FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Pool++: Enhancing Unit Test Coverage Through Audio Data and Mock Generation

**Pedro Pereira Ferreira**


Master in Informatics and Computing Engineering

Supervisor: José Campos

July 14, 2025

© Pedro Pereira Ferreira, 2025

# Pool++: Enhancing Unit Test Coverage Through Audio Data and Mock Generation

**Pedro Pereira Ferreira**

Master in Informatics and Computing Engineering

Approved by . . . :

President: Bruno Miguel Carvalhido Lima

Referee: Ilídio Fernando de Castro Oliveira

Referee: José Carlos Medeiros de Campos

July 14, 2025

# Resumo

A testagem é uma etapa crucial no processo de engenharia de *software*, a fim de obter um código seguro e isento de erros. A comunidade científica tem feito novos avanços para melhorar a experiência de teste, especialmente no que respeita ao processo de geração automática de testes, melhorando o desempenho das ferramentas criadas para este efeito. No entanto, nos casos em que é necessário lidar com dados complexos, várias ferramentas de geração automática de testes, especialmente o EvoSuite, ainda enfrentam alguns desafios.

Esta tese investiga métodos para melhorar a eficácia dos testes unitários gerados automaticamente pelo EvoSuite, com um foco particular em classes Java para processamento de áudio. O estudo identifica uma limitação na capacidade do EvoSuite para gerar entradas complexas para além de primitivas simples, dificultando a sua cobertura de certos casos. Para resolver este problema, foi desenvolvida uma nova ferramenta, *Pool++*, para alargar o EvoSuite através da criação e injeção de *mocks* com dados aleatórios, permitindo uma cobertura mais ampla e profunda do teste.

O impacto da ferramenta foi avaliado em três dimensões: cobertura do código sob teste, o atraso gerado na execução dos dados falsos e o atraso na execução dos testes gerados. As experiências realizadas em duas classes presentes no conjunto de dados SF110, demonstraram aumentos significativos na cobertura de linha, alcançando entre 81% e 97% das linhas, e para a cobertura de ramo, cobrindo 55% e 88% dos ramos das classes. Embora a *Pool++* introduza uma sobrecarga reduta no processo execução dos testes unitários, devido à geração e integração dos *mocks*, a nova ferramenta não provoca um atraso substancial nos tempos de execução de teste comparáveis.

As análises da escalabilidade da solução apresentada sugerem que a *Pool++* pode continuar a produzir benefícios de cobertura para classes maiores e mais complexas, embora com um ligeiro aumento na sobrecarga de produção quando são necessários *mocks* adicionais. De uma perspetiva geral, a abordagem proposta melhora efetivamente a capacidade do EvoSuite para testar cenários complexos de processamento de áudio, contribuindo para a geração automática de testes de maior qualidade e apoiando o desenvolvimento de sistemas de *software* mais seguros e robustos.

**Palavras-chave:** Testagem de *software*, EvoSuite, Dados de áudio, Mocks, Geração de dados falsos.

# Abstract

Testing is a crucial step in software engineering process, in order to achieve safe, error-free code. Novel advancements were accomplished by the scientific community to enhance the testing experience, especially for the automatic test generation process, improving the performance of the tools for those kind of tasks. Nevertheless, in cases it is needed to handle with complex data, several automatic test generation tools, especially EvoSuite, still faces some challenges.

This thesis investigates methods to improve the effectiveness of unit tests generated by EvoSuite, with a particular focus on Java classes for audio processing. The study identifies a limitation in EvoSuite's ability to generate complex inputs beyond simple primitives, hindering its coverage of certain cases. To address this, a novel tool, *Pool++*, was developed to extend EvoSuite by creating and injecting mocks with randomized data, enabling broader and deeper test coverage.

The tool's impact was evaluated along three dimensions: the impact on code coverage, the mock generation's overhead produced, and the test execution's detain generated. Experiments on two classes from the SF110 dataset demonstrated significant increases in line coverage, reaching 81% and 97%, as well as for branch coverage, reaching 55% and 88% of the classes' branches. Although *Pool++* introduces execution overhead due to the generation and integration of mocks, it does not cause substantial output delay and retains comparable test execution times.

Scalability analyses suggest that *Pool++* can continue to yield coverage benefits for larger and more complex classes, with a slight increase in production overhead when additional mocks are required. Overall, the proposed approach effectively enhances EvoSuite's capability to test complex audio processing scenarios, contributing to the automatic generation of higher-quality test suites and supporting the development of safer and more robust software systems.

**Keywords:** Software testing, EvoSuite, Audio data, Mocks, Unit test generation

# UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

This thesis aligns with specific SDGs, as presented in Table 1. Despite the fake data generation on this work is focused on audio content, the extension of the mocks to components can contribute to a more sustainable development.

Firstly, the research contributes to **SDG 9: Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation** by proposing a replicable approach to enhance automatic test generation tools through the integration of mocks with fake data, to simulate behavior of systems, and software which supports infrastructures. This ensures their reliability and safety, supporting their performance and maintainability.

Secondly, this works contributes as well to **SDG 11: Make cities and human settlements inclusive, safe, resilient and sustainable** by inspiring the creation of mocks for testing services in smart cities, such as smart lighting and efficient transportation systems. By improving the testing processes, the safety and sustainability of urban services are reinforced, contributing to resilient and sustainable urban development.

Moreover, the performance indicators for these contributions include improvements in code coverage metrics (line and branch coverage) and enhanced quality assurance practices for targeted systems and services. Those indicators will ensure the work has improved their testing processes, promoting safer, and more sustainable infrastructures and services available for citizens.

Table 1: Alignment of this thesis with the United Nations Sustainable Development Goals

| SDG | Goal | Contribution | Performance indicators and metrics |
|---|---|---|---|
| 9 | 1 | Replicable idea to enhance an automatic test generation tool containing mocks with fake data, to simulate the behavior of the code utilized by infrastructures and systems. This will ensure the safety of infrastructures, enhancing their performance, as well as their maintenance procedures. | Code coverage scores (line and branch coverage scores). Quality Assurance procedures of the infrastructures and systems. |
| 11 | 2 | The novel idea can be used to inspire the generation of mocks which use fake data, to test services present in smart cities, such as smart lighting, and efficient transportation systems. As a result, it will be ensured the services used are safe, improving the sustainability of the cities, and their communities. | Code coverage scores (line and branch coverage scores). Quality Assurance procedures of smart cities' services. |

# Acknowledgements

Although this is a large-scale, individual piece of work, I could not have done it without the help of several people in my life, and I would like to express my gratitude to them in this section.

First and foremost, I would like to thank Professor José Carlos Medeiros de Campos, for all guidance, being always available to help me with any obstacle found during the thesis' progress.

I would also like to thank my family, especially to my parents, my grandparents, my brother, and my cousins who provided all the necessary conditions for me to succeed in every step in my academic career. Thank you for all your love and support throughout this journey and my life.

I would also like to thank my girlfriend Beatriz Sousa who has always accompanied me throughout the thesis development process, providing all her love, availability, and help so that I could complete this work with success.

I would like to thank my friends from the course, who have accompanied me throughout my journey at FEUP over the last five years, especially Inês Gaspar, Fábio Sá and Lourenço Gonçalves. To Inês and Fábio, who have accompanied me since the first day of university, and to Lourenço, since the 12th year of secondary school. It has been a pleasure to have worked with you on various projects, and all the moments of friendship and companionship over these five years.

Finally, I would also like to thank my high school friends with whom I have kept in touch throughout college life. To you, António Parchão, Bernardo Relvas, Diogo Salgado, Francisco Gil, and Marcos Aires, thank you for all the support, and for believing in me and in my success.

To all, my most deep and sincere thank you.

Pedro Pereira Ferreira

*Tenho em mim todos os sonhos do mundo.*
by *Fernando Pessoa*

# Contents

# List of Figures

# List of Tables

# List of Listings

# List of Acronyms

# Chapter 1

# Introduction

## 1.1 Context

In the software development process, it is becoming increasingly important to develop test cases to make systems expect the requirements and identify vulnerabilities in them.

Over time, the process of developing them has become progressively more effective and automated [4, 18, 19, 20, 32, 33, 34, 35, 42, 56, 76], with the help of various automatic test generation tools, such as EvoSuite [10, 31] and Randoop [64, 65] for Java and, Pynguin [55] for Python.

Significant enhancements have been made to these test case generation tools, including the implementation of various strategies for the generation of test cases [72], the enhancement of test cases' seeds through the use of graph architecture [54], and the optimization of object generation through the use of SQL or web queries, meta-heuristics or even bug reports [7, 57, 59, 63]. These developments have resulted in enhanced code coverage and the generation of test cases capable of detecting real faults in the developed code [4, 76]. Furthermore, the readability of the test cases was enhanced to render them more meaningful to developers [88].

## 1.2 Motivation

The generation of effective test cases represents a significant challenge in the field of software development. It is often constrained by the difficulty of identifying critical cases and the pressure to deliver software promptly, as it was documented throughout the last decades where several companies had losses in the order of millions of dollars due to software faults. A recent example of this phenomenon is the CrowdStrike blackout, which continues to impact corporate entities as they endeavor to restore their operations[1]. This incident exemplifies the potential for significant

[1] Services begin recovery after faulty update causes global IT crash, https://www.euronews.com/2024/07/20/services-begin-slow-recovery-after-faulty-software-update-causes-global-it-crash, accessed 7 October 2024.

infrastructure destruction and, most distressingly, the loss of human life, as evidenced by Boeing 737 Max crashes attributed to defective software[2].

Therefore, investing in enhancing automatic test generation tools will thus contribute to better production of test cases, without the human supervision, and to reduce the incidence of errors.

## 1.3 Problem statement

It is well known from the literature that EvoSuite is the state-of-the-art automatic test case generation tool for Java, with remarkable performance for different types of Java classes and problems, as stated in several comparison experiments with other tools [4, 37, 76], as well as its participation in competitions in the recent years [17, 38, 68, 86]. Unfortunately, even the improvements referred to are insufficient to cover all special cases, especially when dealing with Java classes that requires specific data, leading to less effective test cases and thus unsafe software inception.

Listing 1.1 shows an example of a Java class, adapted from SF110 dataset [33], representing an `MP3` player which has the `run` method (lines 14–58). Although EvoSuite is able to generate a test case that partially covers the `run` method (Listing 1.2), it is not capable of generating or mocking valid objects for line 19, as it can not generate the file content. Thus, the lines after line 19, apart from the `catch` and `finally` clauses (lines 50–57), are not covered.

Hence, there is a clear opportunity to enhance EvoSuite at generating test cases. If there is a way to improve EvoSuite's data pool to create data for this kind of objects, it would be possible to cover more particular cases systematically and effectively.

Listing 1.1: Java code for an `MP3` class.

```
1  import java.io.*;
2  import javax.sound.sampled.*;
3
4  public class MP3 extends Thread
5  {
6   AudioInputStream in = null;
7   AudioInputStream din = null;
8   String filename = "";
9   public MP3(String filename)
10  {
11    this.filename = filename;
12    this.start();
13  }
14  public void run()
15  {
16    AudioInputStream din = null;
17    try {
18      File file = new File(filename);
19      AudioInputStream in = AudioSystem.getAudioInputStream(file);
20      AudioFormat baseFormat = in.getFormat();
21      AudioFormat decodedFormat = new AudioFormat(
22         AudioFormat.Encoding.PCM_SIGNED,
```

[2]How the Boeing 737 Max Disaster Looks to a Software Developer https://spectrum.ieee.org/how-the-boeing-737-max-disaster-looks-to-a-software-developer, accessed 8 October 2024.

```
        baseFormat.getSampleRate(), 16, baseFormat.getChannels(),
        baseFormat.getChannels() * 2, baseFormat.getSampleRate(),
        false);
    din = AudioSystem.getAudioInputStream(decodedFormat, in);
    DataLine.Info info = new DataLine.Info(SourceDataLine.class, decodedFormat);
    SourceDataLine line = (SourceDataLine) AudioSystem.getLine(info);
    if(line != null) {

      line.open(decodedFormat);
      FloatControl volumeControl =
(FloatControl)line.getControl(FloatControl.Type.MASTER_GAIN);
      volumeControl.setValue(-20);
      byte[] data = new byte[4096];
      // Start
      line.start();

      int nBytesRead;
      while ((nBytesRead = din.read(data, 0, data.length)) != -1) {
        line.write(data, 0, nBytesRead);
      }
      // Stop
      line.drain();
      line.stop();
      line.close();
      din.close();
    }

  }
  catch(Exception e) {
    e.printStackTrace();
  }
  finally {
    if(din != null) {
      try { din.close(); } catch(IOException e) { }
    }
  }
 }
}
```

Listing 1.2: Test present in test case generated for the `MP3` class by EvoSuite, which does not fully covers the class under test.

```
@Test(timeout = 4000)
public void test2() throws Throwable {
  MP3 mP3_0 = new MP3("bHf+3W^/$-t^0J##^R@");
  PipedOutputStream pipedOutputStream0 = new PipedOutputStream();
  PipedInputStream pipedInputStream0 = new PipedInputStream(pipedOutputStream0);
  BufferedInputStream bufferedInputStream0 = new BufferedInputStream(pipedInputStream0);
  AudioFormat audioFormat0 = new AudioFormat(1304, 746, 746, false, false);
  AudioInputStream audioInputStream0 = new AudioInputStream(bufferedInputStream0,
audioFormat0, 1304);
  SourceDataLine sourceDataLine0 = mock(SourceDataLine.class, new ViolatedAssumptionAnswer());
  doReturn((Control) null).when(sourceDataLine0).getControl(any(Control.Type.class));
  mP3_0.run(audioInputStream0, sourceDataLine0);
  // // Unstable assertion: assertFalse(mP3_0.isDaemon());
}
```

## 1.4 Contributions

There are four major contributions resulting from the work developed throughout the academic year, which consists of the following:

1. The thesis document, describing all the work completed to achieve the solution developed during this thesis work.
2. A novel idea consisting of utilizing fake data to generate complex objects, without depending on external sources, to generate mocks which can improve the test cases' coverage.
3. *Pool*++, which is the extension of EvoSuite, able to generates mocks containing fake data, to simulate the desired behaviour of the CUT, focused on audio processing tasks, which is available online on the following link: `https://github.com/EvoSuite/evosuite/pull/482`.
4. An empirical study, which describes the experiments done to assess the performance of *Pool*++, and hence, validates the aforementioned idea.

## 1.5 Document structure

This thesis is organized in seven chapters. Chapter 1, this chapter, provides a brief introduction to the problem to be solved by this thesis, as well as a short background of the software engineering, and software testing areas. Chapter 2 presents background knowledge in the areas needed to comprehend this thesis' theme, such search-base software testing, as well as the genetic algorithms, mocking and unit testing, which are used in EvoSuite. Chapter 3 provides the state-of-the-art in search-based software testing, as well as test input data generation area. Chapter 4 describes the tool and its architecture. Chapter 5 presents the empirical study conducted on this work, where it is described the metrics used and subjects under experiments, as well as threats that may compromise the success of this work. Chapter 6 presents this thesis' conclusions and future work. Moreover, this document also includes Appendix A, containing an appendix where it is available the text code of the subjects explored and utilized on the experiments, as well as the complete statistical results obtained from the experiments conducted in Chapter 5, for all the metrics collected for each run of the tool developed on this work, and EvoSuite default version.

# Chapter 2

# Background

As this work inserts into the Software Testing area, several topics must be comprehended to understand with success the content of this work. Moreover, the literature review (see Chapter 3) introduces several contributions for alternative approaches than the one used in this work. As a result, this chapter describes the main areas of Software Testing addressed in this thesis.

## 2.1 Search-Based Software Testing

Search-Based Software Testing (SBST) is one of the main approaches for unit test generation. It consists of the utilization of metaheuristics, such as the GA (guided by a fitness function), to find the best solutions, i.e., unit test cases to assess whether the CUT works as expected [57, 58]. The generation procedure includes the generation of test data and test oracles [12, 35].

Several contributions have been made to improve the aforementioned procedure [7, 8, 54, 57]. In these efforts, several aspects of SBST are enhanced, such as the algorithms used, or the test input data generation process.

A key advantage is the automation provided by the algorithms used SBST. The automation of these processes makes software testing tasks less prone to human errors. Furthermore, it speeds up their execution, reducing the time required for them [58].

Another key advantage of SBST is the scalability it provides to unit test generation process. Due to this approach and its wide range of search space, more complex and large projects can be tested with success. Moreover, the SBST provides the versatility needed to address different testing paradigms (such as functional, structural, or temporal testing), when testing complex projects.

However, there are still some debilities: in fact, the algorithms used by this technique always attempt to achieve a local optimal solution, requiring other algorithms to escape that stage [58]. Another challenge faced by this approach is the design of the fitness function, as it is challenging define a proper one in complex problems, and hence conditionates the performance of SBST in those cases. Moreover, it is very common the aforementioned approach faces some challenges when handling with external environment dependencies, such as databases, file systems, due to their poor coverage scores for functionalities which rely on them, etc. [58].

## 2.2 Genetic Algorithm

GA is an evolutionary algorithm utilized for search tools, especially the ones used for Software Testing, which has its principles based on the genetics selection approach present in biology. The approach has appeared for the first time in the 1970's [52].

That algorithm consists essentially of five major steps: the initial population initizalition, the selection, the crossover, the mutation, and termination steps. The initialization step generates the first population which will be modified to generate the further generations. The selection phase consists of choosing the members of the population that will suffer the transformations provoked by the GA. The crossover phase consists of the replacement of the statements, so that they are weaved between each other, making an analogous comparison to the crossover in chromossomes.

Following the crossover phase, the mutation phase is the insertion of small modifications of the evolved code. In the context of Software Testing, those mutations will assert if the modification provokes bugs in the unit tests generated. It also improves the genetic diversity on them, preventing the algorithm from reaching a local optimal solution.

In the end, the termination step will utilize the next population to iterate them over the algorithm, so that it will generate a certain population which satisfies the stoppage conditions.

This algorithm is not only used for Software Testing. It has, in fact, other applicabilities, in particular to Machine Learning (ML) applications, as well as to optimization applications, due to its robustness and generative capabilities. Moreover, the GA provides a solid adaptability to several contexts, without being restricted to a certain domain.

## 2.3 Unit Testing

Unit testing consists of constructing a test suite by using minimal software test cases. This approach usually isolates one target method from the CUT per unit test, so that the functionality to be tested is isolated from the remaining code. Moreover, the aforementioned approach maintains the CUT to independent from external sources, such as databases, file systems, etc. [76].

The generation of unit test can be developed by humans, or automatic generation tools. On one hand, the human-made tests are tailor-made to functionality to be tested. Nevertheless, there are more prone to errors, due to the fact manual testing is a tedious and repetitive task.

On the other hand, automatic test generator tools provide better coverage scores, and can also save time, as they can generate unit tests in a rapid response period, demonstrated in several studies and competitions [17, 38, 68, 86]. There are several tools available in the scientific community for the most popular programming languages. Pynguin is an automatic test tool generator for Python, and EvoSuite for Java [10, 24, 31, 55].

Unit testing provides several advantages to software testing. In fact, the isolation of the unit tests allows testing parts of the CUT in separate, which facilitates the fault localization task. Moreover, that isolation permits tests being more reduced, enhancing the readability, but also the separation of concerns for each test [45].

However, inadequate usage of unit testing can compromise the quality of a software project if used in an incorrect manner. In fact, a poor usage of unit tests can lead to a false sense of security, since the CUT is not properly tested. Moreover, due to the provided isolation by the aforementioned technique, unit testing may not be adequate to test the CUT's integrations.

## 2.4 Mockings

A mock consists of an object which simulates the desire behavior of another object, without having the real characteristics of the object to be replicated. An example of a mock can be a calendar time service, to simulate the behavior of system, which checks whether the current day is a holiday or not. As a result, the system becomes isolated from that functionality, without being dependent on the real date of the machine [9, 11, 79].

Mocks are commonly misconcepted with 'stubs', a small portion of code (method, class, etc.). On one hand, mocks can contain specific values, as well as the minimal logic to simulate the system under test, especially values and methods that are prone to provoke an unexpected behaviour. On the other hand, the latter only contains the minimal values for the system work smoothly, without containing logic implemented inside of it [11].

There are relevant frameworks that already generate mocks in an automatic approach, especially Mockito, which can generate functional mock objects for unit tests in Java [1]. Mockito produces object instances with simple values, which can be set to have a certain value on a specific condition, and ensures if it returns the desired output, by using the `when()`, and `thenReturn()` methods. That open source framework is already present in a significant amount of projects, becoming one of the most paramount frameworks for mocks in software testing area [11].

This concept is fundamental for software testing, especially for unit testing. In fact, mocking components of the System Under Test (SUT) can auxiliate the simulation of external dependencies, as well as of other internal parts, enhancing the isolation of the software tests generated. Moreover, it can enhance the testing process itself, as mocks simulate a missing component which has not been developed yet, accelerating the testing and software development process as well [9, 11].

