the user and the file containing the names of classes under test. A simple trace, which is given as an input for the tool, is very useful for Tpalus. It has been suggested that Tpalus results are not very sensitive to its input execution trace. This is a big advantage as there is no concern that there are no tests being generated for the methods not covered in the initial trace. Tpalus generates regression oracles that capture the behavior of the program under test and it also integrates JUnit theory, permitting programmers to write down domain specific oracles. In terms of generating assertions, Tplaus keeps track of all the sequences that were captured and now it tries to capture the behavior using them by generating regression assertions. It generates parameters with the values by using the rules specified in section 2.3.1.2. A theory is a generalized assertion that should be true for any data. Tpalus can be thought of as an extension to Randoop.

2.4 Mutation Testing

[20] Mutation testing is about fault based testing, which results in a “mutation score". The mutation score quantifies the adequacy of a test suite as far as its capacity to identify faults. The faults utilized by mutation testing are used to represent the mistakes the programmers make. Such faults are intentionally seeded into the first program, by basic syntactic changes, to make a set of faulty programs called mutants.

Example of a mutant:

If (x > y) -> If (x >= y)

Above is an example of a generated mutant.

[21] To assess the quality of a given test suite, these mutants are executed against the input test suite. If the result of running a mutant is different from the result of running the original program for any test cases in the input test suite, the seeded fault denoted by the mutant is detected and is termed killed. The goal of mutation analysis is to raise the mutation score to 1, i.e. to make sure that the test suite is effective enough to kill all the faults in the program. After all test cases have been executed, there may still be a few ‘surviving’ mutants. To improve the test suite T, the programmer can provide additional test inputs to kill these surviving mutants. However, there are some mutants that can never be killed, because they always produce the same output as the original program. These mutants are called equivalent mutants. They are syntactically different but functionally equivalent to the original program. Automatically detecting all equivalent mutants is impossible, because program equivalence is not decidable. The equivalent

mutant problem has been a barrier that prevents mutation testing from being more widely used.

[20] Although mutation testing is able to effectively assess the quality of a test suite; it still suffers from a number of problems. One problem that prevents mutation testing from becoming a practical testing technique is the high computational cost of executing the enormous number of mutants against a test set. The other problems are related to the amount of human effort involved in using mutation testing. Determining if a mutant is equivalent is very time consuming.

For all the above reasons, to make mutation testing into a practical testing technique many cost reduction techniques have been proposed. They are widely divided into two categories:

1. Mutant Reduction:

   a. Mutant sampling is a simple approach that randomly chooses a small subset of mutants from the entire set.
   b. Selective mutation involves using an effective subset of mutation operators.

2. Execution Reduction:

   a. Strong mutation or traditional mutation testing. In strong mutation, for a given program *p*, a mutant m of program *p* is said to be killed only if mutant *m* gives a different output from the original program *p*.
   b. In weak mutation, a program p is assumed to be constructed from a set of components C = {$c_{i,}$ … c_m}. Suppose mutant m is made by changing component c_m, mutant m is said to be killed if any execution of component c_m is different from mutant m. As a result, in weak mutation, instead of checking mutants after

the execution of the entire program, the mutants need only to be checked immediately after the execution point of the mutant or mutated component.

c. Firm mutation: The idea of firm mutation is to overcome the disadvantages of both weak and strong mutations by providing a continuum of intermediate possibilities. That is, the ‘compare state’ of firm mutation lies between the intermediate states after execution (weak mutation) and the final output.
d. Other techniques in execution reduction include compiler based techniques, and interpreter based techniques.

Even though mutation testing has its flaws, applying some of the above techniques make it a more reliable testing technique.

2.5 Major Mutation Framework

Major is a mutation framework and as the name indicates it is responsible for generating the mutants for the specified targets [5]. Major allows us to perform mutation testing on large software subjects written in Java. It mainly works in two steps,

1. Generate and embed the mutants during the compilation.
2. Run the actual mutation analysis, which identifies the generated mutants.

Mutant creation using Major can be done in two ways, using Apache Ant or by using Major’s Driver stand-alone. I chose to generate mutants using Apache Ant to integrate the Major tool along with the Cobertura code coverage tool, which will be explained in a later section. In the process of generating mutants, we have to initially write an MML script, which is a domain specific language for the Major mutation framework. Mutants are then generated using Major’s compiler.

The mmlc compiler, which is a part of the Major Mutation Framework, compiles and converts input into a binary representation.
An example of how to compile an MML script is as follows:

```
major$ mmlc example.mml example.mml.bin
```

The next step is to generate the mutants, which is done using the javac compiler, which also is a part of the Major mutation framework. Major used its own mutation compiler, which extends the “OpenJDK” Java compiler All the generated mutants are integrated into the mutants.log file and each line in the file represents a single mutant that has been generated by mutating a single line of code. Now that we have all the generated mutants for the given source code, our next task is to analyze the mutants. This also is specified as a target in the build.xml file and the results are simultaneously stored in a results.csv file and a killed.csv file. Here is a small description of mutant details that are kept in the mutants.log file.

The file is divided into 7 columns and a mutant is present in each row, where each mutant looks like this.

Mutant: 1:LVR:TRUE:FALSE:Plant@<init>:9:true |==> false
Details of every column:
Column1- Mutant Id number of the generated mutant
Column 2- Name of the applied mutation operator of all the 8 mutant operators
Column 3- Original operator symbol in the source code
Column 4- Replaced operator symbol for the symbol in column 3
Column 5- Fully qualified name of the mutated method
Column 6- Line number in original source file
Column 7- Visualization of the applied transformation (from |==> to)

Following are the details of the implemented mutation operators in the Major mutation framework

1. AOR Arithmetic operator replacement a + b ------> a – b {+,-,*,/,%}
2. LOR Logical Operator Replacement a ^ b -------> a | b {&,|,^}
3. COR Conditional Operator Replacement a || b ------> a && b {||,&&}

4. ROR Relational Operator Replacement a == b ------>a >= b {>=,!=,FALSE}

5. SOR Shift Operator Replacement a >> b ------> a << b {<<,>>,>>>}

6. ORU Operator Replacement Unary -a -------> ~a {+,-,~}

7. STD Statement Deletion Operator: Delete (omit) a single statement foo(a,b) ------> <no-op>

8. LVR Literal Value Replacement: Replace by a positive value, 0 ------> 1
a negative value, and zero 0 ------> -1

2.6 Cobertura

Cobertura is a Java tool that calculates the percentage of source code accessed by tests [9]. Code coverage identifies which part of the source code lacks coverage. The coverage measured can be of several forms. Major forms of coverage include: branch coverage, statement/line coverage. A test case that does not 'increase coverage of the program' and does not 'cause a failure' is considered ineffective. Measuring code coverage is important for testing and verifying code during both development and porting to new platforms [22].

2.7 Apache Ant

Apache Ant is a Java library, which can be used as a command line tool whose main functionality is to implement targets specified in a build.xml file. The major idea behind Apache Ant is that it can be used as a Java build tool i.e. to build Java applications. Processes in Ant are primarily defined in terms of targets.

The build file is written in XML format and each build file must have a single target. A target can depend on another target. Consider an example where you have a target to compile and you have a target to distribute. Distribution can be done only after the compile target is implemented. So the distribute target relies upon the compile target. Ant determines these conditions.

When Ant is run without any specified target, it runs the target specified in the default attribute of the <project> tag. It is also possible to specify one or more targets that should be executed.

Here is an example of how to run Ant from the command line [10]:

```
$ ant
```

runs Ant using the build.xml file in the current directory, on the default target.

```
$ ant -buildfile test.xml
```

runs Ant using the test.xml file in the current directory, on the default target.

```
$ ant -buildfile test.xml first
```

runs Ant using the test.xml file in the current directory, on the target called first.

3. Implementation of the Experiment

This chapter mainly focuses on how the thesis experiment was designed and conducted on the tools. Several heuristics have been considered in evaluating the performance of the tools.

Aspects that were considered for evaluating the performance of these tools [4] are:

1. Size of the generated test suite.
2. Coverage of the test suite on the source code.
3. Mutation testing (How effectively a test suite kills the generated mutants).

Some of the other factors that were considered to measure the performance include: time taken to generate the test suite, readability of the test suite, extensibility of the test suite, and ease of using the tools.

The following sections explain the reason why these aspects were chosen and what problems were encountered trying to evaluate performance.

3.1 Approach

In the first phase of the implementation, I primarily focused on getting the tools working. I worked with the tools on small jar files and small programs. I ran the tools from the command line, which takes an input from the user as a jar file along with several options. Tpalus runs in two main steps. The first step tries to run a sample execution provided by the user (i.e. a driver program) and the tool generates a sample trace (the trace model). This trace is used to build the entire test suite. In the second step, Tpalus requires the name of the file, which contains the names of the classes for which tests are to be generated. EVOSUITE works in a similar fashion, by accepting the class file or the jar file for which the tests are to be generated.

Experimenting required test subjects on which the performance of the tools could be evaluated. Test subjects chosen for the experiment were as follows:

Name of the Class File	Name of the Jar
MyHashMap	HashMap
ArrayIntList	Commons Primitives
Rational	Math4J
AVLTree	AvlTree
Heap	HeapSort
Plant	PlantMutant
WeightedGraph	MinimumSpanningTree
Time2	Time
BasePlusCommissionEmployee	Polymorphism
List	List

Table 3.1: List of all the programs used in the experiment

Among the ten test subjects used, two of them are from the experiment reported in [6]; authors of this paper were the ones who implemented EVOSUITE. Other test subjects used in the experiment include frequently used Java programs taken from [24] [25]. Dr. Brooks provided the plant program. The reason for choosing these programs was to get a better understanding on the generated test cases as the programs were well known and the experiment was only conducted on a single class in every jar file, making it relatively easy to understand the generated test cases.

To proceed with the evaluation process, each test subject was organized into a separate folder. The folder includes the jar files of EVOSUITE and Tpalus, utility Python programs and Bash scripts that are required to run the experiment. Each folder comprises of two subfolders names Major_EVOSUITE and Major_Tpalus, which comprised the mutation analysis work for the tests generated by EVOSUITE and Tpalus. Tests were generated not once but five times for each test subject. So results are averaged over five runs. In the initial run the tests for the test subject are created using the default configurations set for EVOSUITE and Tpalus. There are several options that can be set for Tpalus and EVOSUITE before generating the test cases. The utility python program written allows the user to easily set the recommended configuration required to generate the test cases for his needs.

From the second run to fifth run, options were varied in each run for EVOSUITE and Tpalus to assess the variability in results. I then tried to automate the use of the two tools to make it easier to perform the experimented work. The problem with these tools is that they are complex to use and results might differ by accidentally missing out the command line options, which leads to a varied result and ultimately leads to a wrong comparison between the test subjects. In the process of automating the tools, I wrote utility python programs that support the user in providing the names of the test subjects, its dependencies and various kinds of options available with the tool.

3.1.1 Using EVOSUITE

EVOSUITE is implemented from the command line [7]. EVOSUITE is also available as a plugin for Apache Maven and IntelliJ IDEA plugin, but for the thesis, I chose to run the tool at the command line, which allows us to explore all the options available for EVOSUITE.

Following are examples of how the tools are run from the command line:

```
java   -jar  EVOSUITE.jar  -class  org.Plant   -projectCP  ./PlantMutant.jar
```

This is a simple example of how to run EVOSUITE on a sample class named “Plant” present in the package named *“org”*, -projectCP along with its argument sets the classpath for test generation.

Here is a complex example of running EVOSUITE with several options.

java -jar EVOSUITE.jar -generateSuite -Dsearch_budget=60 -Dstopping_condition=MaxTime -Dsandbox=false -target PlantMutant.jar

generateSuite: use whole suite generation. This is the default behavior
search_budget: Integer value indicating maximum search duration
stopping_condition: Indicates a Boolean value to stop optimization once a goal is covered.
sandbox: Boolean value indicating to execute tests in a sandbox environment

In the above example, EVOSUITE is run on a target jar file named PlantMutant.jar. EVOSUITE is run only for 1 minute. Other options are to disable the sandbox in which tests are executed with a restrictive security manager. EVOSUITE has many parameters, which can be set on the command line by passing properties using -Dkey=value, where key indicates the option name

There is another way we can setup the options for EVOSUITE before generating the test suite. The -setup option for EVOSUITE reads the input for the test generation and generates a file named “EVOSUITE.properties” in the current working directory. We can edit this file and set the required properties before running the tool. In this way, EVOSUITE reads user’s mentioned options from the file and generates the test suite according to the requirements. Here is an example of how to setup the test suite with specific requirements.

```
java -jar EVOSUITE.jar -setup -class PlantTest.Plant -projectCP ./PlantMutant.jar
```

Running EVOSUITE from the command line is a tedious task and there is a risk of missing important configurations and may lead to faulty test suites. To avoid this I have implemented EVOSUITE by creating a python API. The API I have implemented can perform all the tasks EVOSUITE can do when it is run from the command line. The API lets you add additional options other than the default configurations.

