Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Evaluating Code-based Test Generator Tools

M.Sc. Thesis

Author
Lajos Cseppentő

Supervisor
Zoltán Micskei, Ph.D.

May 26, 2016

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Cseppentő Lajos*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2016. május 26.

Cseppentő Lajos
hallgató

Abstract

Software testing is one of the most common way of software verification. Thorough testing is a resource demanding activity, thus, automation of its phases receives high priority in both academia and industry. This might either mean the automated execution of test cases (which is already widespread) or even involve the generation of test cases or test inputs.

There are several techniques that are capable of selecting test inputs based on the source code of the application under test, these are called code-based test input generator tools. In recent years several (mainly prototype) tools have been created based on these techniques and several attempts have been already made to put them in industrial practice. Experiences show that the available tools considerably vary in capabilities and readiness.

The further spread of test input generator tools requires the assessment and evaluation of their competencies. One possible method for this is to create a code base containing the language constructs that are commonly used. With the help of such a code base it is possible to investigate the tools and compare their capabilities.

In the thesis a framework is presented which supports the creation of such code bases, is able to perform test generation using five test input generator tools and to carry out automated evaluation. In addition, the research results achieved using the framework will be also discussed.

Kivonat

A szoftverellenőrzés egyik legelterjedtebb módja a tesztelés. Az alapos tesztelés azonban erőforrás-igényes tevékenység, ezért kiemelten fontos kutatási és ipari feladat a tesztelés különböző fázisainak az automatizálása. Ez jelentheti a tesztek automatikus végrehajtását – ami ma már széles körben elterjedt – de akár a tesztesetek vagy tesztbemenetek generálását is.

Több olyan módszer is létezik, amely tesztbemeneteket választ ki a tesztelendő programkód felhasználásával, ezeket kód alapú tesztbemenet-generaló eszközöknek nevezzük. Ezekhez a módszerekhez több, főleg prototípus eszköz készült az utóbbi években, amelyeknek az ipari felhasználásával is többen próbálkoztak már. A tapasztalatok alapján azonban a rendelkezésre álló eszközök jelentősen eltérnek egymástól mind tudásukban, mind kiforrottságukban.

A kód alapú tesztbemenet-generáló eszközök további elterjedéséhez szükséges azok tudásának pontos felmérése és kiértékelése. Erre mód egy olyan kódbázis összeállítása, amely a gyakorlatban is fontos programelemeket tartalmazza, majd ezek segítségével lehetséges az eszközök vizsgálata és azok képességeinek összehasonlítása.

A diplomamunkámban egy általam fejlesztett keretrendszert mutatok be, amely támogatja hasonló kódbázisok létrehozását, képes öt tesztbemenet-generáló eszközzel ezekhez tesztgenerálást futtatni, illetve automatizált kiértékelést is tud végezni. A dolgozatomban emellett ismertetem az aktuális kiértékelési eredményeket is.

Chapter 1

Introduction

Software testing is a major field of software development, since testing is a widespread way to find flaws during development and prevent them in the final products. Several current industrial practices are based on intensive testing (e.g., *continuous integration/delivery/deployment*), hence the highest possible level automation is preferred in order to cut development costs and provide better software quality.

The idea of automatizing not only test execution, but also test generation was first proposed in the 1970s. At that time due to the lack of enough processing power and memory no industrial solutions were available, however, now four decades later dozens of tools are aiming to solve this problem and some are already advertised to be used by software developers.

However, these tools are not perfect and most of them are not ready to be used in practice. My related research focused on comparing such automated test generation tools which are based on the source code or bytecode of the program. To aid this research with automated experiment execution, I developed a framework which is the subject of the thesis.

1.1 Problem Statement

During my former involvement with test input generator tools I have found out that they are unable to handle several common situations. At that time no comparison was available which analysed the tools based on what they support and they do not. Hence, I have elaborated an evaluation methodology with which symbolic execution-based test input generator tools may be compared. This methodology is based on short programs called *code snippets* for which test input generators should generate parameter values or test cases.

In the last two years with my supervisor we investigated five Java and one .NET tool with 363 code snippets under different type of experiments and executed more than 45 000 test generations. In order to produce this amount of data, not only the tool execution needed to be automated, but also other parts of the evaluation, such as coverage and mutation analysis. The automation of these tasks with a proper software is able to provide consistent and valid results during the analysis.

From the researcher's point of view, the best be if they could pass the code snippets to a black-box, whose output not only contains the result for each test generation, but also categorizes and aggregates them, thus the researcher may focus on examining the results and making conclusions. My assignment was to elaborate a framework that

- provides functionality to define *code snippets* and supply them with meta data (e.g., target coverage),
- calls the test input generator tools to produce tests for the *code snippets*,
- is able to parse the results of the tool executions and if required generate test suites from input values,
- can perform coverage and mutation analysis on the generated test suites and
- can classify and aggregate the results.

1.2 Progress of the Work

These requirements have been implemented in the *Symbolic Execution-based Test Tool Evaluator (SETTE) framework*. This framework is licensed under Apache License 2.0 as an open-source project and is available at

`http://sette-testing.github.io`

I have started to study test input generation tools in the last semester of my bachelor studies and it was the topic of my Student Research Conference report [11] and B.Sc. thesis [10] in 2013 and the results were also published in a Hungarian multidisciplinary conference [12]. In these works an evaluation methodology was proposed for comparing test input generator tools and assessed four Java tools using 201 code snippets. However, all the tools were based on symbolic execution and the work lacked other type of test input generators (search-based, random) and only one experiment was carried out for each tool. This work was already tool-aided, but that program was only able to carry out the test input generation and used a rule-based evaluation, which only classified a part of the test generations, did not measure coverage and did not support repeated experiment executions.

During the first year of my M.Sc. studies the code snippet base was extended to 300 snippets, one tool was removed from the work but two other tools were added, one of them targeting the .NET platform. The first version of the framework was able to properly describe code snippets with meta data, collect the raw results into a common format and perform coverage analysis. The research results were published in *IEEE International Conference on Software Testing, Verification and Validation (ICST) 2015* [13] (acceptance rate: 24%).

Last year the code snippet set has been extended by 63 new code snippets working with system environment, networking, multithreading and reflection. Moreover, one new tool has been added to the investigation. The actual work also includes repeating experiments several times (because two tools are using a random generator), running experiments with different time limits and performing mutation analysis (these results are currently in the publication process). These research goals also required adding new functionalities into the framework.

1.3 Structure of the Thesis

This thesis will first give an overview on the research field of test input generation (Chapter 2) and present the scientific approach I used for evaluating and comparing test input generator tools (Chapter 3). Then, the requirements, specification and the architectural design of the elaborated tool evaluation framework will be explained (Chapter 4), followed by the description of how development was carried out and of several implementation issues (Chapter 5). Finally, the framework is presented in action and the scientific results are discussed (Chapter 6).

Chapter 2

Background

This chapter gives a brief introduction on that subset of *software testing* which sets the scope of my research, and provides a basic overview of test input generation. Thus, this chapter neither gives a complete overview of the field, nor presents exhaustively the current state of research results and tools.

2.1 Software Testing

According to the ISQTB Glossary [19] *software testing* is "the process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects".

During *dynamic testing* test cases are executed against the *software under test (SUT)* which can either *pass* or *fail*. A test case consists of "test inputs, execution conditions and expected results developed for a particular objective" [18]. Thorough testing is resource-intensive, companies sometimes spend even 40–50% of development costs on software verification. Testing is still considered as a quite monotonous process by the majority of developers, however, during the last years the tooling support have evolved significantly.

2.1.1 Black-box and White-box Testing

Software testing methods can be divided along several aspects, one of them is distinguishing *black-box testing* (or functional testing) and *white-box testing* (or structural testing).

During black-box testing the internal structure of the software under test (SUT) is not exposed and testing is performed directly against the specification of the SUT. This approach is capable of discovering required, but unimplemented software features, it can even be carried out by a third-party person or sometimes by the customer or the user. Nevertheless, it is not as efficient as white-box testing in discovering implementation bugs.

White-box testing takes the internal structure of the SUT into account and usually requires the source code. It means that the method requires a personnel who has knowledge about the structure of the SUT and it is usually unable to detect derivation from the specification. However, it is able to find hidden errors and problems and also helps the developer to understand the code better.

The following part of the chapter focuses on the test input generation approach of white-box testing.

2.1.2 Test Automation and Automated Testing

Test automation means the automation of test execution. There are testing frameworks for almost all programming languages and platforms, such as JUnit for Java or MSTest for .NET. Full test automation is essential for *continuous testing*, which is a corner stone of *continuous delivery*.

The expression *automated testing* is mainly used in academia and its purpose is to automate the complete testing process, including the creation of test suites. One subtask of automated testing may be test input generation.

2.2 Mutation Analysis

During software development it is important to measure the efficiency of the process and the quality of the product and this also applies to testing. For example, if for the same code base there are two test suites, it is not trivial how to decide which test suite is more *effective*. One answer might be that the "better" is which reaches higher coverage, however, the original goal of testing is to detect faults in the SUT, and test suite effectiveness does not strongly correlate with achieved coverage [17].

Mutation analysis [29] is based on injecting small modifications into the software and these divergences (*mutants*) from the original code should be detected (*killed*) by a proper test suite. An indicator of test suite quality might be the number of the killed mutants. Mutant generation is performed by applying *mutation operators* to the original code. A part of them usually imitate common programmer mistakes, such as deleting or duplicating a statement, replacing constants, operators or variables.

To give an example for mutation analysis, take the method and the test case in Listing 2.1. The test case reaches 100% statement coverage, but is unable to kill the mutant when the instruction `i > 0` is mutated to `i > 1`. In addition, if the test case would only call the method and did not have the *assertion*, it would be able to detect only severe runtime problems such as invalid array indexing.

Listing 2.1. *Mutation testing example*

```
int sum_pos(int[] x) {
    int sum = 0;
    for (int i = 0; i < i.length; i++) {
        if (i > 0) {
            sum += x[i];
        }
    }
    return sum;
}

void test() {
    assert sum_pos(new int[] {3, -4, 5}) == 8
}
```

It must be noted that some mutants do not change the functionality of the code (e.g., changing `i > 0` to `i >= 0`). These mutants are called *equivalent mutants* and should be ignored from mutation analysis, however, their detection is not trivial and may require major effort if the number of mutants is high.

2.3 Test Input Generation

The goal of *test input generation* (or *test data generation*) is to generate inputs for the SUT which will be later used in test cases. The selection of test inputs is hard, but there are several methods which are aiming to provide a solution for this problem [2, 8]. In addition, there are several challenges which have to be overcome:

Path explosion: the possible execution paths of a program usually grow exponentially as the size of the software increases. Thus, if a technique works with them, it has to overcome this challenge at least to a certain extent.

Complex (path) conditions: some techniques aim to generate test inputs based on solving path conditions in order to reach higher coverage. The conditions are usually transformed to SMT (satisfiability modulo theories) problems. The general SMT problem is undecidable, but even its subsets are usually NP-hard. A good example is when the test input generator has to determine the exact number how many times a loop has to be executed in order to reproduce a bug.

Floating-point calculations: floating-point calculations are quite common, yet require caution. These calculations are usually not precise and might depend on the hardware architecture and/or the operating system.

Pointer operations: pointers may point to anywhere in the memory, even to a value which is correct in the particular situation. One way to overcome this problem is working with *memory snapshots*.

Objects: objects are tougher to test because they often represent an internal state, making the test generation for methods working with objects much more complicated. A common way to handle object parameter values is to represent them as bytes in the heap, however, this solution might create objects with invalid states which may result in *false negatives*.

Strings: strings are similar to arrays (which can be regarded as pointers or objects depending on the platform), but often proper string expected by the SUT may be hard to generate (e.g., valid SQL statement, XML document which is validated by a schema).

Library/native code: many software use codes whose source is not available, however, these cases have to be handled somehow. Although for manual testing *mocking* is a common solution, it is not trivial how to automate.

Interaction with the environment: the SUT may use the environment in several ways, such as reading and writing files, performing actions based on system time or based on a random generator, accessing network resources, etc. Moreover, a test input generator should never do any unintended harm in the developer machine. One solution may be to generate tests which use a *virtual system environment*.

Multithreaded applications: concurrency and synchronisation lead to several other problems since not only thread scheduling may affect the result of program execution.

Reflection and metaprogramming: although this might seem an uncommon case, modern software (especially web frameworks such as *Spring* for Java) heavily use these features. The general proposed solution is to write *testable code* and use design patterns such as *dependency injection* or *inversion of control*, which have to be taken

into account when designing the architecture. Nevertheless, this still does not give a solution for generating robust test suites for dependency injection frameworks.

Non-functional requirements: checking several non-functional requirements is usually done with other methods (e.g., code quality by static analysis, efficiency by performance testing), but sometimes it may reasonable to use test input generation. For example, finding security leaks in the code or inputs which significantly increase the program execution time.

Unfortunately, either the solution of these challenges require more processing power, more memory and better algorithms or it requires a special approach whose implementation is usually hard. Some problems are easier to overcome if the code was written with testability in mind. Currently the majority of the test input generator tools are research prototypes and as it will be presented later some of them are not even able to handle cases which would be not difficult for test engineers. Test input generation can be carried out using various techniques, three of them are presented in the next sections.

2.3.1 Random Testing (RT)

Random testing is a simple and popular way to approach automated testing and it can be also used if the source code is not available (thus, sometimes it is also considered as a black-box testing method). A random-based test generator is basically driven by a random generator and heuristics.

The main strength of random test generation is that it reaches high coverage in quite a short time and a randomly generated immerse test suite might be used for *regression testing*. However, the technique is not ideal in terms of finding complicated faults in the software and covering lines which can be reached only through complex conditions. Furthermore, the high number of test cases might be a disadvantage sometimes and finding the ideal time limit for random generation is crucial.

A few years ago a new technique, *adaptive random testing* [9] was published, which provides enhancements for random testing by allowing the test developer to control the generation by different factors, e.g., specify a set of strings from which the test generator may be able to create valid SQL requests.

Good examples for RT tools are GRT, Randoop [30] and T3.

2.3.2 Search-based Software Testing (SBST)

This technique regards testing as an optimization problem and uses metaheuristic search strategies in order to generate test inputs [26]. The idea was first formulated in the 1970s [27].

In the last decade research continued in this topic and a *genetic algorithm* has been proposed to solve the problem. This algorithm is based on a *fitness function* which aims to predict which parts of the search space should be examined. The fitness function is problem-specific, but for white-box testing it is usually some kind of coverage, such as path coverage or branch coverage and it may also include conditions to kill mutants.

Recently the number of SBST tools has grown significantly and they are now performing better. Some examples are AUSTIN [23] and EvoSuite [14]. Plus, there is annual contest for Java SBST tools [16].

2.3.3 Symbolic Execution (SE)

Symbolic execution is a well-founded technique for test-input generation [22]. Nonetheless it is not spread yet mainly because it requires high amount of processing power (or time) and system memory. The technique utilizes *symbolic variables* which do not have concrete values in order to collect the path conditions of the SUT. When the path conditions are collected, they are transformed to a formal problem (usually SMT) and passed to a solver to satisfy the expression. Based on the solver response, it is possible to determine which inputs cover which paths (and hence lines) of the code. Because of the formerly mentioned challenges, dozens of optimizations have been proposed to SE since the beginning of the millennia.