Nevertheless, mock objects may also require refactoring the developed code, if they are not used in a proper manner. Moreover, an incorrect implementation of mocks may also lead to a false sense of security as well, giving an output which does not correspond to the actual one provided by the system. As a result, the mock usage must be utilized in a moderate manner, to take advantage of the aforementioned software test component [9].

# Chapter 3

# Literature review

This chapter examines key advancements in automated unit test generation, highlighting the evolution of methodologies and tools within the scientific community. The initial phase involved gathering a core set of relevant papers by exploring the existing literature, including references within the original set and citations of these works. This collection then underwent a two-stage filtering process: an automatic screening based on predefined selection criteria, followed by a manual review of titles and abstracts to assess their relevance for inclusion. This process was inspired by other systematic and exploratory reviews conducted by other researchers [51, 78].

The following sections delineate the process undertaken in each step, as well as the topics of the works retrieved. The studies described present novel techniques in the automated test generation and its data. Furthermore, the use of Artificial Intelligence (AI) and Application Programming Interface (API)s to generate the unit tests and their inputs is also explored in these sections. Besides those sections, other related topics, such as the readability are also discussed in chapter, being succeeded by a final discussion section about the works conducted by the scientific community.

## 3.1 Literature review process

### 3.1.1 Collection Process

In this collection process, it was firstly analysed the base literature of the thesis' theme proposal, which consists in the collection of four papers (see Table 3.1).

After the content analysis of the papers from the original set, it was applied two techniques on that collection. The first one consisted in making a backward snowballing review of the original set up to one layer reference. In other words, it was collected the references present in the papers of original dataset. From this extraction, it resulted in the extraction of 345 documents.

To collect the most recent efforts made by the scientific community, the second approach has been applied: a forward snowballing review of the original set of papers. With this, it has retrieved the papers that have cited any of the four papers contained in the thesis' theme proposal. From this step, it resulted in the aggregation of 1,788 cited documents.

Table 3.1: Original literature set of the thesis proposal. From this collection, it was explored the literature cited by it and the papers which have cited any paper present on this set.

| **Title** | **Author** | **Venue** | **Domain** |
|---|---|---|---|
| "Seeding strategies in search-based unit test generation" | José Miguel Rojas, Gordon Fraser, and Andrea Arcuri (2016) | Software Testing, Verification and Reliability journal, 2016 | Search-based test generation |
| "SQL Data Generation to Enhance Search-Based System Testing" | Andrea Arcuri, Juan P. Galeotti (2019) | Proceedings of the Genetic and Evolutionary Computation Conference | Search-based test generation |
| "Graph-Based Seed Object Synthesis for Search-Based Unit Testing" | Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, Jin Song Dong (2021) | Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering | Search-based test generation |
| "Search-Based Test Input Generation for String Data Types Using the Results of Web Queries" | Phil McMinn, Muzammil Shahbaz and Mark Stevenson (2012) | 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation | Automated test data generation |

In addition, the papers published between 1 January 2000 and 30 September 2024 on top-tier conferences and journals in software engineering were also collected, including:

- International Conference on Software Engineering (ICSE);
- ACM International Conference on the Foundations of Software Engineering Conference (previously know as just FSE or ESEC/FSE);
- IEEE/ACM International Conference on Automated Software Engineering Conference (ASE);
- ACM SIGSOFT International Symposium on Software Testing and Analysis Conference (ISSTA);
- IEEE International Conference on Software Testing, Verification and Validation Conference (ICST);
- International Workshop on Search-Based Software Testing (SBST);
- Symposium on Search-Based Software Engineering Challenge (SSBSE);
- IEEE Transactions on Software Engineering journal (TSE);
- ACM Transactions on Software Engineering and Methodology Journal (TOSEM);
- Empirical Software Engineering Journal (EMSE);
- Information and Software Technology Journal (IST);

From this collection, it resulted in the retrieval of 15,083 documents. With this process, it was possible to not only assemble the past contributions that have led to original dataset from this thesis proposal, but also to join the later works of the scientific community. As a result, the

most complete literature was collected, providing the needed insights of the related work done to software testing, resulting in the 4 + 345 + 1,788 + 15,803 = 17,940 documents retrieved.

### 3.1.2 Automatic filtering with inclusion/exclusion criteria

As there were several contributions found in the literature related to software testing, some criteria for selection were created to filter the most relevant ones, in addition of removing the duplicates. As a result, the following list of inclusion criterion were defined:

- The documents to select must be research articles excluding white papers, tutorials, competitions' results, and editor's messages.
- The documents must contain the keywords "automatic test data generation", or "search based software testing".
- The documents selected must be published between 1 January 2000 and 30 September 2024.

Inversely, the exclusion factors consists of the negation of the items in the inclusion ones. In other words, every document which does not satisfy any criteria define is out of the further step.

This process guarantees, that it is maintained the most significant, recent efforts made in the scientific community in the unit test generation and search-based software testing areas. From this action, it reduced substantially the number of works eligible to further steps, from 17,940 to 139.

### 3.1.3 Manual filtering

Following the application of the selection criteria, a further step was required to assess the relevance of the content of each paper for the purposes of this thesis. As a result of this process, 81 documents were excluded from the selection process, either because they were not relevant to the motivation of this work, because there were documents with similar efforts made, or because they were not accessible for review. As a result, the number of papers collected was decreased to 58.

An example of this filtering was the case of the papers of Tuya et al. [83], and Castelein et al. [21], which described contributions to automatic test generation for SQL queries. While the former work used coverage techniques to test the several features of SQL (such as aggegation, selection, etc.), the latter used search-based approaches, as well as Genetic Algorithms, to also cover the same SQL features. As Castelein et al. [21] cited the first work and having a more recent contribution, it was decided to maintain it and remove the work of Tuya et al. [83].

After the filtering process, it is guaranteed that the most up-to-date and innovative advances were selected. As a consequence, the final set of contributions was selected for full-text analysis.

These steps are crucial for not only verifying that the preceding ones filter the most pertinent efforts from the entire literature, but also for guaranteeing a multitude of enhancements to be investigated. The aggregation of all the aforementioned steps enables any member of the community to reproduce the process and collect a comparable literature set for this thesis.

This literature set is described in Sections 3.2 to 3.5. Section 3.2 presents the innovative techniques to improve the seeding process present in the automatic test tools, which ensures what

inputs are used on the unit tests. Section 3.3 describes the breakthroughs made by the scientific community on the automatic generation of tests using API and AI, especially Large Language Models. Section 3.4, presents the novel techniques of generating test inputs for unit tests, while Section 3.5 describing other relevant aspects studied on automatic software test generation area. These sections are followed by a conclusive one, delineating how this work is applicable to the fields of study mentioned, and discussing the efforts found in this literature review process.

## 3.2 Seeding strategies in search-based unit test generation

### 3.2.1 DynaMOSA Algorithm

Panichella et al. [67] proposed the novel algorithm called DynaMOSA: an extension of the Many-Objective Sorting Algorithm (MOSA) [66] which dynamically selects test coverage targets based on control dependency hierarchy. Their solution does not only attempts to augment test coverage scores, but also maximize the code coverage as well. Other test adequacy criteria, such as branch, statement, and mutation coverage, were also improved.

One key benefit is the dynamic target selection, which allows MOSA to dynamically select only reachable values. As a consequence, it reduces the computational costs and enhances the efficiency of the algorithm. Moreover, the preference-based sorting, which combines the sub-set Parteto solutions dominance with the control dependency target, to cover the closest uncovered targets, which enhances the efficacy of their solution. Also the experiments conducted confirms the augment of the code coverage, when DynaMOSA is compared to traditional many-objective algorithms, by increasing the coverage scores to 12% of MOSA.

As future work, the authors suggest the integration of non-coverage criteria in the algorithm, as well as enhancing the parametrization of the test size, Pareto-based ranking, and number of uncovered targets. Furthermore, the combination of their algorithm with random testing is another planned enhancement to be done; however, the authors predict it will face some challenge when it comes to handling with complex-structured input and data types.

### 3.2.2 Addressing External Dependencies

Arcuri et al. [8] address the challenges present in automated unit test generation for Java classes related to the external environments, such as file system, system time, or network. To solve them, they propose an enhancement to the tool, to manage those dependencies: by adding bytecode instrumentation and environment mocking for class isolation from the environment under test, it leads EvoSuite to the generation of stable, high-coverage unit tests.

Their approach is tested in more than 11,000 classes, which agrees with the stated enhancements: in fact, their solution achieves increases in 80% and 90% of branch coverage scores in practical real-world cases. Moreover, the number of unstable tests is decreased. Nevertheless, their apporach still faces some challenges with particular project examples, in particular with multi-threading projects, as well as GUI software, due to the fact there are external dependencies

that are not fully handled. Other cases, as projects related to file handling, networking, as well as databases still needs more improvements to be fully implemented on their approach.

As their work still faces some challenges with file processing and handling cases which requires file inputs, An eventual enhancement to this work would be using mock data, so that EvoSuite is capable to test other behaviors which are not covered with the previous mocks. Moreover, it can also improve the handling of complex inputs, due to the our solution's capability to populate object of complex classes, without needing to be dependent to the external environment.

### 3.2.3 Comparing Random and Evolutionary Search Techniques

The work of Shamshiri et al. [77] compares random and evolutionary search techniques (GA and Chemical Reaction Optimization (CRO)) for automatic test suite generation tasks, particularly for object-oriented Java programs. To assess their solution's performance, the authors conducted experiments utilizing EvoSuite over 1000 classes of SF110 dataset. Additionally, they examine the impact of several branch types on different kind of approaches.

Their solution is, in fact, a novel contribution to the scientific community, as they are the earliest work to introduc the CRO. Moreover, the experiments demonstrates that both approaches achieves similar branch coverage scores throughout the different classes under test. However, it is also possible to observe that both approaches cover different kind of branches: while GA has more effectiveness on branches that are benefitial to the its fitness function, the random-based approach outperforms the others on branches which does not have those kind of charactheristics.

The Shamshiri et al. [77]'s effort provides a paramount study about the impact of the algorithms utilized in EvoSuite. However, the authors observe there is potential to improve their solution, as there is a significant amount of branches that can be converted to gradient ones, enhancing the evolutionary algorithms' performance. Moreover, as their approach cover different types of branches, a relevant enhancement to the aforementioned work would be the exploration of tailored fitness functions is also another improvement regarded, according to the authors.

### 3.2.4 FinHunter Framework

Ding et al. [30] presented a novel technique called FinHunter: a search-based test generation framework which improves the GA (used in EvoSuite and Randoop) to address challenges in structural testing of FinTech systems. Some of the debilities are related to the poor test input generation for the unit test generated on that area, such as insufficient seed inputs, and implicit constraints between fields.

Moreover, their solution includes two major techniques: the gene-pool expansion, and a novel technique that handles with several levels of crossovers. While the former addresses insufficient seeds inputs, the latter one aims to break implicit constraints. The experiments conducted by the authors show that FinHunter not only outperforms the traditional genetic algorithm, but also provides a better performance than a practical case, which is the one used by Ant Group.

However, the effort made by Ding et al. [30] still faces some challenges, as they are not capable to distinguish different fields present in the constraints throughout the genetic phase, due to the randomness of the assignment of the genetic operators FinHunter uses. Moreover, handling with a significant amount of historical data in a large-scale project is still a time-consuming process, as well as with the algorithm's data bias, which decreases the effectiveness of their approach. Hence, the ambiguous test inputs created in distinct contexts can also impact FinHunter's performance.

### 3.2.5 TACKLETEST: Type-Based Combinatorial Testing

TACKLETEST is a tool built on EvoSuite and Randoop, which enhances automated unit testing in Java applications using a novel type-based combinatorial testing technique, created by Tzoref-Brill et al. [84]. Using Combinatorial Test Design (CTD), their approach enables efficient computation of type-related coverage scores, that are overshadowed by the traditional code-coverage criteria (such as branch and line coverage) in a white-box test generation application. By addressing type-related behaviours, TACKLETEST allows a more comprehensive exploration of application methods, especially for handling complex object states, as well as type hierarchies.

Through the implementation of real-world case studies, such as the IBM and SF110 datasets, the authors have provided empirical evidence that substantiates the efficacy of the tool and its industrial applicability. Moreover, the conducted experiments facilitates the acquisition of insights by the industry, enabling the exploration and enhancement of their tools for such environments.

However, TACKLETEST faces challenges related with scalability for more methods with simpler interfaces. Furthermore, the dependency on the test generation tools limit the tool's utility on other contexts. Thus, it leads to a limited utility, as type-based testing requires more complex interfaces for the methods unded test. Nonetheless, the aforementioned approach faces difficulties while handling with complex objects states of certain domains.

### 3.2.6 Adaptive Fitness Functions for SBST

The effort made by Xu et al. [91], the authors propose an adaptive fitness function for SBST, aiming to enhance path coverage of the generated unit tests. The fitness function calculates the Expected Number of Visits which are modeled using an absorbing discrete time Markov chaining tuple. Thus, they are used to measure the branch hardness of the CUT.

The experimental results demonstrate a significant effectiveness improvements of the novel fitness function, when compared to the traditional equivalents. Moreover, by adapting itself to different testing scenarios by heuristically tuning its parameters, the function provides flexibility and better performance compared to static counterparts.

However, the metrics used to measure the function's effiency do not show any significant enhancements, due to the computational overhead, resultant from the parameter optimization and branch analysis. Moreover, their solution still faces some challenges when the function is used in large-scale problems. Their solution can still be improved in the parameter tuning, by utilizing more advanced techniques in the heuristic methods, as suggested by the authors.

### 3.2.7 EvoSuiteAmp: Enhancing Developer-Written Unit Tests

Another paramount contribution is the work of Roslan et al. [74], which presents EvoSuiteAmp, an altered version of the two state-of-the-art test generation tools EvoSuite and DSpot aimed to improve developer-written unit tests by killing their mutants. While their solution utilizes the unit tests as seed and evolves them via fitness-guided mutation analysis, DSpot attempts to enhance the existent tests with minimal modifications.

To assess their tool's performance, the authors conduct experiments on Detects4J a well-stablished benchmark in software testing studies. In fact, more capable to kill more and target more unique mutants, when compared to the DSpot. Furthermore, their proposed solution is more capable to modify the developer-written tests written tests, enhancing its flexibility.

Nevertheless, the authors claim the mutations in the tests cause significant differences between them, which hence reduces the readibility of them. To solve the mentioned challenge, Roslan et al. [74] suggest enhancing DSpot with meta-heuristic parameterization, as well as better utilization of the test input generated by EvoSuite and its data structure.

### 3.2.8 EvoObj: Object Construction Graphs for Test Seed Synthesis

Lin et al. [54] put forth a novel, systematic approach to test seed synthesis on object-oriented programs. It involves constructing an object construction graph to capture relevant object states and synthesizing test templates with mutation points, which they refer to as EvoObj. The results demonstrated that the tool outperforms EvoSuite in terms of branch coverage, as evidenced by the outcomes observed in 2,750 methods across 103 open-source Java projects.

As a consequence, the enhancement in branch coverage is more pronounced when utilizing more robust genetic algorithms, such as DynaMOSA and MOSA. On the one hand, EvoObj employs algorithms that are sufficiently general and flexible to be implemented in any programming language that necessitates complex object inputs for test generation. On the other hand, it exhibits some performance issues when constructing extensive object construction graphs.

### 3.2.9 PUT: Pattern-Based Unit Testing for TypeScript

Thu et al. [81] expose the development of PUT, a system that uses the ***P***attern-based ***U***nit ***T***esting approach to generate test data for web applications written in TypeScript. The work has contributed with two innovative steps on the test generation, improving the function coverage (with increases varying from 3.60% to 23.80%), statement coverage (with increases varying from 0.50% to 27.40%) and branch coverage (with increases varying from 3.30% to 39.20%).

Nevertheless, there are numerous patterns that remain undetected due to the constraints of the implemented approach. Moreover, the static analysis of the code under test has the effect of increasing the time complexity of the algorithms. Furthermore, Thu et al. [81] aposits the outcome is reasonable, given the utilization of multiple APIs in web development. Potential avenues for advancement include the automation of mocking frameworks and the expansion of supported syntaxes, which could prove beneficial for its endeavor.

### 3.2.10 Meta-GA: Hyper-Parameter Tuning for Test Case Generation

The work of Zamani and Hemmati [93] addresses challenges in hyper-parameter tuning for search-based test case generation. The authors propose a novel technique, Meta-GA, which estimates the potential gains for each class under test. Their approach aims also to enhance the test efficiency and effectiveness of SBST.

The proposed tuning method significantly improves test case generation performance, by reducing computational costs and enhancing the precision of generated test cases. In fact, the experimental results demonstrate that optimized hyper-parameter lead to higher coverage metrics compared to default settings. Moreover, the analysis of those results confirms reproducibility and robustness across different testing environments.

Nevertheless, the aforementioned approach does not provide enough consideration on non-functional aspects, such as the execution time and resource utilization. As a result, their algorithm requires a significant amount of resources on the initial tuning, especially for projects of lower dimensions. When it comes to further work, the authors suggest to expand their solution to other domains of software testing, as well as exploring different interaction between hyper-parameters. As some instatiators method's parameters have some crucial parameters, as well as on the prompt parameter tuning, another future effort to be conducted would be the parameter-tuning of those parameters, to enhance the novel tool's capabilities.

### 3.2.11 Reproduction of Crashes in Search-Based Strategies

Derakhshanfar et al. [29] assess reproduction of crashes in search-based strategies, comparing the traditional test seeding and behavioral model one. It is observed the increase in performance of these techniques, when compared to the base test seeding and no seeding used at all. The study is concepted by the analysis of various approaches of that kind.

The behavioural model seeding technique consists in the learning of how classes are used of the SUT, as well as on the unit tests. This approach has a better performance when compared to the others studied, with better results in initializing the search, as well as reproducing the crashes. The increase of crashes reproduced is at least by 6%, when compared to the other studied techniques.

However, some deficiencies are found with these approaches. In fact, the generation of the seeds in the behavioral mode can be time consuming, as it is an inference model. Additionally, as the lenghth of the abstract object is static, it does not provide the flexibility needed for the model seeding, resulting in another negative factor. The existent test cases are not as general as they should be, which leads to a prevention from the crash reproductions.

A future contribution to the aforemention work is the enhancement of seed generation's process for unit tests. By delegating the responsibility for seeding to the solution presented in this paper, being just responsible for learning the usage of classes in SUT, the effort Derakhshanfar et al. [29] can be free of the time-consuming tasks that decrease the performance of the approach.

### 3.2.12 Fitness Landscape and Genetic Algorithms for Unit Test Generation

The work of Albunian et al. [2] analyzes the fitness landscape of unit test generation, while using Genetic algorithms. The presented methodology consists in calculating the metrics, as well as the number of improvements of the algorithm while generating the unit tests. It is stated that the landscape is often dominated by detrimental plateus, compromising the success of the methodology.

Those results were observed in the experiments conducted by the authors, where they compare the use of the fitness landscapes and traditional indicators. In fact, the classical ones are indicative of well-searchable problems, in unit test generation context; however, the former indicator for the majority of problem instances is characterized by the aforementioned plateus. Those plateaus were generated by private methods, boolean flags, and exception-throwing methods.

A possible improvement to the Albunian et al. [2]' work would be the generation of mock instances for the respective unit tests generated by their solution. In fact, some methods can throw exceptions, due to the poor creation of the necessary instances. Hence, providing mocks that can simulate the required object or method would prevent the generation of the aforementioned exceptions, and thus enhance the coverage scores obtained for the respective CUT.

### 3.2.13 Enhancing Automated System Test Generation for Web/Enterprise Systems

Arcuri and Galeotti [7] represents a novel search-based approach to enhance automated system test generation for web/enterprise systems that interact with SQL databases, by introducing SQL heuristics as secondary objectives to optimize and by generating SQL data directly as part of the search. In this experiment, it was found that generating SQL data directly as part of the search led to strong improvements in code coverage, with statistically significant improvements on three of the five SUTs and increases the average code coverage in 18%. Furthermore, the main contribution of the paper is the fact it is the first in the literature that automatically generates "white-box", system-level tests.

However, the aforementioned approach has some limitations, as the heuristics used isolatedly limited improvements to code coverage, and only one of the five SUTs was used for that. Because of that, the authors of the paper recommend the exploration of other types of system testing, as well as other database technologies. Moreover, the authors also attempts to reach more accurate unit tests that use fake. Hence, it is possible to conclude this work is a similar to work done by Arcuri and Galeotti [7] data to simulate more real and complex inputs on the systems under test.

### 3.2.14 SUSHI: A Tool for Generating Complex Test Inputs

Braione et al. [15] develops SUSHI: a standalone tool to generate complex test inputs for the cases. It can be used in conjunction with other test generation tools to complement the unit tests generated by them. SUSHI collects the dependencies between the system under test and the input data structure, represented by path conditions. Then, generative algorithms are used to create the methods satisfying the path conditions collected. Finally, EvoSuite is used to refine the test sequence with iterative mutations and combinations of the methods created.

