Master thesis

An experimental study on combining automated and stochastic test data generation

Patrick Haverkate - rickh@student.chalmers.se

Marufa Binte Mostafa - binte@student.chalmers.se

June 4, 2020

Acknowledgements

We would first like to thank our thesis advisor Dr. Francisco Gomes Oliveira Neto of the IT faculty at Göteborgs Universitet for his immense support and guidance throughout the whole thesis period. Also we would like to express our heartiest gratitude to our examiner Prof. Robert Feldt for his valuable reviews and advices.

Patrick Haverkate and Marufa Binte Mostafa, Gothenburg, June 2019

Abstract

Test data plays an important role in improving the quality and effectiveness of test cases and automatic generation of meaningful test data saves a lot of human effort. There are several automatic test data generation approaches; stochastic test data generation is one of them. To investigate the benefits and challenges of using a stochastic test data generation technique, this study presents `JuliaTest`, an automatic test data generation tool integrating stochastic test data generation framework with the unit testing framework JUnit5. Using `JuliaTest` two empirical studies were conducted on open-source projects to compare different automatic test generation techniques and to investigate the behavior of `JuliaTest` in different settings (e.g., varied number of data generated, different generators used). Performed experiments of limited scope showed promising results indicating test data generated by stochastic technique is able to provide better mutation coverage and detect more faults when compared to other existing alternatives. These experiments aims at being used as a baseline for future work with broader scope and setup. Moreover, the study discusses different design choices made during the implementation of the tool. Finally, a number of future research concepts are discussed to open the door for researchers interested in this field.

***keywords**: Test automation, Automatic test data generation, Stochastic test data generation, GödelTest*

Contents

Chapter 1

Introduction

Software testing is a crucial part of software development, such that the creation and the quality of test cases affect the quality of the software product itself. Since manual creation of test cases is often costly and time consuming in complex systems [1], current research propose several approaches to automatically generate test cases (e.g., Evosuite [2], Randoop[3], Quickcheck [4]). However, there are also challenges in test automation such as selecting the right tool, demanding skilled resources, getting the developers to trust the generated test and so on. Moreover, to automatically validate the correctness of the test cases there needs to be a mechanism which takes input and determines the correctness of the test by comparing the generated output with expected one. This mechanism is called test oracle. To make the comparison between actual and expected output a test oracle uses artifacts like source code, specifications etc. These kind of artifacts are often not available to the test engineers or are very expensive which makes test automation more challenging. This research will try to investigate and advance test automation techniques, with a particular contribution to automated and stochastic test data generation.

In software testing there are several different techniques and tools being used for automated test generation, some of these are: Randoop[3], Evosuite [2] and GödelTest [5] [6]. All of these test data generators use random or meta-heuristic data generation. In random data generation, data is randomly generated and used to test the software, e.g., if a method requires a string value, a string with random characters and random length is generated. Such techniques are easy to implement, but has as a disadvantage that it can lead to a lot of test data testing the same or similar cases.

On the other side of this scale, there are systematic approaches to test generation, where the drawn samples follow a predefined structure and don't have any randomness. In the middle of these techniques would be stochastic

test generation where the test engineer would define a "distribution" on one or more properties of the System Under Test (SUT), or the test cases and sample from that distribution, this allows her to have some influence on the test data, while not completely being structural.

These tools also differ on whether they require knowledge or some instrumentation of the source code (white-box) or if they don't (black-box). With white-box techniques most techniques try to create a graph of the different "paths" (e.g., control flow graph) through the software in an attempt to create test cases covering as much of the code as possible[2]. Conversely, black-box techniques uses the specifications of the program (e.g., which inputs are expected) to see if the executed test cases actually provide the expected output.

A test data generator for the more primitive types (strings,numbers etc) can easily be created by using random numbers/strings, whereas some testing frameworks (e.g., GödelTest[5], EvoSuite[2]) also support the generation of more complex or structured data such as XML files or trees [7], [8]. These tend to have very specific requirements and require configuration info such as schemes. One promising technique for software testing is diversity-based testing, where a diverse set of test data is used to test the software [9], [10].

These techniques for automated test generation have been evaluated in industrial cases but technology transfer and adoption is limited [11], particularly since there should be a clearer link between automated test generation and manual testing.

In summary, there are several dimensions when considering automated test generation. One can consider automated techniques that can generate data using more random, stochastic or systematic approaches, either using different software artifacts such as the source code (white-box) or the specification[1] (black-box) of the System Under Test (SUT). Consequently, the challenges lie on using those dimensions to generate test data that adds value to software testing as well as the tester responsible for analyzing the outcome of the tests.

The purpose of this thesis is to investigate how test automation can be improved to be used in combination with current test frameworks. Research literature have come up with advanced and complex testing techniques but most of the common tools that are used today do not support these new mechanisms. We aim to bridge this gap, make testing activities (e.g., test instrumentation) easier and enable human testers benefit from automated and stochastic test generation.

We propose an extension to unit testing frameworks where test generation can be combined with manual testing. Particularly, we extend JUnit5[2] to use

[1] Note that specification does not necessarily refer to a documentation (e.g., requirements documents), rather we mean the definition of input and the expected output of the program.

[2] `https://junit.org/junit5/`

GödelTest[3] [5] generated test data and differential test oracle with support for both local and CI environment with a view to improving the quality of the source code by reducing test case preparation time for designing or instrumenting, dependency on source code. We evaluate our proposed approach by conducting an experiment on open source data (e.g., JFreeChart from Defects4J[4]) where stochastic test generation approaches will be compared to existing manual tests. We focus on answering the following research questions:

RQ1: How can we instrument xUnit Frameworks to enable automated and stochastic test generation?

RQ2: What are the challenges in applying stochastic test data generation to unit testing frameworks?

RQ2.1: What are the challenges in instrumenting test frameworks for stochastic test generation?

RQ2.2: What are the characteristic of xUnit Frameworks that can foster usage of stochastic test generation?

RQ3: What are the trade-offs between different test generation techniques for unit testing?

RQ3.1: How do different test generation techniques compare in terms of fault detection?

RQ3.2: How do different test generation techniques compare in terms of line coverage?

RQ4: What are the effects of different settings/setups in stochastic test generation?

RQ4.1: How do stochastic test generation technique change in terms of fault detection for different number of data generated?

RQ4.2: How do stochastic test generation technique change in terms of line coverage for different number of data generated?

RQ4.3: How do stochastic test generation technique change in terms of fault detection for different type of generator used?

RQ4.4: How do stochastic test generation technique change in terms of line coverage for different type of generator used?

The expected contributions of our study are the following:

[3] `https://github.com/robertfeldt/DataGenerators.jl`
[4] `https://github.com/rjust/defects4j`

- C1: Identification of challenges in extending Unit testing framework to enable automated and stochastic test generation approaches. These challenges comprise mainly, our lessons learned, and aim to support future advances in this field by leading to different design choices on how to integrate xUnit frameworks and automated test generation.

- C2: Creation of a prototype extension to JUnit, a popular xUnit framework for Java programs and support for generating JSON from Gödeltest. The prototype is an addition to the existing options of tools combining automated test generation and unit testing.

- C3: An experimental study comparing manual test suites, stochastic test generation approaches and other automated test generation techniques (e.g., Evosuite), in open source data. The comparison reveals trade-off for the investigated techniques.

- C4: An experimental study investigating deeper into stochastic test generation approach with different settings/setups, in open source data. The experiment is automated and reproducible artifacts are available online[5] for future replications and validation of our findings.

As a consequence, C1, C3 and C4 compose our scientific contributions, whereas C2 comprise our technical contributions.

[5] `https://github.com/evpxregu/TRAM`

Chapter 2

Background and Related Work

In this chapter we present the background theories and existing techniques of automated test generation. Moreover we discuss the works related to the concepts of our study, such as stochastic and diversity based test data generation, existing tool support in automated test generation, the challenges in addressing the oracle problem.

Unit testing is the most fundamental method in the software testing process. Main goal of unit testing is to test individual units of a software (e.g. source code, modules with associated data and procedures) for determining the fitness for use [12]. However, it is nearly impossible to capture all test combinations manually in a fairly large application and the situation becomes even worse in case of regression testing. Unit test automation is therefore considered to be less expensive in terms of time, cost and labor and more reliable for covering larger amount of test cases. JUnit [13] is a Java unit test framework for automated, repeatable, self-verifying tests and is widely adopted in organizations using Java. The framework allows test engineers to compare output values with expected values using methods from the assert class. Since the context of our study is unit testing, our focus is to improve testing capabilities with a tool which hooks up GödelTest generated data to JUnit and evaluating the effectiveness of that.

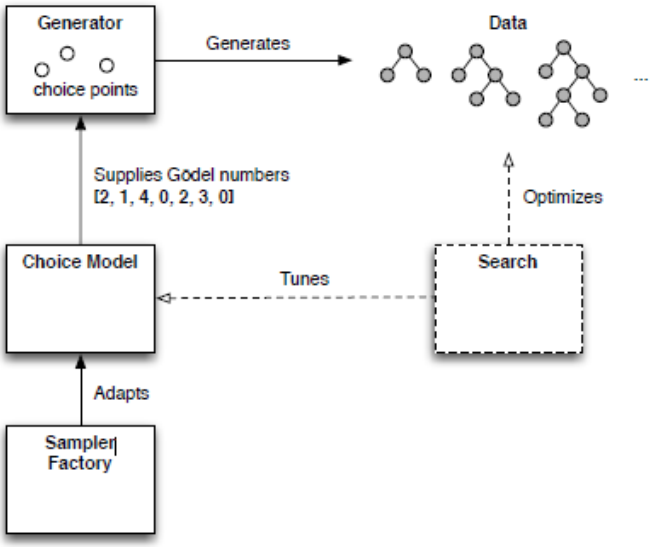

Figure 2.1: GödelTest Framework Architecture [5]

2.1 Test data generation

2.1.1 Stochastic Test data generations

In test automation, data generators are used to feed test cases with generated test data. Generating test data randomly is one of the simplest way but not very effective if the generated random data distribution is not similar to the real data. [14].

Feldt and Poulding proposed the GödelTest [5] [6] framework consisting of generators and choice models that generates stochastic test data. Figure 2.1 shows the architecture of the GödelTest Framework. In GödelTest data generators create specific data structures and choice models control the distribution of test data through parameters [5]. The 'choice model' concept introduces non-determinism by deciding values or execution paths to be taken during each choice point [5]. Therefore using GödelTest framework it is possible to tune the distribution of generated test data so that it has the desired bias objectives [5]. GödelTest uses metahuristic search with differential evolution algorithm [15] to meet the global bias objectives which has shown promising results compared to Botlzman sampler or QuickCheck [5].