Below is a code snippet that is a test suite generated by EVOSUITE when run on a class named *"Plant"* in the jar file *"PlantMutant"*.

```
/*
* This file was automatically generated by EVOSUITE
* Fri Jun 12 19:27:20 CDT 2015
*/
package org;
import static org.junit.Assert.*;
import org.junit.Test;
import org.Plant;
public class PlantEVOSUITETest {
 //Test case number: 0
 /*
  * 2 covered goals:
  * 1 org.Plant.talkToPlant(Ljava/lang/String;)V: I27 Branch 10 IFLE L65 - false
  * 2 org.Plant.talkToPlant(Ljava/lang/String;)V: I43 Branch 12 IF_ICMPNE L67 - false
  */
 @Test
 public void test0()  throws Throwable  {
     Plant plant0 = new Plant((-1));
```

```
    plant0.talkToPlant("please");
    assertEquals(1, plant0.getSize());
}
//Test case number: 1
/*
 * 4 covered goals:
 * 1 org.Plant.talkToPlant(Ljava/lang/String;)V: I14 Branch 8 IFLE L64 - true
 * 2 org.Plant.talkToPlant(Ljava/lang/String;)V: I23 Branch 9 IFLE L65 - true
 * 3 org.Plant.talkToPlant(Ljava/lang/String;)V: I34 Branch 11 IF_ICMPNE L66 - true
 * 4 org.Plant.talkToPlant(Ljava/lang/String;)V: I43 Branch 12 IF_ICMPNE L67 - true
 */
@Test
public void test1()  throws Throwable  {
    Plant plant0 = new Plant((-1));
    plant0.talkToPlant("");
    assertEquals(false, plant0.isAlive());
    assertEquals(0, plant0.getSize());
}
//Test case number: 2
/*
 * 1 covered goal:
 * 1 org.Plant.animalBite(I)V: I3 Branch 7 IFGE L54 - false
 */
@Test
public void test2()  throws Throwable  {
    Plant plant0 = new Plant((-1));
    // Undeclared exception!
    try {
      plant0.animalBite((-883));
      fail("Expecting exception: IllegalArgumentException");

    } catch(IllegalArgumentException e) {
       //
       // neg. amount
       //
    }
}
```

The test cases produced by EVOSUITE are very short and the total number of test cases generated by EVOSUITE are few in number. They are readable and easily understood, and have the comments that are automatically created. These comments mention the goals that were covered in the creation of a test case. Each goal makes note of the methods in the original program.

3.1.2 Using Tpalus

The process of generating a test suite from Tpalus is more complicated than using EVOSUITE. As mentioned it is usually done in two steps. Here is the example of how to obtain a trace:

```
java  -javaagent:./palus-0.2-nodept.jar=Plant –cp
./palus0.2nodept.jar:./PlantMutant.jar org.PlantDriver
```

After the above command is run, a trace model is generated in the current working directory named Plant_trace.model. In the above command, the -javaagent option is to invoke a JVM agent class, which has the premain class so that it is loaded after the JVM starts. Tpalus uses the ASM framework to perform bytecode instrumentation. This allows Tpalus to add constraints and also method dependence analysis is calculated at the bytecode level.

The second step in using Tpalus is test generation. An example of the command to generate tests is:

```
java  -Xmx2G -cp ./palus-0.2-nodept.jar:./PlantMutant.jar palus.main.OfflineMain --time_limit 10 --class_file ./testfile.txt --trace_file Plant_trace.model
```

The above command informs Tpalus to generate tests in 10 seconds, for classes defined in the testfile.txt file, using a model built from trace file Plant_trace.model. Heap size is set to 2GB for the test generation. There are several other options available for Tplaus. As observed, running Tpalus is more complex than EVOSUITE, so I have again tried to implement Tpalus by creating a command line parser using the python Argparse module to make it easier to use. Instead of adding the command line arguments and the additional options required for the test generation, the command line parser for Tpalus makes it easier and a less error prone way of test generation. Anything additional that a user needs for test generation can be added using the command line parser. Two steps in Tpalus test generation are now reduced to a single step. Tests generated by Tpalus are stored in a folder named tests in the current working directory.

In the process of comparing the test suites, the first and important task is to find a way to generate mutants. The Major mutation framework [5] was used for the purpose of mutation testing. The primary aim of mutation testing is to check how effective a test suite is in detecting a mutant. Mutants are nothing but artificial faults based on what is thought to be real errors commonly made by programmers. A good test suite must be capable enough to detect the mutant and kill it. Mutation analysis [16] is a technique to assess a test suite in how great it is at recognizing issues, and to give knowledge into how and where a test suite needs change. In the process of mutation analysis, test cases generated by Tpalus and EVOSUITE are executed on a program variant (mutant containing one such fault) to see if any of the test cases can identify that there is a flaw. A mutant that is identified thusly is viewed as killed. A live mutant, on the other hand, demonstrates a situation where the test suite needs to be improved.

Below is a code snippet which is a test suite generated by Tpalus when run on a class named *“Plant”* in the jar file *“PlantMutant”*.

```
public void test4() throws Throwable {

  org.Plant var0 = new org.Plant();
  java.lang.Integer var1 = new java.lang.Integer(3);
  var0.addFertilizer((int)var1);
  int var3 = var0.getSize();
  java.lang.Integer var4 = new java.lang.Integer(5);
  var0.addFertilizer((int)var4);
  int var6 = var0.getSize();
  var0.addWater();
  boolean var8 = var0.isAlive();
  int var9 = var0.getSize();
  var0.addWater();
  boolean var11 = var0.isAlive();
  int var12 = var0.getSize();
  boolean var13 = var0.isAlive();
  int var14 = var0.getSize();
  boolean var15 = var0.isAlive();
  int var16 = var0.getSize();
  var0.addWater();
  boolean var18 = var0.isAlive();
  var0.addWater();
  var0.addWater();
  int var21 = var0.getSize();
```

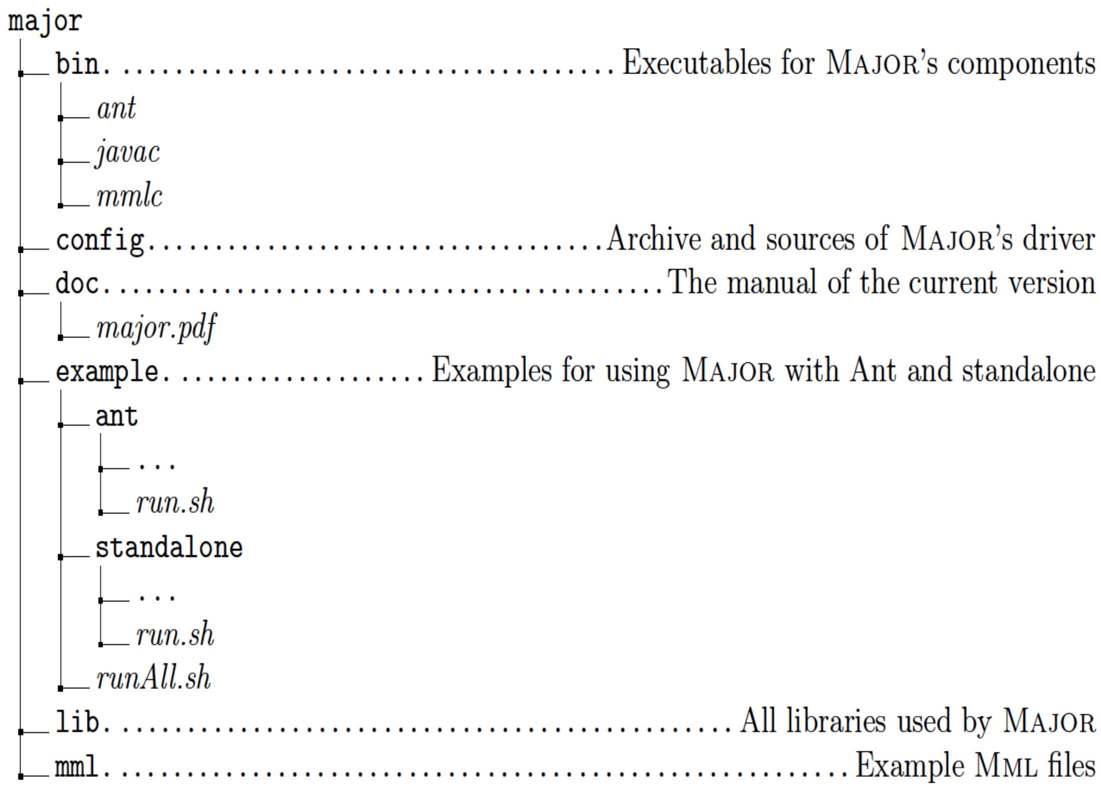
```
java.lang.Integer var22 = new java.lang.Integer(999);
var0.addFertilizer((int)var22);
int var24 = var0.getSize();
boolean var25 = var0.isAlive();

// Regression assertion (captures the current behavior of the code)
assertTrue(var3 == 4);
// Regression assertion (captures the current behavior of the code)
assertTrue(var6 == 4);
// Regression assertion (captures the current behavior of the code)
assertTrue(var8 == true);
// Regression assertion (captures the current behavior of the code)
assertTrue(var9 == 4);
// Regression assertion (captures the current behavior of the code)
assertTrue(var11 == true);
// Regression assertion (captures the current behavior of the code)
assertTrue(var12 == 2);
// Regression assertion (captures the current behavior of the code)
assertTrue(var13 == true);
// Regression assertion (captures the current behavior of the code)
assertTrue(var14 == 2);
// Regression assertion (captures the current behavior of the code)
assertTrue(var15 == true);
// Regression assertion (captures the current behavior of the code)
assertTrue(var16 == 2);
// Regression assertion (captures the current behavior of the code)
assertTrue(var18 == false);
// Regression assertion (captures the current behavior of the code)
assertTrue(var21 == 0);
// Regression assertion (captures the current behavior of the code)
```

```
  assertTrue(var24 == 0);
  // Regression assertion (captures the current behavior of the code)
  assertTrue(var25 == false);
 }
```

The above code is a single test case generated by Tpalus. When compared to EVOSUITE they are relatively very long. When compared to EVOSUITE test cases, Tpalus does not generate any comments for the generated test cases.

3.1.3 Implementing Mutation Analysis



```
major
 |__bin. ...................................... Executables for MAJOR's components
 |   |__ant
 |   |__javac
 |   |__mmlc
 |__config. ................................. Archive and sources of MAJOR's driver
 |__doc. ......................................... The manual of the current version
 |   |__major.pdf
 |__example. .................. Examples for using MAJOR with Ant and standalone
 |   |__ant
 |   |   |__...
 |   |   |__run.sh
 |   |__standalone
 |   |   |__...
 |   |   |__run.sh
 |   |__runAll.sh
 |__lib. ............................................... All libraries used by MAJOR
 |__mml. ......................................................... Example MML files
```

Fig 3.1: The directory structure of the Major tool

To perform mutation analysis, mutants have to be generated. To generate the mutants using the Major tool [17], I wrote an mml script describing the type of the mutants that are to be generated. There are two options for generating the mutants:

- Generate all the possible mutants
- Generate a set of mutants by writing an mml script (mml is the domain specific language of Major)

I chose the second approach because using all the mutants was time-consuming. The number of mutants generated using the first method is very large and the effort of checking all tests against all mutants can bring about huge computational costs. A statement in an mml script can be one of the following entities: Variable Definition, Invocation of Mutation Operator, Replacement Definition, and Definition of an own operator group or a comment.

The following sections show the syntax definition of a flat name in an mml script. A flat name identifies a certain entity within a Java program [17].

These are examples of valid flatnames of a class, package and a method in a class:

- “org.Plant”
- “org”
- “org.Plant@Growth”

It can also be extended to constructors and inner classes.

Following is an example of an MML script written for the experiment used in the system, which has a record of mutation for relational and conditional operators.

```
// Use sufficient replacements for ROR
BIN(>)->{>=,!=,FALSE};
BIN(<)->{<=,!=,FALSE};
BIN(>=)->{>,==,TRUE};
BIN(<=)->{<,==,TRUE};
BIN(==)->{<=,>=,FALSE,LHS,RHS};
BIN(!=)->{<,>,TRUE,LHS,RHS};

// Use sufficient replacements for COR
BIN(&&)->{==,LHS,RHS,FALSE};
BIN(||)->{!=,LHS,RHS,TRUE};

// Enable all operators
AOR;
LOR;
SOR;
COR;
ROR;
LVR;
ORU;
STD;

}

target0="org.Plant";
targetOp<target0>;
```

The last two lines of the script show the target for which the mutants are to be generated and mutation testing has to be performed. In the above script, the target class is *“Plant”* in the *“org”* package. After writing the mml script, which specifies the specific Java entity for which the mutants are to be generated, the mml file has to be compiled into an intermediate binary representation. This is done using the compiler (mmlc) provided in the Major mutation framework. The targets for which the mutants have to be generated are added to the mml script by a Python program. This program reads a file which contains the names of the classes for which test have to be generated and writes the valid flat name for the target class names into the mml script. The next step in the process involves using the mml script for generating the mutants.