An improved version is *dynamic symbolic execution (DSE)*. DSE assigns values to the symbolic variables during the execution and executes the SUT with the concrete generated values[2]. Tools using DSE are often referred as *concolic tools*. [8]

Currently available SE tools cover several platforms, e.g., CATG [32], jPET [1], Symbolic PathFinder [28] for Java, IntelliTest (formerly Pex [33]) for .NET, KLEE [7] for C and SAGE [5] for x86 binaries.

2.4 Tool Evaluations and Comparisons

The publications in the topic can be usually classified into one of the following categories:

- Tool developers and researchers create publications about new tools, innovations and enhancements and these papers mainly focus on presenting one tool and sometimes comparing it with other solutions in one or two aspects.
- Several surveys, comparisons and case studies [24, 6, 31] have been carried out lately, which usually present the actual state of the research area and notable tools. Some of these publications focus on test input generation in general, some of them are dealing with a particular technique. Comparisons are mainly based on comparing the tools' achieved coverage on open source projects, I only found one paper which focused on a *fine-grained* survey [15].

When I started my work, I found out that some tools fail for even simple programs and because they are research prototypes, they usually lack user manual and often only have a short description of usage. My goal was to compare symbolic execution-based test input generators: what do they support and what they do not.

2.5 Summary

This chapter gave a short overview on *test input generation* and its challenges which made the basis of my research, whose approach is discussed in the next chapter.

Chapter 3

Methodology for Tool Evaluation

This chapter will provide a brief overview of the evaluation methodology of test input generator tool comparison and will provide the reason why the subsequently presented framework had to be created. The foundations of this approach were laid down in 2013 [11] and it was improved and published during the last two and a half years [12] [13].

3.1 Evaluation Methodology

The goal of the research was to analyse and compare the capabilities of several test input generator tools. As discussed in the previous chapter, currently there are several challenges of automated test generation and my first experiences with available tools have shown that it is not always clear what they are capable of. For example, some tools cannot handle floating-point numbers or one tool may run out of memory in a few minutes for a simple piece of software while another may provide a test suite reaching 100% coverage in a few seconds for the same code.

Thus, the comparison had to take into account the general programming practices and the current challenges of test input generation. In addition, it was also a goal that the methodology should be language independent which allows the comparison of tools of different languages or platforms. Although the research targets tools of the Java and .NET platforms, it is also possible to involve tools of other languages.

The overview of the scientific approach is illustrated on Figure 3.1.

1. *Language Reference and Challenges:* Collecting program organizational structures and language elements for C/C++, Java and C#.NET (ranging from primitive data-types to complex features such as inheritance and API) and challenges of test input generation.
2. *Features:* Each feature draws up a concept which should be handled by a test input generator tool and it can be also formulated as comprehensive question, e.g.,

   Is the tool able to handle inheritance?
3. *Code snippets:* A code snippet is straightforward piece of code for which a test input generator tool has to generate such inputs or test suite which reaches the maximal achievable coverage[1]. A code snippet formulates a more specific question, such as

[1] As a software may contain unreachable code, some of the implemented code snippets are injected with unreachable branches.

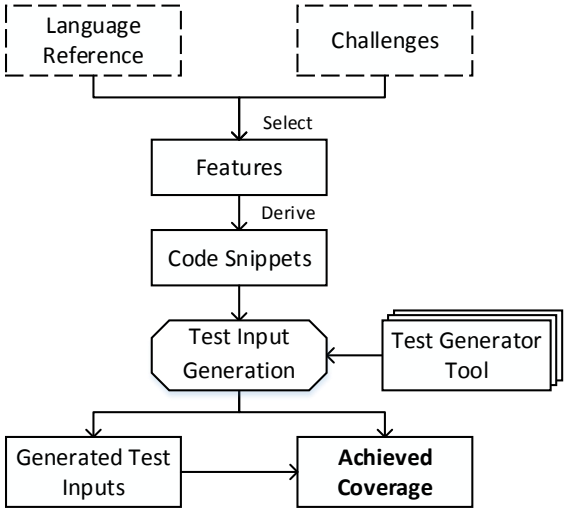
Is the tool able to provide a concrete object parameter value for a function if only the type of the interface is specified?

A code snippet always targets exactly one feature, however, it is possible that other features are also involved (for instance, the example question assumes that the tool is able to call functions with object types).

4. *Test Input Generation:* After the code snippets had been declared, test input generation had to be performed on them separately using the tools which were under investigation. Moreover, in this step variables had to be fixed, such as the parametrization of the tool and the available time limit.

5. *Generated Test Inputs and Achieved Coverage:* Using the outputs of the generations, the result (whether the tool terminated successfully, if yes, then what was the achieved coverage, etc.) could be determined for each tool and code snippet and they could serve as data for evaluation and comparison.

Figure 3.1. *Overview of the scientific approach*



3.2 Features

As written above, features formulate requirements for test input generator tools. A feature is derived from either a program organizational structure (e.g., recursion), from a language construct or element (e.g., Java autoboxing or language API) or from a test input generation challenge (e.g., path explosion). The guidelines during the selection of the features were the following:

- *Coverage:* in order to get basic and detailed feedback on the tools, the most important language elements shall be covered at least once. It must be noted that because of the large number of elements and combinations covering all of them cannot be a reasonable objective.

- *Clarity:* the methodology should be clear for each programming language since sometimes the common concept in two different programming languages might have different meanings.

Table 3.1. *Features of Comparison*

B	Basic language constructs, operations and control flow statements	
	B1	Primitive types, constants and operators
	B2	Conditional statements, linear and non-linear expressions
	B3	Looping statements
	B4	Arrays
	B5	Function calls and recursion
	B6	Exceptions
S	Structures	
	S1	Basic structure usage
	S2	Structure usage with conditional statements
	S3	Structure usage with looping statements
	S4	Structures containing other structures
O	Objects and their relations	
	O1	Basic object usage
	O2	Class delegation
	O3	Inheritance and interfaces
	O4	Method overriding
G	Generics	
	G1	Generic functions
	G2	Generic objects
L	Built-in class library)	
	L1	Complex arithmetic functions
	L2	Strings
	L3	Wrapper classes
	L4	Collections
	LO	Other built-in library features
Others		Other features (e.g., enum, anonymous class)
Env	Working with the environment	
	Env1	Standard I/O
	Env2	Files and directories
	Env3	Networking (sockets and ports)
	Env4	System properties and system environment
T	Multi-threading	
	T1	Threads
	T2	Locks
	T3	Indeterminacy, classic threading problems
R	Multi-threading	
	R1	Classes
	R2	Methods
	R3	Objects
N	Native code	

- *Well-organized structure:* it not only increases clarity and helps maintenance, but all the partial and final results should have the same structure, which makes evaluation easier.
- *Compactness:* the number of code snippets should not be unnecessarily large, otherwise the maintenance, the test execution and the evaluation would require more resource.
- *Minimizing the dependencies:* inevitably there will be dependencies between the features. For example, to use a conditional statement, support for the used type is essential. These dependencies should be only present in one direction between two criteria and there should be no circular dependencies.

Before discussing the concrete features, some notions must be clarified, as the differences between C/C++, Java and C# can be significant:

- *Function:* a program code which can be always called directly, i.e., functions in C/C++, static methods in Java and C#.
- *Structure:* a complex type which can contain other types (even another structure), but does not declare any methods and all parts of it are accessible, i.e. structs and classes without methods and with only public fields.

For the concrete research, several features were selected which are listed in Table 3.1.

3.3 Code Snippets

The methodology defines the *code snippet* as a language-specific, straightforward and directly callable piece of code. Code snippets are usually short (5–20 lines long) and very similar to an ordinary *main()* method expect its parameter list and return value can vary.

An example code snippet can be seen in Listing 3.1 which serves as SUT for a test input generator. The entry point is the `useReturnValue(int, int)` method for which such test inputs or test cases should be generated. The test suite should reach maximum coverage (which is in this case 100%) on both the entry method and the called method. A code snippet (entry point) should be always `static`, the main reason for wrapping them into classes is that in Java and C#.NET methods must always belong to a class.

Regarding the class, in the terminology of the approach such a class is called a *snippet container* and its main purpose to enable putting snippets, usually which target the same or two closely related features, next to each other. A snippet container should not be inherited and should never be instantiated.

Listing 3.1. *A sample code snippet*

```
public final class B5a2_CallPrivate {
    private B5a2_CallPrivate() {
        throw new UnsupportedOperationException("Static class");
    }

    // used method which should be also covered
    private static int calledFunction(int x, int y) {
        if (x > 0 && y > 0) {
            return 1;
        } else if (x < 0 && y > 0) {
```

```
            return 2;
        } else if (x < 0 && y < 0) {
            return 3;
        } else if (x > 0 && y < 0) {
            return 4;
        } else {
            return -1;
        }
    }

    // entry point
    public static int useReturnValue(int x, int y) {
        if (calledFunction(x, y) >= 0) {
            return 1;
        } else {
            return 0;
        }
    }

    // other code snippets...
}
```

For each code snippet meta data has to be specified, mainly the required coverage and the list of methods which should be also involved in coverage analysis. Optionally, *sample inputs* might be specified with which the desired coverage can be reached.

In total, 363 code snippets have been implemented – 300 of them target the *B*, *S*, *O*, *G*, *L* and *Others* categories (see Table 3.1) and referred as *core snippets* while the rest (*Env*, *T*, *R* and *N*) are called *extra snippets*.

In comparison my B.Sc. work [11] were based on 201 code snippets targeting the first 6 categories. These code snippets were first revised in order to minimize their number, but it was realized soon that they did not even cover some important cases and especially features targeting objects were not specific enough. The *extra* categories were added lately since experience showed that some tools (especially *EvoSuite*, *Randoop* and *Pex/Intellitest*) might be able to handle these cases, however, their number is low since first trials pointed out that the tools are not ready for these complex cases.

3.4 Experiment Execution

Experiment execution is a process when a tool is ordered to generate test inputs for a particular code snippet project. Since the methodology targets the analysis of *what cases a tool can support*, tests should be generated separately for each snippet with a smaller timeout rather than for the whole project in one process with a longer one. The main steps of the process are the following:

1. Determine which tool and code snippet set to use, the time limit for an individual generation and tool parametrization.
2. Call the tool to generate test inputs for each code snippet separately using the specified time limit.
3. Analyse the results of the generations individually: decide whether the tool has terminated successfully, generate test cases if needed[2] and measure the achieved coverage.

[2]Some tools are able to generate test suites, however, some of them write the generated inputs into a file or to the standard output.

4. Aggregate results and perform scientific analysis.

The first step already assumes that the parametrization and the usage of the tool is already known, however, this is not trivial since sometimes a tool only has a one-paragraph user documentation. In addition, preliminary experiments are required to determine a time limit which will lead to a meaningful scientific result. Moreover, the analysis of the first experiments may lead to other interesting experiments either with other parametrization or even with new code snippets.

The automation of the second step with at least a batch script is a must, since hundreds of commands for each tool should be never called manually. Plus, the majority of tools cannot directly work on the code snippets and require a *test driver* which is a special `main()` method and sometimes a tool may even require a configuration file for each execution. Examples for both can be seen in Listing 3.2.

Listing 3.2. *Example for Test Drivers*

```
// Test driver for CATG
public final class B5a2_CallPrivate_useReturnValue {
    public static void main(String[] args) throws Exception {
        // create symbolic variables through the tools interface
        int param1 = catg.CATG.readInt(1);
        int param2 = catg.CATG.readInt(1);

        // print the parameters and the return value
        // (they are not saved by the tool)
        System.out.println("B5a2_CallPrivate#useReturnValue");
        System.out.println("  int param1 = " + param1);
        System.out.println("  int param2 = " + param2);
        System.out.println("  result: " +
                           B5a2_CallPrivate.useReturnValue(param1, param2));
    }
}

// Test driver for SPF
public final class B5a2_CallPrivate_useReturnValue {
    public static void main(String[] args) throws Exception {
        B5a2_CallPrivate.useReturnValue(1, 1);
    }
}

// Configuration file for SPF
target=hu.bme...B5a2_CallPrivate_useReturnValue
symbolic.method=hu.bme...B5a2_CallPrivate.useReturnValue(sym#sym)
classpath=build/,/home/sette/sette/.../snippet-lib/sette-snippets-external.jar
listener=gov.nasa.jpf.symbc.SymbolicListener
symbolic.debug=on
search.multiple_errors=true
symbolic.dp=coral
```

After the generation has finished, the result of each execution has to be classified into one of these categories:

- `N/A`: the tool was unable to handle the particular code snippet because either parametrization was impossible or the tool failed with a notification that it is unable to handle the case.
- `EX`: the tool has failed during test generation due to an internal error or exception.
- `T/M`: the tool did not finish within the specified time limit or it ran out of memory during generation.

- `NC` or `C`: generation has terminated successfully and coverage analysis is needed to be done in order to determine whether the required coverage was reached (`C`, stands for *covered*) or not (`NC`, stands for *not covered*).

If the result is NC or C, coverage has to be measured, which in this case is *statement coverage*. The tools usually measure the achieved coverage and write it to their output, however, their interpretation of the notion of *code coverage* vary, hence the coverage has to be measured for each tool in the same way. Although some tools produce only test input values and not an executable test suite, it is possible to generate a test suite from these values and measure the coverage using that.

After the categorized result is decided for each individual execution, they can be aggregated and the comparison of the tools can be made. The selection of experiments enables the comparison in different aspects. For example, from running several experiments with different time limit we can conclude how time-efficient the tools are and *mutation analysis* may give a feedback about how strong the generated test suite is.

3.5 Summary

In conclusion, the methodology of test input generator tool evaluation and comparison discussed in this chapter has already proved that it is strong enough to provide relevant scientific outcome [13], however, experiment running is a long and monotonous process.

In my B.Sc. work the experiment execution and the analysis of the output was automated. The analysis of the coverage was partly automated, which means that the framework was able to determine whether a result belonged to NC or C for the simplest code snippets (no branches or fixed finite set of possible return values), but it did not generate a test suite, did not perform automated coverage analysis and for some cases I had to write the test cases and analyse coverage manually using Eclipse and EclEmma. In that time, four tools were involved in the evaluation and only one experiment happened for each tool and the manual part of evaluation took several hours.

Later, it became necessary to run experiments with different timeout values and repeat experiments with the same parametrization several times since other tools which are partly based on randomness were taken into account. This would have lead to several days, if not weeks of manual analysis (not mentioning how error-prone it is), which was undesired if the *SETTE framework* would not have been developed in parallel with the research.

Chapter 4

Designing the Automated Evaluation Framework

In order to run experiments discussed in the previous chapter, I have developed the *Symbolic Execution-based Test Tool Evaluator (SETTE) framework*[1], which is able to automatically execute experiments on test input generator tools targeting the Java platform. This automated process includes result categorization, coverage and mutation analysis. This chapter presents *why* it was necessary to develop SETTE, *what* it does and briefly *how* it works.