Figure 2.2 shows an example how data generators of GödelTest written in Julia look. The generator example shows four different generators being

```
@generator FirstNameGen begin
    start() = choose(String, "[A-Z]{1}[a-z]{2,15}")
end

@generator LastNameGen begin
    start() = choose(String, "[A-Z]{1}[a-z]{2,10}")
end

@generator SalaryGen begin
    start() = choose(Int,29000,70000)
end

@generator PersonGen(firstNameGen,lastNameGen,salaryGen) begin
    start() = Person(choose(firstNameGen),choose(lastNameGen),choose(salaryGen))
end

firstNameGen = FirstNameGen()
lastNameGen = LastNameGen()
salaryGen = SalaryGen()
personGen = PersonGen(firstNameGen, lastNameGen,  salaryGen)
```

Figure 2.2: persongenerator example

defined. The first three generators are used to generate individual values and the fourth one is used to generate a `person` object which contains a value of each of the generators. The regex on `FirstNameGen` states that the first character should be capital. After the first character, any number of lower case characters between the range 2 to 10 can be generated which means there is a limit of at least 3 characters to the first name. `LastNameGen` also follows similar structure. `SalaryGen` generator generates a `Integer` number between the range 25000 to 70000. Finally the `PersonGen` generates a `person` object that has the properties- first name, last name and salary. Figure 2.3 shows the result of calling the `choose()` method on the person generator a number of times.

```
julia> choose(personGen)
Person("Iuhy","Ycu",61340)

julia> choose(personGen)
Person("Lzv","Fodn",49733)

julia> choose(personGen)
Person("Ibpayl","Nnfqvtjcrpr",62110)

julia> choose(personGen)
Person("Rkq","Midzq",59545)
```

Figure 2.3: example `persongenerator` results

Quickcheck [4] [14] is another popular framework for randomly generating

complex and varied test data. Using some libraries of QuickCheck it is possible to automatically derive generators for specific algebraic data types [16] [17].

SmallCheck and Lazy SmallCheck [18] libraries follows the lead of QuickCheck but differs in data generation technique. While QuickCheck uses a sample of randomly generated values, SmallCheck and Lazy SmallCheck use all values upto a limiting depth. Claessen et al. [19] improved this mechanism by using size instead of depth and allowing random search in large sets and proposed an algorithm for generating uniformly distributed test data. Combining the algorithm with a backtracking-based generator, they presented a hybrid generator that produces almost uniform test data and satisfy given boolean predicates. In case of generating large test case the performance of their proposed generator is however still an issue. EasyCheck [20] is another library for generating random test data written in curry functional logic programming language. By narrowing it can satisfy given predicates but has the problem of generating data of similar values for the same test run.

Mista and Russo [21] extended DRAGEN to DRAGEN2 that can generate algebraic data types by synthesizing QuickCheck generators. It allows the test engineers to adjust the distribution of the random values at compile-time. Simple combinatorial structures (graphs and trees) can be created using Boltzmann sampling [22] or using the tool proposed by de Oliveira Neto et al. which randomly selects baseline graph constructs and combines them. [23]. Basing on Boltzmann model Bendkowski et al. [24] proposed boltzmann-brain framework that specifies and synthesizes standalone Haskell random generators. To acquire desired distribution of values it allows tuning of parameters and discards the samples of unwanted sizes.

2.1.2 Diversity-based Test data generations

There are several test data generation techniques that ensures diversity within the input domain implicitly or explicitly. When each data point in an input domain has the same probability of being selected, some sort of implicit diversity is achieved [25]. Another example of implicit diversity technique is to have data partitions based on different properties of the data and then make sure to select data from each of the partitions [26]. Among the test data generation techniques that imposes diversity explicitly, some (e.g., Anti random testing [27], Adaptive random testing [28]) use Euclidean distance metric between two test inputs as the measure of diversity. Taking spatial distribution of the test cases into consideration Mao et al. [29] presented a distance-aware forgetting strategy for adaptive random testing that reduces the cost of computing distance by ignoring out of "sight" test cases. The techniques proposed by Bueno et al. [30], Cartaxo et al. [31], Hemmati et al. [32] define their diversity metric taking consideration of the test input set as a whole.

Based on information theory and using the normalized compression distance diversity metric [33], Feldt et al. introduced test set diameter diversity metric that can be used to create diverse test sets [9].

In their research Feldt and Poulding [34] found a hill climbing search to be efficient for feature-diversity. A method for finding boundary between valid and invalid regions of the input space was proposed in another research of Marculescu and Feldt [35].

As the mechanism to achieve diverse test input in our extension of the JUnit Framework, we have used the GödelTest framework and it's implicit diversity from the distributions used to sample datums.

2.2 Automated unit test generation tools

A number of automatic test generation tools are available for Java, both commercial and open-source. Main test data generation methods used in these tools are- random, search based and symbolic execution based [36].

Randoop[1] is a tool for generating feedback-directed unit tests by creating method sequences randomly [3]. Randoop creates assertions by using the results from the execution [3]. Its random but smart test generation techniques has been claimed to be highly effective in finding bugs and generating regression tests[37].

Evosuite[2] use search-based test data generation method with the special evolutionary algorithm - Genetic Algorithm [2]. Evosuite offers optimization based on different coverage criteria (e.g., lines, branches, outputs and mutation testing)[38]. Evosuite also minimizes tests based on which ones are contributing to achieve coverage[38].

Parasoft Jtest[3], Agitar AgitarOne[4] and Jcute[5] use the symbolic execution based method for Java unit test data generation which is popular among other methods for high code coverage [39]. Some other tools for automatic java unit test generation are CodePro Analytix, Jwalk, CATG, GRT, JTExpert, Symbolic Path Finder, T3, Jcrasher, PET [36].

Lakhotia et al. [40] introduced a search based software testing tool AUSTIN for C language that uses stochastic meta–heuristic search algorithms. Using similar algorithms Papadakis and Malevris [41] proposed an automatic mutation based test data generation framework Metallaxis for Java. However, GödelTest

[1] `https://randoop.github.io/randoop/`
[2] `http://www.evosuite.org/`
[3] `https://www.parasoft.com/products/jtest`
[4] `http://www.agitar.com/solutions/products/agitarone.html`
[5] `http://osl.cs.illinois.edu/software/jcute/`

framework has not been implemented in any tool yet.

2.3 Oracle Problem

One of the challenges of implementing a unit test framework is distinguishing the desired/correct behavior of the SUT from potential incorrect behavior which is addressed as the oracle problem [42]. The difficulty lies in making the test framework automatically "guess" the expected output while the test input is being automatically generated in parallel. As this is a bottleneck in the applicability and effectiveness of most test case selection strategies, the unavailability and expensiveness of test oracles makes it a fundamental problem in software test automation [43]. It is very difficult to make software testing fully automated unless this problem is solved [44].

Barr et al. [42] addressed the issue of automation of test oracle generation getting less attention than required and presented a comprehensive survey of approaches in test oracle problem. In their survey they divided test oracles in three categories: specified test oracle, derived test oracle and implicit test oracle. Specified test oracles being heavily dependant on software specification, some challenges of specified test oracles are: 1. the lack of formal specifications 2. possibility of specifications being imprecise, including unfeasible behavior or not capturing all the relevant behavior 3. possibility of misinterpreting model output and the challenge of equating it to concrete program output [42].

On the other hand, derived test oracles use information derived from Software properties or different software artifacts related to the SUT [42]. Some different versions of derived test oracles are: pseudo-oracle [45] or N-version programming [46], [47] that can be produced using genetic programming [48] but is considered very expensive [49], metamorphic testing [50] in which still remains the challenge of discovering metamorphic relations automatically [42], test oracles from system execution traces [51] using invariant detection technique [52], [53] (where inferring perfect invariant is very challenging [54]) or specification mining technique [42] etc.

Implicit test oracles distinguish between a SUT's correct or incorrect behavior basing on implicit knowledge such as system crush or execution failure [55]. Moreover, there are AI based test oracles where the system learns the correct behavior from running with known answers using artificial neural networks, support vector machines and decision trees [49].

Regression test oracle is another type of derived test oracle [42]. A regression test oracle assumes that the previous version of the SUT is correct and is used as the oracle for the existing one in order to find disruptions due to modification in the new version [42]. Regression testing can be expensive in practice

[56]. There have been a number of studies on different approaches for test case minimization, selection and prioritization in regression testing [57].

However, regression testing is complemented with differential testing which compares exhaustive test results of two or more version of a system in order to find potential candidate for bugs [58]. Basing on the black box differential testing Jung et al. proposed "Privacy Oracle" that is used to discover information leaks in an application [59]. To overcome the oracle problem in our proposed solution we intend to use *differential test oracle* as a solution.

Chapter 3

Proposed Solution

Automated test data generation tends to be better at finding specific corner cases of which humans are unaware, however this can be costly to execute or implement in practice [60]. Conversely, knowledge about the system also helps testers in identifying, e.g., severe defects. For instance, if we were to give an experienced tester some sort of control over test cases generation she could use her experience to find more meaningful defects then just using general diversity tests [61]. This is where stochastic test data generation technique comes into play.

Aiming at further investigating the applicability of stochastic test generation frameworks using unit testing frameworks we created a tool for testers to easily use GödelTest generated test data in JUnit 5 with support for doing this both locally as well as in a CI environment using differential test oracles. We believe this would allow for: saving time in designing test cases, reducing the instrumentation or dependency on source code to create tests, and ultimately, improve the quality of the SUT. Further in this chapter we discuss different elements of our proposed tool along with the architecture and design decisions which is the technical contribution of this study.

3.1 Our proposed architecture

The created toolchain consists of four modules- 1. `JuliaTest`, 2. `juliatest-maven-plugin` 3. `Junit5` 4. `A SUT`. Figure 3.1 shows how they relate to each other. Most of the functionalities are in the `JuliaTest` project, which are:

- Calling GödelTest for the generation of test data

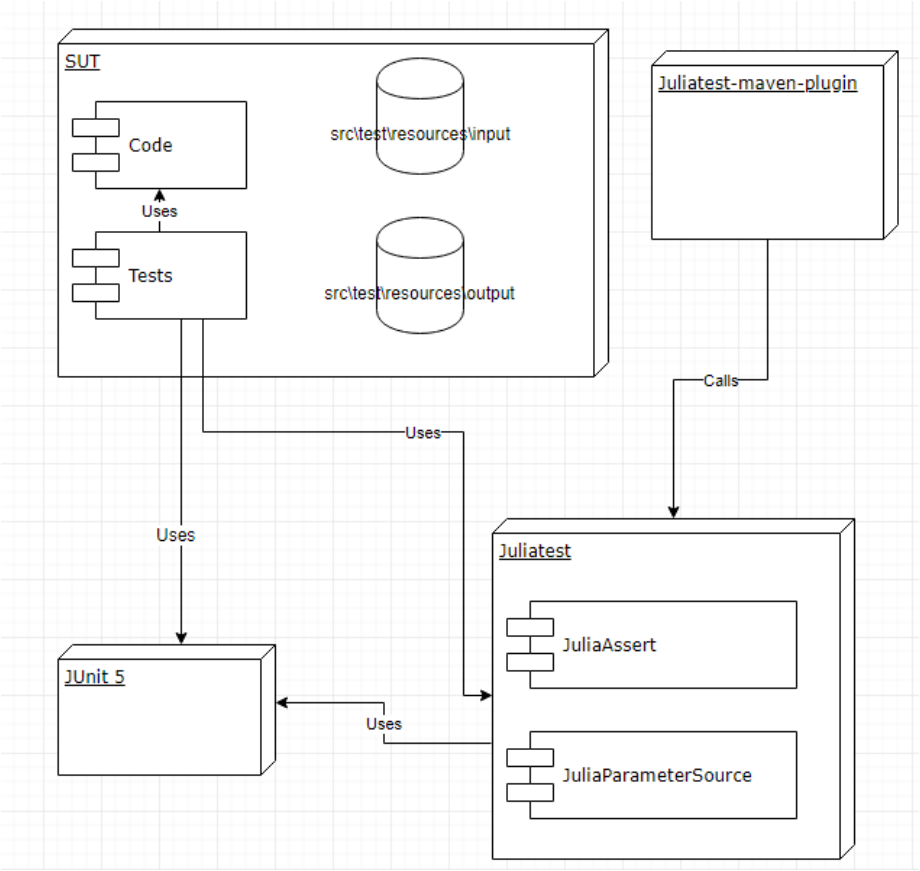

Figure 3.1: Communication between the modules of the tool

- Passing test data onto Julia 5
- Functionality for the support of differential oracle testing
- Code for running the test with the data

Automatic execution of tests can be done without using the `juliatest-maven-plugin` as well- by just calling maven test. It contains the functionality which allows for CI integration and easier running of the tests.