The experiments conducted indicate SUSHI achieves higher branch coverage, while compared with other test generation tools, as it can reaches up to 97% of branch coverage. Additionally, it can handle with complex data structures and abstract data types, as well as generate test cases with more structural dependencies on them. Moreover, the tests generated are valid and diverse, as well as robusted by the symbolic execution and meteheuristic search used in the moment of their generation. Nevertheless, the resultant computational overhead prevents the mentioned solution from scaling to complex programs, limitating their performance on projects of that kind.

The purpose of this work is similar to the effort made by Braione et al. [15]: generate more complex inputs for unit tests generated by an external tool, such as EvoSuite. However, the Braione et al.' work is not capable to provide the required data for all the cases. Hence, a possible improvement to the aforementioned effort is to enhance the range of data SUSHI can provide, such as synthetic data to simulate complex file's content. As a result, the work conducted in this thesis can improve and overcome the contribution made Braione et al. [15].

### 3.2.15 Whole Test Suite Approach to Search-Based Test Generation

The work conducted by Rojas et al. [73] presents an in-depth empirical study comparing the effectiveness of the traditional one-goal-at-a-time apporach to search-based test generation versus whole test suite approach. It is stated in the experiments that the whole test suite technique generally outperforms the traditional one. Because of that, the presented technique is now utilized on several test generation tools, as the case of EvoSuite.

In fact, that out-perfomance is supported by the better coverage results achieved by the approach presented in their work. Furthermore, the use of an archive also improves the results in the significant amount of the cases. Nevertheless, it is planned to test their work in industrial context, to assess the performance in the real-world context.

The effort made by Rojas et al. [73] is fundamental for validating the purpose of this solution. In fact, this work is intended to increment the data pool of EvoSuite, this can lead to a paramount improvement for that technique to generate more complete unit tests.

### 3.2.16 Multiple-Searching Genetic Algorithm for Test Suite Generation

The effort of Khamprapai et al. [49] examines the efficiency of a Multiple-Searching Genetic Algorithm (MSGA) for creating whole test suites, focused on achieving higher branch coverages and fault detection in complex programs. Their study assesses the performance of MSGA being integrated in EvoSuite, by comparing it with other traditional GAs.

In fact, their evaluation indicates an increase of efficacy, as MSGA augments the fault detection (being between 39.40% and 39.70%), as well as the code coverage (up to 55.70%), when compared to the other traditional algorithms. Moreover, as MSGA generates more test suites in the same time budget, the experiments indicate an enhancement of the efficiency as well. As a result, the Khamprapai et al. [49]'s solution enhances the whole test suite generation process.

Nevertheless, the parameter tuning is not fully explored in their work, which can potentially enhance their approach. Furthermore, the authors suggest a future exploration of the algorithms used, to enhance the performance of the genetic selection process. If this work enhances the EvoSuite, it will also be a reasonable enhancement in the Khamprapai et al.' contribution, for handling more complex inputs, when it comes to the data input generation process for unit tests.

### 3.2.17 Input Domain Reduction in Search-Based Test Generation

Several works attempts to augment test input pool, in order to provide more adequate inputs for the unit tests. However, Hassoun et al. [44] investigate the impact of input domain reduction for SBST, by removing irrelevant input variables using static dependence analysis derived from program slicing. The authors also explore the effiency, as well as the effectiveness, of their approach in Search-Based (SB) techniques, such as hill climbing, evolutionary testing, etc.

The experiments conducted by Hassoun et al. [44] indicate the increase on the performance for specific algorithms, by achieving 100% success rate on the branch coverage with their novel approach, as well as for the evolutionary algorithm. However, the performance on the Hill Climbing does not always provide succesful results, as significant amount of branches does not show any enhancement, according to the stochastic variation metric. Moreover, it is stated that there is no improvements of their approach, when used upon Random search, due to the lack of statistical evidence of the existence eventual enhancements.

The aforementioned work is, in fact, a novel approach that utilizes the opposite method used in a significant amount of the contributions done by scientific community, by removing input variables that does not impact the branch coverage. Despite the focus of this work is to augment the test input pool, so that EvoSuite has more available data to generate enhanced test suites, the work of Hassoun et al. [44] also inspirtes this effort to generate the essential data needed for the test suites, without additional inputs needed.

### 3.2.18 Seeding Strategies for Search-Based Unit Test Generation

Rojas et al. [72] explores different seeding strategies for search-based unit test generation, such as constant and dynamic seeding, seeding constants, types and previous test cases, and the values observed at run time. They also evaluate the effects caused on branch coverage and the effectiveness while detecting faults and come to the conclusion that the effectiveness varies with the number of constants available on the system under test.

Their work provides strong statistical evidence that the use of appropriate seeding strategies boosts the unit test generation. However, the study is restricted to EvoSuite's parameter configurations and seeding strategies used, meaning that this thesis can be studied, analyzed, and, eventually, improve the techniques mastered in the aforementioned effort.

### 3.2.19 Networking Testing for Java Projects

Arcuri et al. [9] presents a novel technique for networking testing, by simulating TCP/UDP functionalities in Java projects. It consequently allows testers to automatically test projects with those kind of functionalities. The effort made constitutes a paramount contribution, as networking is a common feature, being present in 11% of the projects explored for the experiments.

The experiments conducted by the authors indicate the proposed solution on the paper has contributed positively for the solution of the mentioned problem. In fact, it is stated that 63% of the networking-related classes are fully tested, increasing in average to 20% of the line coverage. Moreover, it is raised up to 40% for classes that significantly depend on those functionalities. The behaviour simulation is achieved by creating mocks, that simulate the behavior of the network-related methods of the classes assessed.

Nevertheless, the resulting overall increase in coverage across the entire corpus is only 1.50%, which indicates that the aforementioned approach can be enhanced. As the authors themselves acknowledge, the proposed solution is not adequately equipped to handle cases involving specific types of packages, particularly RMI and NIO. Furthermore, the authors propose enhancements to the readability of the generated tests and the reuse of the framework developed for use with multiple test generation tools, which they believe would be beneficial for the project.

The aforementioned approach is crucial to the advancement of that work, as none of the mocks utilized in this solution possess the capacity to generate networking packets, or useful input data. Consequently, this solution is unable to accommodate tests of that nature, as it does not have the data needed to replicate the intended behavior. It would be advantageous for these two contributions to complement one another, thereby addressing the shortcomings of both.

### 3.2.20 Defect-Prediction Guided Test Generation

The work of Perera et al. [69] introduces a novel approach to SBST (SBST$_{DPG}$), guided by defect prediction to prioritize testing efforts on potentially defective code regions. Their work is also relevant work, as traditional techniques aim to maximize the code coverage, which does not necessarily leads to the maximization of bug detection. Their proposed solution uses Schwa, a defect-prediction tool, to allocate resources based on defect likelihood present in EvoSuite.

The experiments conducted by Perera et al. [69] validate the effectiveness of their approach, as it captures more 13.10% of the bugs on average, when compared to traditional search-based techniques. The improvement of the efficacy is caused by the detection of more unique bugs that there once undetectable. Moreover, the authors observe an increase of the performance of their tool throughout the experiments, when comparing to the traditional ones, in cases it has sufficient budget time to generate the test suites.

However, their approach is limited to class-level defects, which constraints the range of defect coverage, besides the heuristics used for defect priorization may not cover missed bugs. To address that problem, the authors plan to use a more powerful defect-predictor, using wide range of features to those tasks, and reduce the size of the test suites generated. The scalability to other systems can

still be tested with more complex cases, which may be solved by expanding the experiments to distinct datasets, as authors suggest.

### 3.2.21 Relational Schema Integrity Constraints for Test Generation

The work of Mcminn et al. [60] investigates the usage of relational schema integrity constraints and their usage in testing purposes. The authors design an appproach which assess the test coverage criteria by validating systematically integrity constraints, such as primary and foreign keys, the uniqueness and not null constraints, and integrity constraints defined by CHECK constraints. These criteria are framed to address inconsistencies in DBMS interpretations of SQL standards and evaluate the fault-detection of the test suites generated by their approach.

The authors have two proposed different strategies to address the problem presented: the primary one uses search-based algorithms to find the keywords needed for the generation of the tests; while the latter one utlizes a data generator called Random+, which creates test input data based on the CHECK constraints found in the schema. The experiments conducted indicate that their contributions can generate valid unit tests that achieve the high-value coverage scores for the criteria found in the schemas under test.

As future work, the usage of real-world examples, as well as expanding the Database management systems used, will enhance the validity and the generalizability of the results presented, as suggested by Mcminn et al. [60]. Techniques to identify missing constraints can also be improved. Moreover, the metrics can also be studied, to perceive their impact on the experiments conducted.

### 3.2.22 Memetic Algorithm for Test Suite Generation

Fraser et al. [36] present a Memetic Algorithm which extends the GA for EvoSuite to generate test suites with branch coverage by incorporating local search operators to optimize various types of test data. Their work provides advantages, as it increases EvoSuite's performance. In fact, the Memetic Algorithm approach can increase branch coverage by up to 53%, when compared to the standard GA. Those results are due to the fact it is able to cover significantly more branches to the traditional, surpassing existent challenges on a specific benchmark.

### 3.2.23 Search-Based Heuristics for Model-Based Testing

The work proposed by Ali et al. [3] explores a novel approach using search-based heuristics to generate test data from Object Constraint Language (OCL) constraints to automate model-based testing in industrial applications. The approach presented on the paper also evaluates the usage of those heuristics using three algorithms (Genetic Algorithm, (1+1) Evolutionary Algorithm, Alternating Variable Method) and evaluate the approach presented's performances.

The experiments conducted on the work indicates that the approach presented improves the effectiveness and efficiency of the generating data process. In an overall observation, the Alternative Variable Method is the one with the best performance. Furthermore, the branch distance heuristics presented also increases the performance of the search-based algorithms.

Nevertheless, the aforementioned approach does not fully cover all the cases needed to transform the OCLs into the desired Unified Model Languages (UML)s, as it is stated by its authors. That is due to the fact it does not cover the requirements needed for more complex cases, which also happen in other solutions in the existing literature. A valid enhenacement to be made to this work is to expand its range of the fake data generation, so that it can cover more complex inputs, as the ones presented in Ali et al. [3]'s work.

### 3.2.24 Parameter Tuning in Search-Based Software Engineering

Arcuri and Fraser [6] present the results of an empirical analysis on parameter tuning in search-based software engineering. The authors find that process useful to improve its performance in certain situations, despite being possible to achieve a reasonable performance using the default settings present in the literature.

The author of the mentioned study claim parameter tuning can impact positively the performance of search algorithms. However, there is no significant inventivation to perform that kind of task, as it is an expensive and an extreme time-consuming process. Moreover, parameter tuning does not always ensure improvements of the performance (and can lead to even worse ones), as it is must adapt to the conditions, the context of the unit test, and its parameters to be tuned.

### 3.2.25 SBST and DSE Integration for Test Generation

The work of Galeotti et al. [39] explores the joint integration of the SBST with the Dynamic Symbolic Execution (DSE) for unit test generation. The combination tries to address the debilities of each part, as SBST is capable of creating test suites, but faces challenges when handling with intricate data or constrained input domain, while effectively addresses these limitations; however, its efficacy is limited by its reliance on constraint solves and the management of complex objects. Their approach is implemented in EvoSuite and evaluated in diversed benchmarks, showing improvements in code coverage.

In fact, the experiments conducted demonstrate the robustness of their method, as their solution uses a solid method to fine-tune the various parameters used in their hybrid approach. Moreover, their adaptative approach ensures and equillibrium of the computational overhead with effectiveness. Those enhancements whether on the efficacy, as well as on the efficiency are also comproved with a rigorous statisical validation, using the SF110's benchmarks.

However, the authors also claim that DSE's implementation limits the efficiency of their approach. Moreover, the potential scalability issues deducted from the marginal improvements on large datasets, as well as the test size management, are other aspects that can be improved on the aforementioned approach. Furthermore, their effort faces challenges when handling complex environment dependencies, such as handling files or TCP connections.

### 3.2.26 Web Queries for Test Data Generation

McMinn et al. [59] presents an effort to generate string inputs for test data generation by formulating web queries based on program identifiers and using the respective resulting web pages to augment the input data pool used. From the experiments conducted in the work, it results in finding 96% of the string types analyzed, and led to significant improvements in branch coverage (by 14% on average), allowing the discovery of various "hard-to-execute" branches, which contains a bug that could not be found until then.

However, the effort is open to further enhancements, whether in terms of providing more meaningful words or expressions, as the tool developed faces some challenges when generating whitespace-separated strings, as well as abbreviations. Consequently, an approach to improve the aforementioned solution may be the usage of fake data generators, as they can originate string values, such as names, or sentences, that are separated by whitespaces and, thus, provide more natural readability [26, 46].

### 3.2.27 Search-based Testing using Enabledness-Preserving Abstractions

The approach of Godoy et al. [40] discusses automated test generation techniques for object protocols, which define the order in which methods should be called on objects. It introduces an approach based on Enabledness-Preserving Abstraction (EPA)s, which partition the object's state space into abstract states based on enabled method sets and extend that approach on EvoSuite by incorporating method terminations and employ SBST to achieve high coverage, calling xEPAs. The results obtained suggest their approach offers a better failure-detection capabilities to the traditional random and structural testing techniques.

However, their approach faces some challenges in CUT that require complex inputs, based on the experiments conducted by the authors. Moreover, it needs manual intervention to generate proper queires for each public class method. As a result, their solution is prone to human errors, which may effect thier solution's effectiveness, which it is planned to be reduced as future work, according to the authors.

Godoy et al. [40]'s work is another novel approach which uses SB techniques to generate improved test suites, focused on EPA. As a result, the aforementioned work may also benefit from the advantage of mocks with synthetic data, in a future improvement of their work. By delegating the generation of complex inputs tasks to fake data generators, their work would thus create the input needed properly and, hence, increase the performance of the test suites generated.

### 3.2.28 Mocking Access to Private APIs

Arcuri et al. [11] studies EvoSuite's performance by enhancing the testing process of classes that contain complex dependencies. Due to the fact a significant amount of those cases do not have public methods, it prevents the tool to generate the adequate tests for them. As a result, the authors use Mockito framework, to generate mocks that simulate the behaviour of those methods.

Arcuri et al. [11]' approach for the scientific community, not due to usage of the mocking strategy, but using a mocking framework to simulate the behavior of methods of third-party libraries, as well as advanced programming techniques, such as encapsulation and reflection. The authors present a novel technique, by adding functional mocks during EvoSuite's test statement generation process. Moreover, their technique increases EvoSuite's branch coverage by 3.30%, leading to a higher efficacy of the tool. In addition to that, the detection of more failures in Defects4J has been increased up to 10,8%, with the usage of their solution.

Nevertheless, the aforementioned approach leads as well to higher false positives, due to the fact the mocks generated by their solution tend to fail by slight changes ocurring on the API methods. Moreover, it is not guaranteed that the mocks generate the necessary input, since they return the default values.

### 3.2.29 API-Aware Search-Based Testing

The work of Ren et al. [71] introduces Kat, a novel approach for SBST, which is aware of the use of APIs. Contrarily to the traditional methods, Kat generated the needed assertions for validating programs, according to the APIs specifications. That approach is achieved by constructing a knowledge graph from APIs' documentation, to detect defects in the CUT, resulting in a more tailored approach to real-world applications.

The tests performed by the authors demonstrate an increase in efficiency and accuracy by Kat. By comparing to two state-of-the-art test generation tools (EvoSuite and Catcher), the results show their solution detects more bugs, with strong evidential statistics (the Kat's F1 score is 0.60) is greater than the other ones (EvoSuite has 0.24 and Cather has 0.30). Moreover, Kat detects more bugs than the other tools for the same time budget, which also indicates that their solution outperforms them in the efficiency aspect.

However, the authors discuss the validity of the experiments, considering the time budget used not being the most adequate one to assess Kat's performance. Moreover, Ren et al. [71] suggest to improve their solution by expanding the knowledge graph's knowledge pool, as well as make Kat with more capabilities of program repairing.

## 3.3 Test automation tools with APIs and AI

### 3.3.1 Comparative Analysis of EvoSuite and ChatGPT

The work of Tang et al. [80] consists in a systematic comparion between the two state-of-the-art tools EvoSuite and ChatGPT (using the model GPT-3.5), assessing their capabilities of unit test generation. The evaluation addresses several topics, such as code coverage, bug detection, and accuracy. With that comparison, the authors try to evaluate the current performance and future role of AI in software testing.

As Tang et al. [80] state, EvoSuite outperforms ChatGPT in various aspects, either in code coverage (up to 18.80%), or bug detection (up to 5%). Moreover, a significant amount of unit tests

created by ChatGPT can be run (69.60%) and used for evalution. However, the authors find the unit tests generated simple, which some of them contain incongruities with the test writing styles.

According to the authors, the fact that missing methods definitions, as well as accessing private methods leads are caused by ChatGPT's inability to access entire projects. Furthermore, EvoSuite faces some challenges when handling with complex Java classes for unit test generation. As future work, Tang et al. [80] believe the standarization of the prompts used with ChatGPT would increase the performance, as the Large-Language Model (LLM) is sensitive to the input phrasing.

It is possible to conclude with the aforementioned work AI still faces some difficulties, when it comes to generate accurate unit tests, as they still outperform by automated generation tools. However, it does not prevent from being used together, as there are capabilities the current state-of-the-art LLMs can be helpful for those tools. Nevertheless, their integration must be made with precision, since they are still not able to always generate the right responses for the prompts used.

### 3.3.2 SINVAD: Testing Deep Neural Networks

Kang et al. [47] presents SINVAD, a novel approach for generating valid test inputs for Deep Neural Networks (DNNs), using Variational AutoEncoders (VAEs), focused on image classification tasks. Contrarily to traditional approaches, which depend on local pixel variations and other algorithms, SINVAD operates in a more approximated distribution of realistic images and hence generates valid test inputs. As a result, their technique aims to test the robustness of DNNs, focusing on identifying boundary cases and debilities in the training aspect.

The experiments indicate the decrease of the space image enhances the performance of Deep Neural Network (DNN), when compared to raw simple images. With that approach, the DNN is closer to identify the label correctly. Morever, the fitness function designed for their solution achieves higher sucess rates for cases DNN faces challenges identifying similar images.

As future work, Kang et al. [47] suggest to validate their approach in other ML areas. They also plan to integrate with related state-of-the-art works of other ML fields. Furthermore, the generalization of their method to test input generation tasks is an effort to explore.

Despite this work can not take advantage from their solution, the [47]'s effort is a relevant approach on a different software testing field for this thesis. In fact, it demonstrates that search-based approaches can generate complex test input data (particularly images for their case), validating the effort made on this work. Moreover, this contribution can be more adequate to SBST challenges, which can potentially provides a more significant contribution to the scientific community, due to the combination of the generation ability inheren of the DNN utilized in the aforementioned work.

### 3.3.3 DeepREL: Fuzz Testing for Deep Learning Libraries

In fuzz testing context, Deng et al. [28] introduce DeepREL, a novel approach to test Deep Learning libraries, employing API inference to generate broader and more effective tests in PyTorch and TensorFlow. Their approach formalizes notions (such as value and status equivalences) to identify

similar APIs, expanding the test coverage which enhances bug detection. Moreover, their tool utilizes the same test inputs for related APIs, useful to identify inconsistencies in their behavior.

Deng et al. [28] executes experiments which indicate their solution enhances the performance in the effectiveness aspect. In fact, DeepREL is capable to find 13.50% of the most critical bugs on a PyTorch's system, as well as 106 unique bugs in total. However, on the FreeFuzz dataset, DeepREL is detecting incorrectly API bugs, which leads to an increase of false positives rate.

The work of Deng et al. [28] is a paramount contribution to scientific community, as it also upgrades the automatic test generation process in an area which is gaining more importance on the current days. Furthermore, it is relevant to this approach to be aware of similar approaches enhancing other areas than the SBST, as they can also be benefic for solutions from that area. Nonetheless, this effort may not be an ideal approach to enhance their work in the future, as it is intended to expand the data pool of SBST, the test generation strategy used by EvoSuite.

### 3.3.4 Catcher: Detecting API Misuse in Java Applications

The work of Kechagia et al. [48] introduces Catcher, a novel method to detect API misuse in Java applications. Their solution combines the analysis of static exception propagation with SBST generation, by comparing their results with EvoSuite, to detect those misuses. Moreover, Catcher provides a robust solution to detect irregularities between the API usage and its documentation.

With the experiments in 21 Java applications reveal the effectiveness of their approach, as Catcher is able to identify 243 unique misuses on those projects. Moreover, their solution enhances the efficiency of the tool, by reducing 20% of the time necessary to generate the unit tests. The authors state as well that a signficant amount of the defections while utilizing APIs are not expected by the developers, which can indicate that they exceptions are not present in APIs' documentation.

However, the lack of ground truth for the evaluated projects prevents Catcher to find all the APIs misuses in the experiments, which leads to a shortage of potential efficacy. Furthermore, the automatic analysis for the documented exception still needs to improved, for being imprecise. As future work, Kechagia et al. [48] plan to expand the Catcher's coverage of more API misuses, as well as the use of third-party libraries to runtime exception analysis, increasing the robustness and completeness of their work.