The program which I started with was just a simple experiment execution tool developed during my B.Sc. studies, which also had limited parsing abilities. During the last 2.5 years this tool has been transformed and extended to a framework which can perform tool evaluation automatically and whose main features are the following:

- improved handling of experiments: parsed data is now in common XML files, experiment execution is split into several parts which can be re-run individually, ability to handle set of experiments
- ability to parse all the outputs which were encountered during the years
- generating complete test suites from input values
- code coverage analysis
- mutation testing
- complete evaluation of five Java tools and mutation testing for IntelliTest (.NET) as well

4.1 Motivation for an Evaluation Framework

As the previous chapter pointed out, there was a need to run several experiments. To put the number of experiments in context, currently the research targets six tools (five Java and one .NET) and requires the execution of the following experiments:

- 10 repetitions of experiments with the 300 core snippets with one time limit value

[1] When the framework was named it only dealt with SE-based tools, but it can work with other types of test input generator tools too.

- 10 repetitions of experiments of the extended code snippet set (63 code snippets) with one time limit value
- 10 repetitions of performance-time experiments (129 code snippets selected from the core snippets) with four timeout values
- mutation analysis of the test generations for the core snippets

Altogether it is 8 790 test input generations per tool making up which is making up a total 43 950 of for the five Java tools. It was obvious that this task had to be completely automated with an extensible framework. Such a framework creates an opportunity to run new experiments (or re-run previous ones) at any time without a major effort. Although the time needed for the development of a framework might be even more than manual experiment execution, for the long run it is more profitable, especially that manual coverage analysis is error-prone.

4.2 Requirements

It is sure that the users of the framework have general IT, software development and software testing knowledge. From the methodology I have identified the user workflow:

1. *Experiment planning:* the user specifies what kind of code snippets and parametrization is needed.
2. *Code snippet implementation:* the user implements the code snippets and supplies it with metadata (required coverage, sample inputs, etc.)
3. *Automated evaluation:* the user orders the framework to perform automated evaluation.
4. *Evaluation Analysis:* the user analyses the experimental results based on the aggregated results and individual outputs.
5. *Refinement:* the user might alter or extend the code snippets, change the parameters or run other experiments.

The tool evaluation framework should provide a solution for the second and third steps. In order to make the framework not only functionally satisfactory, but also usable and robust, I have identified five major requirements.

Handling Tool Evaluation Artifacts The framework should save all the raw and parsed data enabling it to be processed by other software.

The user is not only interested in the final results, but other files have to be preserved or created: raw outputs, information about the tool executions, output parsed into a common format, executable test suite, code coverage visualization and aggregated results. All the data produced by the framework should be in a standard format. Coverage visualization should be easy to understand for humans and should be a one-page summary for each test generation which only contains the SUT. In addition, the framework should provide tools (preferably graphical user interface) for browsing the data easily.

High Level of Automation The framework should be able to automatically carry out experiments with minimal user interaction.

The user would like to focus on experiment planning and result analysis, but proper research results require a vast number of test input generation executions. Thus, the ideal situation is that the user calls the framework and passes the experiment specifications and they get a notification when evaluation has finished.

Customizability The framework should be parametrizable with the code snippet set, tool and execution timeout and it should also allow the user to re-run not all, but only one particular test generation.

Since the user would like to perform different kinds of experiments, the framework should provide an opportunity to set the experiment parameters (code snippets, time limit, tool to use). Moreover, a possible scenario is that due to a temporary problem a tool failed test generation for one snippet and only this case should be re-run.

Extensibility The framework should be able to work with other tools and code snippets without modifying its source code.

Currently the main users of the framework are my supervisor and me, who conduct research on evaluating test input generator tools. It was important for us from the beginning to make it easy to extend the framework with a new tool or code snippets. Moreover, in the future it is possible that somebody (tool developers, researchers) will want to use this framework to carry out experiments with other code snippets, tools or parametrizations.

Validity The framework should never provide invalid results and rather fail on unexpected events.

Invalid results can undermine the credibility of the scientific results. The framework should extensively validate its inputs, especially the code snippets, and immediately fail if it detects something error-prone or unknown. In addition, the framework should provide detailed error messages if it found something invalid in the user input and should report as much errors as it can at once rather then reporting errors one by one. This is not a common strategy used in general user software, however, this would enable the user to fix several errors in the same step.

4.3 Specification

Based on the requirements, I have elaborated the use cases, which are represented in Figure 4.1 and the overview of the framework can be seen in Figure 4.2.

In order to satisfy the requirements, I have elaborated what inputs can be specified for the framework, what output is expected and how can the evaluation process be split into several parts. The latter was important to plan in this step because it affects both the product outputs and the usage.

Figure 4.1. *Major Use Cases*

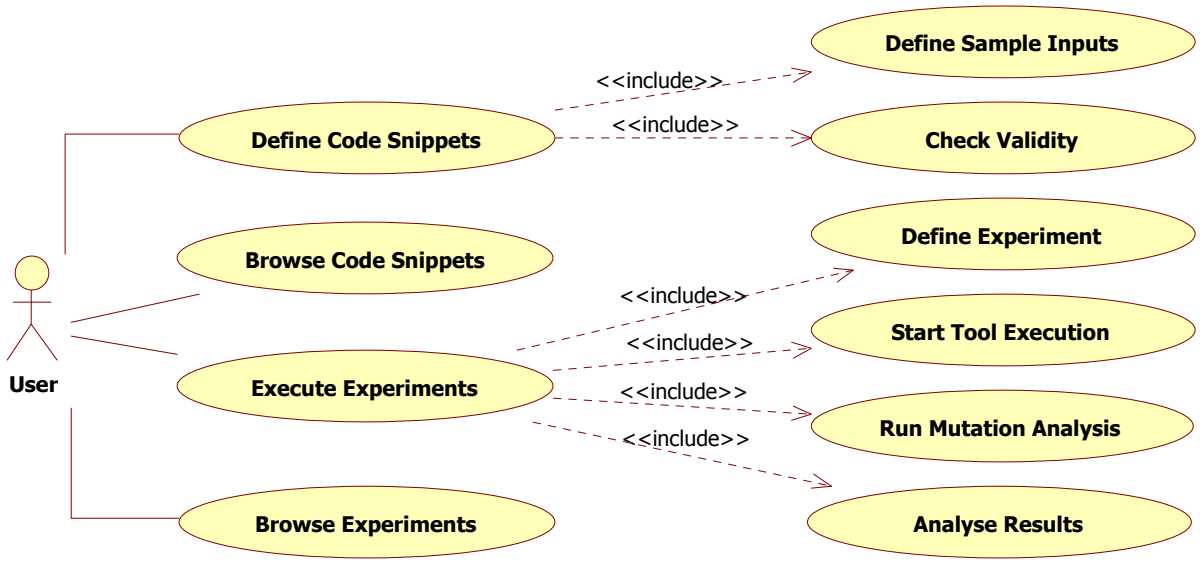
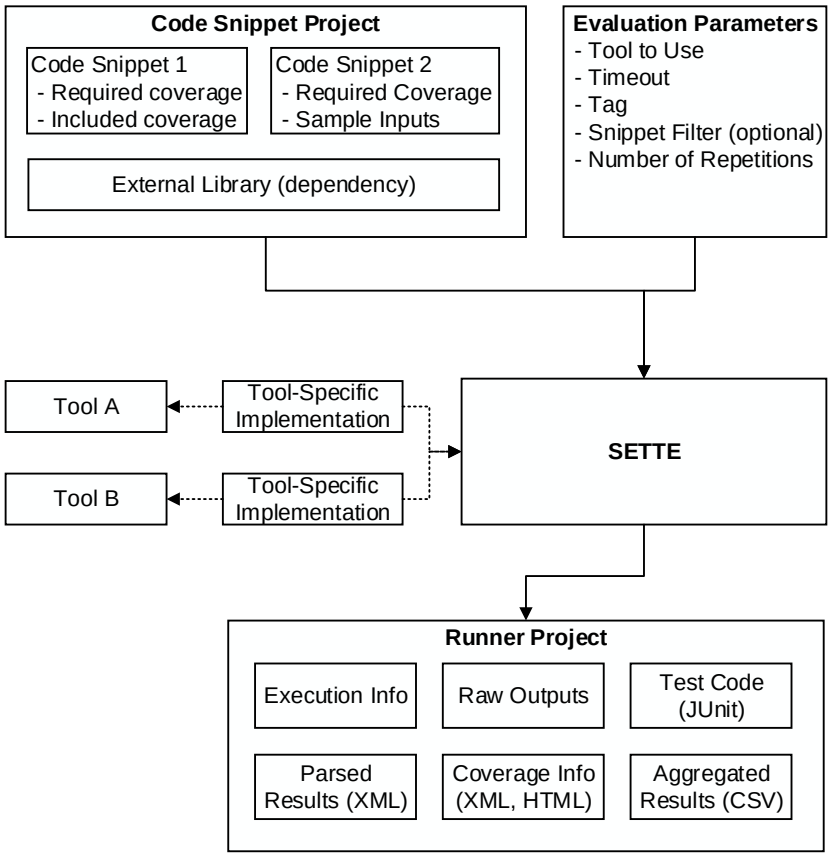


Figure 4.2. *SETTE as a Black Box*



4.3.1 Glossary

This section clarifies some notions used within the context of the framework.

(Code) snippet Source code which serves as SUT for test input generator tools.

(Code) snippet container A class containing one or more code snippets.

(Code) snippet input factory An optional method that returns the sample inputs that reach the required coverage on a particular code snippet.

(Code) snippet input factory container An optional class which contains the snippet input factories for the snippets of a particular snippet container.

(Code) snippet project A project which consists of one or more snippet containers and snippet input factory containers.

Required (statement) coverage The expected coverage which should be reached by the generated test inputs in order to justify that the tool supports a certain case.

Include coverage A code snippet may call other methods and coverage analysis should also consider these methods.

Sample inputs Set of such inputs for a particular code snippet that reach the required statement coverage.

Generated inputs Set of inputs which are generated by the tool.

Experiment Evaluation of one tool for one snippet project with a certain parametrization. In the terminology of the framework, performing evaluations on several tools or with different time limits are separate experiments.

Runner project A project containing the artifacts of one experiment.

Test execution The process when the generated test suite is execution.

Tool Test input generator tool.

Tool execution The process when tools are called to generate test inputs.

4.3.2 Target Platform and Tools

The framework should work on both Linux and Windows and since it should mainly work with Java tools it would be the easiest to develop it in Java. This means that the evaluation should be possible to be carried out on both platforms, however, some tools might be bound to a certain operating system. This fact only affects the tool execution part of the evaluation, but not the other functionality, such as coverage analysis. Nevertheless, the main target platform was *Ubuntu 14.04 LTS*.

Regarding the test input generator tools, the framework have the ability to handle five Java tools, namely CATG, EvoSuite, jPET, Randoop and SPF but should provide an interface through which list can be extended, even for other languages.

4.3.3 Inputs of the Framework

The framework should handle the following inputs:

Code snippet project a standalone, compiled Java project which contains code snippets. Meta data is also supplied for all code snippets. Optionally, the user may declare sample inputs in order to ensure that the required coverage can be reached. If sample inputs are present, the framework should be able to check that they reach the required coverage.

Tool the tool with which the evaluation should be carried out. The user may choose from the tools which are internally supported by the framework or they supply an unsupported tool with the required tool drivers.

Tag a label for experiments. The code snippet project, the tool and the tag together identifies the runner project of the experiment. Using a tag the user can add any identifier to the runner project.

Timeout the time limit to use for test input generation, which can be set for each experiment execution and applies to all tool executions within one experiment.

Filter (optional) if specified, experiments will be carried out only for a subset of code snippets. This parameter should be flexible and allow the user to make any kind of selection among the snippets.

Number of repetitions (optional) it is possible that the user would like to run the same experiment several times.

The snippet project should be in the following layout:

- `build`: directory containing the compiled files of the project.
- `snippet-input-src` (optional): directory containing the source files of the sample inputs.
- `snippet-lib` (optional): directory containing the dependencies of the snippet project (third party JARs, native `.so` and `.dll` files).
- `snippet-src`: directory containing the source files of the snippets.
- `snippets-src-native` (optional): directory containing the source files of the native libraries.

The snippet project has to be compiled by the user and they may use any build tool, such as *Ant*, *Maven* or *Gradle*. The user is responsible for making sure that in the `build` directory the bytecode of the actual source files are present and not an older version.

Supplying the code snippets with meta data should be available using annotations as it can be seen in Listing 4.1. The required annotations are listed in Table 4.1.

The framework has to validate that all the classes are marked as a snippet container or a snippet dependency and all `public` snippet container methods have a declared required coverage or they are marked as not snippets.

Listing 4.1. *Example for supplying code snippets with meta data*

```
@SetteSnippetContainer(category = "B5",
        goal = "Check support for private function calls",
        inputFactoryContainer = B5a2_CallPrivate_Inputs.class)
public final class B5a2_CallPrivate {
    // ensure that the snippet container cannot be instantiated
    private B5a2_CallPrivate() {
        throw new UnsupportedOperationException("Static class");
    }

    private static int calledFunction(int x, int y) {
        if (x > 0 && y > 0) {
            return 1;
        } else if (x < 0 && y > 0) {
```

```
            return 2;
        } else if (x < 0 && y < 0) {
            return 3;
        } else if (x > 0 && y < 0) {
            return 4;
        } else {
            return -1;
        }
    }

    // snippet ID: B5a2_useReturnValue
    @SetteRequiredStatementCoverage(value = 100)
    @SetteIncludeCoverage(classes = { B5a2_CallPrivate.class },
            methods = { "calledFunction(int, int)" })
    public static int useReturnValue(int x, int y) {
        if (calledFunction(x, y) >= 0) {
            return 1;
        } else {
            return 0;
        }
    }

    // other code snippets...
}
```

Table 4.1. *Annotation Types for Supplying Meta Data*

`@SetteDependency`	
Marks non snippet container classes.	
`@SetteIncludeCoverage`	
Marks snippet methods to order the framework to also take into account the coverage measured on other methods.	
`classes`	Array of classes whose methods are referred.
`methods`	Array of referred methods. The asterisk literal (`*`) denotes that all methods of the corresponding class should be taken into account.
`@SetteNotSnippet`	
Explicitly marks `public static` methods which are not code snippets.	
`@SetteRequiredStatementCoverage`	
Defines the required statement coverage.	
`value`	The required coverage value in percent, e.g., 95.61.
`@SetteSnippetContainer`	
Marks snippet container classes.	
`category`	Category of the snippet container, used for aggregating results.
`goal`	Short description of the goal of the snippets.
`inputFactoryContainer`	Optional reference to the class which produces the sample inputs for the snippets.
`requiredJavaVersion`	Required Java version of the snippets (default: Java 1.6).