When `JuliaTest` is used, all test annotated with `@juliasourceparameter` are checked for having already generated test data for the specific annotated method. If this is not the case test data is generate and stored in the output folder and stored in a simple CSV format. Complex data with relations of sub-entities are saved as JSON and then read in the JUnit code. When `JuliaTest`

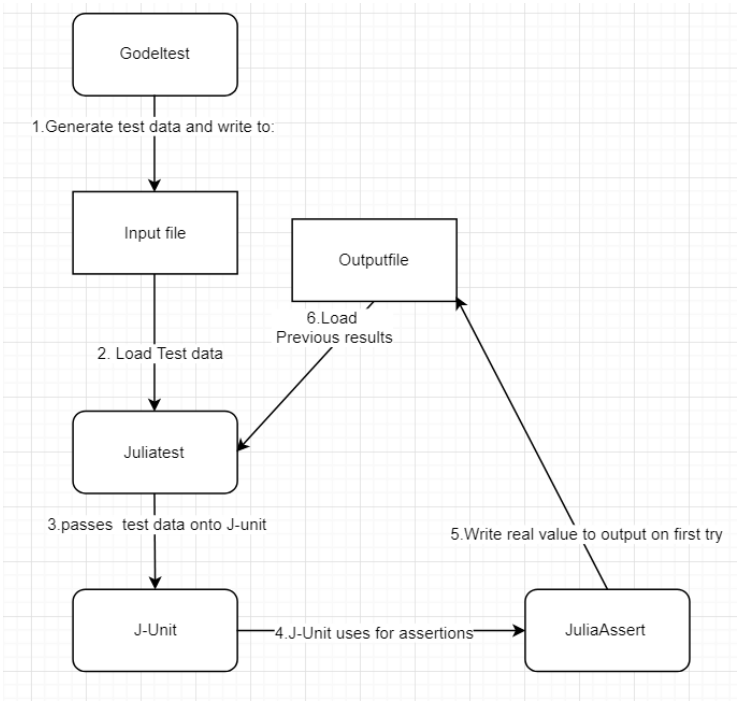

Figure 3.2: Data flow of the JuliaTest

is executed via maven a unit test run is generated for each test data. There is support for using a differential oracle in the form of the `JuliaAssert` class which takes an actual[1] and an expected value. The actual value is the result of an operation (in the given example in figure 3.4 the code in figure 3.5 is called. It calculate taxes which is depended on the input value salary). If a test is run for the first time, the `JuliaAssert` class writes the actual value to a file and on the run it is passed on by `JuliaTest` to the unit test in the form of a variable of an instance of the `Output<T>` class. The instance of the `Output<T>` is then passed on to the `JuliaAssert` class for checking if is correct.

Figure 3.2 shows the flow of information through the system when the system is called once from Maven. First, the Test code will invoke GödelTest and Julia to generate test data that will then, write it in an input file. The file is read by `JuliaTest` an ran as individual unit tests by JUnit. The `JuliaAssert` class writes the output value of a test to a file which is loaded on sequential runs as the expected value by `JuliaTest`. If the expected value and the actual output value are not same the test fails according to differential testing approach.

[1]By actual we refer to the value returned by invoking a method on the SUT

3.1.1 Usage

Following are the steps a tester needs to follow to use `JuliaTest`:

1. Add `JuliaTest` to the project's project's dependencies (i.e., the `pom.xml` file if using Maven).

2. Create a generator for the previously mentioned unit test (e.g. figure 2.2).

3. Create a `TestAdapter` unit test according to the earlier mentioned structure(see figure 3.4).

4. Run the unit test for the first time so it generates both the input data as well as the values for the expected output which is used for differential testing.

5. Refactor the SUT

6. Rerun the system and see if the test failed due to a change in the values for the expected output.

3.2 JSON generator

As an extension to GödelTest, a JSON[2] data generator was developed, which takes a JSON schema[3] as input and generate JSON according to the schema using GödelTest. For GödelTest there are already several generators for other schema formats such as XSD[4] and juliatype for GödelTest these are maintained in the DataGeneratorTranslation github repository[5].

The reason for generating JSON instead of another format is that JSON nowadays is commonly used by webservices and for storage by a wide array of applications, furthemore the size of JSON is quite small compared with older types like XML which makes for more efficient transport and it's easier to read compared to XML. This comes at a cost though, JSON has less features e.g no namespaces or built-in support for inheritance.

JSON schema has been under ongoing development since 2009 and has gained traction as a format for describing responses of web services and other applications,

Since JSON schema is intended for validation and documentation of the

[2]https://www.json.org/
[3]https://json-schema.org/
[4]https://www.w3.org/XML/Schema
[5]https://github.com/robertfeldt/DataGeneratorTranslators.jl

Figure 3.3: JSON schema example

data, generating proper data for all schemes can be hard, for instance when a property is just denoted as a string, what kind of length and what kind of characters should be generated for the string? We have tried to find sane defaults for such situations, but they might still give a poor representation of production data. In order to solve this problem, JSON schema defines some limitations to the data, for the most part these are implemented, e.g. patteren and max length, min length for string minItems and maxItems for an array etc.

At the time of writing the JSON schema format is still in the drafting phase and properties and validators are still being added, we have tried to implement generators for the more stable functionalities, the following types and validators for them have been implemented:

Types	Validators
Object	Properties
String	Pattern, minLength and maxLength
Integer	Minimum and maximum
Number	Minimum and maximum
Array	MinItems, maxItems and properties

The implementation could be extended by using other functionality from the JSON schema standard, such as reference to other objects which would

```java
@ParameterizedTest
@JuliaParameterSource(value="salaryGen", nrOfSamples=100)
public void simpleTestDataGeneratorJuliaTest(Integer value, Output<Integer> expected) {
    int realVal =  Calculate.calculateTax(value);

    JuliaAssert.IsCorrect(realVal, expected);
}
```

Figure 3.4: JuliaTest unit test example

```java
public static int calculateTax(int salary)
{
    double tax = salary * 30 / 100;
    return (int)tax;
}
```

Figure 3.5: The method called by the test in figure 3.4

allow for generation of recursive structures. Another example would be adding support for the required properties allowing the ability to define which property to always generate and which properties to leave out, unlike the current system where all properties are always generated. Furthermore, support for arrays where the contained items are not always the same entity could also be added, these we see as future work since they are more advanced functionality that seems like something that is used by every schema.

3.3 Unit test example

A unit test written with `JuliaTest` looks similar to a regular JUnit parameterized unit test but with an added `@JuliaParameterSource` which annotates the generator to be use. Figure 3.4 shows an example of a unit test written with `JuliaTest` which test the method shown in figure 3.5. in figure 3.6 the generator which is used by the test is shown.

After running the unit tests for the first time in the `src/test/resources/input` folder a file named after the unit test is created with the input values for the test adapter, an example of the contents of this file are shown in figure 3.7a in

```
@generator SalaryGen begin
    start() = choose(Int,29000,70000)
end
salaryGen = SalaryGen()
```

Figure 3.6: The generator used by the test in figure 3.4

```
55197
39890
30232
68931
41324
60652
66250
40425
63929
34335
```

(a) contents input file

```
"16559"
"11967"
"9069"
"20679"
"12397"
"18195"
"19875"
"12127"
"19178"
"10300"
```

(b) contents output file

Figure 3.7: Created files from `JuliaTest`

the `src/test/resources/objects` for an output file is created for this input, an example of this is shown in figure 3.7b the contents of this file are used on the second run of the unit test by the test oracle as the value to compare to.

3.4 Limitations of the tool

The tool is only the first step in integrating GödelTest with existing unit testing framework and is still in its initial phase. With further development it is possible to make it fully matured and more efficient. Some limitations of the current version are:

- No support for writing exceptions to the output file i.e., in case the generated datum would cause an exception in the SUT, the current version of `JuliaTest` does not log such failures.
- Referenced generators needs to be instantiated, passing the name of a non instantiated generated does not work yet, hence, has to be invoked during the Maven build process, which adds a small during the process, which adds a small overhead (seconds per test) to the execution process. This can hinder usability of the tool, and could be improved in future work.

3.5 Architectural choices and lessons learned

This section explains the architecture choices that were made while creating our proposed tool.

For the integration with a unit testing tool two things are needed: 1)

the test class variables need to be bound to the generated variables 2) the tests needs to run N times where N is the number of samples. For the first point the generated values can be bound to the parameters of the test method which is easily done with the JUnit framework. However, generally using class variables is seen as a best practice. To accomplish this these variables needs to be annotated somehow and mapped to the generated datums. Because of the added complexity of this we chose the first solution, for further research it might be interesting to find a solution to this problem.

We solved the second point by extending a JUnit `"parametersource"` which are responsible for supplying the variables of a test method to call. By returning an array of more then one element JUnit run the same test multiple times. Another solution for running a a test for each datum is to automatically create at est case for each generated test datum, we decided not to do this since it would lead to extremely large test classes. After deciding on the way to supply the data to the unit test, a decision needs to be made about how to get the data from the generator to the unit testing framework. There are several options, such as:

- passing the data straight from the generator to the unit testing framework without storing it in an intermediate format somewhere
- storing the data first in some format somewhere and then passing it along.

The first option has as an advantage of being simpler, the data does not need to be saved in a format to a file storage and then loaded and parsed and then parsed to the testing framework. On the other hand the second option has the advantage of rerunning the tests with the same data without having to wait to generate the data again. This would also make the tests be more reproducible and controllable since they would be run with the same test data after the generation of the data, and would not have to handle new data unless the file was removed. Note that for differential oracles, one would have to re-generate the oracles every time new data is generated, hence option one would need a different approach to solving the oracle problem (e.g., pseudo-oracle, metamorphic testing, test oracles from system execution traces etc.). Storing the test data also means that a regression test oracle can be used, since the results of the first test run can be saved and then be reused after the code is refactored whereas the first option would require a non regression test oracle which tends to add complexity.

GödelTest supports the use of choice models where the generation mechanisms (i.e., methods and distributions to sample datums) can be changed. For example, the size of the resulting element when calling the `mult`, `plus` or `reps` choice points, for this a custom choice model could be used, these can be written in Julia themselves. It might be interesting to give some feedback to the choice model from the testing framework, to "steer" GödelTest generations to

sample datums differently (e.g., more interesting test data), either by choice of the tester or by some automatic process. To accomplish this some feedback mechanism could be made where the unit testing software sends information back to the generator to change the choice model depending on the results of the previously generated test data.

Chapter 4

Methodology and Evaluation

The evaluation of our implemented tool is conducted via an experimental study. Our study covers two experiments using the tool- the first experiment (ES1) investigates the efficiency and effectiveness of our tool compared to other test generation techniques; the second experiment (ES2) captures the efficiency and effectiveness of different settings/configurations of the test generation (e.g. different number of test data generated and test data from differently written generators).