Kim et al. [50] makes efforts on attempting to test twenty REST API services, using the most advanced techniques in the literature. Their approach includes the use of some automatic test generator tools, such as EvoMasterWB and EvoMasterBB, and black-box tools. With the experiments developed, it is possible to conclude that the tests generated have failed to achieve high score values, when it comes to branch (approximately 36%), line (approximately 53%), and method coverage (approximately 53%), due to the fact of not collecting successfully the correct parameters for the APIs used, and some discrepancies between the specifications and implementations of the APIs tested. To solve these debilities, the authors suggest improving the system created with an enhanced input generation process, by extracting strings from input logs and using Natural Language Processing tools to infer the dependencies present between the operations.

### 3.3.5 Keeper: Testing Software with ML APIs

Wan et al. [87] creates the Keeper- a novel testing tool for software that employs ML APIs. It is designed to integrate and generate pertinent inputs, evaluate the accuracy of outputs, and identify various forms of failure, including crashes. In our tests, Keeper achieved an average accuracy of 21% to 38% and successfully detected 35 unique crash failures across 25 of the 63 applications on the experiments conducted on their work.

The approach described above is relevant to the conception of the solution presented in this thesis, as it also aims to generate important inputs, especially images, to improve the generation of test cases. However, it is not capable of generating different kinds of files input, such as text or audio. Moreover, it is important to note that the effort in the solution described above is oriented to ML and Computer Vision fields, which are usually programmed in Python, whereas this work is intended to improve the pool of EvoSuite, which generates unit tests for Java code.

### 3.3.6 Combining LLMs with SBST for Test Generation

The work of Lemieux et al. [53] introduces CodaMosa, a novel approach for SBST that leverages LLM, such as OpenAI Codex, to avoid coverage plateaus found throughout the generation of test cases. As SBST faces challenges generating specific inputs for the test cases, the authors address their problem by utilizing LLMs to generate the test cases, in case coverage stalls. Hence, it redirects the search process to unexplored areas of the CUT.

Their proposed solution reaches, in fact, to higher levels of code coverage, due to the higher temperature utilized in the sampling algorithm. Moreover, the authors observe that CodaMosa improves the effectiveness by addressing methods that have low coverage, when compared to random functions. The increase of complexity of the prompts also improves the coverage scores; however, the consistency of the results is not always obtained.

To improve the aforementioned work, the authors suggest enhancing the prompts utilized for the LLMs, using a more complex, but structured ones to generate better test cases requiring complex inputs. Moreover, the use of uninterpreted lines of the CUT would not only improve the performance of CodaMosa, but also integrate test cases generated by humans. However, the use of those statements can compromise the performance of the search algorithm by adding irrelevant statements for the input tool.

## 3.4 Test input generation techniques

### 3.4.1 Search-Based Test Input Generation

The work of Sakti et al. [75] presents an automated SBST data generation approach to improve unit-class testing for Object-Oriented Programming (OOP), called JTExpert. Their approach leverages static analysis to identify relevant methods and introduces an instance generator that

diversifies the instatiation process using means-of-instation, seeding, and diversification strategies. The authors test JTExpert against EvoSuite, demonstrating improved performance, in terms of search time and code coverage.

Due to the algorithms utilized by their tool for test data instatiation tasks, JTExpert achieves higher code coverage scores with a stricted budget of time, when compared to other state-of-the-art tools, as it is possible to observe in their experiments. Moreover, their approach is a full automated process, becoming less prone to human errors. JTExpert is a valid solution to the problem it addresses, as it can improve test generation for classes that represent several OOP challenges.

Nevertheless, JTExpert faces some challenges when it comes to handle file input dependencies, as the experiments show for the Sqlsheet library, which requires an specific path for a Excel file, and sheet. Moreover, the authors claim other SB algorithms may be explored in the future, to enhance test input generation. The expansion of testing criteria can also be a good approach to enhance the performance of JTExpert, as it may utilize, or generate different test inputs that can enhance the test suites.

Moreover, the aforementioned work is another contribution that still faces challenges when handling complex inputs, reinforcing the importance and relevance of the problem this thesis attempts to fill. Despite the challenge faced, Sakti et al. [75] explores algorithms that utilizes static analysis to generate the inputs, mitigating the risk identified. Moreover, integration of fake data generators would also be a valid approach to overcome the challenge found.

McMinn [57] surveys the application of meta-heuristics search techniques, such as the GA and simulated annealing in test data generation process. His work involves the analysis of several testing approaches, such as the structural, functional testing, and grey-box. The effort also investigates of those techniques for non-functional testing, focusing on the worst-case results, such as the execution time in systems operating in real-time.

In the structural testing context, the author discusses the challenges found by coverage-oriented methods, as they do not handle properly loops and dynamic data which is modified in runtime, while claims the structure-oriented ones are more capable to generate input data for strutured tests, due to their capability to adapt the test inputs according to the goal to be accomplished. McMinn [57] also observes how functional testing utilizes search-based approaches for data generation-those algorithms are inteded to mutate the test suites so that the inputs detect a defection in the respective test suite. Furthermore, the author finds grey-box testing utilizing a compounded strategies of the previous approaches, which have relevant results in real-world cases.

His work also identifies gaps and areas of improvements for the aforementioned strategies. For instance, the author states how search-based strutured tests face challenges with creating string and complex inputs for the respective test suites. Moreover, to overcome the barrier for complete automation in search-based functional testing, it is possible to utilize encodings, as well as inserting the structure of the states present in the system under test. When it comes to improve grey-box testing, the author suggests the usage of black-box approaches, as well as the reutilization of components to augment the search for data process.

The work of McMinn [57] is a paramount contribution to this work, as it gives relevant information on state-of-the-art strategies for the generation of search-based tests and their approaches for generating data. As EvoSuite still faces some challenges, when it comes the handling with the dependencies of complex inputs for test, it shows the problems found in that time are still valid. Moreover, it is found a similarity between this work and the aforementioned one, as both attempt to improve input simulation in EvoSuite, despite the different nature of the inputs to be simulated.

### 3.4.2 LLMs for Test Data Generation

Baudry et al. [13] develops a proof-of-concept for the use of LLMs for generating test data generation tasks. The experiments evaluate three kinds of prompts and models. In the aforementioned assessments, it was observed that 63 cases indicated that LLMs demonstrated an understanding of the application domain. Additionally, 42 of the prompts assessed resulted in the generation of executable code containing popular libraries commonly utilized for data generation purposes.

However, it is stated a lack of performance when using less popular, human languages (such as Farsi), due to the lack of training by LLMs on them. Additionally some data generated contained hallucinations. As an improvement Baudry et al. [13] suggests the exploration of the usage of few-shot prompts, as well as chain-of-thoughts.

The study enriches the insights of the use of LLMs and generative AI models, such as ChatGPT, as it exposes some issues that should be taken into consideration when using more complex data inputs or relevant metadata, such as the language to be used for the prompts. Furthermore, an analysis of the behaviour of the model integrated with other fake data generators is required when it is used for the execution of tasks of that nature.

### 3.4.3 GANs for Test Data Generation

When it comes to using the Deep Learning for test data generation, Guo et al. [43] explore Generative Adversarial Networks (GAN)s to create fake inputs for unit tests. The experiments indicate that the mentioned approach has successfully improve the branch coverage score. The authors test three different GAN models, the WGAN-GP, the BiGAN and the standard.

The experiences indicate the first one has a better performance the the others, due to the data instances created are more similar to the data used for training them. In addition to it, the fact it was decreased the complexity of the tests, as well as the GAN models boost the accuracy for predicting the values to be input, it is possible to state the approach is adequate for unit testing, since it leads for better results. Furthermore, it benefits given by the models increase the branch and test coverage, when compared to other approaches, such as random testing.

Nevertheless, the GAN-based models increase the time complexity, as are heavier algorithms. In addition to it, the models face difficulties when trying to cover conditional branches using the equality sign (‘==’), and for the ones which contains complex conditions statements, as well as generating values for float types. Moreover, models encounter challenges in covering methods when the discrepancy between the target and training data is significant.

The aforementioned work is insightful for the development of this work, as it uses a framework which is commonly used in different fields from testing (especially in Computer Vision one). Despite this effort being a simpler approach, its inherited idea would be a valid solution to overcome the obstacles found on Guo et al. [43]'s work.

### 3.4.4 Bug Report Mining and Test Input Extraction

Ouédraogo et al. [63] describes how the BRMINER was developed: a technique that automatically extracts relevant test inputs from bug reports to improve the effectiveness of automated test case generation. The approach presented has extracted 68 with success. 68% of the relevant inputs from the bug reports when using regular expressions, compared to 50.20% without regular expressions. Furthermore, the tests generated using the relevant inputs extracted by BRMINER detected 45 bugs that were previously undetected by the baseline approach.

Another observation stated that incorporating the relevant inputs extracted by BRMINER into the test generation process led to higher code coverage than the baseline approach without using these inputs. Despite the findings found, BRMINER has some limitations related to effectiveness, as the parser used (Javalang), as well as the tokenizer, has restricted the literal extraction from test cases and bug reports, compromising the relevant input extraction rate. Moreover, constraints on the time budget and the number of iterations have also decreased the performance of it.

### 3.4.5 Symbolic Execution for Test Input Generation

The research conducted by Pham et al. [70] presents a novel method which exploits symbolic execution for heap-based program based on separation logic. The approach presented has the ability to generate test inputs that are valid and fully initialized, contrary to lazy initializtion approaches. The proposed method is capable of generating valid test inputs which allows their unit tests achieving high branch coverage, outperforming the former approaches that often instatiate invalid ones. In fact, the new unit tests reaches up to 99% of branch coverage on average.

The paper employs a comparable methodology to that proposed by Pham et al. [70] in generating test data inputs that are valid for the SUT. However, there is a difference between this work, and the aforementioned one: while the latter is based on symbolic execution and is limited to heap-based programs, this thesis generates the required test inputs for Java programs that need audio input data. As a consequence, both approaches contribute to the generation of test inputs, albeit with differing objectives.

### 3.4.6 Domain-Specific Input Generation

Another contribution made to the scientific community is TestMiner, a work executed by Toffola et al. [82]. TestMiner is a tool comprehends from the existent test corpus what are the most reasonable inputs to insert on new ones, regarding the domain and context of the SUT. It is also integrated in Randoop, as proof-of-concept of the work conducted.

The input values extracted resulted from two main steps: the first one consists in the static analysis and the retrieval information processes. When the first step is finished, the inputs are index based on the context, which are used to rank them according to the domain they are inserted in. With that, the test generator selects the inputs with best indexes.

The approach referred to above not only improves the score coverage (increase the test coverage from 57% to 78% in 40 state-of-the-art classes), but the analysis process scales up to a significant amount of projects. Furthermore, the queries executed to obtain the inputs are efficiently responded, as well as it is possible to generalize to other software domains than the analyzed ones, according to the experiments made on the work mentioned.

The effort made by Toffola et al. [82] is a reasonable approach to generate inputs as well. However, the reffered paper exposes some debilities with some input strings, as well as with data types and tokenizers. The work conducted can overcome the difficulties the mentioned one, by augmenting the seeding pool of the words used in the test cases generated with more suitable data for the context of the SUT. Moreover, they are integrated on two different test generator tools, and hence is not expected to face the problems encountered on EvoSuite.

### 3.4.7 Adaptive Algorithms for Test Input Generation

Galeotti et al. [39] propose a novel approach for softare testing, using a Kalman Filter-based Adaptive Genetic Algorithm (KFAGA). The algorithm adjusts its parameters dynamically throughout the optimization process to enhance test data generation process, following the Kalman's filter heuristic. It outperforms the traditional GAs, by providing improvements on coverage and efficiency for large and complex problem instances.

In fact, the experiments conducted evidence that the adaptative algorithm uses less method and functional calls when compared to other solutions, which enhances the usage of resources, and hence improves efficiency. Furthermore, their novel approach also adapts according to the charactheristics code under test, in a significant amount of the benchmarks used in those experiments, making the test data generation more meaningful.

However, the authors also state that the multi-objective optimization is poorly explored in their effort, relying significantly in the effectviness of the Kalman's filter. Moreover, the lack of focus on unreachable branches reduces heavily the potential of the coverage score increment, which are intended to be addressed in the future work. Another improvement suggested is the further investigation betweeen the parameter adjustments and the search space structures, which may eventually lead to an improved adaptability of the algorithm.

Another detail mentioned is the algorithm faces some challenges in projects containing infeasible branches, due to the usage of uncalled private methods. It is not obvious whether this effort can help the work presented by Galeotti et al. [39], as it is not clearly stated if the impossibility of calling those private methods is due to the fact the algorithm is not capable of creating the objects containing those methods in the proper manner. If that happens to confirm, this effort may improve the aforementioned work, by augmenting the pool of the KFAGA, contributing to a more varied input pool for the unit tests to be created.

### 3.4.8 Continuous Test Generation

Campos et al. [19] demonstrate a proof-of-concept of an innovative approach called Continous Test Generation (CTG). The work described consists in the reuse and the conception of new unit test cases upon previous ones from (and for) a certain software project tested in EvoSuite.

The aforementioned approach offers a number of notable advantages. The maintenance of the score coverages of the project's test suites is ensured by the fact that, as the unit tests are not rebuilt upon project completion, new unit tests for the code under test, which is modified, are generated. Consequently, the minimum code coverage is identical to that of the project's preceding test cases. Moreover, the experiments indicate improvements in branch coverage (up to 58%), as well as an increase in the number of undeclared exceptions raised (up to 69%). Furthermore, it saves up to 83% of the time needed to execute new tests.

The authors suggest using more advanced seeding techniques or coverage requirements to improve the study conducted. Moreover, the effort made by Campos et al. [19] provides significant insights for this thesis, as it is an approach already implemented in EvoSuite. As a result, the aforementioned effort can also benefit from an augment of input data, enhancing the combination of previous tests with novel inputs, providing a more varied set of unit tests to cover the CUT.

### 3.4.9 Property-Based Testing

An alternative methodology for the generation of test inputs is property-based testing. In Claessen and Hughes [22]'s work, a tool called QuickCheck was developed for the generation of tests for Haskell programs. The primary distinction between testing Haskell and Java programs is that the former prioritizes the examination of program properties, as opposed to the program's final state, which is the focus of the latter. As a consequence, QuickCheck is capable of evaluating the program's properties, generating a diverse array of random inputs for testing, and thereby assessing the code's behavior in scenarios that may be considered extreme or anomalous.

The aforementioned work demonstrates that the property-based testing approach yields reasonable results for functional programming languages. The random input generated for its unit tests provides comprehensive coverage of the program under test. Indeed, the experiments conducted confirm QuickCheck has successfully created tests in numerous applications, as it enables testers to control the parameters used for testing the properties under test.

Nevertheless, it is also stated some debilities in the aforementioned tool: in fact, the need of a significant amount of criteria implies the programs must be reinterpreted before applying to Haskell program. Furthermore, using that criteria obligues to use heavyweight methods, to generate the test for the properties, utilizing user's input of the test data generators.

This presents a promising opportunity for this work to address these types of deficiencies. However, applying this effort on the aforementioned approach would deviate the focus of the problem proposed, as EvoSuite provides search-based and property-based paradigms. Moreover, this effort is intended to improve the search-based paradigm present in EvoSuite, in order to generate better input for the unit tests created by the mentioned approaches.

### 3.4.10 Novel Input Data Generation

Yoo and Harman [92] present a technique that generates novel input data from the existent one utilized on the test unit cases. Their work is relevant for the scientific community, particularly for the software testing field, where the availability of various and effective test data can significantly improve the identification of software faults. The aforementioned technique aims as well to enhance test coverage and effectiveness while it minimizes human intervention.

In fact, their effort provides a cost-effective solution, as there is no need for novel, expendious resources to generate the novel input data. Moreover, it demonstrates significant improvements in test coverage and fault detection. However, the Yoo and Harman [92]'s work still has some opportunities to improve, as the authors suggest the utilization of more complex operators, as well as the exploration of the input domains for the test input generation tasks. The optimization of the meta-heuritics can also be lead to significant enhancements on the performance of their solution whether on the effectiveness, or the effiency aspects.

The aforementioned approach has also the potential to take advantage from this work. As the aim of this work is to improve the effectiveness of the test suites, using a similar approach to Yoo and Harman [92] would lead to the enhancement of the resources for the test input available data, and thus improve the effectiveness.

## 3.5 Other relevant literature

### 3.5.1 MR-Scout Framework

The work of Xu et al. [90] presents MR-Scout a novel framework designed to address in Metamorphic Testing (MT), and to define their Metamorphic Relation (MR). Their approach consists of leveraging developer-written unit tests and automates the discovery and synthesis of MRs, enhancing the quality of the unit tests produced. As a result, the study explores the potential of MR-Scout enhancing the performance of the automated test generation process.

Their experiments show that, in fact, MR-Scout provides improvements in test generation: 97.20% of the MRs generated from the tool were labeled as high-quality and adequate for unit testing. Moreover, those MRs improve the test coverage of unit tests incepted by EvoSuite and the ones present in the OSS projects. Furthermore, a significant amount of the MRs are easily-comprehensible by the developers (between 55.80% and 76.90%), which indicates that MR-Scout does not compromise the tests' readability.

Nonetheless, the authors have found that their effort can not provide robust MRs, when its complex data inputs are used. Moreover, as MRs are only restricted for individual classes, their solution has limited applicability for multi-class or system-level projects. As MR-Scout needs human-written test cases, it faces challenges in projects lacking of unit tests of that kind.

### 3.5.2 Readability Factors in Test Cases

Winkler et al. [88] identifies unique readability factors in scientific literature, highlights overlaps with grey literature, and demonstrates through an empirical study that applying industry best practices improves test suites' readability. That is an important reference to the solution to be developed, as it is fundamental to have tests that are understood by the testers and developers using the tool developed by this thesis. The aforementioned concern with readability helps to identify potential failures or bugs in the code. Hence, the content generated should be meaningful, comprehensible, and coherent for each unit test, following the mentioned guidelines, as well as good practices present in the industry.

### 3.5.3 Enhancing Test Names

The work conducted by Daka et al. [25] has as main goal to enhance the name creation for the tests generated by the most relevant automatic test generation tools. The main approach attempted consists in the use of an algorithm developed by the authors to extract the main goals of the test and rank them. With that, they are selected according to their priority in the unit test.

The experiments indicate significant positive results of the approach aforementioned. In fact, 53% of the students who participated in the study approved the names generated for the tests. Furthermore, they demonstrated enhanced accuracy in associating names with their corresponding tests and in recognizing the purpose of the tests based on their synthesized names, although some limitations were observed in achieving the desired length for the names.

However, Daka et al. [25] identify some possible measures that may improve the performance of the mentioned approach, such as the adaptation of the names generated to the most commonly used patterns and conventions of SBST. Furthermore, investigating how the maintenance of unit tests would benefit from creation of more meaningful names for the tests generated is another possibility of improvement. Another possible enhancement of the solution previously mentioned involves improving the technique used to extract better results of the tests' purposes.

Deljouyi et al. [27] investigates the understandability of tests generated by EvoSuite which were complemented with test data generated with UTGen. UTGen is a system which integrates a LLM into the search-based software testing process, enhancing the generation of test names, data, and variable names. The study tests UTGen under 346 classes and a controlled experiment with 32 individual with different expertise background.

Similarly to other approaches which use LLMs, some of the content contains hallucinations, which decreases the quality of the tests built. In fact, it is stated a worse performance when compared the automated test generation by only EvoSuite. To cover the defficiencies found, the authors suggest to refine the creation of a fine-tune LLM specialized for that kind of tasks, instead of using pre-trained LLMs available on the market, as well as the enhancement of performance efficiency, and the minimization of the need of re-prompting.

The aforementioned study demonstrates that the deployment of AI models as a standalone solution for enhancing automated test generation tools is not optimal. Instead, the utilization of

these models should be subject to rigorous oversight and integration within the existing framework. In light of the fact that the tool in question also employs alternative data generators, it would seem prudent to reserve the incorporation of AI tools for tasks that cannot be accomplished with existing techniques, such as the generation of test files.

Similarly to the Deljouyi et al. [27]'s work, Biagiola et al. [14] and da Silva [23] utilize LLMs to enhance the generation of input variables, as well as test names, and thus improve test readibility. While Biagiola et al. [14] utilize LLMs to enhance search-based-generated test suites, da Silva [23] uses ChatGPT [61] and compares its performance with other similar tools, such as DeepTC-Enhancer, TestDescriber, etc., to improve understandability of JUnit test suites. Both works conduct user studies, whose users confirm the enhancement of the comprehensability of the test generated without abdicating the effectiveness of those tests, providing stability to the enhanced test suite.

### 3.5.4 LEARN2FIX Approach

Another paramount approach is the work made by Böhme et al. [16], which presents LEARN2FIX, a semi-automatic program that generates failing tests. The differentiation factor of their work is the human intervention in the test generation process, where the tester helps the program to improve the unit tests generated, which are used to train an automatic bug oracle using active learning. It is also stated that their solution creates higher-quality unit tests than the manually constructed ones.