The framework is expected to validate its inputs, especially the snippet project. A snippet project must satisfy the following requirements:

- The `snippet-src` and `snippet-input-src` directories must only contain `.java` files.
- The `snippet-lib` directory may only contain `.jar`, `.so` and `.dll` files.
- All the classes should be either marked as `@SetteSnippetContainer` or as `@SetteDependency`.
- Snippet container classes
  - must be `public final`,
  - must not be inner classes,
  - must declare a category and a goal,
  - must have a name which starts with the main category and the subcategory is separated by an underscore (`_`) character,
  - may declare only `static` fields,
  - must have exactly one `private` constructor, which takes no arguments and throws an exception and
  - must contain only `static`, non-`native` methods.
- Snippet methods
  - must be `public static`,
  - must be placed inside snippet containers,
  - must have a unique name in the container, a unique identifier in the snippet project
  - must declare the required statement coverage (between 0% and 100%) and
  - must only include the coverage of valid methods.
- Snippet input factory container classes
  - must be `public final`,
  - must not be inner classes,
  - must have a name which is the name of snippet container and the `_Inputs` suffix,
  - must not declare any `static` fields,
  - must have exactly one `private` constructor, which takes no arguments and throws an exception and
  - must contain only snippet input factory methods.
- Snippet input factory methods
  - must be `public static` and non-`native` and
  - must not declare any parameters.
- *Synthetic* (compiler-generated) elements are not subject of validation.

These rules might seem quite rigid, however, they pursue to ensure that the snippet project is correct and prevents inconvenient mistakes which can be made by the users. Of course, everything cannot be checked (such as ensuring that a code snippet does not do any harm to the system) and the code snippet project is the responsibility of the user. However, it comes handy if the framework would fail on probably unintended cases, such as the user forgot to specify the required coverage for a code snippet.

Validation of the snippet projects should happen in as few steps as possible and the framework should rather report several errors at one time. It spares time for the user, since the user may fix several issues in one step before recompiling the project and running validation again. My first experiences has shown that when the user is creating code snippets, they are focusing on these short codes and the sample inputs, not on the validity of the annotations. It is inevitable to fix them in order to obtain valid results from the framework and the framework should be rather strict than have even a little chance that it produces invalid scientific results.

To summarise, the framework should reject the code snippet projects which have inconsistent or improper naming, invalid annotations or structure.

4.3.4 Outputs of the Framework

Based on the requirements, I have elaborated that the framework would provide the following outputs:

- All the experiment results should be written into a separate project, which is called *runner project*.
- For each code snippet execution the framework should:
  - save all the data that was gathered during evaluation (raw output, information about the tool process execution)
  - generate XML files with a common schema containing the results, thus, it can be processed with other software as well
  - generate a user-friendly HTML file in which the measured coverage is visualized
- A CSV file containing the aggregated result of the experiment
- Log files: the log files of the framework should provide feedback for the user and debugging information for the developer. If the evaluation is successful, the log files of the framework can be discarded.

The *runner project* should contain the transformed copy of the code snippets which can be passed to a test input generator tool. This transformation includes removing the framework-specific annotations, generating tool-specific test-drivers and configuration files. The framework should also be able to compile this project automatically before tool execution. All generated files shall be placed into a directory called `gen` and all the code-snippet output specific files shall be placed to the `runner-out` directory, while the files which contain aggregated results should be in the root of the runner project's directory. The runner project should preserve the directory naming of the snippet project (such as preserve the `snippet-src` directory) since some tools may require their own `src` directory. The runner project must not contain the sample inputs so test input generator tools cannot access them.

4.3.5 Behaviour

Since evaluation is a long process, it shall be split up into the following steps:

1. *Runner project generation:* the first step is to generate the runner project, including the tool-specific files.
2. *Tool execution:* in this step the tools are called to generate test inputs or test suites for the code snippets.
3. *Parsing raw output:* the outputs of the tools are parsed into a common format and if required, the generated test suite is also transformed (e.g., removing tool-specific but unnecessary dependencies). The fact whether a tool finished successfully or not is also decided in this step.

4. *Test suite generation:* if the tool generated only test input values rather than test suites (which is executable test code), a test suite has to be generated in order to make coverage analysis possible.

5. *Coverage analysis:* the coverage is measured for each code snippet and it is decided whether the tool has reached the required coverage or not. In order to make it convenient, the coverage should be reflected on the original code snippets, not on the transformed ones. Since it is possible that a test case calls an infinite loop and it never finishes, it is required that during test case execution the framework is able to force a timeout, detect deadlocks and even kill threads.

These steps are referred to as *evaluation steps* and are carried out by *evaluation tasks*. The main reasons for splitting up the evaluation are that it provides a clear overview of the evaluation process and also helps testing. If the user wants to re-do a step, it is enough to re-do the particular step and all the steps following it. For example, the second step may take up even several hours if the time limit and the number of code snippets is big and if some changes are made in the parser, it is enough to rerun steps 3–5 (why it was common is detailed in the next chapter).

While first three steps are tool-specific, the latter two are tool-independent. It becomes handy since the implementation of coverage analysis is not trivial and it is enough to implement it once for all the current and future tools.

4.3.6 User interface

The main requirements of the user interface is that it should be easy to use for a professional, thus it is enough that the evaluation can be performed from a console interface which follows the KISS[2] design principle and it is not a problem if the user has to provide several command-line parameters for an application or a script.

However, there are two use cases when a graphical user interface becomes convenient: browsing the snippet project and examining the experiment results. For the former, a simple GUI is needed that provides basic feedback about what snippets were detected, what are the categories and what is the total number of snippets. For the latter, it must be known that users might use code snippet projects with hundreds of code snippets and have dozens of experiment executions. The user might be interested in simple questions such as checking the raw output of a particular snippet for several experiments and checking the differences between the different runs. Browsing several directories and tracing down a particular file in each project might be hard for users who prefer using a GUI rather that writing scripts. Thus, another simple GUI is required which is able to:

- detect the runner projects,
- filter the runner projects by code snippet project, tool and tag,
- filter the code snippets and
- provide shortcuts for each snippet in order to make the user able to open a particular output file quickly and easily.

[2]Keep it simple and stupid

4.4 Architecture of the Framework

The framework specified above should serve one purpose and the complexity of the automated tool evaluation lies in implementation details. The framework can be easily split into several components, however, the low-level design of these components are often technology-specific and closely bound to the implementation.

The visualization of the architecture can be seen in Figure 4.3 which are discussed in the next paragraphs from bottom to top. It must be clarified that the *SETTE Framework* consists of

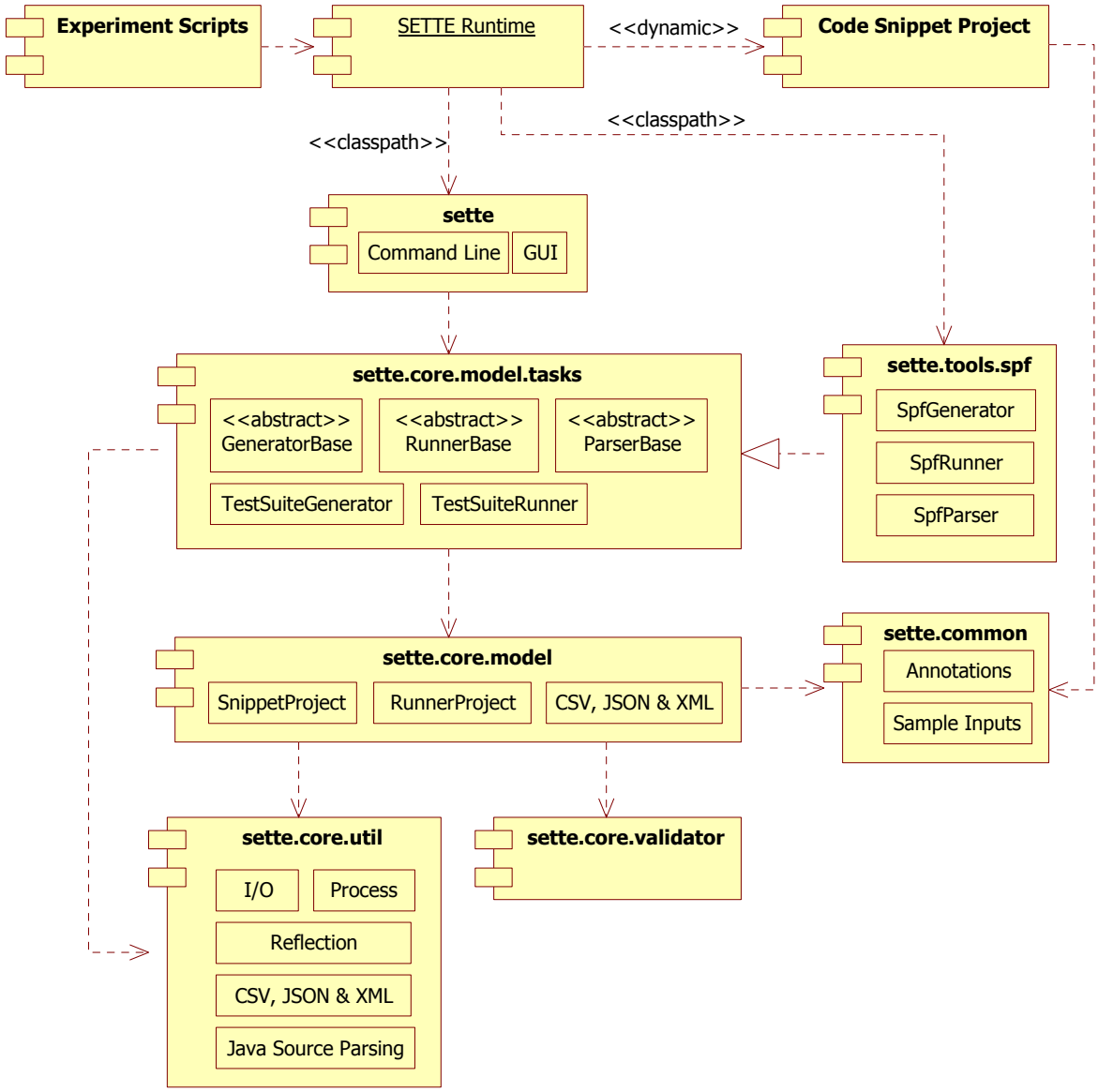
- a Java application (referred to as *SETTE Runtime*), which is able to perform one experiment execution and evaluation and provides the formerly mentioned two GUIs and
- a set of *experiment scripts* which can call the application to run several experiments on different code snippet projects, tools with different timeout values.

sette.common This package builds up the standalone *sette-commons* library, which has no dependencies and contains the annotations which are required for code snippet description (discussed in Section 4.3.3) and classes with which the sample inputs can be specified. When generating the runner project, SETTE automatically removes all annotations from the code and references to this project in order to avoid any interference with the test input generator tools.

sette.core.util SETTE has to perform several low level tasks, namely extensive file handling, process execution, parsing the code snippet project using reflection, reading and writing CSV, JSON and XML files and parsing Java source code. The following problems are solved with this component:

- A utility I/O component which shall be used for everything inside SETTE that provides convenient methods for easier file handling and logs all I/O events.
- Process handling and utility component, which is able to call processes with a timeout and to extract the result of process execution, provides a listener interface and kills processes forcefully using OS calls.
- Reflection is heavily used when parsing code snippet projects, thus, helper classes were needed such as comparator and annotation extractor.
- CSV, XML and JSON data is easy to handle (sometimes even without third-party libraries), however, these utility classes are able to make reading/writing a one-line code and also transform the thrown exceptions in order to avoid boilerplate code.
- Java source parsing: Java source parsing is done using a third-party [20] component, however, I have encountered a few bugs which are fixed using the extensible interface of the library.

Figure 4.3. *Architecture of SETTE*



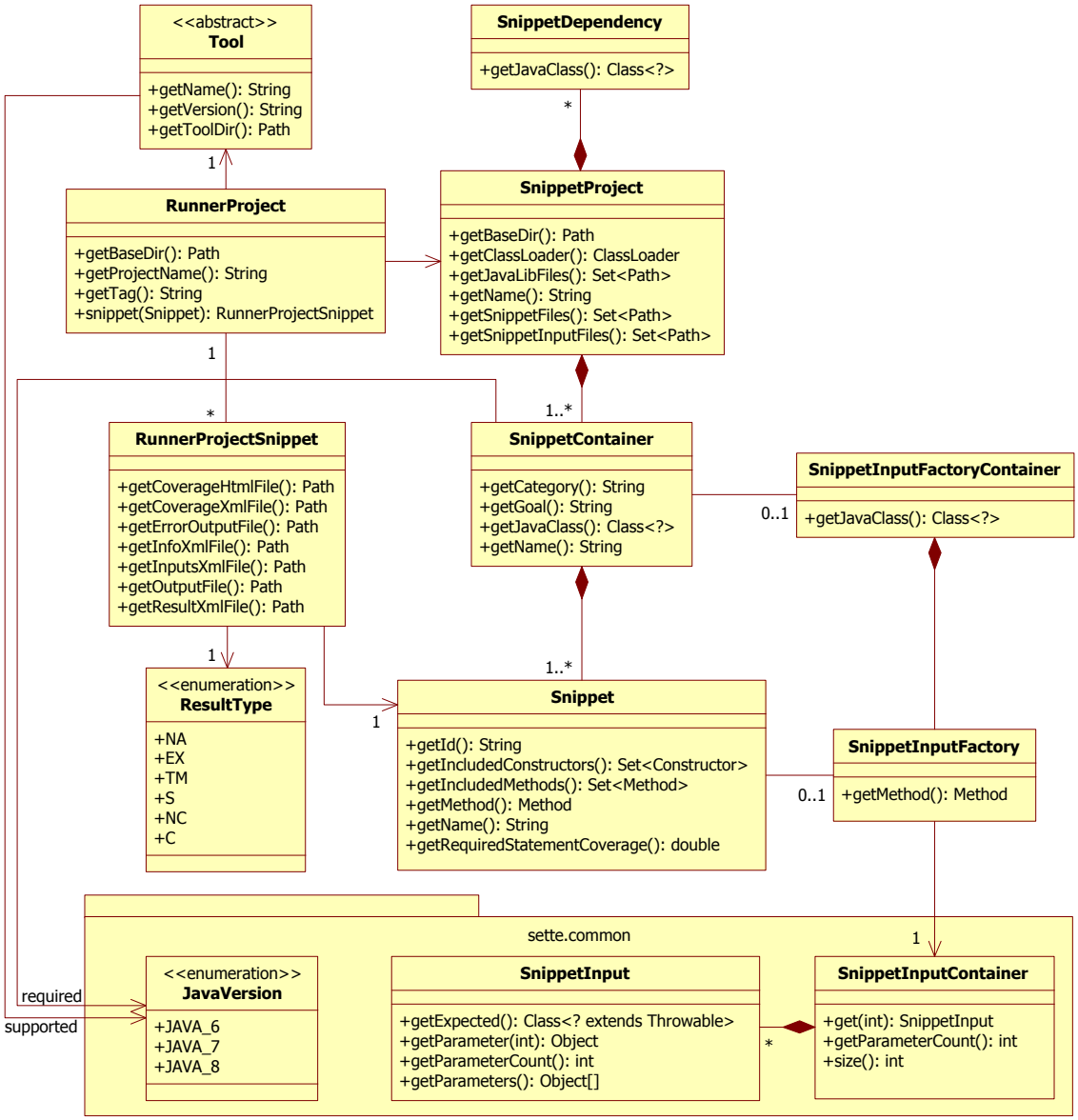
sette.core.validator Since batch validation was essential, a complete validation layer was implemented. This component provides several types of validators (files, reflection, etc.) which may report several errors at once. These validators can be arranged in a tree hierarchy and error would be reported in the same structure.

sette.core.model This component contains model classes which represent the snippet project, runner project and the data files and also contains the algorithms responsible for parsing and exporting these classes. The classes and their properties are represented in Figure 4.4.