Both experiments consist in: i) creating test generators in the implemented platform, ii) instrumenting the SUTs with a set of faults (e.g., seeded by mutation) as well as monitoring the generation procedure, iii) collect quantitative data on efficiency and effectiveness of the investigated approaches, iv) analyze and present results. The steps of our experimentation process is designed following the guideline mentioned in the book "Experimentation in Software Engineering" [62]. Each step is described in detail in this chapter.

4.1 Scoping and Planning

In order to define the scope of our experiments we use the template proposed by Wohlin et al.[62]. There, our experiments aims to: Analyze automated test generation technique; for the purpose of comparing the trade-offs between automated and manual testing; in terms of fault detection rate and coverage of test input; from the point of view of a test engineer or developer; in the context of unit testing.

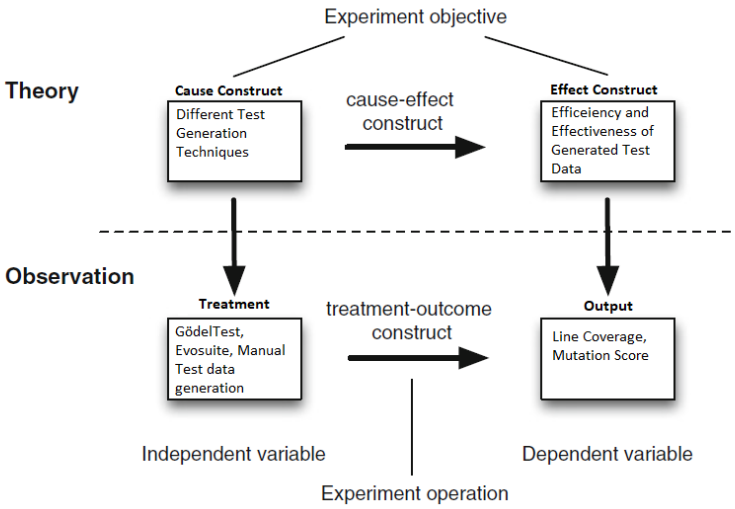

Figure 4.1: Adapted experiment principles for the first experiment [62]

4.2 Context

Since this thesis is not done in collaboration with a company, a lab based experimental study is done consisting of a quantitative approach for checking the efficiency and effectiveness of the developed test data generation tool compared to other techniques.

4.3 Variable selection

4.3.1 Experimental Study 1 - ES1

ES1 has one factor (unit testing technique), with three levels: i) stochastic test generation using GödelTest, ii) unit tests written manually by human developers, and iii) test generating using EvoSuite. Efficiency and effectiveness of stochastic test generation technique implemented in our tool are measured in terms of two dependent variables: i) percentage of mutation detection rate, ii) percentage of line coverage of the test cases. We chose mutation score and line coverage since they have been widely used in literature as constructs to measure effectiveness and efficiency [63]–[65]. Namin and Andrews showed that there is an influence of coverage on the effectiveness of a test suite [66].

Figure 4.1 shows the cause-effect relationship of our first experiment ac-

cording to the basic principles of experiment suggested by Wohlin et al. [62]. Through this experiment we find out how different test generations techniques affect the efficiency and effectiveness of the generated test data which reflects the benefits (if any) of stochastic test data generation and eventually answers our RQ3.

4.3.2 Experimental Study 2 - ES2

Our second experiment aims at finding out if there is any effect of varying number of generated data points or generators designed/written differently on the efficiency and effectiveness and answer to our RQ4. There are two factors in the experiment- different sizes of sampled datums and different designs of generators.

Our hypotheses is whether increasing the number of generated datums would allow the stochastic generation to explore the different areas of the input space. Therefore, our first factor "different sizes of sampled datums" has eight levels- 1, 5, 10, 20, 50, 100, 200 and 500. The specific values were chosen in order to see a progression by nearly doubling the size of the pool. This is an initial investigation, but future studies can include different strategies on how to define the sample sizes.

Our another hypothesis is that the design of the generator affects performance since choice points specified in the generator (choose, mult, plus) trigger the choice models implemented in GödelTest. Consequently, the sampling of datum is affected, however, we aim to observe to what extent that affects the actual testing. Therefore, the second factor "different designs of generators" has three levels- type A, type B and type c which corresponds to three differently written generators generating same type of data. For this run, we did arbitrary design decisions when writing the generators, such as changing the range of types, or choice points. Figure 4.2 shows the three type of generator used for generating an `array` of `Integer` numbers. The first one chooses the numbers with a range of -10000 to 10000. The second one generates the numbers without any range which means any value can be generated within the minimum and maximum limit of `INT` type. Finally the third has three different methods for generating a number from which it randomly chooses one while generating each of the numbers. Figure 4.3 shows the cause-effect relationship of the second experiment.

According to Juristo et. al. [67] this experiment design is termed as 8x2 factorial design (eight alternatives/levels for one variable/factor and two for

```
using DataGenerators
using CSV
using DataFrames
using DataStructures


@generator NumGen begin
    start() = join(numlist(), ',')

    numlist() = reps(num(),1,)
    num() = choose(Int,-100000, 100000)
end

@generator NumGen2 begin
    start() = join(numlist(), ',')

    numlist() = reps(num(),5,)
    num() = choose(Int)
end

@generator NumGen3 begin
    start() = join(numlist(), ',')
    numlist() = reps(num(),5,)
    num() = choose(Int,-10000,10000)
    num() = choose(Int)
    num() = choose(Int,1,100)
end
```

Figure 4.2: Example of different designs of generators for generating same type of data

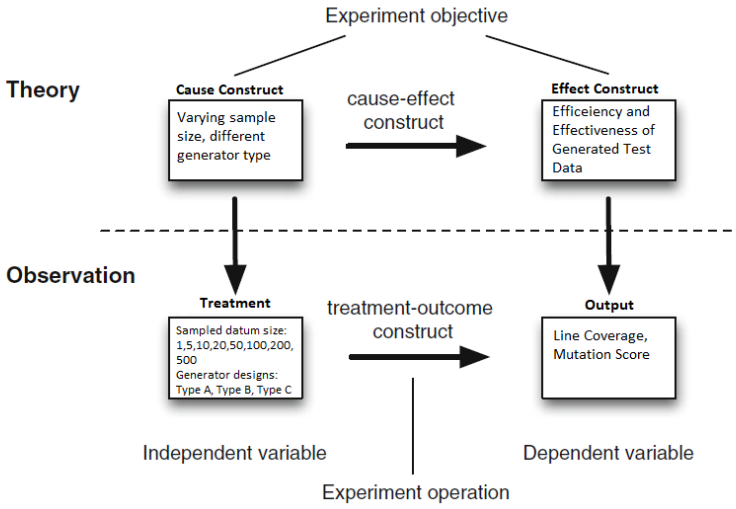

Figure 4.3: Adapted experiment principles for the second experiment [62]

another) which can be described by means of the linear statistical model

$$y_{ijk} = \mu + \alpha_i + \beta_j + (\alpha\beta_{ij}) + e_{ijk}$$

where μ is the grand mean, α_i is the effect of the i^{th} alternative of the row factor (different sizes of sampled datums), β_j is the j^{th} alternative of the column factor (different designs of generators), $\alpha\beta_{ij}$ is the effect of the interaction between α_i and β_j and e_{ijk} is the error associated with the unitary experiment concerned with the i^{th} and j^{th} alternatives.

However, Since we are not exploring the interaction effects of the factors in this study, this experiment is considered as two separate one-factor analysis with K alternatives/level. Mathematical model for one-factor experimental design according to Juristo et. al. [67] is-

$$y_{ij} = \mu + \alpha_j + e_{ij}$$

where y_{ij} is the value of the response variable in the i^{th} observation with the factor valued j (that is, the j^{th} alternative), μ is the mean response, α_j is the effect of the alternative j, and e_{ij} is the error.

10 observations are taken while executing the experiments with each alternative. Due to the call to Julia, each run takes at least several seconds, such that more executions per level could lead to prohibitive experimental execution cycles.

First part of the experiment analyzes the effect of different sizes of sampled datums on line coverage mutation coverage keeping the designs of generators fixed, second part of the experiment analyzes the effect of different designs of generators on line coverage and mutation coverage keeping the sizes of sampled datums constant.

In both experiments, for all the levels, different objects are used to allow the execution of the techniques, and hence the experimental study. Different parts of the SUT is chosen to be tested, since exhaustive coverage requires the creation of too many generators which would compromise the time of the thesis. We decided to control these variables, so that the results can be evaluated in terms of varying the unit testing techniques, as opposed to differences due to using different projects for each technique.

4.4 Hypothesis formulation

Our first experiment (ES1) focuses on the comparison of the line coverage and mutation coverage among different test data generation techniques. The general hypothesis behind the experiment is that the investigated test generation techniques differ in terms of efficiency and effectiveness. However, trial runs during

setup revealed that the performance of different tools is quite stable, hence little to no variance on efficiency and effectiveness was seen during data collection. Consequently, for "ES1" we do not perform statistical tests, rather we provide a comparative analysis of our experiment result.

Our second experiment (ES2) investigates the effect of different sizes of sampled datums and different designs of generators on the efficiency and effectiveness of the generated test data; therefore, the null hypothesis and alternative hypothesis for our second experiment are:

Analysing different numbers for sampling datums:

$H_0 1$**:** There are no differences in GödelTest's **line coverage** (LC) when using various numbers of generated datums;

$$\mu(LC_1) = \mu(LC_5) = \mu(LC_{10}) = \mu(LC_{20}) = \mu(LC_{50}) = \mu(LC_{100}) = \mu(LC_{200}) = \mu(LC_{500})$$

$H_A 1$**:** There are differences in GödelTest's **line coverage** (LC) when using various numbers of generated datums;

$$\mu(LC_1) \neq \mu(LC_5 \neq \mu(LC_{10}) \neq \mu(LC_{20}) \neq \mu(LC_{50}) \neq \mu(LC_{100}) \neq \mu(LC_{200}) \neq \mu(LC_{500})$$

$H_0 2$**:** There are no differences in GödelTest's **mutation coverage** (MC) when using various numbers of generated datums;

$$\mu(MC_1) = \mu(MC_5) = \mu(MC_{10}) = \mu(MC_{20}) = \mu(MC_{50}) = \mu(MC_{100}) = \mu(MC_{200}) = \mu(MC_{500})$$

$H_A 2$**:** There are differences in GödelTest's **mutation coverage** (MC) when using various numbers of generated datums;

$$\mu(MC_1) \neq \mu(MC_5 \neq \mu(MC_{10}) \neq \mu(MC_{20}) \neq \mu(MC_{50}) \neq \mu(MC_{100}) \neq \mu(MC_{200}) \neq \mu(MC_{500})$$

Analysing different designs of generators:

$H_0 3$**:** The fact that different type of data generators are used makes no difference to the line coverage of the stochastic test data generation technique;

$$\mu(LC_{TypeA}) = \mu(LC_{TypeB}) = \mu(LC_{TypeC})$$

$H_A 3$**:** The fact that different type of data generators are used makes a difference to the line coverage of the stochastic test data generation technique;

$$\mu(LC_{TypeA}) \neq \mu(LC_{TypeB}) \neq \mu(LC_{TypeC})$$

$H_0 4$**:** The fact that different type of data generators are used makes no difference to the mutation coverage of the stochastic test data gener-

ation technique;

$$\mu(MC_{TypeA}) = \mu(MC_{TypeB}) = \mu(MC_{TypeC})$$

H_A4**:** The fact that different type of data generators are used makes a difference to the mutation coverage of the stochastic test data generation technique;

$$\mu(MC_{TypeA}) \neq \mu(MC_{TypeB}) \neq \mu(MC_{TypeC})$$

There could be more hypothesis addressing the confounding aspects of the factors (size of generated data and generator design) but considering the complexity of two-factor multi-level test analysis and the time constraint of the thesis, instead of running a two factor analysis we decided to do a fractional factorial analysis by analyzing each factors individually. We leave the analysis of interaction effect of the factors for future work.