One main finding of the aforementioned work is LEARN2FIX is able to produce test oracles that can predict with precision the labels of the validation tests, using only one single test suite as input. Moreover, the human effort is reduced, by decreasing of the generated tests ready for the label task, while maintaining the proportion of actual failing tests. As a result, the generated test suites lead to fewer repairs, and the quality of them, as well as their validation score increases.

The authors suggest as future to maximize the automation of their process, by only needing human intervention to identify the bug on the code under test. Moreover, the exploration of statistical metrics, such as the enhancement of the binary classification, by adding the abstention output. Those suggestions would lead to an improvement of the prediction rates of the aforementioned solution, where the only intervention needed was for uncertain cases.

## 3.6 Discussion

Chapter 3 delineates the most significant advancements made by the community, which pertain to the generation of optimal inputs for tests generated by tools such as EvoMaster, EvoSuite, and Randoop. These tools share a similar objective, as they are aimed to generate superior unit tests. Furthermore, the contributions enhance the understanding of diverse approaches within the scientific community, including the utilization of search-based seeding strategies for test generation and the employment of API and artificial intelligence for the generation of tests.

It is also relevant to highlight the works from Braione et al. [15], Mcminn et al. [60], Sakti et al. [75], Tzoref-Brill et al. [84], due to their similarities with our work. In fact, all of them have

a common objective: generate more complex data inputs to enhance the test cases generated by automated test tools. All of the works discussed, as well as this one, present novel approaches for the same problem, although they cannot generate tests to solve all the possible cases.

When it comes to using search-based seeding strategies, the scientific community has also improved the automatic generation tools to take advantage from it. This would possibly solve the gaps found in others works like the ones from Arcuri and Galeotti [7], Kim et al. [50], McMinn et al. [59], Ouédraogo et al. [63], etc.

The aforementioned work also contributes for innovative techniques to generate complex input files, as the studies mentioned in Section 3.3. In fact, the effort made in the work of Wan et al. [87] is guided on that purpose, as it generates more accurate images for the ML APIs. Due to the purpose of their work, it has not tried yet the generation of other kinds of files, such as Portable Document Format (PDF)s or audio files and, as a consequence, enhance the coverage of SUT requiring those inputs.

An additional factor to consider is the readability of the unit tests generated. Indeed, the contributions referenced in Section 3.5 have influenced the exploration of the unit tests generated in a positive manner. [27, 88]. Nevertheless it is important to recall this work is limited to augmenting the efficiency and accuracy of the tests generated for complex inputs.

Despite the advances made in the scientific community, there are still some challenges with the generation of complex inputs. Moreover, there is reduced amount of works that attempts to generate test files for test suites, or tried to use third-party generators for complex input creation. As a result, the work presented in this thesis may have an advantage of presenting another innovative solution, if attempting the inception of other relevant outputs for testing purposes.

# Chapter 4

# *Pool++*

From the analysis of the existent literature in Chapter 3, there is a lack of approaches that enhances automatic test generation tools, especially EvoSuite, that can handle with more complex inputs, such as files. A significant amount of works on the community have improved, indeed, the data pool of those test generation tools [7, 63, 81], using various techniques and strategies for the effect. Nevertheless, they face some challenges with systems that require inputs of the aforementioned type. Moreover, there are efforts that handle with complex inputs, enhancing the performance of state-of-the-art test generation tools when operating with more sophisticated data [15, 84], but they are not capable to cover particular cases for complex input processing.

Furthermore, none of the works present on the literature have explored the usage of data instatiators, neither handle with the usage of complex files. In fact, a significant amount of works found in the literature, uses dependencies patterns or other algorithms to generate the tests, but they do not address the mentioned problem. Moreover, the idea of using mocks to increase coverage is already a proved and valid idea in the scientific community [9, 11]. However, generating mocks containing required data for specific input processing has never been experimented with before.

In these circumstances, there is a clear opportunity to attempt a novel approach to solve it, motivating the creation of *Pool++*. *Pool++* consists of an extension of the mocking behaviour of EvoSuite: **it generates mocks for complex classes, which possess specific data for the effect**. In the case of this work, *Pool++* is capable of generating audio input data, so that EvoSuite can use mocks to simulate the behavior of the main Java classes for audio processing. This way, it is expected EvoSuite can increase the coverage for CUTs which has that executes audio processing tasks, such as open an audio file, read or write an audio input stream.

The following sections describe the development process of the solution, including the inherent phases of it, such as the exploration phase, which contains the selection criteria for projects containing important classes to be mocked, as well as the main findings from it. Moreover, the development phase describes the ideas explored, and how the chosen one is implemented, adding to the illustration of the novel mocks' organization on EvoSuite and its structure. Furthermore, a discussion section can be found, where it is reflected the advantages and disadvantages, as well as the summarization of the aforementioned phases of the solution's development process.

Our tool named *Pool++* is available in https://github.com/EvoSuite/evosuite/pull/482.

## 4.1 Development process

The development process consists in three steps: the exploration, the development itself and testing. The exploration phase consists in the investigation of which projects from SF110 dataset that may be adequate to analyze, before developing *Pool++*. The second step is the development phase, consisting of the creation of the solution and integrate it in EvoSuite. In the last, the testing phase will assess the good functioning and integration of *Pool++*.

The following subsections will describe each step in the development process of the solution proposed in this work. It will be a dedicated subsection for each aforementioned phase of the process, where it will be detailed the main contours, challenges and approaches used on them. There is also a final conclusion paragraph, the development process in a general overview.

### 4.1.1 Exploration phase

The exploration phase consists in the examination of the performance of the unit tests generated by EvoSuite for a significant amount of classes present in the SF110 dataset. It is a paramount phase for the development process, in order to understand in what extent the fake data generators will improve the performance of EvoSuite. This section will describe the criteria to select the classes to be under test, as well as a analysis of the the projects explored, its relevant Java classes, and EvoSuite's test generation performance on those classes.

#### 4.1.1.1 Selection criteria

Since there is a wide range of variety in the classes present in SF110 dataset, test cases would not benefit from the usage of the novel mocks, as they do not use the objects or methods from the `javax.sound` library. Furthermore, there are several projects which are not intended to perform audio-processing-related tasks. As a result, it is necessary to select a significant amount of classes, so that it has a minimal representation of the dataset. For that, the following selection criteria was defined to elect the projects under test:

- EvoSuite can generate the unit tests for the project's classes, without generating exceptions.
- Select projects which utilizes similar Java classes to the ones used in the Problem Statement section Chapter 1.
- Select projects containing classes with alusive names for files.

Those criteria will ensure that there are projects that contain similar problems to the one identified in Chapter 1. Hence, a smaller set of alike projects and classes will be filtered, turning the exploration phase less time-consuming. From them, it resulted in the following set of five projects:

- `56_jhandballmoves`;
- `75_openhre`;

- `95_celwars2009`;
- `99_newzgrabber`;
- `109_pdfsam`;

From this set, there are two paramount projects for analysis: one is the 95th project of SF110 dataset "celwars2009", containing the `MP3` Java class, since it is the one utilized in the Problem Statement presented in Chapter 1, and "pdfsam" project (SF110's 109th project), containing the class `SoundPlayer`, which also utilizes classes for audio processing. Those classes represent a niche set for audio purposes, where EvoSuite faces challenges on generating effective unit tests. As a result, those classes were chosen for exploration.

The `MP3` class' content can be found on the Appendix A.2. After analyzing the source code of the class, it is possible to conclude there it six audio-related classes, which are the following.

- `javax.sound.sampled.AudioInputStream`;
- `javax.sound.sampled.AudioFormat`;
- `javax.sound.sampled.AudioSystem`;
- `javax.sound.sampled.DataLine`;
- `javax.sound.sampled.FloatControl`;
- `javax.sound.sampled.SourceDataLine`;

In fact, and as shown in Chapter 1, EvoSuite generates a test suite, which can only cover code until the "`AudioInputStream in = AudioSystem.getAudioInputStream(file);`" (line 18). That stoppage is caused by EvoSuite is not capable to extract an audio input stream from a file, due to the fact the mock files generated by the tool are not valid. As it is not reasonable to generate a mock file containing data, stored in the file system, as it would make the unit tests dependent from an external source. As a result, **mocking the behaviour of the class, providing fake input data on that mock**, as proposed by this solution, might be a valid approach to overcome this problem. Moreover, mocking other classes will not only make EvoSuite independent from eventual failures, which were not covered before, due to the stoppage on a previous line in CUT, but also provide an alternative to EvoSuite, in cases where the automatic test generation tool is not covered before, due to the stoppage on a previous line in the CUT, contributing with more options during the initialization of its unit test population.

The `SoundPlayer` source code can be analyzed in the Appendix A.2, where it is possible to observe the class utilizes four classes from Java libraries, for audio content handling, which are the following:

- `javax.sound.sampled.Clip`
- `javax.sound.sampled.AudioInputStream`
- `javax.sound.sampled.DataLine`
- `javax.sound.sampled.AudioSystem`

When EvoSuite attempts to generate tests for the `SoundPlayer` class, the tool faces challenges as well, as the test creates a `NullPointerException`, when it is executed, due to the

fact the tool is not capable to generate a valid instance of an internal object `Configuration`, and assess if the object's service is set to play sounds.

As this object is not directly related to audio processing, it has also been run tests with the lines and branches containing `Configuration` instances were removed. For that version, EvoSuite has created unit tests, where the coverage has stopped on the obtention of an `AudioInputStream`, with the provided Uniform Resource Locator (URL), from an `AudioSystem` instance in line 88. As a result, mocking those classes, providing a `AudioInputStream` with fake content, so that EvoSuite is not dependent from an URL.

As a result, the analysis of the aforementioned classes has guided the development phase to explore the available audio-processing Java classes. That exploration has permitted to understand that EvoSuite does not possess the required data to simulate the audio-processing classes' behavior. As a result, **mocking those classes, and its dependent ones, providing to them a fake set of the required data**, can be a novel approach which may solve the problem discussed on this work.

### 4.1.2 Development phase

After the exploration phase, the main step to take on the development is to plan how to integrate a fake data in the audio mocks. In fact, the main paramount aspect with which EvoSuite encounters difficulties is the generation of complex objects. As stated in the Problem Statement in Chapter 1, EvoSuite faces challenges in generating certain kind of objects, in that particular case, the ones that process audio content. In order to solve the aforementioned challenge, *Pool++* mocks the maximum number of complex objects EvoSuite cannot simulate in its vanilla version.

It is intended as complex object any class that is contained in a Java library, which is used to execute a specific set of tasks (generate a XML file, for instance). As there are several classes of that kind, it is impossible to mock all of them in the time planned for this work. As a result, it is necessary to select a significant group of them, so that EvoSuite is capable of simulating a reasonable amount of behaviors, without being dependent on external sources.

To achieve that group, it was used the classes of the selected projects from SF110 dataset and identified 14 complex objects, in addition of the ones identified in the Problem Statement (that is, the `javax.sound.sampled.AudioInputStream`, `javax.sound.sampled.Clip`, and `javax.sound.sampled.Control` classes). Those objects were identified by assessing the ability of EvoSuite generate unit tests for that classes, with a high code coverage score. In order words, if a certain unit test has low code coverage, due to the fact EvoSuite could not generate an instance of a specific object, then that object is added to the list of objects to mock.

The Table 4.1 presents the classes to be mocked by *Pool++*. As they are present in the SF110 dataset, there is evidence that these objects are commonly used in real-world applications. Hence, identifying those classes helps to define a scope of what are **the most relevant classes to be mocked for audio-processing purposes, which need a specific input data to function as pretended.** With the necessary mocks to simulate the audio system, it is mandatory to integrate them in the EvoSuite system, so that they can be instatiated in the test suites generated.

Table 4.1: List of classes mocked and the corresponding mock class. This way, it is guaranteed that is generated instances of those classes with fake input data.

| **Mocked Class** | **Mock Class** |
|---|---|
| `javax.sound.sampled.AudioInputStream` | `MockAudioInputStream` |
| `javax.sound.sampled.Clip` | `MockClip` |
| `javax.sound.sampled.Control` | `MockControl` |
| `javax.sound.sampled.Control.Type` | `MockControl.MockType` |
| `javax.sound.sampled.AudioFormat` | `MockAudioFormat` |
| `javax.sound.sampled.AudioSystem` | `MockAudioSystem` |
| `javax.sound.sampled.BooleanControl` | `MockBooleanControl` |
| `javax.sound.sampled.DataLine` | `MockDataLine` |
| `javax.sound.sampled.DataLine.Info` | `MockDataLine.MockInfo` |
| `javax.sound.sampled.FloatControl` | `MockFloatControl` |
| `javax.sound.sampled.Line.Info` | `MockLineInfo` |
| `javax.sound.sampled.Mixer` | `MockMixer` |
| `javax.sound.sampled.Mixer.Info` | `MockMixerInfo` |
| `javax.sound.sampled.SourceDataLine` | `MockSourceDataLine` |
| `javax.sound.sampled.TargetDataLine` | `MockTargetDataLine` |

The mock of those objects consists in simulating their real behaviour, without being dependent of external factors. Those factors can be a file system, a database, etc. Therefore, their implementation will be simpler than the actual one present in the real class. Moreover, in case it is needed to generate test content (such as a stream of bytes in a specific format, for instance), supplied by the `Randomness` class, will generate according to the data required.

As it is illustrated in Figure 4.1, EvoSuite creates an instance of a test cluster generator (of `TestClusterGenerator` class), which is responsible to define what objects will be instatiated and their respective dependencies between them, according to the SUT's attributes. For instance, if it has an attribute of string type, then the cluster will add to it all the possible generators for string attributes. In case are also used other classes to do so, it will include the necessary dependencies for generating strings.

Furthermore, it is also assessed if EvoSuite can convert those extraneous classes into a mock one. All EvoSuite's mock classes list are traversed and it is checked if any of them has the target class as super class. If so, they replace the target class with the respective mock one. As a result, after the implementation of the novel mocks, they are added to the list, so that they can map the objects they could not be mocked before.

Once the mocks are created, the TestCluster generator wil create the chromosomes for the initial population of unit tests, which will include tests containing the novel mocks. The GA will mutate and create novel generations of tests. While that evolution occurs, and as the tests using the mocks are expected to reach more coverage, they will prevail in the next generations. When the GA reaches a test suite which cannot provide more coverage than the most optimal one, or when it reaches to 100% coverage score, the algorithm stops. Hence, EvoSuite will provide the test suite created which achieves the highest coverage score.

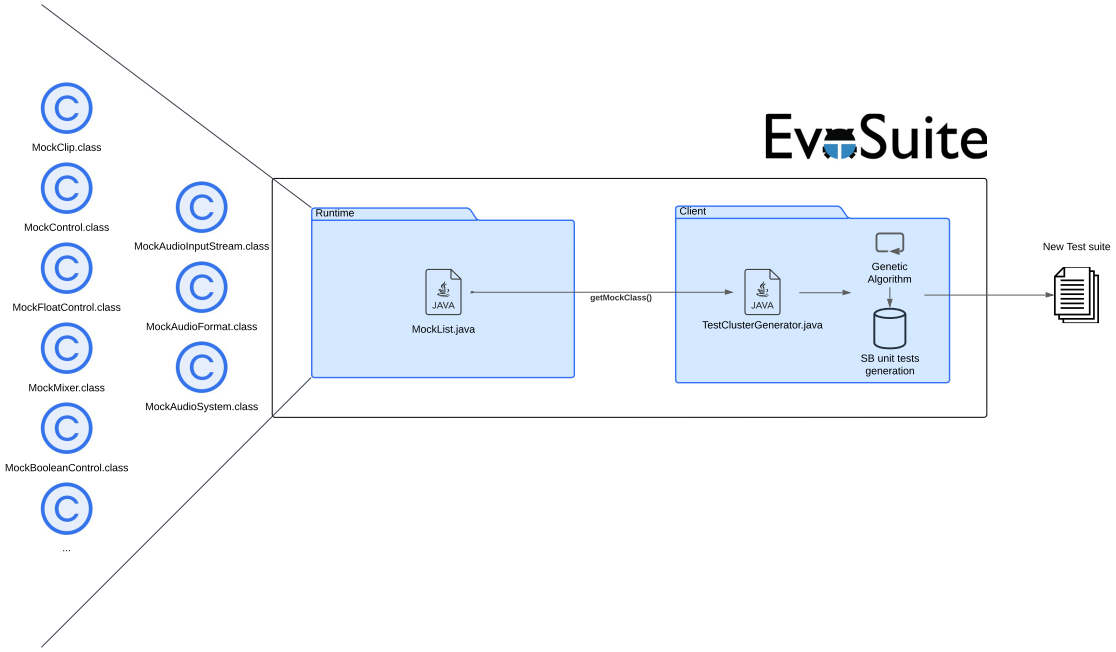


Figure 4.1: *Pool++*'s implementation architecture. The mocks are added into the EvoSuite's `MockList` class, which will the target classes present in the dependencies by the respective mock.

In this way, it is ensured the behaviour of the classes to be mocked is simulated by the mock. **Moreover, the respective mocks are sustained with the data needed, being capable to replicate what is the expected behaviour.** As a result, it is avoided to be dependent for other external sources to have the required data, and hence improve the coverage of SUT.

#### 4.1.2.1 Audio mocking and fake data generation

The struture of the novel mocks consists in replicating the behaviour of the main Java classes for audio processing. The 15 classes identified in Table 4.1 are not only simulating classes for processing files, but also they provide a sample data, if a real one is not available for testing, or the mocking framework is not activated. That is provided an auxiliar class (the `MockAudioUtils` class), which will generate data for three main components of an audio file: the sample rate, the duration, and the channels.

The sample rate consists of the frequency of sound samples that are processed per second. It can generate samples from 8.00 KHz to 48.00 KHz. The duration of the sound is also an integer which can vary from one to five seconds. Moreover, the number of channels may be one or two, depending of the sound type (if it is a mono or a stereo sound). Apart from the audio duration parameter, which only it has the value reduced, to not generate a heavy resource to the mocks, the fake data for the remaining parameters will contain a random value in the ranges supported by Java 11 [62].

With these parameters fulfilled, it is possible to generate a byte array, containing a simple, monotonic sound, varying with the characteristics provided. The array is later converted to an audio input stream, which is used for the audio mock classes, in order to simulate their respective behaviour. Furthermore, the channels and sample rate are also used to configure the format of the audio. This format is not only used for the audio input stream, but also to configure the controllers, as well as the `Clip` class, which simulates the audio playing and interaction.

Hence, the `MockAudioInputStream` and the `MockAudioFormat` classes, can be **equipped with fake content to simulate the the classes' behavior**. The former consists in an specialized `InputStream` for audio content. It has the methods needed to read, open and close those kind of streams. The latter defines possible, valid formats for audio content, which are used by `Clip`, or the `AudioInputStream` instances.

Furthermore, another paramount mock class is the `MockControl`. This class ensures that `Clip` instances, can be played, paused, opened, or closed. This Control class can also divide into a FloatControl, or a BooleanControl, which are be used to either control the source, or target data lines containing the input or output streams of content, respectively, represented by `MockSourceDataLine`, and `MockTargetDataLine`. The `MockMixer` class, will contain the information of the lines received, which are helped by the `MockLineInfo` and `MockMixerInfo`, to be configured according to the line being processed. This class will simulate Java's sound mixer.

With this classes simulating all the sound-related classes, using fake data generated by the `MockAudioUtils`, it is possible to simulate the behavior of the Java's sound audio system, which will provide fake elements for `MockAudioSystem` class. This class will be responsible to simulate the extraction of sound input data, by calling the aforementioned mock classes, and thus replicate in as whole the processing of a sound.

As a result, the main classes for audio processing will be eligible to be tested, due to the fact there will always be adequated data to simulate the intended behaviour for those classes. This leads that EvoSuite has what it takes to test those methods, and hence increase the coverage for unit test which cover classes of those kind. It can be found on Appendix A a class diagram, illustrating how the aforementioned organization could be visualized.

#### 4.1.2.2 Challenges

One of the main obstacles throughout the development phase is the determinism of the random data generated by the mocks. A clear example of it is related to `MockAudioUtils`, the class responsible to generate the fake audio data:

Listing 4.1: Sample rate generation by `MockAudioUtils`. It returns one of the valid frequencies.

```
1 public static float generateSampleRate() {
2    return (float) Randomness.nextDouble(8000, 48000);
3 }
```

Listing 4.1 illustrates how `MockAudioUtils` generates the sample rate of the audio data. In the "generateSampleRate()" method, it is selected one of the valid values contained in the range

from 8.00 KHz to 48.00 KHz, with the help of the `Randomness` class. However, this snippet of code isolated does not assure that the executions will always generate the same value, as the `Randomness` seed is not the same as the one used for EvoSuite to generate the tests suites. This means that if the test case generated once the EvoSuite is executed, with a certain randomness seed, the values selected could be different in another execution, even with the same seed. Thus, the snippet of code is incomplete, as it needs to ensure that the values selected for a certain randomness seed value is the always the same, without being dependent the number of executions of EvoSuite with that value.

A solution to overcome the aforementioned challenge consists of using the static initialization block, that looks up for environment variable which stricts EvoSuite seed's value. If that variable is set, then it is possible to set the `Randomness`' seed value, according to the environment one.