The class hierarchy maps the notions declared in the specification and reflects the connections between them, but some parts may need further explanation.

ResultType The category of the evaluation result for one snippet, valid values are: `N/A`,

Figure 4.4. *Model Classes Defined by SETTE*



`EX`, `T/M`, `S`, `NC` and `C`. `S` is needed because of splitting the evaluation process into several steps. After parsing the results of a tool execution, it can be decided whether the tool finished properly or not, but it is undecided whether the generated inputs has reached the required coverage. `S` means *successful* and during the coverage analysis it will be decided whether it should be replaced by `NC` or `C`.

RunnerProjectSnippet Represents a code snippet in context of a runner project. While the `Snippet` class focuses on code snippet description (required coverage, reference to Java method, etc.) this class focuses on the files belonging to a code snippet (raw outputs, XMLs, etc.) within the runner project.

(Currently there are two more classes in the implementation: `RunnerProjectSettings` and `RunnerProjectUtils`. These classes are static helpers for runner projects and are being replaced by `RunnerProject` and `RunnerProjectSnippet`. The latter two represents the object model of runner projects better.)

sette.core.model.tasks This package focuses on *evaluation tasks*, which perform the steps of an evaluation.

`GeneratorBase` Generates the runner project for a tool.

`RunnerBase` Calls the tool for each code snippet to generate test inputs.

`ParserBase` Decides whether the result of the execution is `N/A`, `EX`, `T/M` or `S` (see Section 3.3). If the category is `S` and the tool produces input values, parses them into an XML format. If the tool produces a test suite, but a transformation is needed to make it usable by the framework, the transformation is carried out in this step.

`TestSuiteGenerator` If the tool generates input values, this task will generate a test suite from the input values exported to the XML files.

`TestSuiteRunner` Performs test execution, coverage analysis and decided whether the result is `NC` or `C`.

Since the first three tasks are tool-specific, these classes are `abstract` and applying the *template method* design pattern they had to be implemented for each tool separately. These classes also provide extensibility to alter the default mechanism. However, thanks to parsing everything into a common format, the last two tasks are tool-independent, cannot be altered (generally these tasks should not be changed at all).

The *tasks* component also provides other functionality, such as a controller for CSV generation and compiling projects using Ant.

sette This top-level component provides general application functions, such as parameter handling, reading start-up configuration, backing up runner projects if needed, handling user interactions and also the two GUIs for better user experience. This layer basically connects the user with the evaluation tasks.

Extensibility It was a requirement that the framework should be extensible by a new test input generator tool by anyone. This practically means that the generator, runner and parser have to be implemented for the new tool, these classes have to be passed to the JVM along with the framework and the name and location of the tool shall be declared in the configuration.

Chapter 5

Implementation

This chapter presents the implementation of the *SETTE framework*, how it was developed and highlights the major engineering problems.

5.1 Platform and Development Tools

The development of the framework started using Java 6, then replaced with Java 7 in 2014 and about I switched to Java 8 in 2015. The main reasons were that both Java 6 and Java 7 have reached their end of life and some tools started to support Java 8. In addition, Java 8 came with dozens of useful features, such as the new Streams API (functional programming), default methods and bug fixes in process execution which made it possible to remove hundreds of lines from the source code, reducing its complexity and increasing its maintainability. In addition, third-party dependencies used by the SETTE framework have also evolved.

SETTE was developed using the Eclipse IDE. Originally, I used pure Ant for compiling the framework and downloaded the framework's third party libraries manually, however, as the number of dependencies grew I switched to Gradle. Gradle is a quite young build tool, which is similar to Maven in terms of dependency management, however, it can be configured using Groovy (which is a Java-like scripting language written for the JVM) and writing custom tasks (such as checking that all files have the license declaration) is convenient.

SETTE itself depends on the following libraries:

- Project Lombok: this library saves the developer from writing boilerplate code by providing annotations to generate source code during compile time (`@Data`, `@Getter`, `@NonNull`...).
- JUnit: JUnit is not only used for running the tests of SETTE, but also used for running the tests generated by the tools.
- JaCoCo: this library is used for instrumenting the source of code snippets and measuring coverage.
- JavaParser: this library is able to parse Java source code to an object model (like DOM for XMLs) and allows the developer to parse the source file, perform transformations or even create Java source files. SETTE uses JavaParser for transforming

the source of the code snippets. Although these transformations could be carried out by transforming the bytecode, in this way the source of the transformed files would not be available.

- Apache Commons Lang3, Guava: common libraries which extend the Java API. SETTE uses utility classes for the OS, exception handling, reflection and immutable collections.
- Jackson Databind and Jackson Dataformat CSV: handling JSON and CSV files.
- SimpleXML Framework: mapping XML files to objects.
- SLF4J with Logback: logging libraries.
- Args4j: mapping command line program arguments to objects.

For improving code quality and finding implementation flaws, formerly I have used FindBugs, PMD and CheckStyle. However, last October (when refactoring began) I switched to SonarQube (formerly Sonar). This tool is a web-based application which can be also run from a developer's machine and its main purpose is to check code quality, to calculate metrics and to notify the developer about the detected flaws. SonarQube is a piece of cake to integrate with Gradle and its default ruleset also contains rules from the formerly mentioned static code analysis tools. Since the tool measures technical debt and visualizes where the problems are, it enabled me to identify which parts of the source code need the most urgent modification.

5.2 Development Iterations

The development can be split up into the following iterations:

- February–May 2014: Development of SETTE has started, extending the core snippet project for 300 code snippets, introducing the current annotations, code snippet project validation, test suite generation, limited coverage analysis.
- June 2014–May 2015: Mapping the code snippets to C#.NET, improving coverage analysis, open-sourcing the framework, public documentation, two tutorial screencasts.
- June 2015–May 2016: Extending capabilities for new experiment requirements, improving the user interface, scripts for batch executions, mutation analysis (including C#.NET). Majority of tool development time was spent on refactoring existing code.

The development of the framework often happened on-the-fly before October 2015 since it served as a tool for scientific experiment execution and evaluation. Since the scientific results were the most important, the framework was initially weakly designed and implementation was carried out as fast as possible. This resulted in bad code quality (mainly uncommented and duplicated code, dozens of TODO comments, missing documentation for crucial function etc.) and refactoring required major effort and it has not been completely finished yet. The framework evolved during the years and the list of requirements was constantly growing, not to mention several dead ends which contributed to the final structure of snippet and runner project and evaluation tasks.

5.3 Major Difficulties

This section summarizes the major difficulties I have encountered during development.

5.3.1 Proper Class Loader Usage

In Java, `ClassLoader`s are responsible for loading classes and directly interacting with them is only required in specific cases (especially when one would like to load classes dynamically), however when they are needed the developer has to find the way out from the *class loader maze*. The main class loaders in Java are the following:

- *Bootstrap class loader*: responsible for loading classes which are part of the Java API.
- *Extension class loader*: responsible for loading JARs placed next to the JDK and from the directory specified in the `java.ext.dirs` VM parameter.
- *Application class loader*: responsible for loading classes from the application classpath.

Class loaders are arranged into a parent-child hierarchy, where the bootstrap class loader is in the root. If a class loader is unable to load a class, it passes that to its parent. Sometimes the *thread (context) class loader* is also mentioned, which is the particular class loader of a thread (it is usually the application class loader unless it was changed by the program).

SETTE needs to load the snippet project dynamically, however, the location of the snippet project only turns out when SETTE is already started. Since the classpath of class loaders cannot be changed through the public API, a separate class loader was needed for loading and validating snippet projects, which can also use the classes of SETTE (thus, it was needed to make the application class loader its parent).

Moreover, when it comes to code coverage analysis, the source code of the code snippets has to be instrumented and loaded into a separate class loader and test execution has to be performed using that one. To clarify, now there are three class loaders to consider, first the application class loader which loads the classes of SETTE, the snippet project class loader which contains the untouched bytecode of the code snippets and the coverage analysis class loader which contains the instrumented bytecode of the code snippets and the test classes (practically two versions of each code snippet is loaded at the same time).

The biggest problem with the *class loader maze* was that I did not have sufficient knowledge and I had to learn to be aware of which class loader has to be used and why.

5.3.2 Source Code Generation

SETTE has to provide several features that requires source code manipulation:

1. removing annotations from code snippet classes,
2. generating test-driver classes and
3. cleaning-up generated test suite (e.g., EvoSuite).

Although all these problems may be solved by general text parsing, it can be only a good long-term solution for the second one, since the others require parsing code that has strict grammar. In the first implementations the third step was not needed, and the first was carried out by using simple text searches for lines starting with `"@Sette"` and the user could only use one line annotations and they must have had avoided automatic code formatting for code snippets.

After extensive search I have found the JavaParser [20] project, which was not maintained for years and only supported Java 1.5 syntax at that time. This had a bad impact on the code snippets, since Java 1.7 language elements (such as the diamond (`<>`) operator) for auto-completing generic types) must have been avoided.

Fortunately, the project was revived and now it supports Java 1.8 syntax. I have encountered two bugs and they were fixed from my side. (As a side note, one of the bugs detected by me is already fixed in the current version.) In addition, later this library also became handy when I had to clean up the generated test code.

This problem was challenging because I was in need of a library that could parse the source code into an object model that supports the actual Java version and is actively developed. If I did not found this library I either would have had to go with a heavyweight solution such as *Eclipse JDT* or write it myself.

5.3.3 Runner Project Compilation

In an ideal world, runner project generation and compilation would look like the following:

1. Generate runner project layout
2. Copy transformed code snippets
3. Create tool-specific test-drivers and configuration files if needed
4. Compile the project for test input generation

However, it is not always the case. For example, CATG is special and in order to make it work, it has to be compiled with the code snippets and generated files. It means that building a runner project might also have a tool-specific part. Runner projects are compiled by starting an Ant process, but it means that the buildfile also depends on the tool. This problem was not difficult to overcome but it was unexpected and I wanted common functions to be part of the framework.

Moreover, another problem was that some tools (especially Randoop) generated a gigantic test suite, for example, the size of source code of the generated tests is 331 MB (core snippets, 30 second timeout). Of course, this amount of test code for a project containing independent methods of 10–20 lines is unreasonably large, from the framework point of view even this amount of code has to be handled and the compilation of this amount of source code is not trivial. As a fast solution, the heap memory for Ant was increased to 4–8 GB, but compilation still takes 5–30 minutes (depending on the CPU) and we are talking about dozens of experiments. This means that users either have to wait for recompilation or they have to preserve the compiled bytecode before re-running analysis.

However, Ant is quite an old tool and although it is simple, current build systems perform better in terms of performance, mainly because in industry zero build time would be ideal. I have created a pilot version of enhanced code compilation using Gradle and it is able to

compile the same code using only 2 GB memory within 2.5 minutes (using 1 GB memory it is 4 minutes). In the future this solution will be integrated into SETTE, however it can be already used manually by the users – they simply need to compile the code without SETTE and the framework will detect that it is already compiled. This solution also needs further investigation, since the generated test suite has an important characteristic: test code for a code snippet does not depend on other test code. If a set of source files are passed to a traditional build tool, it must assume that there might be dependencies between the source files, however, here it is known that there are not any, thus compilation may happen in separate smaller steps. Build tools like Maven and Gradle already have optimizations, so this solution requires further investigation and benchmarks.

5.3.4 Test Generator Tool Execution with Timeout

There were several problems with test tool execution. One was that half of the examined tools do not provide a time limit command-line parameter and only stop when they have finished the test input generation or have failed due to an error (e.g., internal error or out of memory error). Since some code snippets intentionally contained infinite loops, leading to *path explosion* and in test tool benchmarks a timeout is almost always used, it was necessary to implement a feature which is able to watch the process during execution, measures the elapsed time and is able to terminate the process if the available time has elapsed. Due to bugs in the `ProcessBuilder` class in JDK 6, formerly it also had to save the process outputs to files (however, it was deleted after upgrading to JDK 7).

It was not trivial to kill a process which is started from Java, especially since some tools (e.g., CATG) can be started by calling a script which forks new JVM processes. Killing the complete process tree is not trivial and all the processes must be killed in case of a timeout before starting the test generation for the next code snippet, because a process which remains in the system will still consume a lot of memory and processing power.

Since this kind of process termination is not supported by Java, it had to be done by calling operating system commands. Currently process termination is only supported on Linux (all tools are used on Linux) and done by searching for the processes in the process list and killing them forcefully.

Another problem with tool execution was the parametrization and tool usage. Unfortunately, some tools do not have proper documentation (maybe because they are usually research prototypes). For example, for jPET parametrization is not trivial, the command used by its developers was not published and it was extracted from the Eclipse plugin (which printed the command during generation). Hence, for some tools I had to experiment with its usage and determine how to use it from SETTE.

5.3.5 Handling Raw Tool Outputs

Each tool has its own output format and the parser has to decide whether the generation finished properly or not. Although one might think that a test input generator tool should always terminate properly, my experience has showed that it is not the case. First, my research was started because I wanted to use test input generators for another purpose, however, it turned out that some tools fail for even simple cases, the detailed capabilities are not documented and that is why the failure is even divided into three evaluation result categories. To summarize, the current parser implementations are able to handle all the outputs which were encountered so far and are probably able to handle the outputs of

future executions even for new code snippets, nevertheless, it would be extremely hard to make them complete, since it would need to read and understand the source code and internal behaviour of the test input generator tools.

Detecting the type of failure: The easiest case to detect is when a process was destroyed since it is simple stated in the execution info file. For other tools process exit value may be also used, however, there are tools which always use exit value `0`, even if they had to stop due to an internal error. Hence, sometimes the raw standard output and error output of the tool has to be parsed and errors have to be detected – it is usually not difficult for humans but it is that for programs.

Parsing generated test inputs: For tools that directly generate test code, if the result is present it can be assumed that generation has finished properly and the test suite can be used. For tools that generate test inputs the solution more complicated, because sometimes they only print the generated inputs to the output (e.g., SPF) or do not print anything and the test driver have to print the inputs before calling the code snippet method (e.g., CATG).

All in all, parsing raw tool output is inevitable and the fact that both input values and error messages are often written to the same place makes parsing even more complex. In addition, usually the output of the tool is not documented. Thus, the parser implementations are based on formerly encountered categorized outputs. For each tool there are certain lines, which clearly state that for instance, a language construct is not supported. However, the SETTE framework should fail for any lines or patterns which is not handled and experience has showed that such unhandled lines can appear even after thousands of test generations.