4.5 Selection of subjects and objects

Since our implemented tool automatically generates test data and runs the test cases, the experiments are conducted without any human intervention i.e. without the involvement of any subject. The objects for the experiments are software projects which we call our systems under test i.e. our SUTs. Suitable candidates for our SUTs are selected basing on the following qualities:

- An Open-source project: Since the experiment is done in a lab environment and companies often lack reliable fault/failure information to run experimental studies
- A Java project: Since currently the tool has support for Java projects only
- Existing unit test suite: Since we need to compare the efficiency of our tool with manual unit test writing
- Reasonable SUT size: Since we need to instrument the TestAdapters, the test and production code should be understandable to use in a relatively short time span

To observe the usability impact and generalizability of our tool we decided to select SUTs of different complexities. We wanted to see if we get similar results by running the experiment with a small, simple SUT and with a comparatively large, complex SUT. Considering all the selection criteria we came

up with two different projects- 1. sorting-algorithms[1] 2. JFreeChart from Defects4J[2] [68]

Our first SUT is comparatively simple with test cases accepting only `array` of `integer` numbers. The test suite consists of one test class with six different test methods. Our second SUT `JFreeChart` have been widely used to validate software testing techniques. Since it has too many test classes, we chose specific test classes for our experimentation instead of the whole project. We selected the whole `org.jfree.data.statistics` package containing 13 different test classes for the experiment. The chosen test classes have 86 test methods that accept `arrays`, `tuples` of `double` numbers. However, for some of these classes no `JuliaTest` tests has been written due to limited complexity of the CUT(class under test), for these we leave in the normal Junit test as to get a fair comparison in coverage sizes.

4.6 Instrumentation

This is the last step before the execution in which we prepare our SUTs for the experiments. The instrumentation process of our experiment consists of: i) implementing the TestAdapter in each SUT, ii) writing the generators, iii) applying Evosuite in each SUT and iv) seeding faults into the SUTs.

Our first step in the instrumentation phase is to make our tool work with the selected SUTs. Our tool support is added to the SUTs as a dependency. For each SUT, we create the TestAdapters based on the existing unit tests by copying and refactoring the existing unit tests into new test files. This steps ensures consistency between the SUT invocations done by the manual tests and our TestAdapter. Otherwise, we would risk testing the wrong unit of the SUT.

Next step is to write the generators that provide appropriate data to the `TestAdapter`. For the same `TestAdapter`, we have written three different generators (type A, type B and type C) varying the data generation parameters with a view to capture the impact (if any) of generator writing style on the experiment results. Figure 4.2 shows the three type of generator used for generating an `array` of `Integer` numbers. The first one chooses the numbers with a range of -10000 to 10000. The second one generates the numbers without any range which means any value can be generated within the minimum and maximum limit of `INT` type. Finally the third has three different methods for generating a number from which it randomly chooses one while generating each of the numbers.

Evosuite uses the state-of-the-art techniques for test generation and opti-

[1] `https://github.com/murraco/sorting-algorithms/`
[2] `https://github.com/rjust/defects4j`

mization [2] and has achieved the highest score among the other tools in the 7th SBST Java Unit Testing Tool Contest in 2019 [69]. Therefore, to compare the efficiency of our tool with existing automatic test generation techniques, we chose Evosuite. Adding the tool support of Evosuite into our selected SUTs we automatically generate test suites for each test class.

Our final step is to seed faults in the SUTs. There are a number of java mutation testing tool available for fault seeding- μJava[3], Jester[4], Jumble[5], Javalanche[6], Pitest (PIT)[7]. However, among these PITest is open source and actively maintained whereas the other tools are not as widely used as PIT since those are more suitable for specific academic research rather than real software industry [70]. The fact that PIT is supported by both ANT and MAVEN makes it easier to use and it is scalable and fast [71]. PIT has inherent optimization techniques, supports mutant operators, supports configurations and presents the reports in a user-friendly way [71] which makes PIT suitable for our purpose. Therefore, we have configured PITest into our selected SUTs to allow mutations in the source code.

At the end of the steps described above, the experiments were executed. Results of the experiments are presented in the next chapter along with the analysis of the results.

[3] http://cs.gmu.edu/offutt/mujava/
[4] http://jester.sourceforge.net/
[5] http://jumble.sourceforge.net/
[6] http://www.st.cs.uni-saarland.de/mutation/
[7] http://pitest.org/

Chapter 5

Result and Analysis

This chapter presents the results and analysis of the data received from our experiments. The statistical analysis are done in R[1] using R-studio[2] software.

5.1 Experiment 1

In our first experiment manual test data generation technique, automatic test data generation technique with Evosuite and automatic stochastic test data generation technique with our proposed tool `JuliaTest` was applied on both the selected SUTs and data of line coverage percentage and mutation coverage percentage was collected. Whole test suite of SUT-1 containing one test class with six different text methods was used in the experiment. For SUT-2, a specific package of test suite that contains 13 test classes with 86 different test methods was used for the experiment.

The experiment was repeated 10 times and every time, the same values were retrieved. Since there was no variance in the results, we did not go through the statistical analysis for this experiment. However, use of varying test classes for SUT-2 might have a difference in the yielded result. Due to the time limitation it was not possible to include all the test suite packages in the experiment, which therefore should be considered for future work. Table 5.1 contains the data gathered from the experiment.

The experiment result shows that Evosuite's test data generation technique has better line coverage percentage than the rest two techniques for both SUT1

[1] `https://www.r-project.org/`
[2] `https://www.rstudio.com/`

Technique	Line Coverage (%)		Mutation Coverage (%)	
	SUT1	SUT2	SUT1	SUT2
Manual	94.0	70.0	82.0	54.0
Evosuite	100.0	88.0	73.0	22.0
JuliaTest	94.0	70.0	85.0	54.0

Table 5.1: Line coverage and mutation coverage using different techniques (Total number of mutants: 1038)

and SUT2. Though our proposed tool's stochastic test generation technique could not beat Evosuit's line coverage, it provides the same line coverage as the manual test data generation technique.

However, clearly, for both SUT, our proposed tool won the race by finding more faults through mutation testing than the rest. for the first SUT our tool provides 12 percentage point more mutation coverage than Evosuite and 3 percentage point more mutation coverage than the manually generated test data. For the second SUT our tool provided 31 percentage point more mutation coverage than Evosuite while providing the same coverage as manual test data generation. The total number of mutants generated was 1038. We will explore these differences further in our discussion of results (Chapter 6), specially contrasting both the line and mutation coverage.

5.2 Experiment 2

As mentioned before, two factors of our second experiment are analyzed individually, therefore, this experiment has two part. Due to the time limitation, there was not enough time to run the experiment on both the SUTs. We, therefore, chose the second SUT (JFreeChart) for being bigger and more complex to run our experiment on. The following sections describe the statistical analysis of our second experiment data.

5.2.1 Different sizes of generated data

The first factor of this experiment is number of data generated which has 8 levels/alternatives- 1, 5, 10, 20, 50, 100, 200 and 500. During the experiment the other factor (different designs of generators) was kept constant.

We collected the data gathered from the second experiment. Due to size, the data is presented, entirely, in the Appendix, whereas here we discuss the results of the statistical tests i.e. the analysis of line coverage and mutation

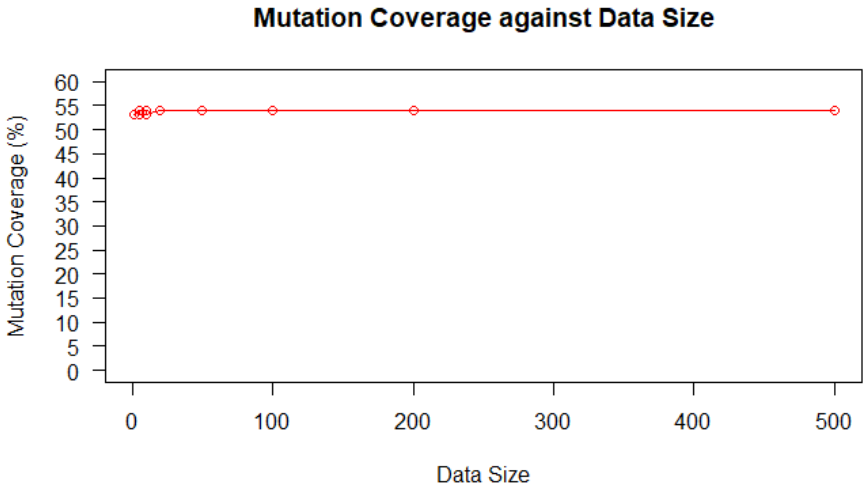

Figure 5.1: Line graph of Mutation coverage against data size

coverage against the generated data size. From the data gathered from the second experiment it can be visually confirmed that the line coverage percentage do not vary depending on the generated data size. For all 80 observations, the line coverage is always the same, therefore, there is no need to perform statistical tests on this data. With this data We have enough evidence to reject the null hypothesis $H_0 1$ in favor of the alternative hypothesis $H_0 1$ concluding that the treatments i.e. the number of data generated do not have any effect on the line coverage.

However, the data for mutation coverage is not exactly the same for all 80 observations. The graph in 5.1 plots the mutant coverage as we increase the number of sampled datums and reveals that there is a small variation in mutation coverage for smaller data size and the value stabilizes with the increasing number of data generated. Figure 5.2 shows the shape of the data distribution and its variability for each alternative group more clearly. To investigate further about this variation we approached to perform statistical analysis.

To perform statistical analysis, it needs to be tested if the data is normally distributed. Among various normality test, we chose Shapiro-Wilk normality test [62] for our purpose. The null hypothesis for this test is that the data is normally distributed. The p-value obtained from performing the test on mutation coverage data is extremely below the α-value with 95% confidence level (0.05) resulting into rejection of the null hypothesis which means that the data is not normally distributed. Therefore, the non-parametric Kruskal-Wallis test is performed on the data since parametric tests assume that the data is

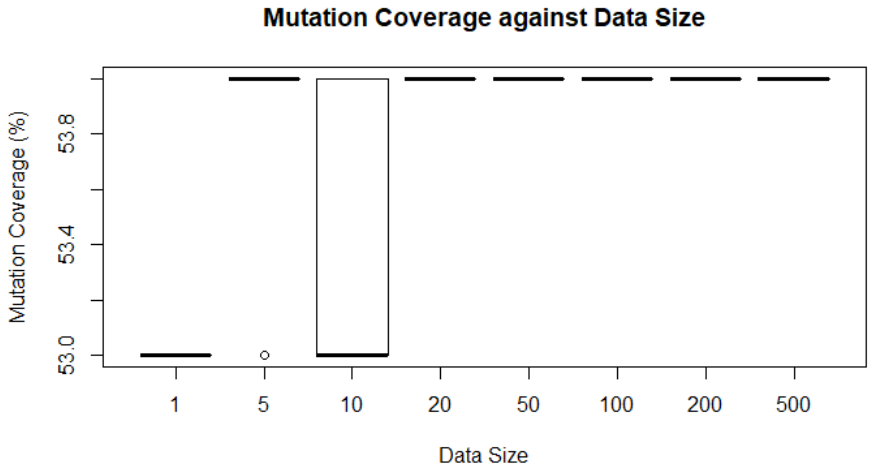

Figure 5.2: Box-plot of Mutation coverage against data size

normally distributed [62].

$$Shapiro-Wilk\ normality\ test\ p-value\ for\ mutation\ coverage\ =\ 8.128e-15$$

$$Kruskal-Wallis\ rank\ sum\ test\ p-value\ for\ mutation\ coverage\ =\ 8.056e-10$$

The p-value obtained from performing Kruskal-Wallis test on mutation coverage against data size is below the α-value with 95% confidence level (0.05), which means it is significant. We, therefore fail to accept our null hypothesis $H_0 2$ and conclude that there is a difference on the mutation coverage depending on the number of data generated. However, the visual analysis show very little variation of the mutation much before 100 datums are used. This is an indication that the difference lies between some of the levels as opposed to all of them (as indicates the Kruskal-Wallis test). Therefore, we did a pair-wise test of the mutation coverage collected in each level.