Listing 4.2: Static initalization block present in `MockAudioUtils`. This block will ensure the determinism of the seed.

```
1 static{
2    String seed = System.getenv("SEED_FOR_MOCKS");
3
4    if (seed != null) {
5       Randomness.setSeed(Long.parseLong(seed));
6    }
7 }
```

This way, the `Randomness` seed will always be the same in every execution for a certain seed, ensuring seed determinism, and thus, test suite's determinism. As each execution of experiment.

Another challenge present in the development is the complexity of the mocks' structure and organization, due to the significant amount of the classes involved and how they are connected to each other. As a consequence, the poor planned development may lead to an incorrect implementation of those classes. Nevertheless, that obstacle can be overcome by ensuring the development of each class at the time, starting from the one with less dependencies, such as `MockAudioUtils`, and `MockControl`, and ending with those who the depends on the other ones the most, as the case of the `MockAudioSystem`.

### 4.1.3 Testing phase

After developing the aforementioned approach, the testing phase has followed, to assess whether the implementation is according to the requirements, and providing the correct output. As a result, it has been conducted a manual testing approach, by using adapted versions of the classes selected from the SF110 dataset, and other created manually, with a significant set of classes that may be mocked by *Pool++*, integrating them in a `SystemTestBase` in the *Pool++*'s source code. The tests cases determine the implementation is correct when it is verified GA is capable to create a set of 50 unit tests, containing the mocks generated, and incorrect otherwise.

From this approach, three tests provide valid examples for *Pool++*'s implementation. The tests are similar to each other, only varying the target class to support the test suite generation. The Listing 4.3 provides an example of one of the system tests created:

Listing 4.3: Example of the system test for the `AudioPlayer` class.

```
1  public class AudioPlayerSystemTest extends SystemTestBase {
2
3     @Test
4     public void systemTestLambdaEA() {
5        EvoSuite evoSuite = new EvoSuite();
6
7        String targetClass = AudioPlayer.class.getCanonicalName();
8
9        Properties.TARGET_CLASS = targetClass;
10
11       String[] command = new String[]{"-generateMOSuite", "-class", targetClass};
12
13       Object result = evoSuite.parseCommandLine(command);
14       GeneticAlgorithm<TestSuiteChromosome> ga = getGAFromResult(result);
15
16       TestSuiteChromosome best = ga.getBestIndividual();
17       System.out.println("EvolvedTestSuite:\n" + best);
18    }
19 }
```

For that particular test, the class under test is similar to the one used in Chapter 1. It contains two objects, one of type `javax.sound.sampled.AudioInputStream` and another of type `javax.sound.sampled.Clip`, as illustrated in Listing 4.4.

Listing 4.4: Example of the class used on the test in Listing 4.3.

```
1  public class AudioPlayer {
2
3     private final AudioInputStream audioInputStream;
4
5     public AudioPlayer(AudioInputStream audioInputStream) {
6        this.audioInputStream = audioInputStream;
7     }
8
9     public AudioInputStream getAudioInputStream(){return this.audioInputStream;}
10
11    public void play(AudioInputStream sound, Clip clip) {
12       try {
13          // load the sound into memory (a Clip)
14          clip.open(sound);
15
16          System.out.println("Playing audio...");
17          clip.start();
18
19          // Wait for the audio to finish playing
20          clip.drain();
21       } catch (IOException | LineUnavailableException e) {
22          e.printStackTrace();
23          throw new RuntimeException(e);
24       }
25    }
26 }
```

Table 4.2: Results obtained from tests for the *Pool++*'s implementation .

| **Test Classes** | **Result (passed or failed)** |
|---|---|
| SoundPlayerSystemTest | Passed |
| MP3SystemTest | Passed |
| AudioPlayerSystemTest | Passed |

Listing 4.3 and Listing 4.4 provide a realistic example of a test generated. This way, it is possible to ensure if the tests generate unit tests with the mocks or not. As a result, the tests have been executed, obtaining the results shown on Table 4.2.

Table 4.2 shows the results obtained from a successful execution of the *Pool++*'s tests. Despite using simple Java classes for testing purposes, all the tests created have passed. As a result, it is ensured *Pool++* is capable of generating mocks for cases that CUT which processes audio inputs, **fulfilling the main goal of this thesis: create mocks using fake data for testing purposes.**

Another observation made throughout on another output obtained from a different system test. In the `MP3SystemTest` a reduced number of unit tests obtained have generated NullPointerExceptions, due to the fact the input given for the file is the null value, when compiled. However, it is not considered as failure of the implementation, as EvoSuite has in fact generated those tests without errors, and it is possible the tests not.

Listing 4.5: Example of the execution of `AudioPlayerSystemTest`.

```
1  Test 2:
2  MockTargetDataLine mockTargetDataLine0 = new MockTargetDataLine();
3  MockAudioInputStream mockAudioInputStream0 = new MockAudioInputStream(mockTargetDataLine0);
4  AudioPlayer audioPlayer0 = new AudioPlayer(mockAudioInputStream0);
5  MockClip mockClip0 = new MockClip();
6  audioPlayer0.play(mockAudioInputStream0, mockClip0);
7
8  Test 3:
9  TargetDataLine targetDataLine0 = null;
10 MockAudioInputStream mockAudioInputStream0 = new MockAudioInputStream(targetDataLine0);
11 AudioPlayer audioPlayer0 = new AudioPlayer(mockAudioInputStream0);
12 audioPlayer0.getAudioInputStream();
```

The Listing 4.5 provides an illustration of an excerpt of output provided by the system tests. In that, case, the tests generated, which contains the mocks created for audio processing classes will appear on the GA's search pool, in the initial population. This way, it is ensured unit tests with the novel mocks are present for the evolutionary search and, in case they provide better coverage than other tests which does not contain mocks, for instance, they will be kept for the next generation, and suffer the mutation, crossover, and reducing processes, which may reach to an optimal solution of better unit tests for the final output test suite.

## 4.2 Discussion

The approach explored throughout the aforementioned phases have advantages, as well as disadvantages. In the exploration phase, reducing the project's sample size would lead to a more objective and scoped study of the dataset, which leads to a better understanding of the characteristics of the problem to be solved by this thesis. In fact, that phase guides to an enhanced comprehension of the problem: increase the tests generated' coverage, by adding the data required for the SUT to complete the expected behaviour. Nevertheless, the sample chosen has the inherited risk of letting other important projects, which could be as helpful as the others, by providing other examples of how the complex objects utilizes their data. That risk is also reduced, by having defined the project's selection criteria list, so that it is guaranteed a significant amount from the main projects and its classes are elected.

Throughout the development phase, it is implemented the solution proposed by *Pool++*: it enhances EvoSuite, by 15 mocks designed ensure that Java audio processing classes have their functions replicated, using the required data to do so, thanks to the generation of the audio inputs by `MockAudioUtils`, as well as the `MockAudioFormat`, and `MockAudioInputStream`. To reduce the malfunctioning of its implementation, which may lead to a consequente decrease of effectiveness of the proposed solution, a small set of tests was incepted, to ensure the expected usage and utility of the mocks, providing enhanced, deterministic unit tests for the pool used by EvoSuite's GA. Despite of having a reduced amount of tests, they provide secure results, as they ensure *Pool++*'s mocks appear in unit tests, when audio classes are present in CUT, as well as in real-world projects containing audio processing tasks. The test cases created for assessing the behavior of *Pool++*'s implementation has given positive results, i.e., they show the capability *Pool++* has to provide **novel mocks which contain fake, random data** for audio processing. As a result, *Pool++* possesses a more diverse population for its GA, having more chances to generate better coverage with **higher coverage scores** throughout its generations.

# Chapter 5

# Empirical evaluation

To assess if *Pool++* has the desired outcome, that is, if it helps to improve its test generation for uncovered cases for complex objects, it is essential to perceive how this solution generates fake data and its potential contribution to test generation, as well as its limitations associated. As a result, the following research questions are considered to be answered in this thesis:

**RQ1:** What is the code coverage achieved by *Pool++*'s fake data?
In this RQ we aim to evaluate whether the generated tests equipped with *Pool++*'s data do indeed exercise more lines, branches, etc., of the CUT. Despite being discussed in the literature an increase of code coverage does not mean that more faults in the code are being detected, the combination of the metrics such as branch coverage, and line coverage could indicate if the EvoSuite's performance benefits or not with the *Pool++* inputs [76].
For this RQ we formulated **Hypothesis (1)** as, test cases equipped with *Pool++*'s data exercise more code under test.

**RQ2:** What is the *Pool++*'s generation overhead when compared with EvoSuite vanilla?
In this RQ we aim to comprehend if the process of the novel mocks' generation process done by *Pool++* create an overhead, which impacts EvoSuite's evolutionary algorithm to create the unit tests, and the test suite for the respective CUT. In fact, this overhead may delay the unit tests production, which can pontentially prevent EvoSuite from generating the optimal unit tests, without maximizing the code coverage for the code under test.
For this RQ we formulated **Hypothesis (2)** as, *Pool++* does not generate any overhead, while creating the unit tests.

**RQ3:** What is the *Pool++*'s output overhead when compared with EvoSuite vanilla?
Another relevant detail to explore is whether *Pool++*'s outputs consume more or less resources, when compared to the test suites generated by the vanilla version. For this RQ we formulated **Hypothesis (3)** as, *Pool++* does not generate output overhead, comparing to the vanilla version.

## 5.1 Subject programs

To perform the experimentations, it is necessary to use Java programs as EvoSuite and therefore *Pool++* only supports Java. For that, SF110, a dataset containing 110 Java projects, which translates in 23,894 Java classes [33], will be used for the effect. *Pool++* is capable to generate unit tests for 5,607 classes, plus another two that were not tested in the vanilla version. However, to compare the novel functionalities introduced by *Pool++*, we only use two classes from two different projects:

- 95_celwars2009 from the `MP3` project.
- 109_pdfsam from the `org.pdfsam.guiclient.commons.business.SoundPlayer` project.

As the aforementioned classes contain objects which process audio content, they constitute the most valid candidates to be replaced by the mocks, in case it is needed to provide mock data to cover their statements. Nevertheless, the latter class contains few, but critical details, which are not related to the class logic, but compromises the test execution. In the `SoundPlayer` class from 109$^{th}$ project, the code relative to the `TextPaneAppender` class, as well as the `Logger` one were removed, as they can not be generated properly, even with EvoSuite vanilla. Since they are not directly related to the audio processing issue, they were excluded from the classes, besides their import statements. Furthermore, simplifying the catch clauses which used `Logger` instances. Moreover, the branch statements which checks if it can play a sound (lines 50 and 72 in Listing A.2) were also removed, as the `Configuration` instance is not being generated by both versions. As a result, removing the `if` statement makes both versions capable to test the audio part in a proper manner.

## 5.2 Baselines

In order to compare the performance of this approach with state-of-the-art tools, the baseline of the experiments is conducted in EvoSuite vanilla, i.e., without the integration of *Pool++*, and using the default settings. The comparison of both tools will indicate in what extent this solution improves or not the efficacy of EvoSuite, and how long are the overheads added to the tool production process and the generated test cases.

## 5.3 Experimental Setup

The experiments conducted manipulate its variables in order to assess *Pool++*'s performance, where the trials are run 30 times, to reduce the randomnness of the algorithm as suggested in other works from the scientific community [5]. Finally, the experiments are run on a Linux Ubuntu 22.04 machine, using a processor Intel Core i7 CPU octa-core, and 16.0 GiB of memory.

Moreover, each baselines' genetic algorithm has a three-minute-search-time budget, to produce the tests suites, as it is known from the literature EvoSuite produces reasonable coverages scores, when utilizing this time budget [37, 63, 76].

## 5.4 Metrics

**RQ1** assesses *Pool++*'s accuracy, that is, if the test suites generated cover more or less statements of the CUT. It is known in the literature that the branch and line coverage are the primary metrics for these kind of measures [8, 34, 76, 94]. However, they are not insightful enough when used in separate conditions, as it is rather to have a lower line coverage which tests the critical branches of the code under test than a main method that calls a significant amount of non-critical methods, leading to a misleading high line coverage score. As a result, the combination of these metrics provides more robust insights for **RQ1** [94].

Furthermore, it would also be beneficial to ensure whether *Pool++* covers or not the lines and branches EvoSuite vanilla also does. As a result, it is also collected a bit string for line and branch coverage (represented by "BranchCoverageBitString", and "LineCoverageBitString" metrics, respectively), which provides two arrays that facilitate the visualization of those occurences. For instance, if the first bit of the "LineCoverageBitString" is set as 1, it means the first line is covered, otherwise is set to 0, meaning is not coverage. This systems repeats for all the remaining lines and it is the same as for "BranchCoverageBitString". Hence, it is possible to analysis of the coverage for specific parts of the CUT, by analyzing those metrics of both baselines.

As **RQ2** discusses the eventual overhead created by *Pool++* when generating the test cases, one metric analyzed is the number of generations provided by the GA. With that metric, it is possible to understand in what extent *Pool++* has created more possible candidates for the final test case to be produced for the CUT, or if that process has been delayed with mock generation throughout the generation's evolution procedure, when comparing to the result obtained with vanilla.

Furthermore, **RQ3** aims to ascertain the overhead introduced by the mocks, throughout the execution of the test cases generated by *Pool++*. To this end, the size (i.e., the number of generated tests), as well as the number of statements, and the execution time of each test suite were considered as metrics. The analysis provided insights into the amount of computational resources required by the novel test suites generated through *Pool++*.

In addition to measuring the metrics, a comparison of the results obtained by *Pool++* and the ones produced by vanilla is also conducted. As the experiments are run 30 times, the weighted average is calculated for each metric, grouped by each project run from the SF110 dataset. Moreover, 20% of the execution times ($30 \times 0.20 = 6$, the three longer and the three shorter), are also removed, before the weighted average calculation for the execution times' metric, to prevent outliers from affecting the experiments.

In order to obtain statistical evidence that would prove the superiority of one approach over another, the Vargha-Delaney $\hat{A}_{xy}$ effect size [85] is utilized together with Wilcoxon Mann-Whitney U-test. For each calculated metric, the aforementioned probability is determined. Each value

obtained provides a distinct insight: assuming that $x$ is *Pool++* and $y$ is the EvoSuite vanilla, if the value is 0.5, it means that both algorithms have similar performance (in this case, *Pool++* will only outperform the vanilla version 50% of the times. Conversely, values greater than 0.5 indicate that the performance of *Pool++* surpasses the standard EvoSuite one in $k\%$ of the times, where $k > 0.5$; while values less than 0.5 suggest that the baseline performs better (in $1 - k\%$ of the times, with $k < 0.5$) [19].

## 5.5 Threats to validity

Another important detail to consider while conducting the experiments is the potential threats *Pool++* might face. Based on the guidelines proposed by Wohlin et al. [89], we have taken all the reasonable steps to mitigate the negative effect of potential threats, which are described in detail in this section. These threats are organized in three distinct groups: the threats affecting the construct validity, the ones affecting the internal validity, and others that impact the external one.

The threats to construct validity of this solution consist of everything that can negatively influence the measures made on the experiments, preventing it from generalizing the results made. The threats to internal validity are the ones related to the design of *Pool++* that can impact the performance throughout the experiments. Finally, the threats to external validity reflects the ability to generalize the results out of the experiments scope.

### 5.5.1 Threats to construct validity

The main threat to the experiments conducted in the construct perspective is the number of runs for the experiments. In fact, EvoSuite uses GA to incept the unit tests, which uses a seed that is randomized. As a result, the results from an experiment can be can random, which affects the causal relationship between the hypothesis and the result. In order to mitigate this threat, experiments are run in experiments 30 times [20, 56, 74], reducing the randomness of the results.

Another salient threat to the construct pertains to the metrics excluded in the research questions, particularly in RQ1. Indeed, the branch and line coverages may prove more suitable for the experiments conducted, as they are frequently employed in such studies. However, it is important to note that the aforementioned scores can potentially lead to erroneous results. Consequently, the Wilcoxon and Vargha-Delaney metrics may emerge as superior alternatives when compared to the former, as they are more robust formulas than the ratio between the number of branches and lines covered and the total number of them.

### 5.5.2 Threats to internal validity

The primary threat to the internal validity is the code implemented. Despite the fact *Pool++* achieves the desired results and satisfies the use cases defined, it is susceptible to the presence of bugs or vulnerabilities that may potentially compromise the integrity of the development. Consequently, a series of tests will be devised to mitigate these weaknesses. By testing the essential

features and their corresponding source code, the risk of *Pool++*'s code having potential vulnerabilities that could compromise the solution is reduced.

One of the main menaces to *Pool++*'s internal integrity is the determinism of the tests. As has been discussed in the preceding Chapter 4, as well as in Section 5.3, it is crucial *Pool++* can generate the same input data for a certain CUT, when it is using a specific seed for EvoSuite's test generation. Without the determinism of the tests, it is not possible guarantee that *Pool++* always generates better, or worst unit test than the vanilla version, or other test generation tool, it is not ensured it will generate the a test with a similar coverage, or structure. To mitigate this vulnerability, the experiments set an environment variable to be passed as argument in the EvoSuite's run command. This way, it is possible to assign a specific value for `Randomness` generator, ensuring the determinism needed for the fake data generation, as explained in Chapter 4.

### 5.5.3 Threats to external validity

The paramount risk for the external validity is the relevance of the problem to be solved. If there is a very reduced amount or none representative code examples that resembles on that problem, it means that there is a few probability of being generalized to other contexts. Consequently, this can result to the reduce the importance, as well as relevance of the proposed solution. As a result, the SF110 dataset is used in the experiments, as it is a data set with significant project examples representative enough to simulate real-world cases.

Another threat for the *Pool++*'s external validity is the selection of the Java classes to be mocked and integrate this solution. Despite being relevant Java classes for audio processing, there is an inherited risk for not mocking other classes for this kind of tests which may also benefit from this approach. To weaken the aforementioned threat, the classes selected for mocking in *Pool++* resulted from the analysis of the classes present in SF110 dataset for audio processing purposes. This way, it is possible to assume the classes selected are the most common ones in real-world applications. Hence, mocking those classes ensures the most pertinent ones, maintaining the relevance of *Pool++* and its approach.

## 5.6 Experimental results

In this section, it is explored the results from the experiments conducted, to answer the research questions proposed in the beginning of the chapter. With the research questions clarified, it is possible to assess if the hypothesis defined for each RQ are confirmed or not. For each RQ, there are two subsections, where it is described the performance on the utilized baselines: the `MP3`, and `SoundPlayer` classes, according to the component analyzed on the question. In Appendix A, it is is possible to observe the results obtained for each baseline in Table A.1, and Table A.2.

Table 5.1: Efficacy performance of *Pool++* and EvoSuite vanilla for `MP3`.

| **Metric** | **LineCoverage** | **BranchCoverage** |
|---|---|---|
| Pool++ (mean) | 0.97 | 0.55 |
| Pool++ (std) | 0.00 | $1.11 \times 10^{-16}$ |
| Vanilla (mean) | 0.40 | 0.18 |
| Vanilla (std) | $1.11 \times 10^{-16}$ | $2.78 \times 10^{-17}$ |

### 5.6.1 RQ1's results

This RQ addresses the efficacy of *Pool++* and the ability of the mocks to improve the coverage score of the unit test generated. To compare it with the vanilla version, two metrics are collected, such as the line coverage, and branch coverage: line coverage calculates the ratio between the tested statements, and the total number of statements in the CUT; while the branch coverage corresponds to the ratio between the tested branches and the total number of them. The combination of both is the most appropriate ones for this study, since this solution aims to increase coverage of the unit tests generated, and that combination demonstrates so: if both increase with *Pool++*'s unit tests, it means the CUT might be *better* tested, otherwise it does not have the desired output.

The "LineCoverageBitString", and "BranchCoverageBitString" metrics provides a simple visualization of the coverage obtained from the tests generated by the baselines. With them, it is possible to compare what lines and branches are covered by each EvoSuite version, and perceive if are more or lesse statements covered in the CUT by *Pool++*, when compared to vanilla version, new lines that were not covered before, or other that become unreachable.

#### 5.6.1.1 MP3

Table 5.1 illustrates the coverage scores obtained on unit tests generated by both baselines, using the `MP3` class as CUT. It is possible to conclude that *Pool++* increases the line coverage to 97%, comparing to the vanilla version, indicating the advantages of the usage of the mocks utilized. Moreover, the low standard deviations obtained indicates both versions have not varied throughout the 30 runs, meaning they always obtained tests with reaching the same results.

Furthermore, the branch coverage scores reached to 55% on *Pool++*'s runs, demonstrating an increase of efficacy performance. That increase is due to the fact the unit tests generated by *Pool++* have the ability to cover further branches inside the `try` statement, as the coverage does not halt on line 19 of `MP3`'s source code (which can be seen on Appendix A.2, on Listing A.1), as occured on the testes provided by the vanilla version. This behavior is expected, due to the fact in the `MP3` source code, it is possible to observe there is one try-catch statement, and hence, it is only possible to either test the `try` branch, or the `catch` one.