Solving this problem was quite time consuming, since, I had to implement the parsers by running them all the time and handling the unhandled cases.

5.3.6 Test Execution and Coverage Analysis

Code coverage analysis is already solved by dozens of specific tools, however, my requirements were slightly different:

- test execution and coverage analysis should be carried out separately for each code snippet,
- statement coverage shall be measured on the code snippet and on the included methods (if any) considering lines/statements and
- coverage analysis should be a fast process, meaning it is undesired to fork a separate process for each code snippet.

When this functionality was implemented, I used JUnit 3, not JUnit 4 and because of the actual plans[1] I could not upgrade JUnit. The test runner of JUnit 3 lacked several important features, such as timeout for test cases (some tools generate test cases which call infinite loops) and passing the custom class loader (on which the code snippets are

[1] At that time I still pursued the old goal, which was test input generation using symbolic execution for Android software, which only supported Java 6 and JUnit 3

instrumented). Thus, I had to implement a custom test execution framework that was able to execute JUnit 3 tests and satisfy the other requirements.

Stopping a thread (test code) in Java from another thread (test runner) is not trivial if the source of the thread to stop cannot be modified. Unfortunately, in certain cases (especially for infinite loops) `Thread.interrupt()` does not always work and I had to use the `Thread.stop()` method which was already deprecated a long time ago. This solution also required other handlers, such as catching `ThreadDeath` errors on the application level, which is not a good practice.

Later I upgraded for JUnit 4 since EvoSuite could only generate JUnit 4 test suites and so JUnit 3 was not needed anymore. In addition, a new feature had to be implemented lately, handling test case set up methods marked with the `@Before` annotations. Since time was limited, I had to extend my own test runner to handle this case as I did not had the time yet to replace my implementation by using the JUnit 4 runner.

Moreover, requirements have changed over time. Previously, fast test case execution was crucial since the number of generated test cases which reached the 30 second timeout was very low while starting processes was slow. However, including EvoSuite and Randoop in the evaluation increased the number of test cases (hundreds or thousands instead of dozens for certain code snippets) and it also resulted in the growth of the number of test cases which cause a timeout and now the total time coverage analysis scales with the number of these test cases.

In addition, during the last half year snippets targeting multi-threading (sometimes intentionally causing a deadlock) have been put in place which require caution. One solution is that the test executor is able to detect which threads were started by the test case and is also able to detect deadlock and relentlessly kills undesired threads (which may even stay active after the test case has returned). Another solution is that for each test case is executed as a separate process. At the moment both implementations are present in SETTE but the former is used since the latter makes each test execution at least two seconds longer.

The next development task will be to clarify and refactor the coverage analysis component. Although the current implementation works, it is very difficult to maintain. The most probable solution is that the test execution component will be replaced by the default JUnit 4 solution. As JUnit 4 provides a rich, well-documented API, it seems possible to extend it through its public interface to satisfy the requirements of the framework.

5.3.7 Mutation Testing

EvoSuite and Randoop generated test suites which reached better coverage and properly finished test generation for almost all code snippets. From the research point of view, it became necessary to measure the quality of the test suite and one method to measure it is mutation testing.

I have decided to use the *Major mutation framework* [21] because its main strength is that it is able to perform mutation testing on different test suites which test the same code base using the same mutant set. However, there are some drawbacks.

First, Major only supports Java 1.7 and does not work on JDK 8. From the code snippet point of view it is a disadvantage, since mutation testing is not supported for any code which either uses Java 8 language constructs or have calls to the Java 8 API. However,

execution is not a challenge since several JDKs may be installed on the same machine and it is enough to set the `JAVA_HOME` and `PATH` environment variables properly before running Major.

Second, Major is sometimes unable to kill the test execution threads which reach the timeout and the process never finishes. Fortunately, in our experiments mutation testing was only required for the code snippets, and codes which or whose mutants potentially lead to infinite loops were removed before mutation testing. Nevertheless, it is still a limitation.

Mutation testing is currently part of the SETTE framework, but not part of the SETTE application. It is planned to be merged into it, but it is not trivial.

5.3.8 Handling .NET Code

Along with the Java tools, a test input generator tool targeting the .NET platform (IntelliTest, formerly Pex) was also evaluated. The Java code snippets were transformed to .NET manually and IntelliTest executions and result analysis was performed independently from SETTE. However, since IntelliTest also generates test suites that reach high coverage, we wanted to perform mutation testing on it as well. However, there were several problems with the methodology:

- How can one compare mutation analysis of Java and .NET test suites?
- How can one ensure that the same mutant set is used for both platforms?

Although the code snippets are almost the same functionally, on the bytecode/IL code level it is not sure. In addition, we did not found an equivalent counterpart of the *Major mutation framework* for .NET, but a better solution was considered. The idea was to transform the generated .NET test cases to Java test cases and perform mutation testing on the transformed source. The main advantage was that the process of mutation analysis would be exactly the same as for Java tools, thus they can be compared. Validity was not a problem, since the code snippets were not language-specific, since .NET specific language constructs were skipped (e.g., `event`, LINQ). Nevertheless, the problem was still there to transform .NET code to Java. Since the .NET test code was quite simple, I have decided to take all the test code which had to be transformed and implemented transformation by using find & replace and regular expression rules.

5.3.9 Lack of Experience and Time

In hindsight, I clearly realize that the greatest problem was lack of experience and that the number of requirements and the implementation challenges discussed before would have required much more time to be implemented according to the clean code principles [25] with a proper test suite.

Although when I started the development I was already fluent in Java, this was my first big software development project in terms of complexity. The development often ran into dead ends and I had to re-plan certain functionality even 3–4 times. However, the implementation is working and is able to satisfy the requirements regardless that there are smaller internal problems.

5.4 Software Quality, Metrics and Technical Debt

Since the time for development was limited, the list of requirements was long and I ran into several issues during implementation, software quality received less care. Testing was mainly manual and based on the fail-fast strategy of the framework. Critical features, such as coverage analysis, were tested thoroughly, yet manually when it was implemented. Additionally, evaluation results are usually checked against former ones and it is always examined if the newer version of a tool performed worse on a code snippet. Thus, SETTE has 426 unit and integration tests which reach about 10% line coverage, but also have smoke tests which check the evaluation process for the tools and the core snippets.

The SETTE application itself contains about 13000 effective lines of Java code out of the total 25000 without the experiment batch scripts (Bash for Linux, PowerShell for Windows) and the source of the code snippet projects. More than 50% of the classes and methods are already documented with JavaDoc and comments make up about the 20% of source. The SonarQube-measured complexity of the code base is about 2500 which reflects the number of how many times can the control flow split (it is practically the total number of the following keywords: `if`, `for`, `while`, `case`, `catch`, `throw`, `return` (if not the last statement of a method), `&&`, `||` and `?`).

Regarding technical debt, when SonarQube was first put into action more than a half year ago, the reported technical debt was a little more than 90 work days and at the moment it is 39 days. The decrease was a result of fixing more than 400 reported issues in the code. The main causes (85%) of the technical debt are spaghetti code in the tool-specific raw result parser classes, unsatisfactory branch coverage and legacy code which is commented out.

Chapter 6

Results

This chapter gives an example how can be SETTE used and also discusses the scientific results.

6.1 Example Experiment Execution with SETTE

The usage of SETTE reflects our workflow in which we performed all the runner project generations and tool executions on Linux, while the evaluation was carried out on Windows. First, SETTE has to be installed according to the manual[1]. In the following example, the `D:\SETTE` directory is shared from Windows over network and is mounted to `/home/sette/sette` on Linux.

First, make sure that SETTE, the test generator tools and the snippet projects are up-to-date:

```
$ cd /home/sette/sette/sette-tool
$ git pull
$ ./build-sette.sh
$ cd test-generator-tools # SETTE is distributed with download/update scripts
$ ./reset-all-tools.sh
$ cd /home/sette/sette/sette-snippets
$ ./build-all.sh
```

SETTE can be started by the `./run-sette.sh` script without any arguments and will ask the user for the details of the execution, i.e., which snippet project and tool to use and which evaluation task should be executed:

```
$ ./run-sette.sh
Please select a snippet project:
[1] /home/sette/sette/sette-snippets/java/sette-snippets-core
[2] /home/sette/sette/sette-snippets/java/sette-snippets-extra
[3] /home/sette/sette/sette-snippets/java/sette-snippets-native
[4] /home/sette/sette/sette-snippets/java/sette-snippets-performance-time
[5] /home/sette/sette/sette-tool/src/sette-sample-snippets
Selection: 1
Selected: /home/sette/sette/sette-snippets/java/sette-snippets-core
Please select a task:
[0] exit
[1] generator
[2] runner
[3] parser
[4] test-generator
[5] test-runner
[6] snippet-browser
```

[1] https://github.com/SETTE-Testing/sette-tool/wiki/Install-Instructions

```
[7] export-csv
[8] export-csv-batch
[9] runner-project-browser
[10] parser-evosuite-mutation
Selection: 1
Selected: generator
Please select a tool:
[1] CATG
[2] EvoSuite
[3] Randoop
[4] SPF
[5] SnippetInputChecker
[6] jPET
Selection: 4
Selected: SetteToolConfiguration [className=hu.bme.mit.sette.tools.spf.SpfTool,
        name=SPF, toolDir=/home/sette/sette/sette-tool/test-generator-tools/spf]
Enter a runner project tag: test
Snippet project: /home/sette/sette/sette-snippets/java/sette-snippets-core
Task: generator
Tool: hu.bme.mit.sette.tools.spf.SpfTool [name=SPF,
        version=4cd8ac11abee_820b89dd6c97,
        dir=/home/sette/sette/sette-tool/test-generator-tools/spf,
        outputType=INPUT_VALUES, supportedJavaVersion=JAVA_8]
Runner project tag: test
Snippet selector: null
Runner timeout: 30000 ms
Backup policy: ASK
Snippet project: /home/sette/sette/sette-snippets/java/sette-snippets-core
Generation successful
```

SETTE can be completely parametrized through program arguments, thus it enables the user to perform evaluation without further interaction:

```
$ ./run-sette.sh --help
Usage:
--backup [ASK | CREATE | SKIP]         : Set the backup policy for runner
                                          projects (used when the runner
                                          project already exists before
                                          generation) (default: ASK)
--runner-project-tag [TAG]             : The tag of the desired runner project
--runner-timeout [ 30000ms | 30s ]     : Timeout for execution of a tool on
                                          one snippet - if missing, then the
                                          value specified in the configuration
                                          will be used (default: 30000)
--snippet-project-dir [PROJECT_NAME]   : The path to the snippet-project
                                          (relative to the base-directory) to
                                          use - if missing, then the user will
                                          be asked to select one from the
                                          projects specified in the
                                          configuration
--snippet-selector [PATTERN]           : Regular expression to filter a subset
                                          of the snippets (the pattern will be
                                          matched against snippet IDs and it
                                          will only be used by the runner and
                                          test-runner tasks)
--task [exit | generator | runner |    : The task to execute
parser | test-generator | test-runner
| snippet-browser | export-csv |
export-csv-batch | runner-project-brow
ser | parser-evosuite-mutation]
--tool [CATG | EvoSuite | Randoop |    : The tool to use
SPF | SnippetInputChecker | jPET]
```

However, the most convenient way to run experiments is to use the provided batch scripts that will call SETTE with the proper arguments. For example, the following command calls the generator and runner tasks for SPF with 30 second timeout using the core snippets for 10 repetitions:

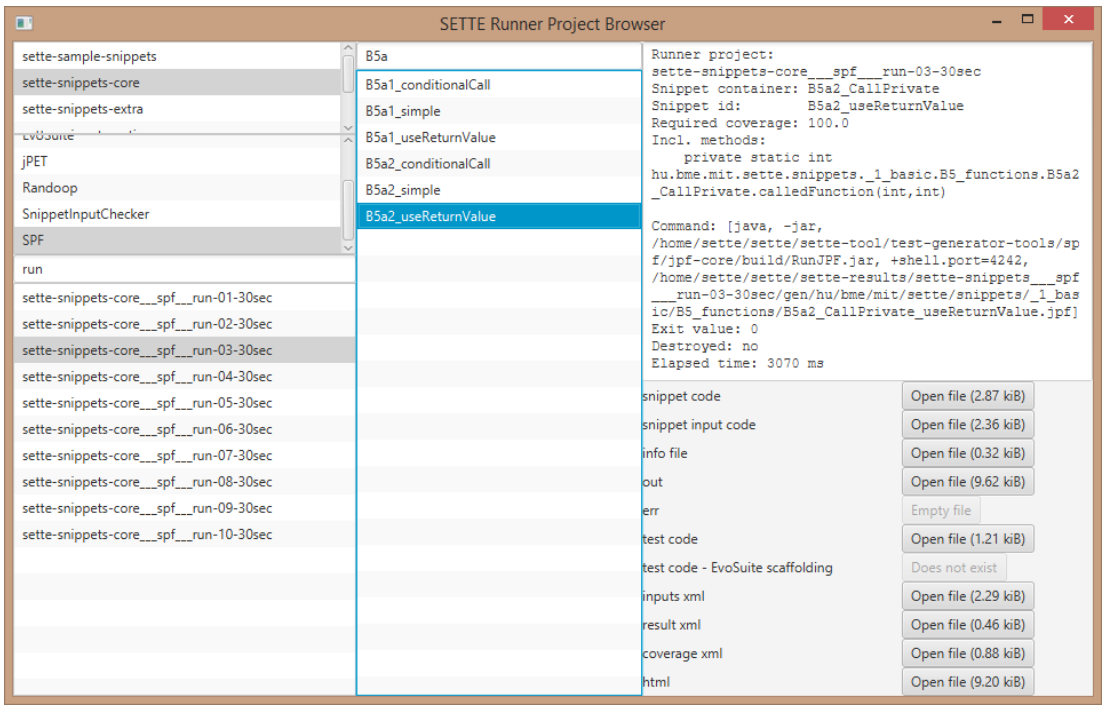
```
$ ./experiment-genrun-30sec.sh spf 01 10
```

Then, the evaluation may be carried out using another batch script from Windows:

```
PS cd D:\SETTE\sette-tool
PS .\experiment-evaluate.ps1 -Project core -Runs (1..10) -Tools "spf" -Timeouts 30
```

After the process has finished, the analysis results are available. The user may directly browse the runner project directory (from `sette-snippets___spf___run-01-30sec` to `sette-snippets___spf___run-10-30sec`) or use the Runner Project Browser component (Figure 6.1). It is convenient to handle dozens of runner projects with this GUI. The interface provides text boxes with which the user may filter the code snippets and it also shows buttons with which the user may directly jump to a particular file belonging to one tool execution.

Figure 6.1. *The Runner Project Browser Interface*



The `*.info` or `*.info.xml` files will contain information about the process execution (called command, exit value, whether it was destroyed by SETTE and the elapsed time) while the `*.out` and `*.err` files contain the standard output and standard error output of a tool execution, respectively.