The pairwise analysis was done using a Bonferroni correction to account for α inflation. The analysis confirms that only specific levels of the factor yield a statistical difference. Moreover, we also used the Vargha-Delaney $\hat{A}_{12}$ metric to reason about the magnitude of such difference (effect-size). The results on Table 5.2 reveals that the difference comes mostly from small sample sizes (i.e., one or five datums generated). Moreover, observing the actual scores on the data (Table 7.1 in the Appendix), the differences are very minimal, hence being not significant in practice.

In short, even though the results reveal that there is a difference, we argue

Pairwise comparison	p-value	Effect size	Effect size
1x5	0.00793	1	large
1x10	0.60461	1	large
1x20	0.00029	1	large
1x50	0.00029	1	large
1x100	0.00029	0.5	negligible
1x200	0.00029	0.5	negligible
1x500	0.00029	0.5	negligible
5x10	1.00000	1	large
5x20	1.00000	1	large
5x50	1.00000	1	large
5x100	1.00000	0.5	negligible
5x200	1.00000	0.5	negligible
5x500	1.00000	0.5	negligible
10x20	0.09029	1	large
10x50	0.09029	1	large
10x100	0.09029	0.5	negligible
10x200	0.09029	0.5	negligible
10x500	0.09029	0.5	negligible

Table 5.2: Pairwise comparison of mutation coverage (Remaining pairwise combinations did not point to any statistical results, because the results had no variance)

that the difference and effect-size are not significant in practice, since unit tests are often faster to execute (specially when compared to higher level tests). In practice, testers would generate thousands or even more datums, leading to a negligible effect-size. Nonetheless, future investigation is required, since we constraint our evaluation to very small parts of the SUT (i.e., a specific package).

5.2.2 Different types of generators used

The second factor of this experiment is the type of generator used for generating the test data. Three levels- generator type A, type B and type C is used in this part of the experiment keeping the other factor (different sizes of sampled datum) fixed.

In this part of the experiment line coverage and mutation coverage is analyzed against the different designs of generators. Like the first part of the experiment, a specific package of SUT-2 is used in the experiment. The three types of generator used for the experiment is described in section 4.6 of Chapter 4.

Table 7.2 in the appendix depicts the data gathered from running the experiment. From the data in table 7.2 it can be seen that there are no change

in line coverage percentage or mutation coverage percentage depending on the different designs of generators as observation values are unchanged with the variation of the alternatives. Therefore, we fail to reject our null hypothesis $H_0 3$ and $H_0 4$.

Results of our experiment is further discussed in detail in the next chapter of this report.

Chapter 6

Discussion

In this chapter we try to answer the research questions (RQs) of the study by discussing our experiences through the thesis work, elaborating the findings from the experiment results and statistical tests, analyzing potential reasons behind the findings and evaluating the validity threats of the experiments. Along with the discussions we also commenced some concepts for future works that may thrive from this study.

6.1 Answer to the RQs

RQ1: How can we instrument JUnit Frameworks to enable automated and stochastic test generation?

With a view to answer this question we created our proposed tool support "`JuliaTest`" that implements an automated and stochastic test generation framework GödelTest and is instrumented with the JUnit framework.

One of the main enablers of our instrumentation was the use of automated builds (Maven) and a TestAdapter to allow communication between the framework to generate tests (GödelTest) and the SUT. Testers should also follow good practices in separating test and production code, along with support from an existing xUnit Framwork (in our case, JUnit) so other aspects of the automation can be handled by the framework itself (e.g., a test Runner and the assertion mechanisms).

The architecture along with other details of the tool is described in detail in Chapter 3. Some ideas on different design and architectural decisions that

could have done differently and the pros and cons of those decisions are also mentioned in Chapter 3.

RQ2: What are the challenges in applying stochastic test data generation to unit testing frameworks?

The experience we had while creating our proposed tool has enabled us to point out some of the challenges in applying stochastic test data generation to unit testing frameworks. One challenge is that the testers need to create an adapter to make the communication between GödelTest generators and the JUnit framework. However, writing an adapter needs only the basic knowledge of java and some understanding of the JUnit 5 annotation property which is assumed to be familiar to most test engineers.

Another challenge we noticed is the overhead that is added to the system for calling the generators of GödelTest written in the language Julia. This challenge can however be further be addressed by using the `JavaCall` library of Julia and then investigating the performance of the tool in future works. Another interesting future aspect of this part of the study would be to extend the tool support for other XUnit frameworks or other languages (e.g., python, C#).

While searching characteristics of JUnit Framework that can foster usage of stochastic test generation we found the JUnit 5 `annotation` property very useful for making the connection between GödelTest generators and JUnit test cases. Also, we exploited the assertion property of JUnit. At present testers sometimes create several tests- each containing one assertion for different test inputs- whereas in our tool one test method can take 100s, 1000s or more different test inputs having the same assertion.

Another challenge we faced during the implementation of our proposed tool is the oracle problem. Different types of oracle entails different design choices, since the expected values are needed to instrument assertions. We used a differential oracle in order to solve the oracle problem in our case. To use differential oracles, we had to adapt the architecture of our tool, we had to make our testing tool write and read the generated test datums to a file and also write and read the results of those tests so they can be used by the differential oracle to see if there has been a regression or not.

	Juliatest				Manual				Evosuite			
Name	Line Coverage		Mutation Coverage		Line Coverage		Mutation Coverage		Line Coverage		Mutation Coverage	
BoxAndWhiskerCalculator.java	88%	61/69	62%	33/53	88%	61/69	55%	29/53	99%	68/69	8%	4/53
BoxAndWhiskerItem.java	71%	35/49	57%	20/35	71%	35/49	57%	20/35	100%	49/49	11%	4/35
DefaultBoxAndWhiskerCategoryDataset.j	60%	128/212	51%	57/111	60%	128/212	51%	57/111	72%	153/212	29%	32/111
DefaultBoxAndWhiskerXYDataset.java	78%	88/113	68%	36/53	78%	88/113	68%	36/53	84%	95/113	0%	0/53
DefaultMultiValueCategoryDataset.java	68%	67/99	41%	28/68	68%	67/99	41%	28/68	87%	86/99	0%	0/68
DefaultStatisticalCategoryDataset.java	78%	150/192	52%	66/127	78%	150/192	52%	66/127	100%	192/192	49%	62/127
HistogramBin.java	80%	20/25	68%	15/22	80%	20/25	68%	15/22	100%	25/25	0%	0/22
HistogramDataset.java	79%	86/109	63%	54/86	79%	86/109	63%	54/86	92%	100/109	0%	0/86
MeanAndStandardDeviation.java	93%	25/27	88%	14/16	93%	25/27	88%	14/16	100%	27/27	25%	4/16
Regression.java	53%	92/174	45%	88/197	53%	92/174	45%	88/197	69%	120/174	0%	0/197
SimpleHistogramBin.java	83%	52/63	66%	41/62	86%	54/63	73%	45/62	98%	62/63	0%	0/62
SimpleHistogramDataset.java	60%	52/87	35%	20/57	60%	52/87	35%	20/57	93%	81/87	0%	0/57
Statistics.java	66%	98/148	58%	87/151	66%	98/148	58%	87/151	98%	145/148	79%	119/151

Figure 6.1: Detail comparison of line coverage and mutation coverage among the three techniques for SUT-2

RQ3: What are the trade-offs between different test generation techniques for unit testing?

We tried to answer this question through our first experiment, the result of which is reported in section 5.1 of Chapter 5. In the experiment Evosuite's performance was better for line coverage for both SUTs. However, for mutation coverage our proposed tool's technique and manual test techniques are pretty close where Evosuite left far behind. To understand the difference better Figure 6.1 shows the class-wise comparison among the techniques for SUT-2. From the figure we can see that for all the test classes Evousuite's mutation coverage is less than the other two, even 0% for some classes.

Since we ran Evosuite with it's default parameters which tries to optimize for certain goals. Among the goals the most important one is line coverage followed by branch coverage, so it makes sense that Evosuite scores high in that regard, while scoring a lesser score for mutation coverage. The reason for choosing the default setting on Evosuite is since this is probably the configuration most people use it as, and therefore seems like a fair setting for our experiment, further experimentation with different configuration can be explored in future work. Perhaps, if Evosuite would have been run on a SUT with less interaction between different objects that the test cases might have scored higher in the mutation coverage, since the mutation coverage score of Evosuite for SUT 1 was a lot higher. In figure 6.1 we can see how Evosuite compares on a class basis. It shows that that for seven different classes Evosuite has a mutation score of 0. After examining the test classes it was identified that all these classes take complex data types like instances of Comparable (i.e., a Java Comparable object) as an input for themselves. Evosuite does seem capable of filling this in but generation of good test data for the other methods which use the variables sat in the constructor seems to be not possible.

It also seems that Evosuite had a hard time with generating meaningful tests for SUT 2 since it is a complex scenario (as opposed to SUT 1 where it performs better but takes very simple data types). A number of test cases where

```
JuliaTest:
- Mutators
=====================================
ConditionalsBoundaryMutator
Generated 101 Killed 24 (24%)
KILLED 24 SURVIVED 44 NO_COVERAGE 33
-------------------------------------
IncrementsMutator
Generated 35 Killed 16 (46%)
KILLED 16 SURVIVED 0 NO_COVERAGE 19
-------------------------------------
VoidMethodCallMutator
Generated 88 Killed 26 (30%)
KILLED 26 SURVIVED 18 NO_COVERAGE 44
-------------------------------------
ReturnValsMutator
Generated 256 Killed 132 (52%)
KILLED 132 SURVIVED 5 NO_COVERAGE 119
-------------------------------------
MathMutator
Generated 239 Killed 136 (57%)
KILLED 136 SURVIVED 19 NO_COVERAGE 84
-------------------------------------
NegateConditionalsMutator
Generated 319 Killed 221 (69%)
KILLED 221 SURVIVED 35 NO_COVERAGE 63
-------------------------------------
```

```
evosuite:
- Mutators
=================================================
ConditionalsBoundaryMutator
Generated 101 Killed 13 (13%)
KILLED 13 SURVIVED 71 NO_COVERAGE 17
-------------------------------------------------
IncrementsMutator
Generated 35 Killed 8 (23%)
KILLED 7 SURVIVED 17 TIMED_OUT 1 NO_COVERAGE 10
-------------------------------------------------
VoidMethodCallMutator
Generated 88 Killed 17 (19%)
KILLED 17 SURVIVED 66 NO_COVERAGE 5
-------------------------------------------------
ReturnValsMutator
Generated 256 Killed 60 (23%)
KILLED 59 SURVIVED 156 TIMED_OUT 1 NO_COVERAGE 40
-------------------------------------------------
MathMutator
Generated 239 Killed 60 (25%)
KILLED 60 SURVIVED 142 NO_COVERAGE 37
-------------------------------------------------
NegateConditionalsMutator
Generated 319 Killed 67 (21%)
KILLED 65 SURVIVED 220 TIMED_OUT 2 NO_COVERAGE 32
-------------------------------------------------
```

Figure 6.2: Comparison of killed mutations by Evosuite and Juliatest for SUT-2

manually changed since they seemed to be unstable (e.g. one test cases generated by Evosuite created a list of a million items inside of an object which caused it to throw a out of memory inconsistently). For this case the passed variable determining the size was changed without changing the other functionality of the test.