There are two ways of performing mutation using the Major mutation framework, but I chose to integrate the process using Apache Ant. The main reason behind using the Apache Ant is its ease of use and using Apache Ant allowed me to use the same build.xml script to integrate Cobertura [9], which is the code coverage tool employed in the experiment. After integrating both Cobertura and the Major mutation framework, we can perform mutation and coverage analyses. The coverage that is obtained using cobertura when it is integrated with Apache Ant is statement/line coverage and branch coverage. Apache Ant is run from the command line and it is run according to the targets available in the build.xml file. I wrote a bash script for specifying the targets without retyping every command on the command line. The Bash script was written to compile the source code and generate the bytecode of the source from which mutation scores and coverage are calculated. All the bytecode of the source is stored in the “bin” directory (executables for major components). Initial source code for the target is stored in the “src” directory and the test cases are stored in the “test” directory. All the mutants that are generated during this step are stored in the “mutans.log” file. Apart from the mutants generated, there is a list of other important files used for analysis. Here is the list of the files generated during the process:

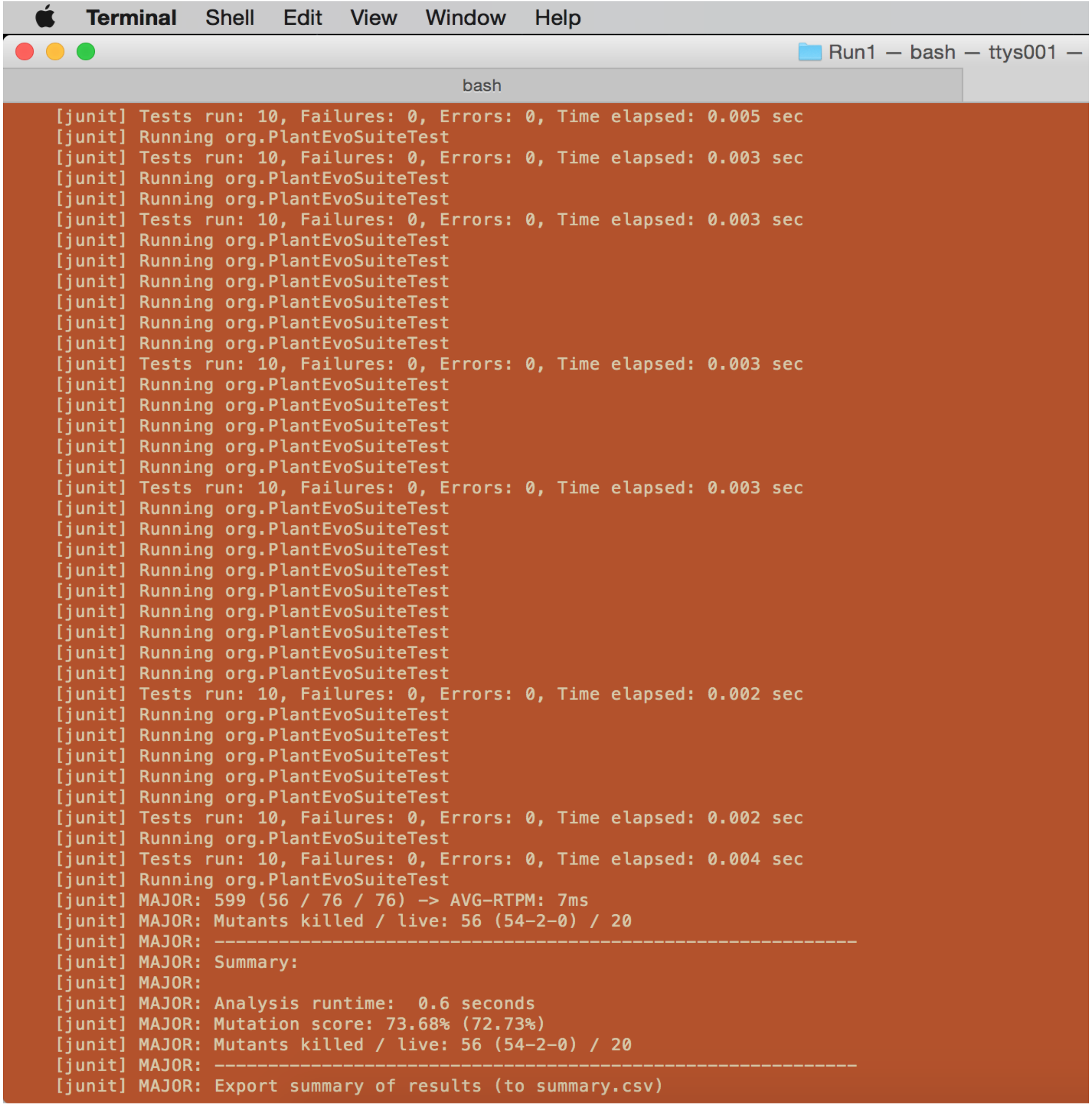
1. summary.csv: This file contains the total number of mutants generated during the process with the number of mutants covered and killed along with the overall time required to generate and analyze the mutants.

2. results.csv: This file also gives an overview of the mutation with more details on how mutants were killed.

3. killed.csv: This file stores the information about how each and every individual mutant was killed during the process (Exception/Failure/Timeout).

4. testmap.csv: This file stores the details of the test suites for which the mutants were generated.

Here is sample output that is obtained after running the mutation analysis on the source code.



```
    [junit] Tests run: 10, Failures: 0, Errors: 0, Time elapsed: 0.005 sec
    [junit] Running org.PlantEvoSuiteTest
    [junit] Tests run: 10, Failures: 0, Errors: 0, Time elapsed: 0.003 sec
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Tests run: 10, Failures: 0, Errors: 0, Time elapsed: 0.003 sec
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Tests run: 10, Failures: 0, Errors: 0, Time elapsed: 0.003 sec
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Tests run: 10, Failures: 0, Errors: 0, Time elapsed: 0.003 sec
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Tests run: 10, Failures: 0, Errors: 0, Time elapsed: 0.002 sec
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Running org.PlantEvoSuiteTest
    [junit] Tests run: 10, Failures: 0, Errors: 0, Time elapsed: 0.002 sec
    [junit] Running org.PlantEvoSuiteTest
    [junit] Tests run: 10, Failures: 0, Errors: 0, Time elapsed: 0.004 sec
    [junit] Running org.PlantEvoSuiteTest
    [junit] MAJOR: 599 (56 / 76 / 76) -> AVG-RTPM: 7ms
    [junit] MAJOR: Mutants killed / live: 56 (54-2-0) / 20
    [junit] MAJOR: -------------------------------------------------------------
    [junit] MAJOR: Summary:
    [junit] MAJOR:
    [junit] MAJOR: Analysis runtime:  0.6 seconds
    [junit] MAJOR: Mutation score: 73.68% (72.73%)
    [junit] MAJOR: Mutants killed / live: 56 (54-2-0) / 20
    [junit] MAJOR: -------------------------------------------------------------
    [junit] MAJOR: Export summary of results (to summary.csv)
```

Fig 3.2: figure shows the output obtained from running Major

56 - number of mutants killed

20 - number of mutants live

73.68% - mutation score

77 – total number of mutants

Above is a summary of the mutation analysis performed on a small file called “Plant” for the test suite generated by EVOSUITE. Numbers in the figure have certain roles associated with them.

Total number of mutant’s generated- Total number of mutants generated for the Plant class by Major mutation framework

Total number of mutants covered- Number of mutants that were discovered by the test suite created by the test generation tool.

Total number of mutants killed- all the mutants that the generated test suite could kill of all the covered mutants.

Coverage for the test suite is calculated along with the mutation score. Reports are generated using Cobertura [9]. All the classes for which the test suites are generated are the ones for which coverage is measured. Coverage types measured include branch coverage and line coverage. Statement coverage is also known as line coverage. Through line coverage we can distinguish the statements executed and where the code is not executed because of blockage. If a test suite finds out that the line coverage is 100% then it can be said that test suite has checked each and every line. The other coverage that is measured using Cobertura is the branch coverage. 100% branch coverage means every branch predicate has been set to true/false. Using Cobertura provides the summary of the reports in the form of an HTML files in the reports folder. After using Cobertura using the Apache Ant, all the reports related to coverage are generated in the reports folder. A description of the generated folders using Cobertura and Major is given below. There are five reports generated for the test subject. These include the following:

1. Cobertura-html : This folder contains the html page for all the files present in the source of the test subject.
2. Cobertura-summary-html : This folder contains an xml file which contains the overall summary of all the files of the source of the test subject.

3. Cobertura-xml : This folder contains an xml file which explains the detailed line coverage and branch coverage on each and every line covered in the source.
4. junit-html : This folder consists of the html pages describing the execution summary of the test cases.
5. junit-xml : This folder contains an xml file which gives the details of the time taken for the test case to run and it also contains the properties of the system under which the test case is run.

After the process of the mutation analysis, a mutant is in one of four forms. It can be killed by the test case (FAIL). The test case can cause an exception (EXC). A timeout is caused by the test case (TIME). The test case failed to the kill the mutant i.e. the mutant is live (LIVE). A mutant in the form FAIL, TIME, EXC is considered killed by the test suite. The sum of all these mutants are used in calculating the mutation score. Mutation score are evaluated using two different metrics. The metrics are taken from [4] and they are raw effectiveness measurement and the normalized effectiveness measurement.

- Raw-effectiveness measurement: The raw kill score is the number of mutants a test suite detected divided by the total number of non-equivalent mutants that were generated for the subject program under test.
- Normalized-effectiveness measurement: The normalized effectiveness score is the number of mutants a test suite detects divided by the total of non-equivalent mutants it covers.

In our case the total number of non-equivalent mutants covered are equal to the total number of mutants covered since it is very consuming to calculate the number of equivalent mutants. Equivalent mutants should be excluded because they cannot, by definition, be detected by a unit test. It is mentioned in [18] that the average time taken to identify an equivalent mutant is 15 minutes. For this reason, the equivalent mutants are not calculated. A partial examination suggests that the number of equivalent mutants are relatively small so might not affect mutation scores too much.

4. Results

This chapter focuses on the results from the two tools on the ten test subjects and the process used in evaluating the two tools. Scripts written in the R programming language were used to analyze the raw data for mutation scores, coverage scores and the size of the test suites.

	Raw Kill Score	Normalized Kill Score	Branch coverage	Line coverage	Test Suite size
ArrayIntList	32.6	36.58	81.88	78.78	10.2
AVLTree	42.45	42.45	99.34	99.50	12.2
MyHashMap	55.09	55.09	93.61	98.19	15.4
Heap	48.037	48.037	100.00	100.00	5.6
List	58.94	58.94	100.00	100.00	8.4
Rational	78.33	78.33	90.00	100.00	11.2
WeightedGraph	42.81	52.79	81.30	79.63	26.4
Plant	71.42	72.36	100.00	94.73	9.6
BasePlusCommissionEmployee	67.14	67.14	80.00	93.33	4.2
Time2	61.98	63.23	90.90	96.87	11.7