To complement those metrics, the bit strings are also analyzed of both line and branch coverages, to perceive the mapping of the unit tests' coverage, as illustrated in the table below:

Table 5.2 illustrates the coverage localization obtained for both baselines for `MP3` class, for the first five runs, due to the similarity with the remaining 25 runs. When analyzing this table,

Table 5.2: Results obtained from the experiments using class `MP3`.

| Version | Class | Seed | Branch Coverage Bit String | Line Coverage Bit String |
|---|---|---|---|---|
| pool++ | MP3 | 0 | **10111011000** | **11111111111111111111111111111101** |
| vanilla | MP3 | 0 | 10000001000 | 11111111110000000000000000000111001 |
| pool++ | MP3 | 1 | **10111011000** | **11111111111111111111111111111101** |
| vanilla | MP3 | 1 | 10000001000 | 11111111110000000000000000000111001 |
| pool++ | MP3 | 2 | **10111011000** | **11111111111111111111111111111101** |
| vanilla | MP3 | 2 | 10000001000 | 11111111110000000000000000000111001 |
| pool++ | MP3 | 3 | **10111011000** | **11111111111111111111111111111101** |
| vanilla | MP3 | 3 | 10000001000 | 11111111110000000000000000000111001 |
| pool++ | MP3 | 4 | **10111011000** | **11111111111111111111111111111101** |
| vanilla | MP3 | 4 | 10000001000 | 11111111110000000000000000000111001 |

Table 5.3: Vargha-Delaney values obtained from the performance of both baselines in `MP3` class. *x*: *Pool++*, *y*: vanilla. The values in *bold* identify the statistical significant effect sizes.

| Metric | LineCoverage | BranchCoverage |
|---|---|---|
| $\hat{\mathbf{A}}_{xy}$ | **1.00** | **1.00** |

it is observed branch coverage bit strings evidences improvements. In fact, *Pool++* enhances the branch coverage, exercising the more branches, besides the ones covered by the vanilla version.

Moreover, the line coverage's bit strings also possess the same pattern: when analyzing them, it is possible to conclude *Pool++* does not only cover the same lines as vanilla does, but also covers more than the latter, confirming the increase of the coverage score without losing the previous achieved. That is due to the fact *Pool++* does not stop the coverage on `MP3`'s source code line 19, present in Listing A.1, in Appendix A.2). Thanks to the mock generated for `Audio.System.getAudioInputStream(file)`, allowing to simulate the load of an audio input stream, from a certain file, as well as cover all other classes depending on audio input processing, present in the CUT.

As a result, it is possible to perceive *Pool++* has improved the efficacy of EvoSuite for this example of audio processing, due to **the usage of the novel mocks with fake data**.

Table 5.3 confirm the results analyzed in previous paragraphs, as it is possible to conclude the line coverage score is improved by *Pool++* 100% of the runs, and hence evidence the increase on the coverage for the baseline. Moreover, the results obtained from the "BranchCoverage" score demonstrates the *Pool++* has also improved the performance in that metric, when compared to the vanilla version, as the novel solution has also covered more branches in all the runs executed in the experiments. Therefore, it is emphasized the improvements accomplished on the code coverage, illustrated in Table 5.1, of *Pool++*'s novel mocks utilized to cover the `MP3` class.

Furthermore, the Wilcoxon Mann-Whitney U-test results in Table 5.4 demonstrate the performance disparity in the efficacy component evaluated in the experiments. As the *p*-values obtained are less than 0.05, the results obtained on the aforementioned metrics are discrepant between the baselines and thus confirming the differences between the baselines' performances. Aggregating

Table 5.4: Wilcoxon Mann-Whitney values for the the performance of both baselines in `MP3` class. It is possible to observe there is statistical difference in the results obtained for both metrics.

| **Metric** | **LineCoverage** | **BranchCoverage** |
|---|---|---|
| $p$ | $\mathbf{2.87 \times 10^{-11}}$ | $\mathbf{2.87 \times 10^{-11}}$ |

Table 5.5: Efficacy performance of *Pool++* and EvoSuite vanilla for `SoundPlayer`.

| **Metric** | **LineCoverage** | **BranchCoverage** |
|---|---|---|
| Pool++ (mean) | 0.81 | 0.88 |
| Pool++ (std) | $3.33 \times 10^{-16}$ | 0.00 |
| Vanilla (mean) | 0.44 | 0.50 |
| Vanilla (std) | $5.55 \times 10^{-17}$ | 0.00 |

this information, as well as the results obtained from the metrics collected, and the Vargha-Delaney tests, it is confirmed the outperformance of *Pool++*, in the class `MP3`, and its mocks containing fake data needed to simulate the desire behavior of the CUT, confirming the correctness of the hypothesis formulated.

#### 5.6.1.2 SoundPlayer

Table 5.5 illustrates the coverage scores obtained in unit tests generated by both baselines, using the `SoundPlayer` class as CUT. It is possible to conclude that *Pool++* increases the line coverage to 81%, when compared to the vanilla version, indicating the advantages of brought by the mocks utilized. In a similar direction, the branch coverage has also increased by the novel tool, covering more branches than vanilla, on average, reaching 88% of the total number of branches.

Without the mocks, EvoSuite has only been capable to cover from lines 53, and 76 of `SoundPlayer` source code (see Listing A.2), as it can not generate the `DataLine.Info` object in a proper manner, preventing the cover further lines in `playErrorSound()` and `playSound()` methods, respectively, due to the null object, value in the previous lines (52, and 75). Moreover, the automatic test generation tool could not also cover the inner class `PlayThread`, as it could not generate the objects to execute the clips from the other methods (see Listing A.2). With the novel mocks, *Pool++* can simulate the generation of the respective `AudioInputStream` objects, as well as the `DataLine.Infos` of the methods, allowing to cover further lines of the respective methods, instatiating the `PlayThreads` required and, therefore covering its code as well.

Alongside the enhancement in the line coverage, the rise of the branch coverage score, as the `if` clause present in the `PlayThread`'s `run()` method is now covered with the mocks (see Listing A.2), maintaining the other branches already covered by vanilla. Moreover, the low standard deviations obtained indicate both versions have not varied throughout the 30 runs, as they always produced tests with reaching the same results. Furthermore, no changes are identified in the branch coverage score, indicating no further branches were covered by the versions, meaning the baselines have always tested the same branches.

Table 5.6: Results obtained from the experiments using class `SoundPlayer`.

| Version | Class | Seed | Branch Coverage Bit String | Line Coverage Bit String |
|---|---|---|---|---|
| pool++ | SoundPlayer | 0 | **10111111** | **11111111111111111111111111110000000** |
| vanilla | SoundPlayer | 0 | 00110101 | 11111111000011011100001101000000000 |
| pool++ | SoundPlayer | 1 | **10111111** | **11111111111111111111111111110000000** |
| vanilla | SoundPlayer | 1 | 00110101 | 11111111000011011100001101000000000 |
| pool++ | SoundPlayer | 2 | **10111111** | **11111111111111111111111111110000000** |
| vanilla | SoundPlayer | 2 | 00110101 | 11111111000011011100001101000000000 |
| pool++ | SoundPlayer | 3 | **10111111** | **11111111111111111111111111110000000** |
| vanilla | SoundPlayer | 3 | 00110101 | 11111111000011011100001101000000000 |
| pool++ | SoundPlayer | 4 | **10111111** | **11111111111111111111111111110000000** |
| vanilla | SoundPlayer | 4 | 00110101 | 11111111000011011100001101000000000 |

Table 5.7: Vargha-Delaney values obtained from the performance of both baselines in `SoundPlayer` class. *x*: *Pool++*, *y*: vanilla. The values in *bold* identify the statistical significant effect sizes.

| Metric | LineCoverage | BranchCoverage |
|---|---|---|
| $\hat{\mathbf{A}}_{xy}$ | **1.00** | **1.00** |

Table 5.8: Wilcoxon Mann-Whitney values for the the performance of both baselines in `SoundPlayer` class. It is possible to observe there is statistical difference in the results obtained for both metrics.

| Metric | LineCoverage | BranchCoverage |
|---|---|---|
| $p$ | $\mathbf{2.87 \times 10^{-11}}$ | $\mathbf{2.87 \times 10^{-11}}$ |

Moreover, Table 5.6 illustrates the branches and lines are covered by the baselines, on the first 5 runs of the experiments. When analyzing the branch coverage bit strings, it is evidenced *Pool++* covers more branches, without losing the ones covered by the vanilla version. Furthermore, *Pool++* not only covers the same lines of the CUT as vanilla do version, but also increases the number of statements covered, thanks to the mocks used, as evidenced in the line coverage's bit strings obtained. As a result, *Pool++*'s mocks provide the necessary data to simulate the correct behavior of the `SoundPlayer` class.

In a similar manner as occurred in the `MP3` class, the Vargha-Delaney test, the superiority of *Pool++* in the efficacy performance, when compared with vanilla: in 100% of the times, *Pool++* has outperformed the line coverage obtained by the latter baseline. In a similar direction, the novel tool has outperformed the branch coverage produced by the vanilla in 100% of the occurrences, confirming the superior power the *Pool++* has in the efficacy aspect. As a result, the combination of the outperformance of both coverage scores evidence the benefits of the mocks created with fake data, when used for testing classes with audio-processing tasks.

Moreover, the Wilcoxon Mann-Whitney U-test metric proves the performance discrepancy between both baselines: since the values obtained are less than 0.05, it means that it is possible to ensure that the experiments conducted of those metrics have demonstrated the behavioral difference of both baselines. Combining this information with the aforementioned conclusions drawn

in the other metrics analyzed, it is clear that *Pool++* provides an enhanced performance in the efficacy component as well for the `SoundPlayer` class.

**RQ1:** *Pool++* increases the coverage scores for both subject programs, when compared with the vanilla version. For `MP3` class, it has increased the line coverage score 97%, and 81% for `SoundPlayer`. Moreover, the branch coverages have augmented to 55% in the former class and in 88%, in the latter one. The mocks generated with fake audio data allow EvoSuite to replace the methods and lines once uncovered, simulating the desire behavior of the CUT.

### 5.6.2 RQ2's results

To assess *Pool++*'s overhead, we collected the number of generations incepted by the EvoSuite's evolutionary algorithm, represented by "Generations" metric. With that criterion, it is possible to assess for each baseline if it requires more or less time to generate the unit tests individually, rather than evolving them, using the GA. This way, it has been collected the metrics for both baselines, using as CUT the two subject programs for these experiments - the `MP3`, and `SoundPlayer` classes, where the results obtained are described in the following subsections, and the conclusions drawn by analyzing them.

#### 5.6.2.1 MP3

Table 5.9 shows how many generations were achieved on average by each baseline, when attempting to test the `MP3` class, as well as the standard deviation. In fact, *Pool++* requires on average up to 98 generations to achieve the final test suite to cover the CUT, when compared to the vanilla version, reaching to 2,948 generations completed, representing a decrease of 96.68% of GA's iterations. Moreover, the standard deviations are reduced, when compared to the respective average scores- *Pool++*'s evolutionary algorithm is more consistent on its exploration phase in the 30 runs; while vanilla version has varied slightly more throughout the experiments.

Therefore, it is possible to observe *Pool++* requires less combinations of unit tests, reducing the ability of GA to improve further generations. That is due to the fact the novel solution demands a significant amount of time to generate the mocks, as well as integrating them in the unit tests (by static replacement for the mocked objects and methods), and executing them, to assess the coverage score. As a result, there is an overhead created by the novel mocks, even though it generates enhanced unit tests, in less GA's generations for the `MP3` class.

Table 5.9: Generations produced by the *Pool++* and EvoSuite vanilla's GAs, for `MP3` class. The vanilla has better performance, by creating a vast amount of generations, outperforming *Pool++*.

| **Metric** | **Generations** |
|---|---|
| *Pool++* (mean) | 97.90 |
| *Pool++* (std) | 19.27 |
| Vanilla (mean) | 2,948.03 |
| Vanilla (std) | 396.60 |

Table 5.10: Vargha-Delaney values from the performance of the baselines in `MP3` class. *x*: *Pool++*, *y*: vanilla. The values in *bold* identify the statistical significant effect sizes.

| Metric | Generations |
|---|---|
| $\hat{\mathbf{A}}_{xy}$ | **0.00** |

Table 5.11: Wilcoxon Mann-Whitney values for the the performance of both baselines in `MP3` class. It is possible to observe there is statistical difference in the results obtained for the metric.

| Metric | Generations |
|---|---|
| $p$ | $\mathbf{2.87 \times 10^{-11}}$ |

In the opposite direction, vanilla version computes more solutions, by mutating and crossovering the unit tests created in the evolutionary algorithm's initial populations, but it cannot produce unit tests with higher coverage scores, as explored in the first research question. Even though the reduced coverage scores, the GA keeps trying in vain to generate unit tests with higher coverage score, due to the speed of GA's mutation and crossover processes. As vanilla does not contain the require mocks and/or data needed to simulate the desire process, it does not have any delay on the unit test generation process, and therefore is capable of producing more generations, despite not increasing the code coverage.

Table 5.10 confirms the results and the conclusions drawn, by analyzing Table 5.9. Representing *Pool++* by *x*, and the vanilla version by *y*, it is possible to see *Pool++* has greater values on "Generations" metric at 0% of the times, comparing with EvoSuite's default version. As a result, it is evidenced the poor performance of the novel tool while producing novel GA's generations.

To ensure that there is statistical evidence of the differences found on the aforementioned metric, a Wilcoxon-Mann Whitney test is also concluded, obtaining the results shown on Table 5.11. As the *p*-value for the "Generations" metric is less than 0.05, there is a discrepancy between the performance of both baselines, where vanilla outperforms the novel tool, as evidenced by the average scores and the Vargha-Delaney tests. Therefore, it is possible to conclude *Pool++* generates an overhead when creating the novel mocks for the class `MP3` as CUT.

#### 5.6.2.2 SoundPlayer

For the `SoundPlayer` class, the differences between the baselines' performances are also evident. In fact, the discrepancy between the two tools is more pronounced, comparing with the performance of the baselines for the `MP3` class. The results obtained from the experiments conducted in this subject program are illustrated in the following table:

Table 5.12 illustrates the results obtained of both versions, when it comes to assess the unit test generation performance, for the `SoundPlayer` class. In fact, it is possible to see a clear difference between both baselines: *Pool++* reaches the final test suite, after 4 iterations, while vanilla can generate 7,052 on average, meaning the novel tool has decreased the search space in 99.94%. Moreover, the values obtained for the standard deviations are on the same magnitude as

Table 5.12: GA's generations performance of *Pool++* and EvoSuite vanilla, using `SoundPlayer` as CUT.

| **Metric** | **Generations** |
|---|---|
| *Pool++* (mean) | 4.23 |
| *Pool++* (std) | 1.75 |
| Vanilla (mean) | 7,052.17 |
| Vanilla (std) | 88.54 |

Table 5.13: Vargha-Delaney values for the the performance of both baselines in `SoundPlayer` class. *x*: *Pool++*, *y*: vanilla. The value in *bold* identifies the statistical significant effect size.

| **Metric** | **Generations** |
|---|---|
| $\hat{\mathbf{A}}_{xy}$ | **0.00** |

Table 5.14: Wilcoxon Mann-Whitney values for the generations performance of both baselines in `SoundPlayer` class. It is possible to observe there is statistical difference in the results obtained for the metric.

| **Metric** | Generations |
|---|---|
| *p* | $\mathbf{2.87 \times 10^{-11}}$ |

the ones collected during the `MP3` experiences, which indicates *Pool++* has been more consistent than the vanilla version.

The aforementioned results denote the reduced amount of generations *Pool++*'s GA, as it is detected the same pattern found in the `MP3` class: the novel tool demands more time to generate the mocks for the unit tests, and hence it reduces the search budget for its evolutionary algorithm to produce more generations.

As `SoundPlayer` is a more robust class, due to the increase of lines, as well as methods (despite having similar code), the vanilla version has needed more generation power to reach the final test suite. However, that has not happened with *Pool++*: the number of generations is lower than the one obtained for `MP3` class, indicating there has been a longer overhead to generate the required mocks and execute the unit tests in the test suite. Since the `SoundPlayer`'s test suites are longer than ones created for the `MP3` class, their execution is therefore longer and, hence, *Pool++*'s GA has less search budget time to iterate over more generations.

As illustrated in Table 5.13, it is possible to confirm the results found in the previous table. In fact, it is stated that *Pool++* generates more generations than vanilla 0% of the times, when compared with vanilla version, confirming the existence of the overhead generated during the production *Pool++*'s novel mocks.

Furthermore, Table 5.14 also illustrates the discrepancy between the baselines, when testing the `SoundPlayer` class. For the "Generations" parameter, the *p*-value obtained demonstrates there are discrepancies between the performance of the baselines, as the value obtained is inferior to 0.05. Furthermore, the aforementioned metrics utilized to measure the baselines' performance and assess the potential delay created by *Pool++*, evidence of vanilla's outperformance, when

compared with novel tool, due to overhead generated by the novel mocks and their insertion in the final test suite's unit tests. As a result, it refutes the formulated hypothesis for this research question.

**RQ2:** *Pool++* generates a significant overhead during the novel mocks production process. In fact, the number of GA's generations explored has dropped between 96.68% and 99.94%, for `MP3` and `SoundPlayer` classes, respectively, when compared with the vanilla version. The production of the mocks process has prevented *Pool++* to generate further generations of the GA's unit tests population.

### 5.6.3 RQ3's results

The final research question investigates the overhead generated by *Pool++*, that is, if more resources (more statements, and / or time to execute) are needed for the generated final test suite, compared to the vanilla version. In fact, it is confirmed an increase in test suite size may suggest that more scenarios are being used to evaluate the behavior of the CUT, potentially improving coverage scores. However, it is important to note that larger test suites can also have a negative effect on execution time, as a greater number of statements must be executed.

To respond to the research question formulated, it has been analyzed the "Size" of the test suite (which is the number of unit tests contained in it), as well as the "Length" of the unit tests, for both baselines. Moreover, the "Execution time" metric is also analyzed, to perceive whether it is required more time to execute the tests produced by *Pool++*, when compared with the ones created by vanilla version. The following sub-subsections describe the results obtained, as well as the conclusions drawn to answer the third research question.

#### 5.6.3.1 MP3

Table 5.15 illustrates the values obtained for the metrics related to the size of the test suites generated by the baselines. On one hand, the vanilla version can generate shorter test suites, with a single unit test inside of them. As a result, that reduction of the tests' size may indicate that the vanilla version can cover the CUT with less resources than the *Pool++*, even though is not as effective as the novel tool, as analyzed in the RQ1.

On the other hand, *Pool++* produces 1.76 more unit tests one average, augmenting the number of statements in 1.62 times, comparing with the ones created by the vanilla version. Nevertheless,

Table 5.15: Size comparison of *Pool++* and EvoSuite vanilla for `MP3`.

| Metric | Size (number of tests) | Length (number of statements) |
|---|---|---|
| *Pool++* (mean) | 2.00 | 4.10 |
| *Pool++* (std) | 0.00 | 0.30 |
| Vanilla (mean) | 1.13 | 2.53 |
| Vanilla (std) | 0.72 | 2.87 |

Table 5.16: Execution time comparison of *Pool++* and EvoSuite vanilla for `MP3`.

| **Metric** | **Execution time (ms)** |
|---|---|
| *Pool++* (mean) | 1,643.79 |
| *Pool++* (std) | 1,325.80 |
| Vanilla (mean) | 684.92 |
| Vanilla (std) | 248.06 |

Table 5.17: Vargha-Delaney values for the size performance of both baselines in `MP3` class. *x*: *Pool++*, *y*: vanilla. The values in *bold* identify the statistical significant effect sizes.

| **Metric** | **Size (number of unit tests)** | **Length (number of statements)** |
|---|---|---|
| $\hat{\mathbf{A}}_{xy}$ | **0.97** | **0.97** |

Table 5.18: Vargha-Delaney values for the execution time performance of both baselines in `MP3` class. *x*: *Pool++*, *y*: vanilla. The value in *bold* identifies the statistical significant effect size.

| **Metric** | **Execution time (ms)** |
|---|---|
| $\hat{\mathbf{A}}_{xy}$ | **0.84** |

the magnitude of them is not altered, meaning the increase of the statements, and the number of test suites is not sufficient to impact in a severe manner the final test suite obtained by the novel tool. Moreover, the standard deviations are reduced for both baselines, inducing they have systematically done test suites with the same length, during all the experiment runs conducted.

In the opposite direction direction, the execution time has reduced with the novel mocks. As illustrated in Table 5.16 the tests generated by *Pool++* require 2.40 more times of the execution time needed by the vanilla version, on average. Moreover, it is also evidenced that *Pool++*'s standard deviation is 5.34 times greater than the vanilla version, indicating a more varied behavior during the novel tool's performance.

The aforementioned results demonstrate *Pool++* generates for this subject program an execution overhead, when compared with EvoSuite vanilla, due to the raise in the execution times of the test suites generated. Furthermore, the increase in the test suites size and length has also contributed to the slower performance of *Pool++*, for this subject program.