All the XML files identify the snippet to which they belong and contain data extracted or measured during the evaluation. The `*.inputs.xml` files contain the generated input values (if the tool produced test data) or the number of test cases (if the tool produced test suite code), while the `*.coverage.xml` files contain the measured coverage data (does not exist if the result is `N/A`, `EX` or `T/M`) and the `*.result.xml` files contain the achieved coverages and the evaluation results. Although the result is obvious from the former two, the latter is justifiable since it will exist for all the result types and will be easier to be parsed by a third-party application.

```
<!-- *.inputs.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<setteSnippetInputs>
  <!-- tool and snippet identification -->
  <tool>SPF</tool>
  <snippetProject>
    <baseDir>D:\SETTE\sette-snippets\java\sette-snippets-core</baseDir>
  </snippetProject>
  <snippet>
```

```
    <container>
      hu.bme.mit.sette.snippets._1_basic.B5_functions.B5a2_CallPrivate
    </container>
    <name>useReturnValue</name>
  </snippet>
  <result>S</result>
  <!-- data -->
  <generatedInputs>
    <input>
      <parameter>
        <type>int</type>
        <value>519067</value>
      </parameter>
      <parameter>
        <type>int</type>
        <value>929928</value>
      </parameter>
    </input>
    <!-- other input values -->
  </generatedInputs>
</setteSnippetInputs>

<!-- *.coverage.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<setteSnippetCoverage>
  <!-- identification like in input.xml -->
  <result>C</result>
  <achievedCoverage>100.00%</achievedCoverage>
  <coverage>
    <file>
      <name>
        hu/bme/mit/sette/snippets/_1_basic/B5_functions/B5a2_CallPrivate.java
      </name>
      <fullyCoveredLines>39 40 41 42 43 44 45 46 48 63 64 66</fullyCoveredLines>
      <partiallyCoveredLines></partiallyCoveredLines>
      <notCoveredLines>34 35 56 74 75 77</notCoveredLines>
    </file>
  </coverage>
</setteSnippetCoverage>

<!-- *.result.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<setteSnippetResult>
  <!-- identification like in input.xml -->
  <result>C</result>
  <achievedCoverage>100.00%</achievedCoverage>
</setteSnippetResult>
```

The achieved coverage is also visualized for each execution in the `*.html` files (Figure 6.2). Green lines mean that all the branches were covered, yellow means the branch was partially covered and red means that a line was not covered.

The aggregated results are saved to the `sette-evaluation.csv` file. The file contains one entry for each code snippet describing the achieved coverage, the number of generated test cases, the tool execution time and the categorized evaluation result. These CSV files also contain the name of the tool and the tag of the experiment, thus the CSV files of several experiments can be easily merged or parsed together.

Figure 6.2. *Example for Achieved Coverage Visualization*

```
31 @SetteSnippetContainer(category = "B5", goal = "Check support for private function calls",
32         inputFactoryContainer = B5a2_CallPrivate_Inputs.class)
33 public final class B5a2_CallPrivate {
34     private B5a2_CallPrivate() {
35         throw new UnsupportedOperationException("Static class");
36     }
37
38     private static int calledFunction(int x, int y) {
39         if (x > 0 && y > 0) {
40             return 1;
41         } else if (x < 0 && y > 0) {
42             return 2;
43         } else if (x < 0 && y < 0) {
44             return 3;
45         } else if (x > 0 && y < 0) {
46             return 4;
47         } else {
48             return -1;
49         }
50     }
51
52     @SetteRequiredStatementCoverage(value = 100)
53     @SetteIncludeCoverage(classes = { B5a2_CallPrivate.class },
54             methods = { "calledFunction(int, int)" })
55     public static int simple(int x, int y) {
56         return calledFunction(x, y);
57     }
58
59     @SetteRequiredStatementCoverage(value = 100)
60     @SetteIncludeCoverage(classes = { B5a2_CallPrivate.class },
61             methods = { "calledFunction(int, int)" })
62     public static int useReturnValue(int x, int y) {
63         if (calledFunction(x, y) >= 0) {
64             return 1;
65         } else {
66             return 0;
67         }
68     }
```

6.2 Scientific Results

This section describes what kind of experiments were carried out on which tools and discusses the results of the measurements.

Description of Tools and Experiments As formerly it was mentioned, five Java and one .NET tool were involved in the investigation:

- *CATG*: an open-source tool that generate test input values with symbolic execution.
- *EvoSuite*: an open-source SBST-based tool that is based on genetic algorithms with decent constraint-solving capabilities.
- *IntelliTest:* a closed SE-based tool developed by Microsoft. Its former research prototype is called Pex. IntelliTest is now proposed for developer usage with `Microsoft Visual Studio 2015`.
- *jPET:* this tool is not maintained any more. In terms of mechanism it is quite unique because it translates the Java bytecode to *Prolog* and performs symbolic execution on that. jPET also has a heap model which enables it to deal with objects.
- *Randoop*: an open-source random-based tool.

- *Symbolic PathFinder (SPF)*: an open-source SE-based tool which does not instrument the bytecode, but uses *Java PathFinder (JPF)*, which is a custom JVM.

The result of each test generation is classified in one of the following categories, as it was described in Chapter 3:

- `N/A`: the tool was unable to handle the particular code snippet because either parametrization was impossible or the tool failed with a notification that it is unable to handle the case.
- `EX`: the tool has failed during test generation due to an internal error or exception.
- `T/M`: the tool did not finish within the specified time limit or it ran out of memory during generation.
- `NC` or `C`: generation has terminated successfully and coverage analysis is needed to be done in order to determine whether the required coverage was reached (`C`, stands for *covered*) or not (`NC`, stands for *not covered*).

The following experiments were carried out (Chapter 4):

1. 10 repetitions of experiments with the 300 core snippets using 30 second time limit per tool execution
2. 10 repetitions of experiments of the extended code snippet set (63 code snippets) using 30 second time limit
3. 10 repetitions of performance-time experiments (129 code snippets selected from the core snippets) and four timeout values: 15, 30, 60 and 300 seconds
4. mutation analysis of the test generations for the core snippets

Experiments with the Core and the Extra Snippets The first two set of experiments targeted to find out how are the formerly mentioned features supported by the tool. The results are presented in Figure 6.4 and (Figure 6.3.

Considering that all tools performed test generation for the same code snippet 10 times, it is not trivial how to aggregate the formerly discussed categories. I have decided to choose the most frequent result for a code snippet and if there are several results with the same cardinality, I chose the better one in favour of the tool. For example, if for one snippet the results were `T/M` two times, `NC` four times and `C` four times, then `C` was chosen to describe how the tool handles the particular code snippet. In fact, this only affected EvoSuite.

My findings for the core snippets show that CATG can handle simple code snippets, however it does not support floating-point numbers and cannot handle the code snippets that declare at least one object parameter. jPET performed quite positively for even structures and objects, however, because its special workaround it cannot handle the majority of the code snippets which use the Java API.

SPF was able to finish test generation for the majority of the code snippets, however, for structures, objects and more difficult features it was unable to generate such inputs which reach the required coverage.

Randoop finished with all the test generations in time, however, the number of `NC`s are high because of the lack of constraint solving capabilities. Regarding the Java tools, EvoSuite reached the best coverage. Nevertheless, in this experiment IntelliTest provided the best results and it could not cover only the most difficult cases, such as dealing with collections and dates.

For the extra snippets, the situation is different. EvoSuite was partly able to deal with code snippets targeting the environment and networking by using a virtual file system and virtual network sockets. Nonetheless, these tools cannot cope with several cases and further research and development is required.

Figure 6.3. *Results for the Extra Snippets*

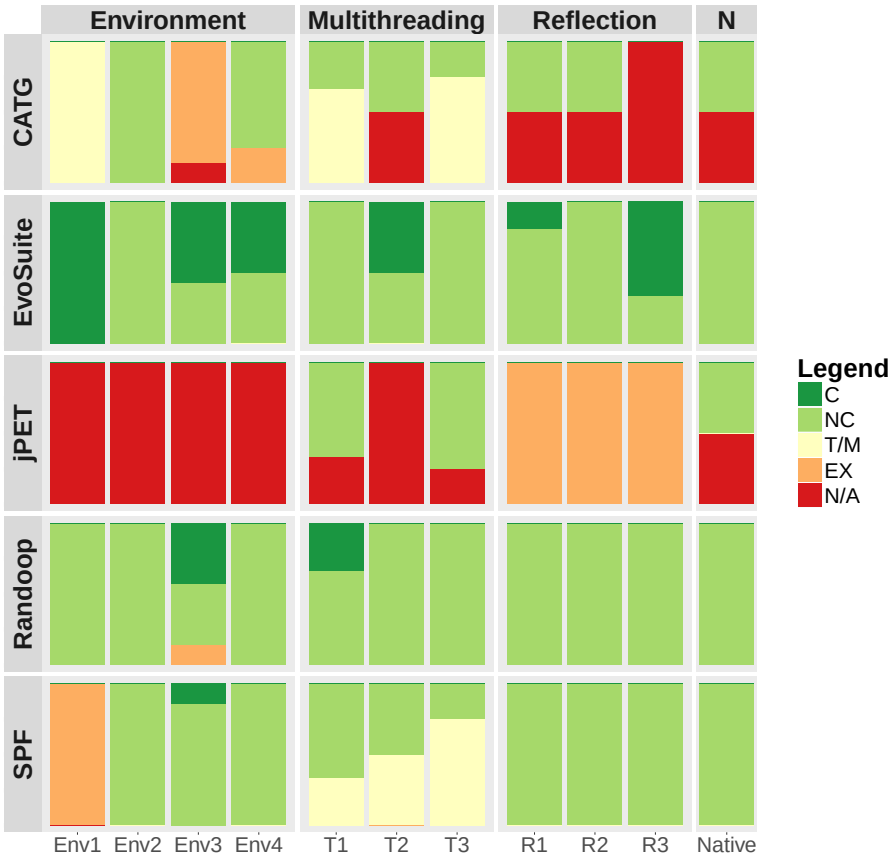


Figure 6.4. *Results for the Core Snippets*

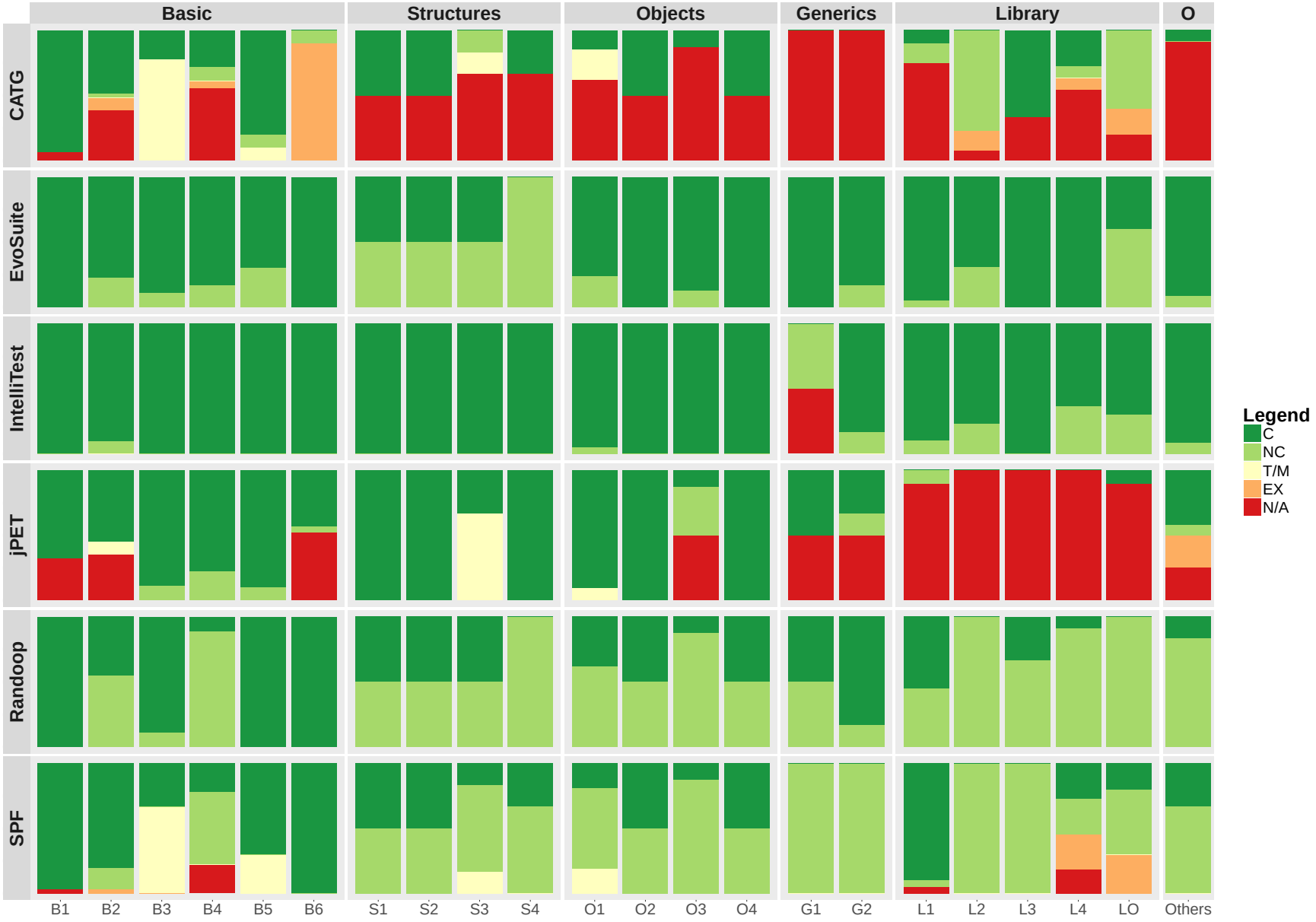
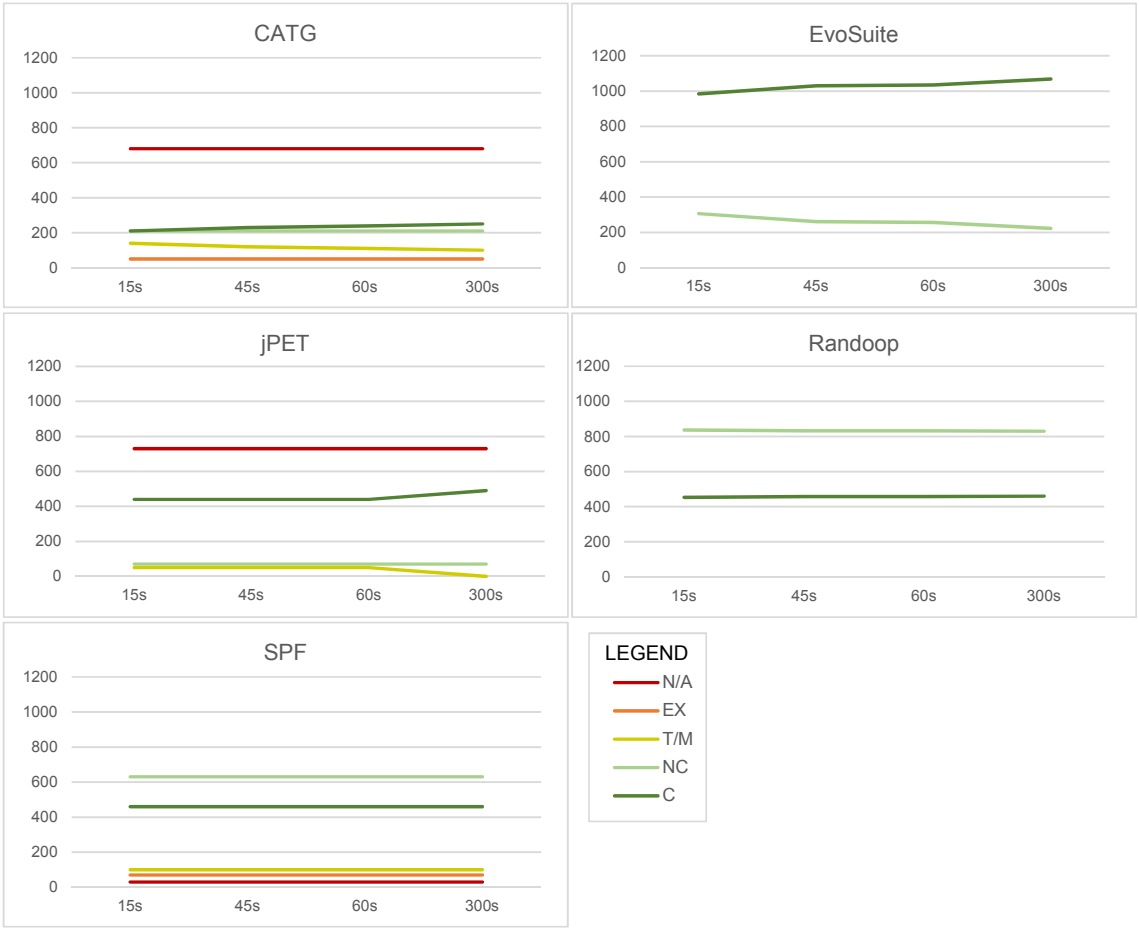