Table 4.1: Shows the results of each class for test cases generated by EVOSUITE

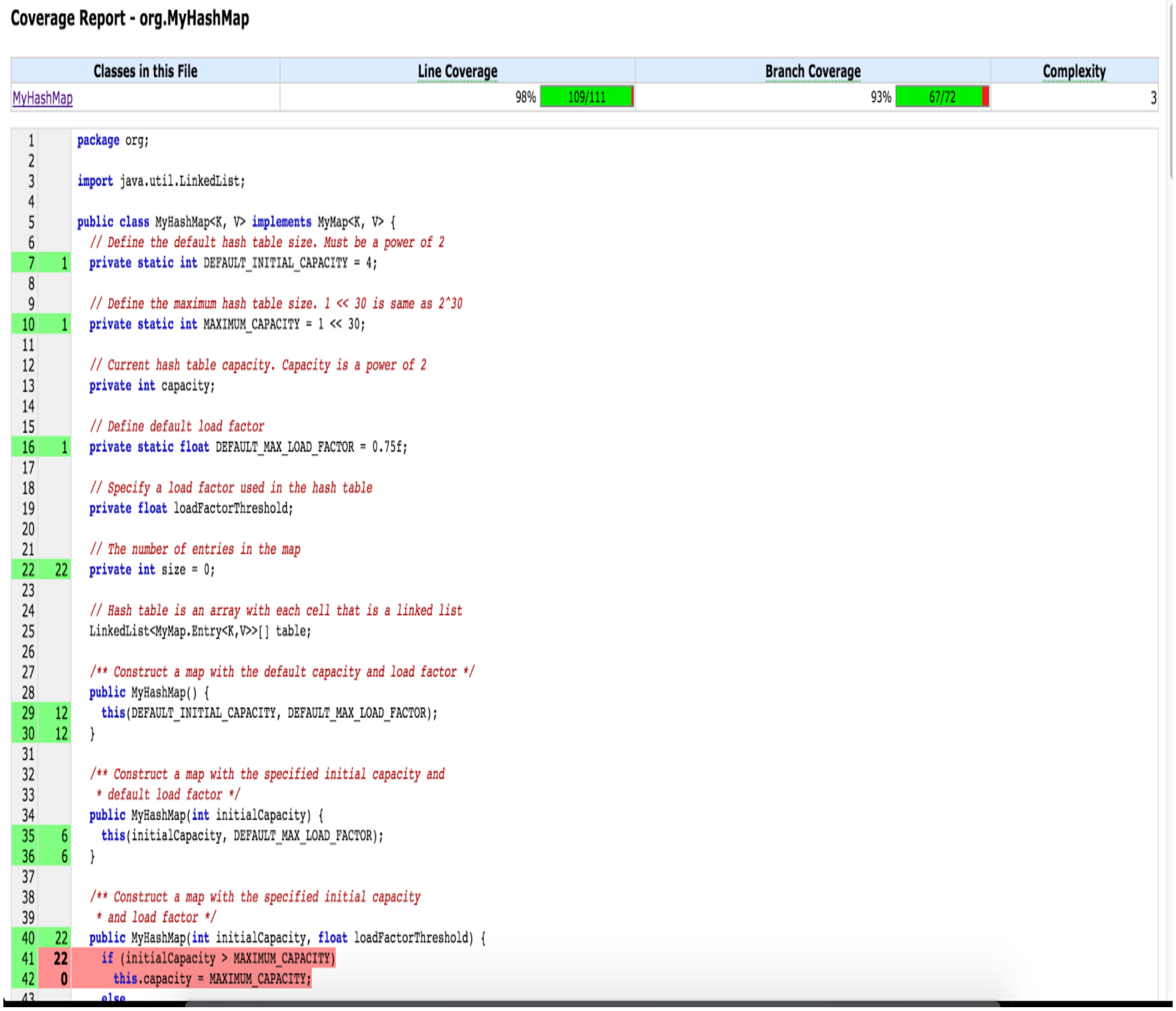
	Raw Kill Score	Normalized Kill Score	Branch coverage	Line coverage	Test Suite size
ArrayIntList	55.65	61.93	84.88	81.88	9778.8
AVLTree	40.00	47.65	63.05	66.22	1479.2
MyHashMap	71.92	74.38	89.16	91.53	1086.2
HeapSort	81.86	81.86	100.00	100.00	2239.8
List	63.15	63.15	100.00	100.00	3484.6
Math4j	88.00	88.00	90.00	100.00	694.2
MinimumSpanningTree	33.28	46.20	62.71	74.59	1015.2
PlantMutants	80.00	84.38	91.66	94.73	4927.4
BasePlusCommissionEmployee	57.14	57.14	50.00	83.33	4857.6
Time2	71.28	72.72	77.27	99.37	15280

Table 4.2: Shows the results of each class for test cases generated by Tpalus

4.1 Coverage Scores

The first part of the results concentrates on the coverage scores of both the tools. The coverage of both tools are divided into line coverage and branch coverage. Branch coverage is considered a more complete measure than line coverage because a test suite with 100% branch coverage will have 100% line coverage and a test suite with 100% line coverage usually does not mean 100% branch coverage.

It is clear that line coverage is better for the tests generated using EVOSUITE more than Tpalus. But the difference in the line coverage is not so great and in most cases it is less than 5%. There are three test subjects where Tpalus performed better in terms of line coverage. Cobertura can also be enabled to get a detailed report of coverage. Below is an image generated by Cobertura, which shows coverage information on each and every line of the source code.



Coverage Report - org.MyHashMap

Classes in this File	Line Coverage	Branch Coverage	Complexity
MyHashMap	98% 109/111	93% 67/72	3

```
    package org;

    import java.util.LinkedList;

    public class MyHashMap<K, V> implements MyMap<K, V> {
      // Define the default hash table size. Must be a power of 2
 1    private static int DEFAULT_INITIAL_CAPACITY = 4;

      // Define the maximum hash table size. 1 << 30 is same as 2^30
 1    private static int MAXIMUM_CAPACITY = 1 << 30;

      // Current hash table capacity. Capacity is a power of 2
      private int capacity;

      // Define default load factor
 1    private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;

      // Specify a load factor used in the hash table
      private float loadFactorThreshold;

      // The number of entries in the map
22    private int size = 0;

      // Hash table is an array with each cell that is a linked list
      LinkedList<MyMap.Entry<K,V>>[] table;

      /** Construct a map with the default capacity and load factor */
      public MyHashMap() {
12      this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
12    }

      /** Construct a map with the specified initial capacity and
       * default load factor */
      public MyHashMap(int initialCapacity) {
 6      this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
 6    }

      /** Construct a map with the specified initial capacity
       * and load factor */
22    public MyHashMap(int initialCapacity, float loadFactorThreshold) {
22      if (initialCapacity > MAXIMUM_CAPACITY)
 0        this.capacity = MAXIMUM_CAPACITY;
        else
```

Fig 4.1: This figure is an example of a coverage report on a class file generated by Cobertura.

The green part of the code indicates that the test case generated covers that part of the code and the red color indicates the test case generated was not able to cover that part of the source code.

Test cases, which have more coverage, should have a greater chance of finding faults in the source code. This is the reason why coverage has been one of the metrics for evaluating the effectiveness of test cases.

Figure 4.2 shows the comparison between EVOSUITE and Tpalus on all the test subjects for the line coverage.

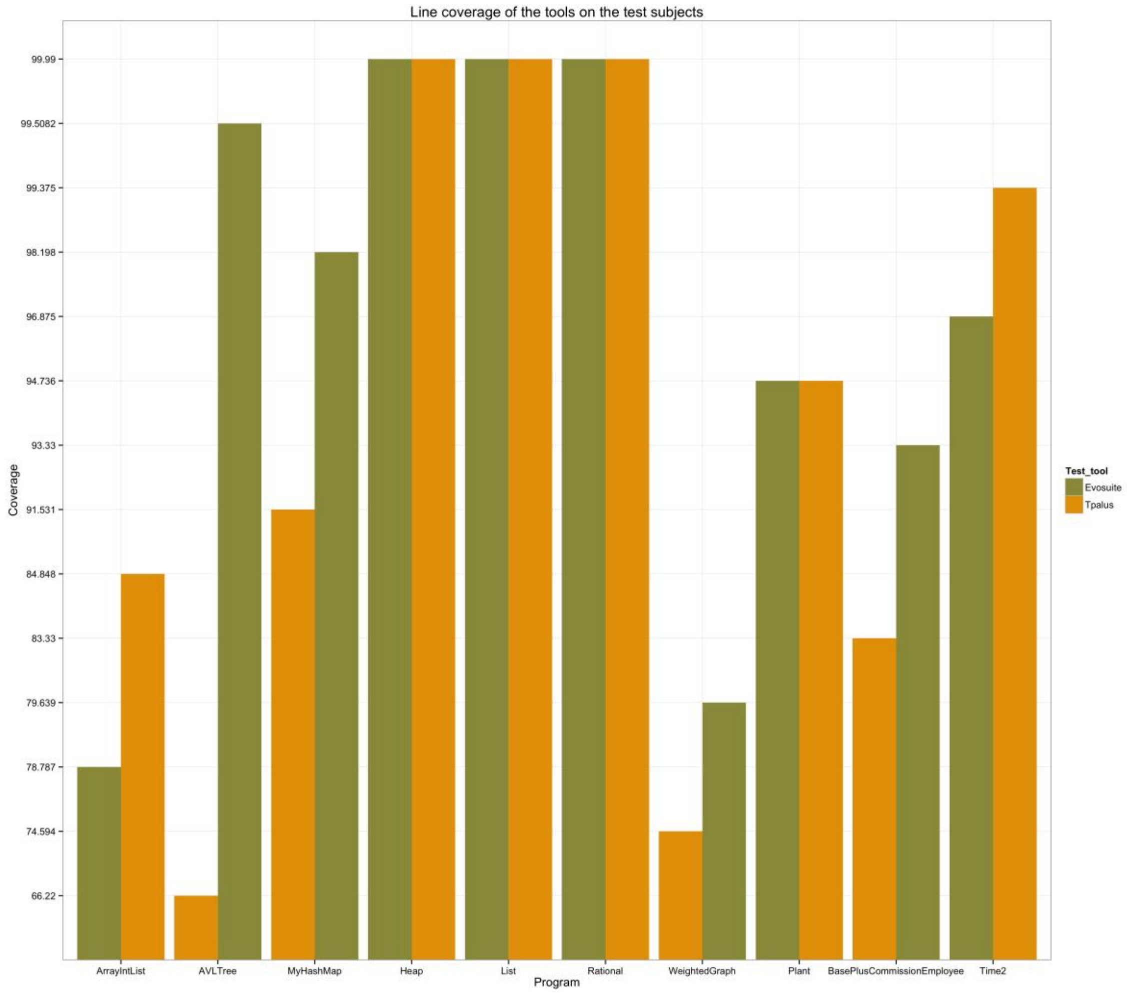


Fig 4.2: Comparison of line coverage between EVOSUITE and Tpalus

The next figure is the comparison of EVOSUITE and Tpalus for branch coverage on the ten test subjects.

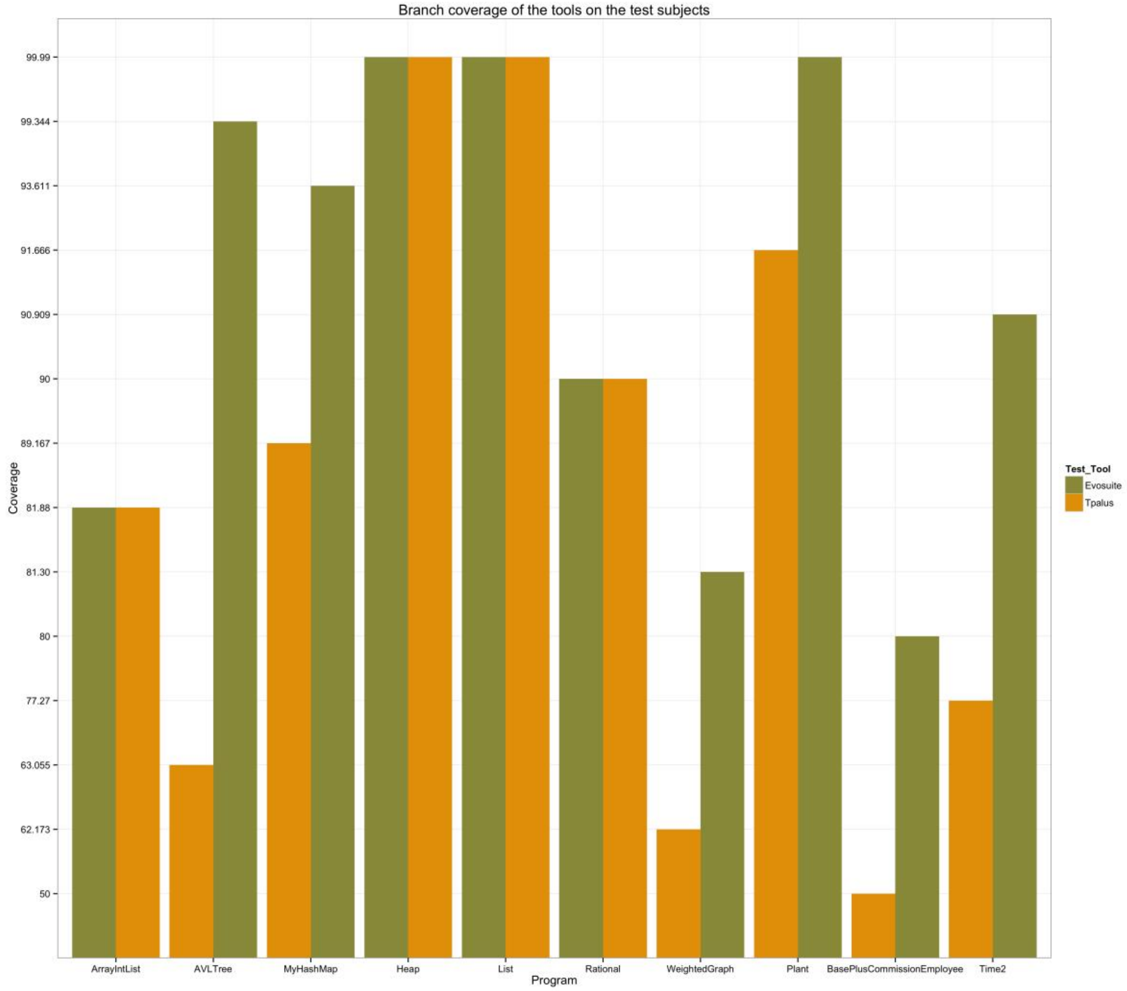


Fig 4.3: Comparison of Branch coverage between EVOSUITE and Tpalus

From comparison it is clear that both EVOSUITE and Tpalus have similar coverage both in line coverage and branch coverage.