To see if the tests generated with Evosuite are simply incapable at finding some specific kinds of faults we also analyzed the output from PITest(see figure 6.2). Here we can see what types of mutations PItest created for the SUT, in total it created a 1038 mutations of the following categories:

- ConditionalsBoundaryMutator - changes a conditional boundary e.g i $<$ 10 gets changed to i $>$ 10
- IncrementsMutator - changes the increment of a local variable e.g. i++ gets changed to i–;
- VoidMethodCallMutator - removes calls to methods void returing methods
- ReturnValsMutator - changes the returned variable of a method e.g. return x; gets changed to return 0;
- MathMutator - changes arithmatic operators to its inverse e.g multiplication gets changed to a division
- NegateConditionalsMutator - changes a conditional operator e.g == gets changed to !=

From figure 6.2 it is clear that Evosuite does not excel or do poorly in a single category, rather it scores about half of what `JuliaTest` does for *all types of mutators*. Evosuite uses genetic algorithm with the source code for creating

tests whereas `JuliaTest` being black-box has no dependency on the source code, all it needs is the input specification. We understand that Evosuite is highly configurable, but to do so a good knowledge of Java is required whereas to use `JuliaTest` the testers needs to know only the basic JUnit for adapters. Though a very basic knowledge of Julia is also required for writing the generators, it is easier to grasp since Julia is a high level language. The generator needs to be created once and it will take care of many test cases. New generator needs to be created only if the test specification changes.

Considering the points mentioned above we argue that performance-wise and usage-wise `JuliaTest` is a better alternative than Evosuite. However, future studies should investigate different configurations of Evosuite. Furthermore, another interesting future work can be to replay the experiment on projects containing real faults (e.g.,defect4J) instead of using mutants to see how many of the faults are captured by manual test data generation technique, `JuliaTest` and Evosuite.

RQ4: What are the effects of different settings/setups in stochastic test generation?

Our second experiment reported in section 5.2 of Chapter 5 aims at answering this question. Our second experiment consists of two parts- first of which analyses the effect of line coverage and mutation coverage on data size i.e. number of data generated. From the results in section 5.2 of Chapter 5 it is obvious that line coverage do not change depending on the different number of data generated. However, we see a small difference in mutation coverage for varying data size. Mutation coverage percentage vary a little until the data size 10. From the data size 20 mutation coverage value stabilizes no matter how many more data is generated. Therefore we identify 20 as the elbow value after which the coverage goes flat meaning that generating 20 input values should be enough to get optimal mutation coverage. Nonetheless, we argue that in practice this change will not be relevant since testers can generate larger unit test suites and still run all tests. However, since the first SUT used in the experiment is very simple which might have an effect on the result, we do not claim the result to be generalized. This is a threat to validity to our study and is open for further experimentation.

The second part of our experiment analyzed the change in line coverage and mutation coverage for different types of generators. Three different type of generators were used on a specific package of SUT-2 (`org.jfree.data.xy`). Our experiment result showed that there is no difference in the line coverage and mutation coverage depending on how the generator is written. However, GödelTest provides four choice points exploiting which a generator can be written in a number of different styles. In our experiment we tried to cover generators written with different ranges which resulted in not having any effect on line coverage

or mutation coverage. Exploring other aspects of generator writing style could not be covered in this study due to the time limitation. Experimentation with these different aspects may result into different outcome from our experiment and therefore, is an interesting candidate for future study.

However, the experiment was done using only one SUT. Though the SUT used is moderately complex, it might be interesting to perform the experiment using multiple SUTs. Analyzing the interaction effect of the two factors of the experiment may also result in interesting findings. Furthermore, considering the SUTs as factors of the experiment observe the change in line coverage and mutation coverage for varying SUTs could be another future work from this study.

6.2 Validity evaluation

Validity of a research investigates the question of how close the conclusion of the study is to the reality and validity threats are the specific ways in which the conclusion might be wrong [72]. There are namely four types types of validity threats- external, conclusion, internal and construct [62].

External validity threat questions the generalizability of the findings of the study outside the scope [62]. Due to the time limitation of the thesis our inability to generalize the result of the experiment into industry practice is one external validity in our case. However, considering Gorscheck's et al. model [73] for technology transfer, this study lies in step 4 (validation in academia), whereas validation with actual industry partners for this study would be Step 5 or higher (static validation), which is aimed for future work. Another external validity threat to our experiments is that we used only two SUTs which may not be sufficient for generalizing the conclusions from our experiment results. Due to time limitation it is not possible for us to include more SUTs in the experiments. Moreover, We, therefore, claim our conclusions to be the the first implication of the potential of stochastic test data generation technique and is widely open for further investigations.

The focus of internal validity concerns if the experiment actually measure the cause-effect construct properly ensuring that no other factors but the treatments are solely responsible for the observed outcome [62]. In our experiments we used a 3rd party mutation testing software called PITest for fault seeding, hence the mutation score obtained from PIT is subject to internal validity. To mitigate this validity threat the output of each step is checked manually. In case of any unusual outcome, the reason was investigated and the whole execution was restarted if necessary. Furthermore, the fact that in our experiment Evosuite was not used in its full capacity is another internal validity threat to our first experiment. Due to time concern instead testing with different config-

urations to optimize the performance of Evosuite we used it with the default configuration. Though we cannot mitigate this validity threat at this moment, future studies should address this aspect.

Conclusion validity threat is concerned about if the treatments used in the experiment are actually related to and have statistical significance on the observed outcome [62]. One conclusion validity threat to our experiment two is that the interaction effect of the two factors are not analyzed, instead, the factors are analyzed individually. Unfortunately this validity threat cannot be mitigated in this study due to the scarcity of time but this analysis is intended to be included in future work. Another conclusion validity threat lies in our experiment two with the choice of statistical analysis method. We mitigate this threat by using non-parametric tests which are more conservative and less constrained by assumptions regarding data distribution.

Finally, construct validity threat focuses on if the right constructs (e.g., techniques, dependent variables, projects) are being chosen i.e. the correspondence between the treatments and the cause of interest [62]. Since the SUTs for our experiments are open source projects, a threat to construct validity for our experiment is that the chosen SUT might be atypical in the quality of test suites. To mitigate this threat, we chose two different SUTs with different levels of complexity. However, the time limit of the thesis period did not allow as to consider more variety in the SUTs.

Chapter 7

Conclusion

Stochastic test data generation is one of the dimensions of test automation that can add values to the software testing field- with this assumption we conducted this thesis work putting a focus on investigating how well stochastic test data generation technique performs with unit testing of open-source projects. We implemented a tool called `JuliaTest` that combines stochastic test generation framework GödelTest with existing Java unit testing framework JUnit. During the process we not only identified some challenges involved with generating JUnit tests using the GödelTest framework but also figured out how we can exploit the JUnit5 `Annotation` property for our purpose. Current version of our tool supports Java Maven projects and can be run via a single maven command to generate as many input data as wish. Data types supported by the tools are- Integer, Double and String. Another Interesting feature of our tool is that it supports creating JSON data from a JSON schema. However, this feature is still in the basic stage which is planned to be developed further in future works.

In order to evaluate our tool's efficiency and effectiveness in terms of line coverage and mutation coverage, we ran an experiment comparing it with manual test data generation and one of the most popular automatic test generation tool- Evosuite. Our experiment showed that our tool provides almost the same line and mutation coverage as manual testing but 31 percentage point more mutation coverage than Evosuite for a moderately complex test suite of 13 test classes and 86 test methods.

To take the evaluation of our tool further, we conducted another experiment investigating if there is an effect of varying number of test data generated or different type of generator used on the performance of the tool. Our experiment did not find any effect of varied data size or differently written generators on the line coverage. However, for optimal mutation coverage the study found the

suitable number of data size to be 20.

Stochastic test data generation is a field of immense possibilities, we tried to investigate a small part of it. There are a lot of scopes to take this study further and conduct experiments from different angles. Some concepts of related future studies have been mentioned in different parts of this report which we think is another implicit contribution of our study. Furthermore, since our study has covered the instrumentation process of stochastic test data generation technique by implementing GödelTest in the proposed tool, it is now easier to generate data of more complex data types which eventually will make the tool compatible to be used in real-life industrial projects. This study allowed us to take a small step into the world of stochastic test data generation which, we believe, with time and more effort will become sophisticated and matured enough to be used industrially by the test engineers.

References

[1] R. Ramler and K. Wolfmaier, "Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost," in *Proceedings of the 2006 international workshop on Automation of software test*, ACM, 2006, pp. 85–91.

[2] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ACM, 2011, pp. 416–419.

[3] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, ACM, 2007, pp. 815–816.

[4] A. Mista, A. Russo, and J. Hughes, "Branching processes for quickcheck generators," in *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, ACM, 2018, pp. 1–13.

[5] R. Feldt and S. Poulding, "Finding test data with specific properties via metaheuristic search," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, IEEE, 2013, pp. 350–359.

[6] S. Poulding and R. Feldt, "Generating structured test data with specific properties using nested monte-carlo search," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, ACM, 2014, pp. 1279–1286.

[7] S. Poulding and R. Feldt, "The automated generation of humancomprehensible xml test sets," in *Proc. 1st North American Search Based Software Engineering Symposium (NasBASE)*, 2015.

[8] N. Havrikov, M. Höschele, J. P. Galeotti, and A. Zeller, "Xmlmate: Evolutionary xml test generation," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2014, pp. 719–722.

[9] R. Feldt, S. Poulding, D. Clark, and S. Yoo, "Test set diameter: Quantifying the diversity of sets of test cases," in *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, IEEE, 2016, pp. 223–233.

[10] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, IEEE, 2016, pp. 523–534.

[11] 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)*, vol. 24, no. 4, p. 23, 2015.

[12] D. Huizinga and A. Kolawa, *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.

[13] *Junit 5*, `https://junit.org/junit5/`, Accessed: 2019-03-28.

[14] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.

[15] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *Journal of global optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[16] G. Grieco, M. Ceresa, A. Mista, and P. Buiras, "Quickfuzz testing for fun and profit," *Journal of Systems and Software*, vol. 134, pp. 340–354, 2017.

[17] N. Mitchell, "Deriving generic functions by example," in *Proc. York Doctoral Symposium*, Citeseer, 2007, pp. 55–62.