Figure 6.5. *Results for the Performance-Time Experiments*



Performance-Time Measurements These measurements focused on performing experiments with a subset of the core snippets[2] with four time limit values. The motivation for this examination was to discover how does the number of the `C` cases change as the time budged increases. The results are presented in Figure 6.5. In the plot all the test generations are considered individually, which means 1 290 evaluated test generations for each tool.

The evaluation results showed that for CATG and jPET the number of `T/M` data slightly decreases in favour of `C`, while `NC` stays the same: CATG performs better for complex loops and jPET is able to handle 5 other snippets which target complex path constraints. It is surprising that SPF produced the same results with the greatest and smallest time limit values and the reason for this might be that for code snippets with path explosions SPF keeps to discover all the paths in order to provide complete results. As a side note, these tools had to be killed by the framework if they reached the time out and they are not aware of the available time limit.

The findings for Randoop illustrate the general nature of random testing: high coverage is reached quite fast by this technique, however, this technique is not the best choice if full coverage is a requirement. EvoSuite always finished the test generation with `NC` or `C` result and increasing the time limit had a notable effect: with 15 second time-limit it covered 76.3% of the code snippets while with 5 minute time limit the tool was able to properly

[2]B2, B3, O1–O4, G1, G2, L1–L4 and LO features.

handle 82.8%.

Mutation Analysis During mutation analysis altogether 6 236 mutants were generated by the Major mutation mutation framework[3]. The mutation score is calculated by the following formula:

$$score = \frac{(killedMutants)}{(allMutants) - (notKilledByAnyTool)}$$

Table 6.1. *Result of Mutation Analysis*

Tool	Covered mutants		Killed mutants		Mutation score
CATG	2 079	(33.3%)	1 285	(20.6%)	0.2842
EvoSuite	4 687	(75.2%)	2 886	(46.3%)	0.6381
IntelliTest	5 198	(83.4%)	3 480	(55.8%)	0.7696
jPET	404	(6.5%)	215	(3.4%)	0.0475
Randoop	4 743	(76.0%)	2 652	(42.5%)	0.5859
SPF	3 344	(53.6%)	2 263	(36.3%)	0.5004

The *notKilledByAnyTool* is an estimation for the *equivalent mutants*, which is 1 714 in this case. Since the number of mutants were high, it would have been time-consuming to check them one by one which is an equivalent mutant. Regarding all the mutants which were not killed by any tool is a common overestimation in academia [3, 4]. The results and the calculated mutation score for the tools is represented in Table 6.1 that contains the means of the measured values for the 10 repetitions.

In this experiment IntelliTest provided the best performance, followed by EvoSuite, Randoop and SPF. However, it must be considered that Randoop often generated 5000 test cases for even a simple code snippet (this was the test case limit set for the tool). This amount of test code is manually unmaintainable, but it may be tolerable for regression testing. CATG and jPET performed significantly worse, this can be explained by that they failed to generate test inputs for several code snippets.

It must be noted that only EvoSuite supports assertion generation, but this feature was turned off. The reason for this is that although EvoSuite allows the user to set maximum time budget for test input search, test case minimization and assertion generation, these time budgets are independent from each other. On the one hand, it would have been unfair if EvoSuite receives a greater time limit than the other tools. On the other hand, it was not trivial how should be the available time limit be split up between the search and assertion generation phases and the tool does not provide a parameter that sets the total maximum time limit.

[3]The code snippets which may result in an infinite loop even indirectly, were excluded from the mutation analysis, because *Major* does not always enforce the timeout for the test cases.

Chapter 7

Conclusion

7.1 Summary of Contributions

The main goal of my thesis work was to develop a software that is able to carry out experiments for the comparison of test input generator tools. The framework generalizes the evaluation process and provides support for performing experiments for any code snippet set and 5 Java tools, with further possibility to easily add other tools later. The framework is also able to carry out batch experiment runs, perform coverage and mutation analysis and provide a convenient user interface.

SETTE has been open-sourced and has already proven that it satisfies the requirements. Although the original problem does not seem difficult, the list of requirements were constantly growing. I ran into several technological problems during implementation which often needed significant time-investment to resolve. Fortunately, the originally designed architecture proved to be solid since it received only minor modifications during development. However, the internal design and implementation of several components have received significant changes and went through refactoring, mainly because their first version was a pilot, even though functionally correct implementation.

Altogether this 2.5-year-long project not only made me familiar with the world of test input generation, but I have also gained a lot of experience. I learnt a lot about uncommon core Java features (especially the reflection API), Java libraries commonly used in the industry (e.g., Apache Commons libraries, Guava, Jackson, Project Lombok) and software development tools (Eclipse, Git, GitHub and SonarQube). Additionally, the long development project taught me several lessons about prioritization, time management and self-management.

Regarding the original thesis problem defined by my supervisor, in this document I have introduced the reader to the common code-based test generation techniques and to my scientific approach for test input generator tool evaluation. Afterwards, the requirements, specification and architectural design of the elaborated framework have been presented, followed by the discussion of the development process and major implementation problems. Finally, an example execution of the framework was described and the actual results were briefly discussed.

7.2 Future Work

As it was stated before, SETTE is still under development. The next main task is to extend the snippets in the `Env2` (file system) feature and to implement the extra and native code snippets for .NET as well. Regarding the test input generator tools, it is necessary to continuously monitor them and add new ones to the evaluation if it is worthy. However, jPET will be probably removed soon since it was last updated in 2011 and is not developed any more.

In addition, I would like to enhance the evaluation of IntelliTest by integrating it somehow into the framework, thus, its evaluation would be automated like for the Java tools. Moreover, I wish to finish refactoring (especially the component which does the coverage analysis) and replace Ant with Gradle for runner project compilation (it could be even two times faster and should use less memory).

Köszönetnyilvánítás

Mindenekelőtt köszönetet szeretnék mondani konzulensemnek, Dr. Micskei Zoltánnak, hogy többéves segítőkész és kitartó munkájával hozzájárult e diplomamunka megszületéséhez. A közös munka eredményeként nemzetközi szinten sikerült hozzájárulni a tesztgeneráló eszközök fejlődésének a vizsgálatához. További köszönetet szeretnék mondani Salánki Ágnes PhD hallgatónak, aki lelkesen segített az eredmények vizualizálásában.

Emellett köszönöm a családom éveken át tartó lelkesítését és támogatását, amely jelentősen hozzájárult a tanulmányaim során elért eredményeimhez. Végül, de nem utolsó sorban köszönöm Barta Ágnesnek, Cseppentő Bencének és Fejes Endrének hogy többször gondosan elolvasták a dolgozatot és észrevételeikkel segítették a munkámat.

Bibliography

[1] E. Albert, M. Gómez-Zamalloa, and G. Puebla. PET: a partial evaluation-based test case generation tool for Java bytecode. In *Proc. of workshop on Partial evaluation and program manipulation*, PEPM'10, pages 25–28. ACM, 2010. doi:10.1145/1706356.1706363.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Software*, 86(8):1978 – 2001, 2013. doi:10.1016/j.jss.2013.02.061.

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, 2006. doi:10.1109/TSE.2006.83.

[4] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014. doi:10.1002/stvr.1486.

[5] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proc. of the Int. Conf. on Software Engineering*, ICSE '13, pages 122–131. IEEE, 2013. doi:10.1109/ICSE.2013.6606558.

[6] P. Braione, G. Denaro, A. Mattavelli, M. Vivanti, and A. Muhammad. Software testing with code-based test generators: data and lessons learned from a case study with an industrial software component. *Software Qual J*, 22(2):311–333, 2014. doi:10.1007/s11219-013-9207-1.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of Operating systems design and implementation*, OSDI'08, pages 209–224. USENIX Association, 2008.

[8] T. Chen, X.-s. Zhang, S.-z. Guo, H.-y. Li, and Y. Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7):1758 – 1773, 2013. doi:10.1016/j.future.2012.02.006.

[9] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse. Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software*, 83(1):60 – 66, 2010. ISSN 0164-1212. doi:http://dx.doi.org/10.1016/j.jss.2009.02.022. URL `http://www.sciencedirect.com/science/article/pii/S0164121209000405`. SI: Top Scholars.

[10] L. Cseppentő. Comparison of symbolic execution based test generation tools. B.sc. thesis, Budapest University of Technology and Economics, 2013.

[11] L. Cseppentő. Comparison of symbolic execution based test generation tools. Student research conference, Budapest University of Technology and Economics, 2013.

[12] L. Cseppentő and Z. Micskei. Comparison of symbolic execution based test generation tools. In *Proceedings of Tavaszi Szél vol. VI. 2014*, pages 139–149, Debrecen, Hungary, 2014. Doktoranduszok Országos Szövetsége. ISBN 978-615-80044-4-2.

[13] L. Cseppentő and Z. Micskei. Evaluating Symbolic Execution-based Test Tools. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, 2015. doi:10.1109/ICST.2015.7102587.di

[14] G. Fraser and A. Arcuri. Whole test suite generation. *Software Engineering, IEEE Transactions on*, 39(2):276 –291, 2013. doi:10.1109/TSE.2012.14.

[15] S. J. Galler and B. K. Aichernig. Survey on test data generation tools. *STTT*, 16(6): 727–751, 2014. doi:10.1007/s10009-013-0272-3.

[16] ICSE. SBST contest. `http://sbstcontest.dsic.upv.es/`, 2016. Last accessed on 19/05/2016.

[17] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi:10.1145/2568225.2568271.

[18] Institute of Electrical and Electronics Engineers. *Systems and software engineering – Vocabulary*, 12 2010. Standard 24765:2010.

[19] ISTQB. ISTQB glossary. `http://www.istqb.org/downloads/category/20-istqb-glossary.html`, 2016. Last accessed on 19/05/2016.

[20] javaparser. Java 1.8 parser and abstract syntax tree for java. `https://github.com/javaparser/javaparser`, 2016. Last accessed on 19/05/2016.

[21] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA)*, pages 433–436, 2014. doi:10.1145/2610384.2628053.

[22] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. doi:10.1145/360248.360252.

[23] 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 *Search Based Software Engineering (SSBSE), 2010 Second International Symposium on*, pages 101–110, Sept 2010. doi:10.1109/SSBSE.2010.21.

[24] K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *J. Syst. Softw.*, 83(12):2379–2391, Dec. 2010. doi:10.1016/j.jss.2010.07.026.

[25] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008. ISBN 0132350882, 9780132350884.

[26] P. McMinn. Search-based software testing: Past, present and future. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 153–163, March 2011. doi:10.1109/ICSTW.2011.100.

[27] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, Sept 1976. ISSN 0098-5589. doi:10.1109/TSE.1976.233818.

[28] NASA. Symbolic PathFinder – tool documentation. `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc/doc`, 2016. Last accessed on 19/05/2016.

[29] J. Offutt. A mutation carol: Past, present and future. *Information and Software Technology*, 53(10):1098 – 1107, 2011. ISSN 0950-5849. doi:http://dx.doi.org/10.1016/j.infsof.2011.03.007. URL `http://www.sciencedirect.com/science/article/pii/S0950584911000838`. Special Section on Mutation Testing.

[30] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Int. Conf. on Software Engineering*, ICSE'07, pages 75–84, 2007. doi:10.1109/ICSE.2007.37.

[31] X. Qu and B. Robinson. A case study of concolic testing tools and their limitations. In *Int. Symp. on Empirical Software Engineering and Measurement*, ESEM'11, pages 117–126, 2011. doi:10.1109/ESEM.2011.20.

[32] K. Sen. CATG web page. `https://github.com/ksen007/janala2`, 2013. Last accessed on 19/05/2016.

[33] N. Tillmann and J. de Halleux. *Tests and Proofs: Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, chapter Pex–White Box Test Generation for .NET, pages 134–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-79124-9. doi:10.1007/978-3-540-79124-9_10. URL `http://dx.doi.org/10.1007/978-3-540-79124-9_10`.

Appendix

A.1 Versions of Test Input Generator Tools

- *CATG*: janala2-1.03
- *Evosuite*: 1.0.3
- *IntelliTest*: Microsoft Visual Studio 2015
- *jPET*: 0.4
- *Randoop*: 2.1.0
- *SPF*: Mercurial changeset 4cd8ac11abee (`jpf-core`) and 820b89dd6c97 (`jpf-symbc`)

A.2 Used Software Development Tools

For development I used Oracle JDK 1.8.0_73, Groovy 2.4.6 and the Eclipse IDE (formerly Juno and Luna and lately Mars) with the following plugins:

- *Buildship*: Gradle IDE
- *C/C++ Developments Tools*: for executing several run configurations in an order
- *Checkstyle*: coding conventions
- *e(fx)clipse* with *SceneBuilder*: JavaFX development
- *EclEmma*: code coverage
- *FindBugs*: static code analysis
- *Groovy-Eclipse*: test cases were written in Groovy mainly because the ease of use of the language and its *assert* language construct
- *MoreUnit*: easier navigation between SUT and tests
- *SonarLint*: code quality analysis
- Misc.: *Easy Shell* and *ZipEditor*

Other development tools:

- *Ant*: compiling snippet and runner projects
- *Git & GitHub*: version control and wiki pages
- *Gradle 2.13*: build automation system
- *SonarQube 5.1*: code quality analysis