4.2 Test Suite Size

The next metric that is compared between the two tools is the number of test cases generated. The number of test cases generated is taken as a mean of five runs. A test suite that has a minimum number of test cases is easier to manage and understand than a test suite with several thousand of them.

In the experiments performed, EVOSUITE produced relatively very few test cases compared to Tpalus and the test cases generated by EVOSUITE were very easy to understand, as they were short. Tpalus produces a very large amount of test cases and they are not very readable when compared to EVOSUITE. Below is a bar graph, which shows the comparison of the number of test cases generated by EVOSUITE and Tpalus.

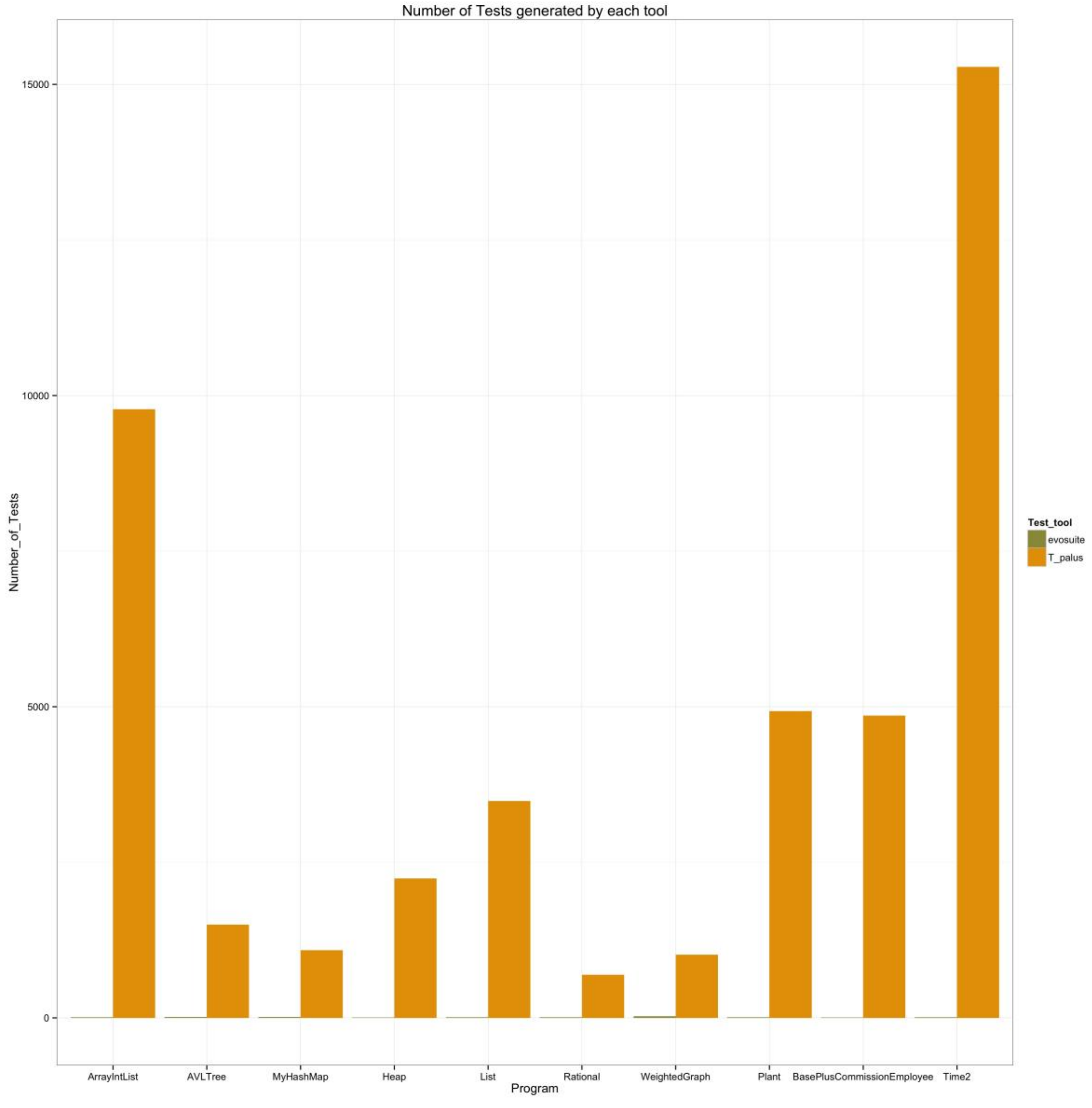


Fig 4.4: The bar graph shows the total test cases generated by EVOSUITE compared to Tplaus

We can hardly observe the green color in the bar graph, which represents EVOSUITE and the test cases it generated. This shows that EVOSUITE performs test suite minimization very well.

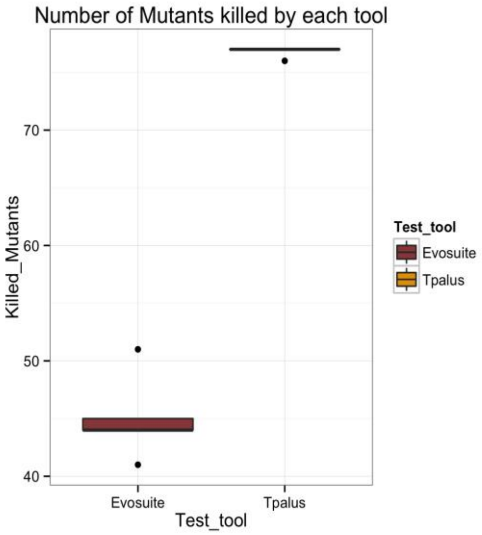
4.3 Mutation Analysis

Mutation testing is the most important part of the comparison and is the most effective metric of the comparison. It is implemented using the Major tool. As before an average of the five runs was taken. If the test suite fails when it is run on a given mutant, we say that the suite kills that mutant. A test suite's mutant coverage is then the fraction of non-equivalent mutants that it kills. [19] Work has also shown that if a test suite detects a large number of simple faults, caused by a single incorrect line of source code, it will detect a large number of harder, multi-line faults. The work thus suggests that the mutant detection rate of a suite is a fairly good measurement of its fault detection ability.

Class Name	Total Generated Mutants
ArrayIntList	138
AVLTree	122
MyHashMap	151
Heap	107
List	19
Rational	120
WeightedGraph	128
Plant	77
BasePlusCommisionEmployee	14
Time2	101

Table 4.3: Indicating the total number of mutants generated for each class

Below are the box plots for the all the test subjects. The box plots show the number of mutants killed by each tool.



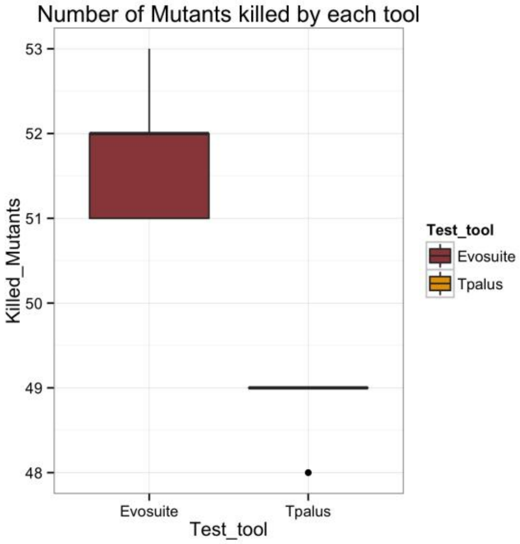
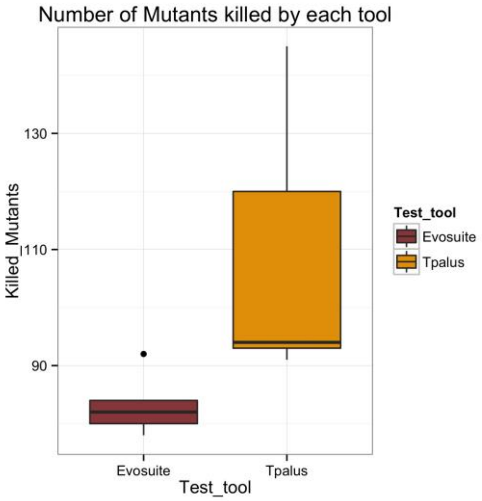


Fig 4.5: Mutation scores on ArrayIntList

Fig 4.6: Mutation scores on AVLTree



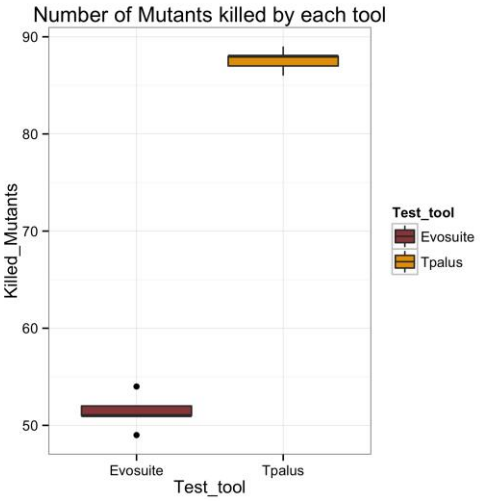


Fig 4.7: Mutation scores on MyHashMap

Fig 4.8: Mutation scores on Heap

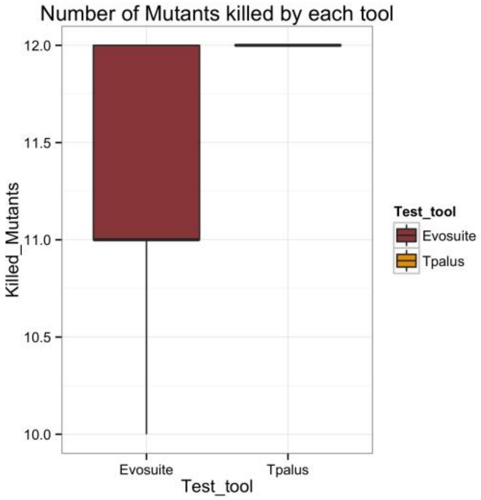


Fig 4.9: Mutation scores on List

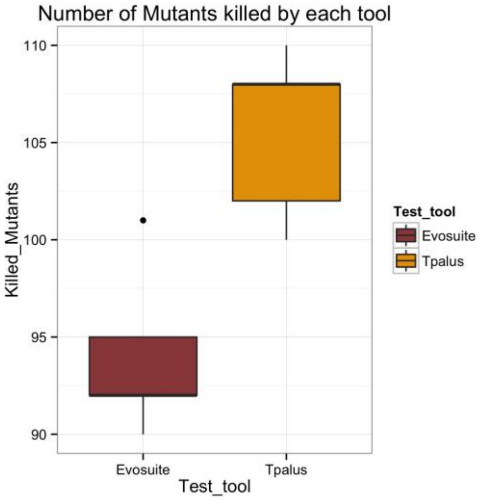


Fig 4.10: Mutation scores on Rational

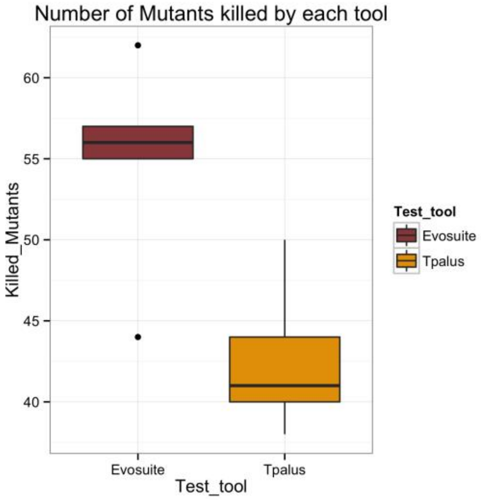


Fig 4.11: Mutation scores on WeightedGraph

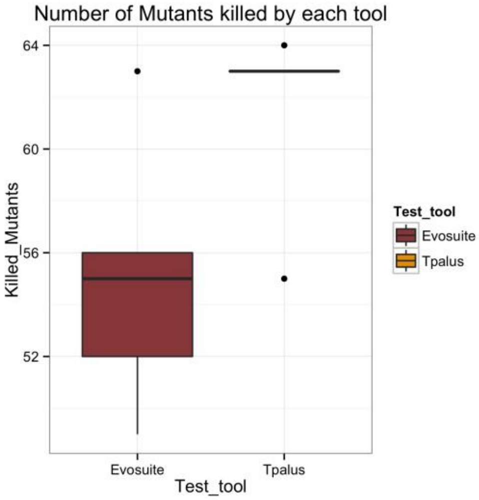


Fig 4.12: Mutation scores on Plant

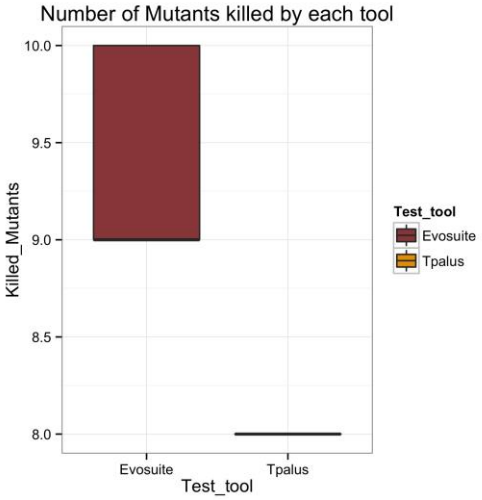


Fig 4.13: Mutation Scores
On BasePlusCommissionEmployee

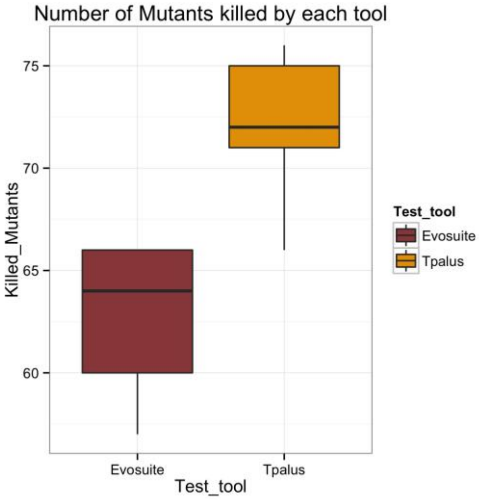


Fig 4.14: Mutation Scores on Time2

In seven cases, Tpalus killed more mutants. In three cases, EVOSUITE killed more mutants.