[18] C. Runciman, M. Naylor, and F. Lindblad, "Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values," in *Acm sigplan notices*, ACM, vol. 44, 2008, pp. 37–48.

[19] K. Claessen, J. Duregård, and M. H. Pałka, "Generating constrained random data with uniform distribution," *Journal of Functional Programming*, vol. 25, 2015.

[20] J. Christiansen and S. Fischer, "Easycheck—test data for free," in *International Symposium on Functional and Logic Programming*, Springer, 2008, pp. 322–336.

[21] A. Mista and A. Russo, "Generating random structurally rich algebraic data type values," in *Proceedings of the 14th International Workshop on Automation of Software Test*, IEEE Press, 2019, pp. 48–54.

[22] P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer, "Boltzmann samplers for the random generation of combinatorial structures," *Combinatorics, Probability and Computing*, vol. 13, no. 4-5, pp. 577–625, 2004.

[23] F. G. de Oliveira Neto, R. Feldt, R. Torkar, and P. D. Machado, "Searching for models to evaluate software technology," in *Proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering*, IEEE Press, 2013, pp. 12–15.

[24] M. Bendkowski, O. Bodini, and S. Dovgal, "Polynomial tuning of multiparametric combinatorial samplers," in *2018 Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, SIAM, 2018, pp. 92–106.

[25] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness?" In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ACM, 2011, pp. 265–275.

[26] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Transactions on software Engineering*, no. 7, pp. 703–711, 1991.

[27] Y. K. Malaiya, "Antirandom testing: Getting the most out of black-box testing," in *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, IEEE, 1995, pp. 86–95.

[28] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Annual Asian Computing Science Conference*, Springer, 2004, pp. 320–329.

[29] C. Mao, T. Y. Chen, and F.-C. Kuo, "Out of sight, out of mind: A distance-aware forgetting strategy for adaptive random testing," *Science China Information Sciences*, vol. 60, no. 9, p. 092 106, 2017.

[30] P. Bueno, W. E. Wong, and M. Jino, "Improving random test sets using the diversity oriented test data generation," in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, ACM, 2007, pp. 10–17.

[31] E. G. Cartaxo, P. D. Machado, and F. G. O. Neto, "On the use of a similarity function for test case selection in the context of model-based testing," *Software Testing, Verification and Reliability*, vol. 21, no. 2, pp. 75–100, 2011.

[32] H. Hemmati, A. Arcuri, and L. Briand, "Empirical investigation of the effects of test suite properties on similarity-based test case selection," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, IEEE, 2011, pp. 327–336.

[33] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, "Searching for cognitively diverse tests: Towards universal test diversity metrics," in *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, IEEE, 2008, pp. 178–186.

[34] R. Feldt and S. Poulding, "Searching for test data with feature diversity," *arXiv preprint arXiv:1709.06017*, 2017.

[35] B. Marculescu and R. Feldt, "Finding a boundary between valid and invalid regions of the input space," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2018, pp. 169–178.

[36] P. P. Mahadik and D. Thakore, "Survey on automatic test data generation tools and techniques for object-oriented code," *Int. J. Innovat. Res. Comput. Commun. Eng*, vol. 4, pp. 357–364, 2016.

[37] *Randop*, `https://randoop.github.io/randoop/`, Accessed: 2019-05-29.

[38] *Evosuite*, `http://www.evosuite.org/evosuite/`, Accessed: 2019-05-29.

[39] T. Xie, "Improving automation in developer testing: State of practice," *North Carolina State University, Tech. Rep*, 2009.

[40] K. Lakhotia, M. Harman, and H. Gross, "Austin: A tool for search based software testing for the c language and its evaluation on deployed automotive systems," in *2nd International Symposium on Search Based Software Engineering*, IEEE, 2010, pp. 101–110.

[41] M. Papadakis and N. Malevris, "Metallaxis: An automated framework for weak mutation," *Department of Informatics, Athens University of Economics and Business Athens, Greece*,

[42] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[43] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 4–22, 2014.

[44] F. Pastore, L. Mariani, and G. Fraser, "Crowdoracles: Can the crowd solve the oracle problem?" In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, IEEE, 2013, pp. 342–351.

[45] M. D. Davis and E. J. Weyuker, "Pseudo-oracles for non-testable programs," in *Proceedings of the ACM'81 Conference*, ACM, 1981, pp. 254–257.

[46] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on software engineering*, no. 12, pp. 1491–1501, 1985.

[47] ——, "On the implementation of n-version programming for software fault tolerance during execution," *Proc. COMPSAC, 1977*, pp. 149–155, 1977.

[48] R. Feldt, "Generating diverse software versions with genetic programming: An experimental study," *IEE Proceedings-Software*, vol. 145, no. 6, pp. 228–236, 1998.

[49] W. B. Langdon, S. Yoo, and M. Harman, "Inferring automatic test oracles," in *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, IEEE, 2017, pp. 5–6.

[50] F. Chan, T. Chen, S. C. Cheung, M. Lau, and S. Yiu, "Application of metamorphic testing in numerical analysis," in *Proceedings of the IASTED International Conference on Software Engineering (SE'98)*, 1998, pp. 191–197.

[51] L. K. Dillon, "Automated support for testing and debugging of real-time programs using oracles," *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 1, pp. 45–46, 2000.

[52] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[53] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[54] N. Polikarpova, I. Ciupa, and B. Meyer, "A comparative study of programmer-written and automatically inferred contracts," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ACM, 2009, pp. 93–104.

[55] K. Shrestha and M. J. Rutherford, "An empirical evaluation of assertions as oracles," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, IEEE, 2011, pp. 110–119.

[56] M. Staats, P. Loyola, and G. Rothermel, "Oracle-centric test case prioritization," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, IEEE, 2012, pp. 311–320.

[57] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[58] W. McKeeman, "Differential testing for software, digital tech," *J*, vol. 10, pp. 100–107, 1998.

[59] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno, "Privacy oracle: A system for finding application leaks with black box differential testing," in *Proceedings of the 15th ACM conference on Computer and communications security*, ACM, 2008, pp. 279–288.

[60] B. Marculescu, R. Feldt, R. Torkar, and S. Poulding, "Transferring interactive search-based software testing to industry," *Journal of Systems and Software*, vol. 142, pp. 156–170, 2018.

[61] F. G. d. O. Neto, R. Feldt, L. Erlenhov, and J. B. d. S. Nunes, "Visualizing test diversity to support test optimisation," *arXiv preprint arXiv:1807.05593*, 2018.

[62] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[63] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," *Empirical Software Engineering*, vol. 22, no. 2, pp. 852–893, 2017.

[64] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, IEEE, 2015, pp. 560–564.

[65] R. Gopinath and E. Walkingshaw, "How good are your types? using mutation analysis to evaluate the effectiveness of type annotations," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2017, pp. 122–127.

[66] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ACM, 2009, pp. 57–68.

[67] N. Juristo and A. M. Moreno, *Basics of software engineering experimentation*. Springer Science & Business Media, 2013.

[68] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, 2014, pp. 437–440.

[69] A. Panichella, J. Campos, and G. Fraser, "Evosuite at the sbst 2019 tool competition," 2019.

[70] *Pitest*, `http://pitest.org/`, Accessed: 2019-05-08.

[71] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: A practical mutation testing tool for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ACM, 2016, pp. 449–452.

[72] J. A. Maxwell, *Qualitative research design: An interactive approach*. Sage publications, 2012, vol. 41.

[73] S. L. T. Gorschek P. Garre and C. Wohlin, "A model for technology transfer in practices," *IEEE Software*, no. november/december, pp. 88–94, 2006.

Appendix

Sample Size	Line Coverage (%)	Mutation Coverage (%)
G1-1	70.0	53.0
G1-2	70.0	53.0
G1-3	70.0	53.0
G1-4	70.0	53.0
G1-5	70.0	53.0
G1-6	70.0	53.0
G1-7	70.0	53.0
G1-8	70.0	53.0
G1-9	70.0	53.0
G1-10	70.0	53.0
G5-1	70.0	54.0
G5-2	70.0	54.0
G5-3	70.0	54.0
G5-4	70.0	53.0
G5-5	70.0	54.0
G5-6	70.0	54.0
G5-7	70.0	54.0
G5-8	70.0	54.0
G5-9	70.0	53.0
G5-10	70.0	54.0
G10-1	70.0	54.0
G10-2	70.0	54.0
G10-3	70.0	53.0
G10-4	70.0	54.0
G10-5	70.0	53.0
G10-6	70.0	53.0
G10-7	70.0	53.0
G10-8	70.0	54.0
G10-9	70.0	53.0
G10-10	70.0	53.0
G20-1	70.0	54.0
G20-2	70.0	54.0
G20-3	70.0	54.0
G20-4	70.0	54.0
G20-5	70.0	54.0
G20-6	70.0	54.0
G20-7	70.0	54.0
G20-8	70.0	54.0
G20-9	70.0	54.0
G20-10	70.0	54.0
G50-1	70.0	54.0
G50-2	70.0	54.0
G50-3	70.0	54.0
G50-4	70.0	54.0
G50-5	70.0	54.0
G50-6	70.0	54.0
G50-7	70.0	54.0
G50-8	70.0	54.0
G50-9	70.0	54.0
G50-10	70.0	54.0
G100-1	70.0	54.0
G100-2	70.0	54.0
G100-3	70.0	54.0
G100-4	70.0	54.0
G100-5	70.0	54.0
G100-6	70.0	54.0
G100-7	70.0	54.0
G100-8	70.0	54.0
G100-9	70.0	54.0
G100-10	70.0	54.0
G200-1	70.0	54.0
G200-2	70.0	54.0
G200-3	70.0	54.0
G200-4	70.0	54.0
G200-5	70.0	54.0
G200-6	70.0	54.0
G200-7	70.0	54.0
G200-8	70.0	54.0
G200-9	70.0	54.0
G200-10	70.0	54.0
G500-1	70.0	54.0
G500-2	70.0	54.0
G500-3	70.0	54.0
G500-4	70.0	54.0
G500-5	70.0	54.0
G500-6	70.0	54.0
G500-7	70.0	54.0
G500-8	70.0	54.0
G500-9	70.0	54.0
G500-10	70.0	54.0

Table 7.1: Line coverage and mutation coverage using JuliaTest with varying number of data generated for SUT-1

Generator Type	Line Coverage (%)	Mutation Coverage (%)
GA-1	24.0	9.0
GA-2	24.0	9.0
GA-3	24.0	9.0
GA-4	24.0	9.0
GA-5	24.0	9.0
GA-6	24.0	9.0
GA-7	24.0	9.0
GA-8	24.0	9.0
GA-9	24.0	9.0
GA-10	24.0	9.0
GB-1	24.0	9.0
GB-2	24.0	9.0
GB-3	24.0	9.0
GB-4	24.0	9.0
GB-5	24.0	9.0
GB-6	24.0	9.0
GB-7	24.0	9.0
GB-8	24.0	9.0
GB-9	24.0	9.0
GB-10	24.0	9.0
GC-1	24.0	9.0
GC-2	24.0	9.0
GC-3	24.0	9.0
GC-4	24.0	9.0
GC-5	24.0	9.0
GC-6	24.0	9.0
GC-7	24.0	9.0
GC-8	24.0	9.0
GC-9	24.0	9.0
GC-10	24.0	9.0

Table 7.2: Line coverage and mutation coverage using JuliaTest with varying generator type