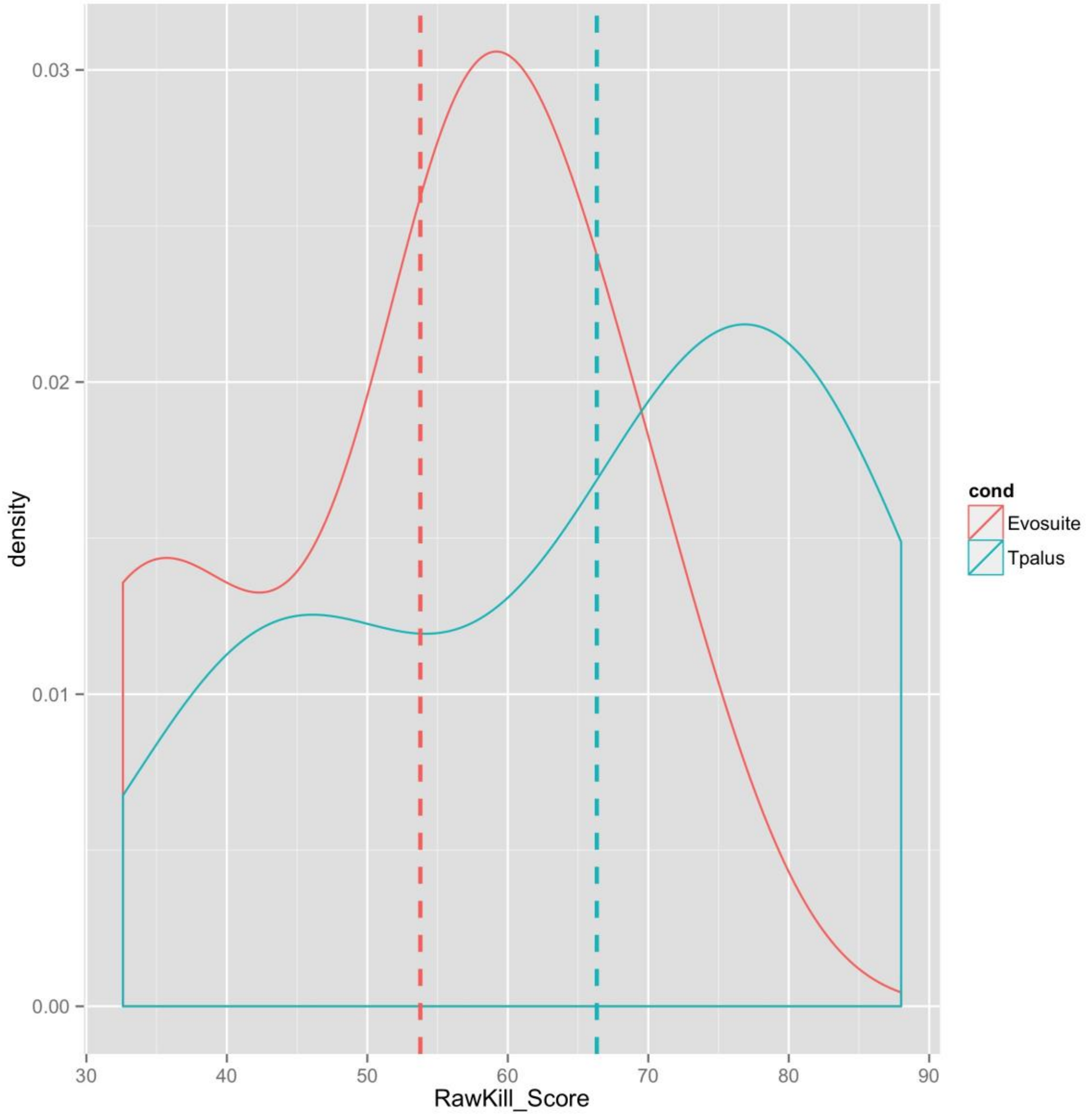


Fig 4.15: A density plot of raw kill scores for EVOSUITE and Tpalus

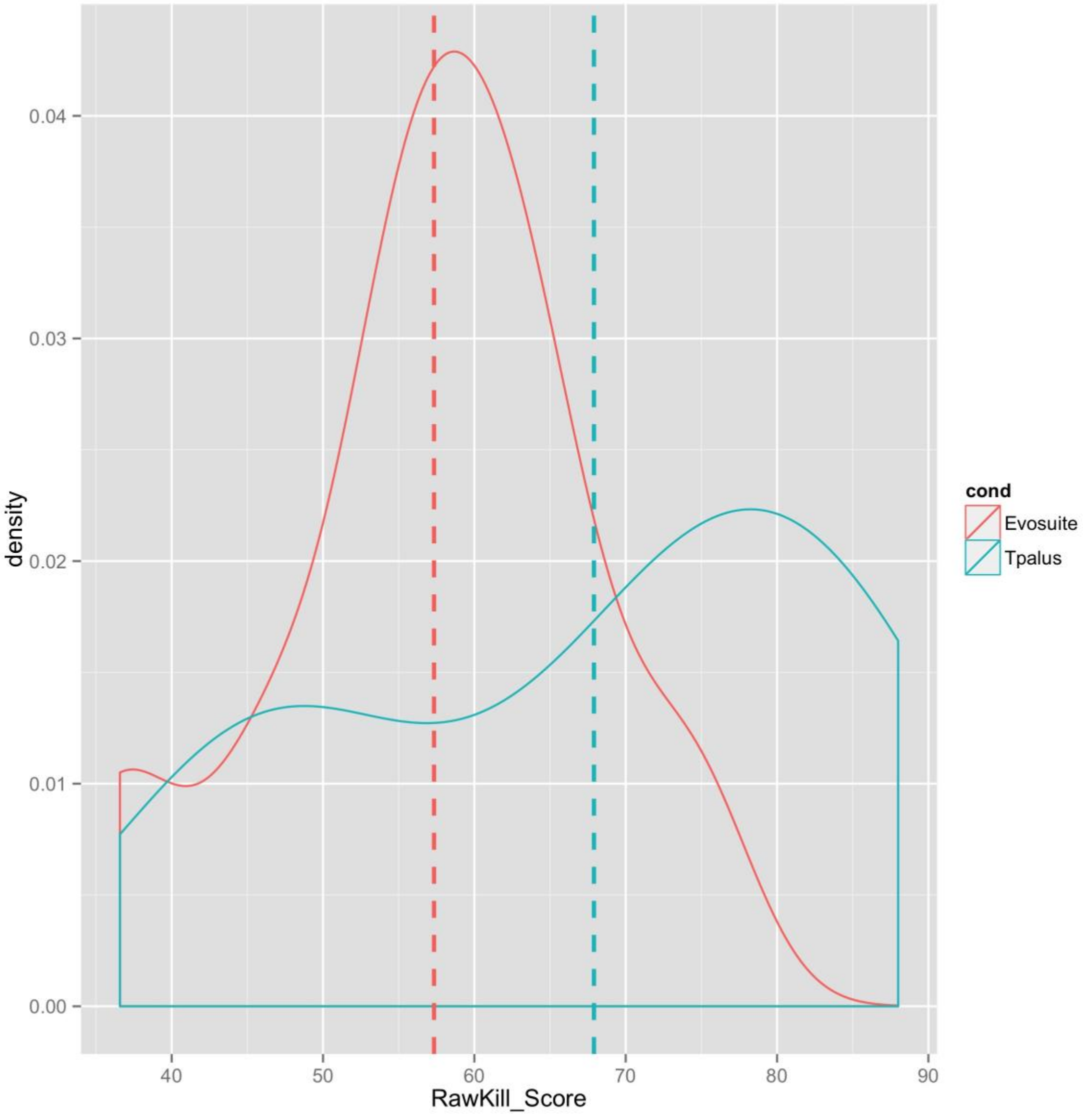


Fig 4.16: Density plot of normalized kill scores for EVOSUITE and Tpalus

From the density plots produced using “R” we can compare the raw kill scores and normalized kill scores. Both plots show that the density is higher for Tpalus in terms of scores. The dotted line indicates the mean of the scores.

4.4 Comparison

The stage is now set for a comparison of the two tools. Coverage scores of the two tools was very high and it was only in two cases did Tpalus have better line coverage and none in terms of branch coverage. In 4 cases EVOSUITE and Tpalus had the same branch coverage. That means EVOSUITE has a slightly better advantage in terms of coverage than Tpalus. But it is not always a good idea to rely on coverage scores, A test suite with better coverage does not mean that the test suite will actually find more faults [4].

Now the second metric is size of the test suite. There are two things that were considered in this aspect, how readable a test case is and the size of the test suite.

a. The size of the test suite generated by EVOSUITE is very small when compared to Tpalus. EVOSUITE never produced more than 30 test cases on all the test subjects, which gives the reader a better chance of understanding the test cases and of extending them if desired.
b. The test cases generated by EVOSUITE are small and easily understandable. Not only is the test suite size minimum, but also the test cases are also small. Along with the test generation, comments are produced indicating the goals that were covered during the process.

These two considerations suggest that EVOSUITE has an advantage over Tpalus.

From the mutation scores it is seen that in 7 cases Tpalus had an advantage over EVOSUITE in terms of killing the mutants. It is important to consider the cases where EVOSUITE is failing. The higher mutation score indicates that Tpalus was more successful in identifying the faulty versions of the program than EVOSUITE. There were several areas where EVOSUITE was failing to detect a mutant while Tpalus did so.
Below is a mutant in the *"Plant"* class, which Tpalus was able to kill, and EVOSUITE could not.

```
28:COR:moisture:TRUE:org.Plant@addWater:41:moisture |==> false
```

The line underlined in the source code below is the mutant, which is shown before and after the mutation.

```
package org;
public class Plant {
private int size; // current size of plant in cms
private boolean moisture; // is the earth wet or dry
private boolean alive; // alive or dead
public Plant () {
    alive = true; size = 1; // the size of a seed (1cm)
    moisture = true; // earth is watered
} //end of constructor
public Plant (int initSize) {
    alive = true; size = initSize; // from 1cm to less than 20 cms
    moisture = true; // earth is watered
} //end of constructor
private void alterSize(int amount) {
    if (isAlive()) size = size + amount;
    if (size <= 0) die();
    if (size >= 20) die(); // plant dies when it has grown to 20cms
} // end of alterSize

private void die( ) { // plant is dead
    alive = false;
```

```
   size = 0;
} // end of die
public int getSize( ) {
   return size;
} // end of getSize
public boolean isAlive( ) {
   return alive;
} // end of isAlive
public void addWater() {
if (moisture) alterSize(-2); // overwatering ->    if (false) alterSize(-2);
if (!moisture) moisture = true;
} // end of addWater
public void addFertilizer(int amount) { // amount between 1..5
   if (moisture) {
      alterSize(amount);
      moisture = false; //water used up growing
   } // else fertilizer is blown away by wind
} // end of addFertilizer
public void animalBite(int amount) { // animals bite up to 10 cms
   if (amount<0) throw new IllegalArgumentException("neg. amount");
   alterSize(-amount);
} // end of animalBite
public void talkToPlant(String s) {
        try {
                String greeting = s;

                 int growth = 0;

                 if (greeting.startsWith("please")) growth = 1;

                 if (greeting.startsWith("please") &&
greeting.endsWith("please")) growth = 2;

                 if (growth==1)alterSize(1);

                 else if (growth==2)alterSize(2);

                 else alterSize(-1); // saying things that are not nice

        } catch (Exception e) {

                // TODO Auto-generated catch block

                System.out.println("it generates a null point exception");

        }

} // end of talkToPlant

}
```

All the test cases generated by EVOSUITE could not recognize the mutant, but a test case generated by Tpalus could kill mutant. Below is the test case that killed the above mutant.

```
public void test7() throws Throwable

{

 org.Plant var0 = new org.Plant();

   boolean var1 = var0.isAlive();

   int var2 = var0.getSize();

   java.lang.Integer var3 = new java.lang.Integer(5);

   var0.addFertilizer((int)var3);
   int var5 = var0.getSize();
   var0.addWater();
   java.lang.Integer var7 = new java.lang.Integer(5);
   var0.addFertilizer((int)var7);
   int var9 = var0.getSize();
   var0.addWater();
   var0.addWater();
   var0.addFertilizer((int)var9);
   // Regression assertion (captures the current behavior of the code)
   assertTrue(var1 == true);
    // Regression assertion (captures the current behavior of the code)
   assertTrue(var2 == 1);
   // Regression assertion (captures the current behavior of the code)
   assertTrue(var5 == 6);
   // Regression assertion (captures the current behavior of the code)
   assertTrue(var9 == 11);
 }
```

This is one of the areas where EVOSUITE was failing to kill the mutants. The reason was that Tpalus was successful in capturing the behavior of the code better than EVOSUITE. EVOSUITE generally aims at generating the test cases by maximizing the branch coverage, which is its primary criterion for test generation. But in cases like this EVOSUITE was able to cover the area where the mutant was generated, but could not generate a test case that could capture the behavior change due to the mutation. As seen from the coverage statistics EVOSUITE was able to get a good coverage score on all the ten test subjects. EVOSUITE generates assertions (chapter 2.2) based on the difference captured by the observer between the original and the mutated program. This technique does not guarantee to capture the mutation applied by major mutation framework. As seen in the comments, Tpalus checks for regression testing, capturing the change in the existing code and it was successful in doing so. Tpalus generates the parameter values for a test case using the static and dynamic information and it uses the regression oracles to check if the test case detects the change in the original program. Even though the Tpalus mutation score is not 100 percent, the way it catches behavior allows it to capture more mutants and kill them.

Here is another scenario where Tpalus gains over EVOSUITE in terms of killing mutants. Following is a class in the *“HeapSort”* jar

```
package org;
public class Heap<E extends Comparable<E>> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<E>();
  /** Create a default heap */
  public Heap() {
  }
  /** Create a heap from an array of objects */
  public Heap(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      add(objects[i]);
  }
  /** Add a new object into the heap */
  public void add(E newObject) {
    list.add(newObject); // Append to the heap
    int currentIndex = list.size() - 1; // The index of the last node

    while (currentIndex > 0) {
      int parentIndex = (currentIndex - 1) / 2;
      // Swap if the current object is greater than its parent
      if (list.get(currentIndex).compareTo(
          list.get(parentIndex)) > 0) {
        E temp = list.get(currentIndex);
        list.set(currentIndex, list.get(parentIndex));
        list.set(parentIndex, temp);
      }
      else
        break; // the tree is a heap now
      currentIndex = parentIndex;
    }
  }
  /** Remove the root from the heap */
  public E remove() {
    if (list.size() == 0) return null;
    E removedObject = list.get(0);
    list.set(0, list.get(list.size() - 1));
    list.remove(list.size() - 1);
    int currentIndex = 0;
    while (currentIndex < list.size()) {
      int leftChildIndex = 2 * currentIndex + 1;
      int rightChildIndex = 2 * currentIndex + 2;
      // Find the maximum between two children
      if (leftChildIndex >= list.size()) break; // The tree is a heap
      int maxIndex = leftChildIndex;
      if (rightChildIndex < list.size()) {
        if (list.get(maxIndex).compareTo(
            list.get(rightChildIndex)) < 0) {
          maxIndex = rightChildIndex;
        }
      }
      // Swap if the current node is less than the maximum
      if (list.get(currentIndex).compareTo(
          list.get(maxIndex)) < 0)
  E temp = list.get(maxIndex);
        list.set(maxIndex, list.get(currentIndex));
        list.set(currentIndex, temp);
        currentIndex = maxIndex;
      }
      else
        break; // The tree is a heap
    }
return removedObject;
  }
  /** Get the number of nodes in the tree */
  public int getSize() {
    return list.size();
  }
}
```

In the above example, EVOSUITE failed to generate a test case that would check the sorting in the Heap class, since the sorting is not specified as a function. Sorting is done as an element is added or removed. Tpalus was able to capture the behavior and create a more effective test case allowing Tpalus to kill more mutants.

```
/*
 * This file was automatically generated by EVOSUITE
 * Fri Jun 12 19:47:41 CDT 2015
 */
package org;
import static org.junit.Assert.*;
import org.junit.Test;
import org.Heap;
public class HeapEVOSUITETest {
  //Test case number: 0
  /*
   * 3 covered goals:
   * 1 org.Heap.remove()Ljava/lang/Comparable;: I84 Branch 6 IF_ICMPGE
L53 - true
   * 2 org.Heap.remove()Ljava/lang/Comparable;: I102 Branch 7 IFGE L54 -
false
   * 3 org.Heap.remove()Ljava/lang/Comparable;: I124 Branch 8 IFGE L61 -
true
   */
 @Test
 public void test0()  throws Throwable  {
    String[] stringArray0 = new String[7];
    stringArray0[0] = "*`~Rm%4skRowf%&";
    stringArray0[2] = "*`~Rm%4skRowf%&";
    stringArray0[3] = "*`~Rm%4skRowf%&";
    stringArray0[4] = "*`~Rm%4skRowf%&";
    stringArray0[6] = "*`~Rm%4skRowf%&";
    stringArray0[1] = ".";
    stringArray0[5] = "Y";
    Heap<String> heap0 = new Heap<String>(stringArray0);
    String string0 = heap0.remove();
    assertEquals(6, heap0.getSize());
 }
 //Test case number: 1
 /*
  * 9 covered goals:
  * 1 org.Heap.add(Ljava/lang/Comparable;)V: I43 Branch 2 IFLE L24 - true
  * 2 org.Heap.add(Ljava/lang/Comparable;)V: I43 Branch 2 IFLE L24 - false
  * 3 org.Heap.add(Ljava/lang/Comparable;)V: I79 Branch 3 IFGT L21 - true
  * 4 org.Heap.remove()Ljava/lang/Comparable;: I72 Branch 5 IF_ICMPLT L51 - true
  * 5 org.Heap.remove()Ljava/lang/Comparable;: I72 Branch 5 IF_ICMPLT L51 - false
  * 6 org.Heap.remove()Ljava/lang/Comparable;: I84 Branch 6 IF_ICMPGE L53 - false
  * 7 org.Heap.remove()Ljava/lang/Comparable;: I102 Branch 7 IFGE L54 - true
  * 8 org.Heap.remove()Ljava/lang/Comparable;: I124 Branch 8 IFGE L61 - false
  * 9 org.Heap.remove()Ljava/lang/Comparable;: I163 Branch 9 IF_ICMPLT
L46 - true
  */
 @Test
 public void test1()  throws Throwable  {
    String[] stringArray0 = new String[4];
    stringArray0[1] = "";
    stringArray0[2] = "";
    stringArray0[0] = "gA_07M$D";
    stringArray0[3] = "!\"-]";
    Heap<String> heap0 = new Heap<String>(stringArray0);
    String string0 = heap0.remove();
    assertEquals(3, heap0.getSize());
 }
 //Test case number: 2
 /*
  * 1 covered goal:
  * 1 org.Heap.remove()Ljava/lang/Comparable;: I5 Branch 4 IFNE L39 -
false
  */
 @Test
 public void test2()  throws Throwable  {
    Heap<String> heap0 = new Heap<String>();
    String string0 = heap0.remove();
    assertNull(string0);
 }
 //Test case number: 3
 /*
  * 5 covered goals:
  * 1 org.Heap.<init>([Ljava/lang/Comparable;)V: I31 Branch 1 IF_ICMPLT L12 - true
  * 2 org.Heap.<init>([Ljava/lang/Comparable;)V: I31 Branch 1 IF_ICMPLT L12 - false
  * 3 org.Heap.add(Ljava/lang/Comparable;)V: I79 Branch 3 IFGT L21 - false
  * 4 org.Heap.remove()Ljava/lang/Comparable;: I5 Branch 4 IFNE L39 - true
  * 5 org.Heap.remove()Ljava/lang/Comparable;: I163 Branch 9 IF_ICMPLT L46 - false
  */
 @Test
 public void test3()  throws Throwable  {
    Integer[] integerArray0 = new Integer[1];
    Heap<Integer> heap0 = new Heap<Integer>(integerArray0);
    assertEquals(1, heap0.getSize());
    Integer integer0 = heap0.remove();
    assertEquals(0, heap0.getSize());
 }
 //Test case number: 4
 /*
  * 2 covered goals:
  * 1 org.Heap.<init>()V: root-Branch
  * 2 org.Heap.getSize()I: root-Branch
  */
 @Test
 public void test4()  throws Throwable  {
    Heap<String> heap0 = new Heap<String>();
    int int0 = heap0.getSize();
    assertEquals(0, int0);
 }
}
```

Above is the test suite for “*Heap*” class created by EVOSUITE, which clearly shows that EVOSUITE could not capture the behavior of the code, as sorting is not specified as a function. Even though EVOSUITE is covering that part of the code where the mutant is generated, it could not detect the behavior. This is one of the drawbacks of the EVOSUITE.

Below is a test case generated by Tpalus, which was successful in capturing the behavior of the “*Heap”* class.

```
public void test256() throws Throwable {
   org.Heap var0 = new org.Heap();
   int var1 = var0.getSize();
   org.Heap var2 = new org.Heap();
   java.lang.Long var3 = new java.lang.Long((-1L));
   var2.add((java.lang.Comparable)var3);
   java.lang.Long var5 = new java.lang.Long(10L);
   var2.add((java.lang.Comparable)var5);
   org.Heap var7 = new org.Heap();
   org.Heap var8 = new org.Heap();
   java.lang.Long var9 = new java.lang.Long(0L);
   var8.add((java.lang.Comparable)var9);
   var7.add((java.lang.Comparable)var9);
   var2.add((java.lang.Comparable)var9);
   org.Heap var13 = new org.Heap();
   org.Heap var14 = new org.Heap();
   java.lang.Long var15 = new java.lang.Long((-1L));
   var14.add((java.lang.Comparable)var15);
   var13.add((java.lang.Comparable)var15);
```

```
var2.add((java.lang.Comparable)var15);
org.Heap var19 = new org.Heap();
org.Heap var20 = new org.Heap();
java.lang.Long var21 = new java.lang.Long(0L);
var20.add((java.lang.Comparable)var21);
var19.add((java.lang.Comparable)var21);
org.Heap var24 = new org.Heap();
org.Heap var25 = new org.Heap();
org.Heap var26 = new org.Heap();
java.lang.Long var27 = new java.lang.Long(1L);
var26.add((java.lang.Comparable)var27);
var25.add((java.lang.Comparable)var27);
var24.add((java.lang.Comparable)var27);
var19.add((java.lang.Comparable)var27);
var2.add((java.lang.Comparable)var27);
org.Heap var33 = new org.Heap();
java.lang.Long var34 = new java.lang.Long(1L);
var33.add((java.lang.Comparable)var34);
var2.add((java.lang.Comparable)var34);
org.Heap var37 = new org.Heap();
org.Heap var38 = new org.Heap();
org.Heap var39 = new org.Heap();
java.lang.Long var40 = new java.lang.Long(0L);
var39.add((java.lang.Comparable)var40);
var38.add((java.lang.Comparable)var40);
var37.add((java.lang.Comparable)var40);
var2.add((java.lang.Comparable)var40);
org.Heap var45 = new org.Heap();
java.lang.Long var46 = new java.lang.Long(0L);
var45.add((java.lang.Comparable)var46);
```

```
org.Heap var48 = new org.Heap();
java.lang.Long var49 = new java.lang.Long((-1L));
var48.add((java.lang.Comparable)var49);
var45.add((java.lang.Comparable)var49);
org.Heap var52 = new org.Heap();
org.Heap var53 = new org.Heap();
java.lang.Long var54 = new java.lang.Long(0L);
var53.add((java.lang.Comparable)var54);
var52.add((java.lang.Comparable)var54);
var45.add((java.lang.Comparable)var54);
var2.add((java.lang.Comparable)var54);
org.Heap var59 = new org.Heap();
java.lang.Long var60 = new java.lang.Long(0L);
var59.add((java.lang.Comparable)var60);
org.Heap var62 = new org.Heap();
java.lang.Long var63 = new java.lang.Long((-1L));
var62.add((java.lang.Comparable)var63);
var59.add((java.lang.Comparable)var63);
org.Heap var66 = new org.Heap();
org.Heap var67 = new org.Heap();
java.lang.Long var68 = new java.lang.Long(0L);
var67.add((java.lang.Comparable)var68);
var66.add((java.lang.Comparable)var68);
var59.add((java.lang.Comparable)var68);
var2.add((java.lang.Comparable)var68);
org.Heap var73 = new org.Heap();
java.lang.Long var74 = new java.lang.Long((-1L));
var73.add((java.lang.Comparable)var74);
var2.add((java.lang.Comparable)var74);
org.Heap var77 = new org.Heap();
```

```
java.lang.Long var78 = new java.lang.Long(1L);
var77.add((java.lang.Comparable)var78);
var2.add((java.lang.Comparable)var78);
org.Heap var81 = new org.Heap();
java.lang.Long var82 = new java.lang.Long(1L);
var81.add((java.lang.Comparable)var82);
var2.add((java.lang.Comparable)var82);
org.Heap var85 = new org.Heap();
java.lang.Long var86 = new java.lang.Long((-1L));
var85.add((java.lang.Comparable)var86);
var2.add((java.lang.Comparable)var86);
java.lang.Comparable var89 = var2.remove();
java.lang.Comparable var90 = var2.remove();
java.lang.Comparable var91 = var2.remove();
java.lang.Comparable var92 = var2.remove();
java.lang.Comparable var93 = var2.remove();
java.lang.Comparable var94 = var2.remove();
java.lang.Comparable var95 = var2.remove();
java.lang.Comparable var96 = var2.remove();
var0.add(var96);
// Regression assertion (captures the current behavior of the code)
assertTrue(var1 == 0);
// Regression assertion (captures the current behavior of the code)
assertTrue("'" + var89 + "' != '" + 10L+ "'", var89.equals(10L));
// Regression assertion (captures the current behavior of the code)
assertTrue("'" + var90 + "' != '" + 1L+ "'", var90.equals(1L));
// Regression assertion (captures the current behavior of the code)
assertTrue("'" + var91 + "' != '" + 1L+ "'", var91.equals(1L));
// Regression assertion (captures the current behavior of the code)
assertTrue("'" + var92 + "' != '" + 1L+ "'", var92.equals(1L))
```

```
    // Regression assertion (captures the current behavior of the code)
    assertTrue("'" + var93 + "' != '" + 1L+ "'", var93.equals(1L))
    // Regression assertion (captures the current behavior of the code)
    assertTrue("'" + var94 + "' != '" + 0L+ "'", var94.equals(0L));
    // Regression assertion (captures the current behavior of the code)
    assertTrue("'" + var95 + "' != '" + 0L+ "'", var95.equals(0L));
    // Regression assertion (captures the current behavior of the code)
    assertTrue("'" + var96 + "' != '" + 0L+ "'", var96.equals(0L));
  }
```

This test case captures the behavior of the Heap class that was missing from the EVOSUITE test case. Results also indicated that there were three classes where EVOSUITE had a better mutation score than Tpalus.

Below is one of the classes in the experiment “*WeightedGraph*” in “*MinimumSpanningTree*” jar. We see how EVOSUITE has a better advantage over Tpalus. Below is a snippet of the class on which mutation is done. The mutant is underlined.

```
  public MST getMinimumSpanningTree(int startingVertex) {
    List<Integer> T = new ArrayList<Integer>();
    // T initially contains the startingVertex;
    T.add(startingVertex);
    int numberOfVertices = vertices.size(); // Number of vertices
    int[] parent = new int[numberOfVertices]; // Parent of a vertex
    // Initially set the parent of all vertices to -1
    for (int i = 0; i < parent.length; i++)
      parent[i] = -1;
double totalWeight = 0;  -> double totalWeight = 1
// Total weight of the tree thus far
// Clone the priority queue, so to keep the original queue intact
List<PriorityQueue<WeightedEdge>> queues = deepClone(this.queues);
// All vertices are found?
while (T.size() < numberOfVertices) {
  // Search for the vertex with the smallest edge adjacent to
  // a vertex in T
  int v = -1;
  double smallestWeight = Double.MAX_VALUE;
  for (int u : T) {
    while (!queues.get(u).isEmpty() &&
        T.contains(queues.get(u).peek().v)) {
      // Remove the edge from queues[u] if the adjacent
      // vertex of u is already in T
      queues.get(u).remove();
    }
    if (queues.get(u).isEmpty()) {
      continue; // Consider the next vertex in T
    }
    // Current smallest weight on an edge adjacent to u
    WeightedEdge edge = queues.get(u).peek();
    if (edge.weight < smallestWeight) {
      v = edge.v;
      smallestWeight = edge.weight;
      // If v is added to the tree, u will be its parent
      parent[v] = u;
    }
  } // End of for
  if (v != -1)
    T.add(v); // Add a new vertex to the tree
  else
```

```
      break; // The tree is not connected, a partial MST is found
    totalWeight += smallestWeight;
  } // End of while
  return new MST(startingVertex, parent, T, totalWeight);
}
/** Clone an array of queues */
private List<PriorityQueue<WeightedEdge>> deepClone(
  List<PriorityQueue<WeightedEdge>> queues) {
  List<PriorityQueue<WeightedEdge>> copiedQueues =
    new ArrayList<PriorityQueue<WeightedEdge>>();
  for (int i = 0; i < queues.size(); i++) {
    copiedQueues.add(new PriorityQueue<WeightedEdge>());
    for (WeightedEdge e : queues.get(i)) {
      copiedQueues.get(i).add(e);
    }
  }
  return copiedQueues;
}
/** MST is an inner class in WeightedGraph */
public class MST extends Tree {
  private double totalWeight; // Total weight of all edges in the tree
  public MST(int root, int[] parent, List<Integer> searchOrder,
      double totalWeight) {
    super(root, parent, searchOrder);
    this.totalWeight = totalWeight;
  }
  public double getTotalWeight() {
    return totalWeight;
  }
}
```

Below is a test case generated by EVOSUITE, which kills the above mutant, but Tpalus could not find and kill the same mutant. The mutant is detected by testing the method getTotalWeight(). It is a relatively simple method that was easily covered by EVOSUITE, But Tpalus could not kill the mutant as it could not produce a test case verifying the method getTotalWeight(). Tpalus failed to detect the test case as the sequence model that was generated during the first step did not cover the getTotalWeight() and the method was missed in the static analysis stage as well.

```
@Test
public void test11()  throws Throwable  {
LinkedList<WeightedEdge>linkedList0 = newLinkedList<WeightedEdge>();
WeightedGraph<Integer>weightedGraph0=
newWeightedGraph<Integer>((List<WeightedEdge>) linkedList0, 1);
WeightedGraph.MSTweightedGraph_MST0=
weightedGraph0.getMinimumSpanningTree(0);
double double0 = weightedGraph_MST0.getTotalWeight();
assertEquals(0, weightedGraph_MST0.getRoot());
assertEquals(0.0, double0, 0.01D);
assertEquals(1, weightedGraph_MST0.getNumberOfVerticesFound());
assertEquals(1, weightedGraph0.getSize());
  }
```

5. Threats to Validity

The experiment that was done on the ten test subjects may seem sound, but there are many threats to the validity of the study.

a. The first threat to validity was not to consider equivalent mutants. This was because finding equivalent mutants is a laborious and a very time consuming task.

b. The sizes of the ten test subject's chosen were small and usually consisted of some hundreds of lines of code. The results might vary when the tools are tested on a larger jar file of hundred of classes. But it would be really difficult to understand the test cases created for large software.

c. None of the test subjects in the experiment were a web application nor connected to a database. Experimenting on different subjects might give different results.

6. Conclusion

There is good reason to argue about the merits of both the tools. Both the tools are very complex with a wide range of options for test generation. EVOSUITE gains an upper hand when it comes to coverage scores and readability of the test cases, but Tpalus had success in terms of killing mutants. But the mutation scores (both raw kill score and normalized scores) were not considerably huge for Tpalus. The difference is around 10% for the test subjects. This is not a significant difference. One of the main advantages of Tpalus over EVOSUITE was that it could generate test cases with regression assertions.

Even though Tpalus had better mutation scores, EVOSUITE is easy to use and very test cases produced are very readable. They generally consisted of very few lines. Moreover Tpalus needs a driver program to implement the test case generation, but EVOSUITE just needs the bytecode of the program. One can easily add/modify/remove test cases from the existing test suite generated by EVOSUITE. The same cannot be said with Tpalus because of the aforementioned difficulties. Tpalus can be improved with the help of a better driver class because that allows Tpalus to generate a better trace model, the most useful thing for generating test cases.

Each tool has its own pros and cons. Both solve the oracle problem effectively. If usability is the paramount consideration, then EVOSUITE is the tool to use.

7. Future Work

The next step in the process is to establish the number of equivalent mutants and calculate the mutation scores more accurately. It would also be good to check if the results are valid on an experiment with large test subjects.

Experiments should also be conducted on a wider range of subjects, including web applications and systems that connect to databases. This would however be a more time consuming experiment. It may also be helpful to check how coverage and mutation scores are related by varying the size of the test suite. It might also be useful to explore the use of different driver programs when using Tpalus.

8. Bibliography:

[1] Fraser, Gordon, and Andrea Arcuri. "EVOSUITE: automatic test suite generation for object-oriented software." *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011.

[2] Fraser, Gordon, and Andrea Arcuri. "Whole test suite generation." *Software Engineering, IEEE Transactions on* 39.2 (2013): 276-291.

[3] Zhang, Sai, et al. "Combined static and dynamic automated test generation."*Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011.

[4] Inozemtseva, Laura, and Reid Holmes. "Coverage is not strongly correlated with test suite effectiveness." *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.

[5] Major Mutation Framework for Java. http://mutation-testing.org/.

[6] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, “Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study,”*ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2014.

[7] EVOSUITE. http://www.EVOSUITE.org/

[8] Tpalus. https://code.google.com/p/tpalus/

[9] Cobertura. http://Cobertura.github.io/Cobertura/

[10] Apache Ant. http://ant.apache.org/

[11] Pacheco, Carlos, et al. "Feedback-directed random test generation." *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007.

[12] Pacheco, Carlos, and Michael D. Ernst. "Randoop: feedback-directed random testing for Java." *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 2007.

[13] Kracht, Jeshua S., Jacob Z. Petrovic, and Kristen R. Walcott-Justice. "Empirically Evaluating the Quality of Automatically Generated and Manually Written Test Suites." *Quality Software (QSIC), 2014 14th International Conference on*. IEEE, 2014.

[14] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. Journal of Documentation, 28:11–21, 1972.

[15] Bytecode Instrumentation http://www.correlsense.com/blog/java-bytecode-instrumentation-an-introduction/

[16] Fraser, Gordon, and Andreas Zeller. "Mutation-driven generation of unit tests and oracles." *Software Engineering, IEEE Transactions on* 38.2 (2012): 278-292.

[17] Just, Rene, Franz Schweiggert, and Gregory M. Kapfhammer. "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler."*Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011.

[18] Grun, Bernhard JM, David Schuler, and Andreas Zeller. "The impact of equivalent mutants." *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*. IEEE, 2009.

[19] K. Kapoor. Formal analysis of coupling hypothesis for logical faults. Innovations in Systems and Soft. Eng., 2(2), 2006.

[20] Jia, Yue, and Mark Harman. "An analysis and survey of the development of mutation testing." *Software Engineering, IEEE Transactions on* 37.5 (2011): 649-678.

[21] Offutt, A. Jefferson, et al. "An experimental evaluation of data flow and mutation testing." *Softw., Pract. Exper.* 26.2 (1996): 165-176.

[22] Tikir, Mustafa M., and Jeffrey K. Hollingsworth. "Efficient instrumentation for code coverage testing." *ACM SIGSOFT Software Engineering Notes*. Vol. 27. No. 4. ACM, 2002.

[23] Runeson, Per. "A survey of unit testing practices." *Software, IEEE* 23.4 (2006): 22-29.

[24] Test Subjects- http://www.cs.armstrong.edu/liang/intro9e/examplesource.html

[25] Test Subjectshttp://www.deitel.com/Books/Java/JavaHowtoProgram10eEarly Objects/tabid/3656/Default.aspx

[26] Tahbildar, Hitesh, and Bichitra Kalita. "Automated software test data Generation: Direction of Research." *International Journal of Computer Science and Engineering Survey* 2.1 (2011): 99-120.

[27] J. Edvardsson, A Survey on Automatic Test Data Generation," In Proceedings of the Second Conference on Computer Science and Systems Engineering(CCSSE'99), Linkoping, pp. 21-28 10/1999.

[28] B. Korel, “Automated software test data generation,” IEEE Transactions on Software Engineering, pp. 870–879, 1990.

[29] P. McMinn, “Search-based software test data generation: A survey,” Software Testing, Verification and Reliability, vol. 14, no. 2, pp. 105– 156, 2004.