Alongside the average scores and standard deviations, the values obtained in the Vargha-Delaney tests confirm that the test suites created by *Pool++* are longer than the ones produced by the vanilla version in 96.67% of the occurrences, as evidenced in Table 5.17. The aforementioned results are also expected, as *Pool++*'s average scores are superior to the ones resulting from the vanilla's performance. As a result, it confirms the increase in the amount of resources needed by *Pool++*'s test suites to cover the CUT, despite the increase of the coverage scores obtained in the experiments.

In a similar direction, it is verified as well the superiority of the execution time by *Pool++*'s test suites, comparing with the vanilla version. As illustrated in Table 5.18, the test suites created

Table 5.19: Wilcoxon Mann-Whitney values for the size performance of both baselines in `MP3` class. It is possible to conclude there is statistical difference in the results obtained for both metrics.

| Metric | Size (number of tests) | Length (number of statements) |
|---|---|---|
| $p$ | $\mathbf{5.32 \times 10^{-10}}$ | $\mathbf{5.32 \times 10^{-10}}$ |

Table 5.20: Wilcoxon Mann-Whitney values for the time performance of both baselines in `MP3` class. It is possible to observe there is statistical difference in the results obtained for this metric.

| Metric | Execution time (ms) |
|---|---|
| $p$ | $\mathbf{6.95 \times 10^{-5}}$ |

by the novel tool require more time in 84% of the occasions, than the ones produced by vanilla. Therefore, it is demonstrated the existence of the output delay generated by *Pool++*, as the tests require a greater temporal budget to be executed in completion.

Furthermore, it is important to also analyze the Wilcoxon Mann-Whitney U-test metric, to ensure the discrepancy between the results. For that purpose, Table 5.19 reveals the Wilcoxon Mann-Whitney values obtained for the test suites' size comparison by both baselines. In fact, it shows there is evidence for the performance discrepancy between the tools utilized, as the $p$-value is smaller than 0.05. Hence, it is possible to conclude *Pool++* creates longer test suites, as they possess lengthier unit tests, comparing with the ones generated by the vanilla version.

Morover, it is also evidenced the performance discrepancy in the tests execution's performance. As shown in Table 5.20, Wilcoxon Mann-Whitney U-test's $p$-value obtain is considerabily lower than 0.05 level of confidence stipulated for this study. Therefore, it is proved the results obtained from the executions from *Pool++*'s test suites are discrepant from the ones generated by the vanilla version.

Therefore, it is evident the output overhead created by *Pool++*'s tests, for the `MP3` class. In fact, the magnitude on the "Size", and "Length" variables do not differ in a significant amount, as the vanilla version generates test suites with one unit test, while *Pool++* generates with two unit tests, increasing as well the number of statements in 1.62 times. Moreover, the test suites produced by *Pool++* needs 2.40 times more time to be executed, when compared vanilla version, due to the runtime replacement of the mocks generated, as well as the lengthier tests *Pool++* produces.

As a result, this induces *Pool++* decreases the performance of the execution unit tests generated, due to the output overhead generated.

#### 5.6.3.2 SoundPlayer

For the `SoundPlayer` subject, the differences between the baselines' performances are also evident, when it comes to size terms. Notwithstanding, the discrepancy between the two tools is not as pronounced as would initially be anticipated from the `MP3`'s results. The results obtained from the experiments conducted in this subject program are illustrated below in the following table:

Table 5.21: Size performance of *Pool++* and EvoSuite vanilla for `SoundPlayer`.

| Metric | Size (number of tests) | Length (number of statements) |
|---|---|---|
| *Pool++* (mean) | 3.00 | 8.00 |
| *Pool++* (std) | 0.00 | 0.00 |
| Vanilla (mean) | 3.00 | 6.00 |
| Vanilla (std) | 0.00 | 0.00 |

Table 5.22: Execution time comparison of *Pool++* and EvoSuite vanilla for `SoundPlayer`.

| Metric | Execution time (ms) |
|---|---|
| *Pool++* (mean) | 4,451.04 |
| *Pool++* (std) | 958.76 |
| Vanilla (mean) | 850.17 |
| Vanilla (std) | 39.87 |

Table 5.23: Vargha-Delaney test for size comparison performance of both baselines in `SoundPlayer` class. *x*: *Pool++*, *y*: vanilla. The values in *bold* identify the statistical significant effect sizes.

| Metric | Size (number of tests) | Length (number of statements) |
|---|---|---|
| $\hat{\mathbf{A}}_{xy}$ | 0.50 | **1.00** |

Table 5.21 illustrates the test suites' sizes obtained from both versions, for the `SoundPlayer` class. It is possible to conclude that vanilla generates shorter test suites, as they have less statements when compared with the ones generated by *Pool++*, despite both baselines generate them with the same number of tests. Moreover, the values obtained for the standard deviations from both metrics are equal to 0.00, meaning *Pool++* and vanilla version always generate tests with the same length for `SoundPlayer` class, indicating *Pool++*'s test suite does not provide any advantage, nor disadvantage, in the size aspect. Adding to this, it is fundamental to analyze the vargha-delaney metric, as well as the Wilcoxon Mann-Whitney U-test, to ensure the conclusions drawn with the aforementioned metrics.

As observed in Table 5.21, there is a significant difference between the time execution required by *Pool++*'s test suites, and the ones resulting from the vanilla version. In fact, former ones requires 5.24 more times to be executed, when compared with the latter ones. As a result, the increase in execution time coverage confirms the existence of an execution overhead, generated by the mocks utilized in the test suites present in *Pool++*, as occurred to the `MP3` class.

As illustrated in Table 5.23, the evidence presented confirms the findings discussed in the aforementioned table. In fact, there is no significant difference between the two baselines in terms of the number of unit tests. Nevertheless, it is evidenced *Pool++* produces them with more statements, when compared to the vanilla version. As a result, the values obtained are consistent with the results shown in Table 5.21.

Moreover, it has also been analyzed the Vargha-Delaney test, to assess if there is an outperformance between the results obtained from *Pool++* and the vanilla version. As shown in Table 5.24,

Table 5.24: Vargha-Delaney values for the execution time performance of both baselines in `SoundPlayer` class. *x*: *Pool++*, *y*: vanilla. The value in *bold* identifies the statistical significant effect size.

| Metric | Execution time (ms) |
|---|---|
| $\hat{\mathbf{A}}_{xy}$ | **1.00** |

Table 5.25: Wilcoxon Mann-Whitney values for the the performance of both baselines in `SoundPlayer` class. The values in *bold* indicate there has been statistical difference in the results obtained for the corresponding metric.

| Metric | Size (number of tests) | Length (number of statements) |
|---|---|---|
| $p$ | 1.00 | $\mathbf{5.32 \times 10^{-10}}$ |

Table 5.26: Wilcoxon Mann-Whitney values for the time performance of both baselines in `SoundPlayer` class. It is possible to observe there is statistical difference in the results obtained for this metric.

| Metric | Execution time (ms) |
|---|---|
| $p$ | $\mathbf{2.88 \times 10^{-9}}$ |

the execution time for tests created by *Pool++* is superior 100% of the times, when compared to one obtained from the execution of test suites generated by EvoSuite vanilla. That result indicates that it is likely to observe an execution overhead for test suites by *Pool++*.

Adding to this, Table 5.25 also illustrates the absence of discrepancy between the baselines, for the `SoundPlayer` class. As the $p$-value obtained in the "Size" metric is 1.00, which is greater than 0.05, it is possible to state that there is no statistical difference between *Pool++* and vanilla version, as they generate tests of the same size. In the contrary direction, the one calculated for the "Length" metric is inferior to the confidence value, proving there results obtained by both baselines in this metric are discrepant.

Moreover, the $p$-value is less than 0.05, for the "Execution time" variable, as illustrated in Table 5.26. The result evidences the existent statistical difference between the execution times for both sets of test suites generated by the baselines. Aggregating that conclusion with the results obtained on previous metrics, it is possible to conclude there is produced an overhead on the output execution, due to the test suites size, as well as due to the mocks utilization on unit tests.

When analyzing the results obtained from an overall perspective, it is possible to conclude *Pool++* produces unit tests containing an output overhead associated with them, refuting the hypothesis formulated for this research question. In fact, there is evidence it generates longer test suites for the `MP3`, while it has maintained the test suites' size for the `SoundPlayer` subject program. Nevertheless, it is relevant to recall *Pool++* has improved coverage scores in a significant proportion for that class, indicating the novel tests are necessary, and have passed through the minimization process.

When it comes to the execution time, the overhead produced is evident, as the test suites produced by the novel tool take more time to be processed, when compared to the vanilla version.

The delay produced in the execution time is due to the time added to replace the target target classes and their methods, with the respective mocks, and generate the needed data to it.

**RQ3:** *Pool++* has increased the test suites in their length, when compared to the vanilla version, while augmenting the number of unit tests in 1.76 times, for the `MP3` class, and maintaining that number in the `SoundPlayer` one. Furthermore, the execution time has augmented in 5.24 times for the `SoundPlayer` class, and 2.40 in `MP3` class. These results confirm *Pool++* produces an output overhead, caused by the novel mocks, and their integration in EvoSuite's test suites.

## 5.7 Discussion

The conducted empirical study is important to explore the advantages, and disadvantages of *Pool++*. In fact, the controlled runs have mitigated all the risks considered, especially the randomness of the novel tool, inherited by GA, giving the security of the results collected in the experiments. Moreover, the metrics utilized have allowed to extract relevant conclusions about *Pool++*'s performance. Several aspects have been explored during these experiments, to compare both baselines in three main components: their efficacy, the test production's delay, and the output overhead of their unit tests.

On one hand, *Pool++*'s mocks with fake data, generated for audio processing purposes, have indeed enhanced the coverage scores of the CUTs. Thanks to replacing the target methods, as well as objects in the code under test, *Pool++* provides an alternative to EvoSuite and its GA to simulate the respective behaviour of the system under test. As a result, it is confirmed the hypothesis formulated, and more crucially, the primary objective of this work: **increase the coverage scores, while using mocks containing fake data.**

On the other hand, the experiments have also evidenced drawbacks in this novel tool: in fact, ***Pool++*'s test generation process produces overhead**, when compared to the vanilla version, as the novel produces few generations of the unit tests population, for both baselines, refuting the hypothesis formulated in RQ2. By producing the mocks, as well as integrating them in the unit tests and respective execution, the GA have less time available in the search budget to produce more generations of tests, otherwise vanilla do. Hence, there is an opportunity for improvement in the future work, so that *Pool++* can generate better unit tests with the mocks, spending less time to generate and integrate them, and hence saving more time for the GA's search process.

Moreover, **the size of the tests generated by *Pool++* has increased**, comparing with the ones produced by vanilla, due to the augment of the produced test suites' size for `MP3` class. However, this increase is expected, as *Pool++* has improved the coverage score in the aforementioned class by a significant amount, thanks to the two unit tests generated for the CUT. As the number of unit tests is still reduced, allied with the fact *Pool++* has maintained the same size for `SoundPlayer` class, it is possible to state the increase of the size is controlled, for the utilized baselines.

When it comes to the test execution, **it has been observed an augment of the execution time**, for both subject programs, in test suites created by the novel tool. That increase is explained due

to the integration and replacement of the mocks, adding an additional time to the required to run the statements present in CUT. As a consequence, it leads to the conclusion *Pool++* generates an output overhead while creating the unit tests for the code under test. Moreover, with the test suites increasing in their size, as occurred in the `MP3` class, it also caused an increase in execution time, as more lines are being run, being more contained in the `SoundPlayer` class, as the sizes of the test suites produced by both baselines for that class are equal.

Furthermore, another important detail to explore is the scalability of the output generated by the novel solution: as evidenced in the experiments conducted, *Pool++* has doubled the size of the test suite for `MP3` class, and maintaining it for the `SoundPlayer` one, when it comes to the number of tests. Despite the controlled length for the simple baselines utilized throughout the experiments, it is expected *Pool++* can generate longer test suites, for more complex cases. Nonetheless, those augment of the test suites' extension is controlled by EvoSuite's minimization algorithm, to reduce the size of the test, only containing the unit tests and minimal structure that maximizes the coverage of the CUT [32].

An increase of the generated test suites' length has also impact on other relevant aspects, such as the readability aspects. In fact, longer test suites are less likely to be better comprehended by the testers and developers. Nevertheless, a A/B user tests would be needed to assess and analyze the novel tests suites generated by *Pool++* on the readability aspect, to comprehend if they are more or less readable than the ones generated by vanilla version.

# Chapter 6

# Conclusion and future work

## 6.1 Conclusions

This study explores how to enhance the efficacy of the unit tests created by EvoSuite. We have concluded in the Chapter 1 the automatic test generation tool does not have the input needed, often more complex than the primitives, to cover specific cases, in particular focus on audio processing classes. For that, it has been created *Pool++*, which extends the tool by creating mocks which possess random data, to simulate the behavior of audio-processing Java classes, creating alternative solution to cover objects, methods, and classes that have not been covered once.

It has been evaluated *Pool++*'s performance on three parameters: on efficacy, on execution overhead, and on output overhead, utilizing two classes present on SF110 dataset, exposing the advantages and disadvantages of this novel solution. In fact, the increase of the coverage is notorious, as being collected an increase of line coverage from 40% to 97% and from 18% to 55% for branch coverage, in `MP3` class, while the tool has risen the line coverage from 44% to 81% in `SoundPlayer` class, and enhancing the branch coverage from 50% to 88%. Nevertheless, *Pool++* also creates an execution overhead, to create the novel mocks and add them for the generate tests suites.

Another relevant factor to be analyzed is the solution's scalability: in case it happens the CUT be longer, with more complex cases, the experiments realized in this work evidence *Pool++* may increase the coverage scores obtained when using the vanilla version. Nonetheless, as the class under test is longer, it is also possible to generate a slight increase on the production overhead, in case it is needed more mocks.

This work's findings indicate there is evidence the approach attempted in this study has reached the objective: improve the coverage scores of the generated test suites. As a result, the mocks containing the random created by *Pool++*, EvoSuite can test more complex cases related to audio processing tasks that have been uncovered once. Therefore, it is enhanced the automatic test software generation, contributing to the development of safer, and more robust software.

## 6.2 Future work

Despite the contribution made in this work, there is still room for improvements to enhance *Pool++*. A possible future work would consist of enhancing the mocks generation, by reducing the overhead generated when creating them. Hence, it would have more time available for EvoSuite's GA to produce more generations and improve the test generation process.

In this thesis, it is demonstrated the findings made to generate mocks containing fake data to process audio content. Nevertheless, and as shown in the Chapter 5, it is possible to conclude there has been utilized a reduced range of subjects in SF110 dataset, meaning a few range of classes have benefited from the novel mocks. As a result, a possible enhancement to this work would consist of expanding them to cover other classes which can process different type of content, in particular complex files such as Extensible Markup Language (XML), or PDF.

In fact, it has been indentified other projects which had not contained audio-processing-related methods or classes. For instance, it has been found several classes containing objects which process components an XML document (such as the `Node`, and the `Element`, and `Document`), on `31_xisemele` project from the SF110 dataset. As a result, those classes would also be valid candidates to be mocked with the necessary data, to increase the coverage scores for the test suites generated for those classes.

Another direction for future work is to conduct the experiments on a different dataset, in order to evaluate *Pool++*'s performance across a broader and more diverse set of examples. Despite being one of the most popular datasets in the scientific community [33, 72, 77, 93], the examples provided in SF110 do not utilize the most up-to-date frameworks, for audio processing, not exposing the efficacy of *Pool++*'s on those frameworks. Hence, exposing *Pool++* to novel scenarios beyond those included in the SF110 dataset would enable more robust and generalizable conclusions regarding its effectiveness in audio-related classes.

Moreover, due to the increase of test suites' size, as well as their unit tests, the readability of the novel tests may be decreased. This growth in complexity might introduce challenges by quickly understanding the purposes and structure of the tests. To confirm whether the readability aspect has been compromised, an A/B user test is performed to determine whether testers and developers understand what the test suite contains and what it tests [14, 23].

The following subsections will also discuss other approaches to enhance the materialization of the idea of this work: producing fake data to be utilized in unit tests to increase their coverage. Moreover, there is a discussion subsection regarding the ups and downs of the ideas considered.

### 6.2.1 Fake Data Generators

One question raised during *Pool++*'s development is how often a fake data instatiator must be invoked during the test generation process. A valid approach is to invoke the generator whenever a non-primitive attribute is invoked a in a test suite, removing that responsibility from EvoSuite. One studied generator is `Instancio` [46], a fake data generator for Java which can generate primitives, or Java objects, that can be directly utilized in the test suites created by EvoSuite.

To insert fake data generators on tests, it is necessary to modify the way EvoSuite instatiates the constructors in the chromosomes of the evolutionary algorithm. For that, the constructor-visitor methods, the ones responsible for the non-primitive class instatiators, will be modified, as well as create another one responsible for writing the instantiation of an `Instancio` object, which will create a fake object of the non-primitive target class.

Listing 6.1: Unit test generated for a Person class by EvoSuite, without `Instancio` integration.

```
1 Test 0:
2 Person person0 = new Person("", "", 0, (Date) null);
3 person0.getAge();
4 person0.getName();
5 person0.getName();
6 person0.getBirthDate();
```

Listing 6.2: Unit test generated for the Person class by EvoSuite with a simple instatiation of `Instancio`.

```
1 Test 0:
2 Person person0 = Instancio.create(Person.class);
3 person0.getAge();
4 person0.getName();
5 person0.getName();
6 person0.getBirthDate();
```

After introducing `Instancio` in the visitors, the tests generated by EvoSuite evolve from having similar examples as Listing 6.1, to unit tests similar to the one illustrated in Listing 6.2. In fact, there is no significant difference between the examples illustrated, as the generation of objects using `Instancio` is close to the Java's traditional one. As a result, there is no significant impact on the construction of unit tests, for CUT containing simple objects and classes.

It has been explored how to insert the fake data generator tool and there has been evidence of the benefits from that approach. In fact, the simple introduction of `Instancio` in the unit test generated by EvoSuite has increased `BatchDriver` from `newzgrabber` project, from SF110 dataset. For that class, it enhances the branch coverage from 2% to 4%, and line coverage from 7% to 11%.

However, when it comes to generate fake instances of abstract classes, or interfaces, `Instancio` is not capable to generate without specific methods (its `.set()` and `.supply()` methods), besides the definition of how those classes should be implemented or extended, utilizing a `Settings` instance, and the target classes [46]. As a result, adapting the code needed to generate the required instances would be cumbersome and prone to generate instances with default or even null values. Hence, it is necessary to modify our approach, to take more advantage of the fake data generator.

Nonetheless, we also found a common limitation both `Instancio` and EvoSuite have. Neither the automatic test generation tool, nor the fake data generator, can not instatiate in a proper manner objects containing interfaces, or abstract classes as attributes. However, it is known that `Instancio` is capable of that, by using specific methods (`.subType()` one, which indicates to `Instancio` what classes should be invoked that implement those interfaces or classes, specified in a `Settings` object from `Instancio`) [46]. As a result, a proper usage of these methods might

contribute to the proper generation of more complex objects and, therefore, provide better data for unit tests, allowing them to better cover the CUT.

Another tool explored is `Datafaker`, which is an alternative fake data generator for Java, capable of generating meaningful strings, quotes, and values for other Java primitives [26]. The particularity of this generator is the meaningfulness of the values produced: due to the wide variety of its providers, the values generated are closer to the ones present in real-world applications. Hence, utilizing them in EvoSuite's test suites may enhance the simulation of the behavior to be assessed, by the code under test.

### 6.2.2 Complex files generation

Furthermore, another approach considered to enhance the coverage of the tests created by EvoSuite is the generation of complex mock files. As shown in the problem statement in Chapter 1, EvoSuite faces challenges extracting information from certain kind of files, in that particular case, audio files. Hence, a possible solution is to create a mock file, containing the respective mock data (for the case of the problem found in problem statement, it would be an audio file possessing an audio input stream), so that it can generate a valid input for the faulty line, as illustrated in Figure A.2.

To implement the aforementioned idea, it is necessary to answer two questions related to EvoSuite's functioning:

- How does the mapping between the file and mock files occur?
- What objects are used in the generated tests' pool?

When generating its test suites, EvoSuite creates an instance of a test cluster (of `TestCluster` class), which is responsible to define what objects will be instatiated and their respective dependencies between them, according to the CUT's attributes. For instance, if it has an attribute of string type, then the cluster will add to it all the possible generators for string attributes. In case there are other classes generating strings, it will include them with the necessary dependencies.

In that specification, if the code to be tested contains a `File` instance, the cluster will define how it will be mapped on code to be written. EvoSuite has the ability to generate a `File` instance, despite containing the null value. Furthermore, it is also assessed if they can convert to a mock file. All EvoSuite's mocks are present in the `MockList` class, in a list which is traversed and it is checked if any of them has the `File` class as super class, as it happens with the `MockFile` class. As a result, they convert the `File` class into the Mock class one. This way, EvoSuite will be capable of converting file objects to enhanced mock ones, augmenting the pool with more possibilities for file handling in test generation. Therefore, EvoSuite's test pool will have more powerful individuals, that is, unit tests which utilize the mock files, which can be used to cover more functionalities in the SUT.

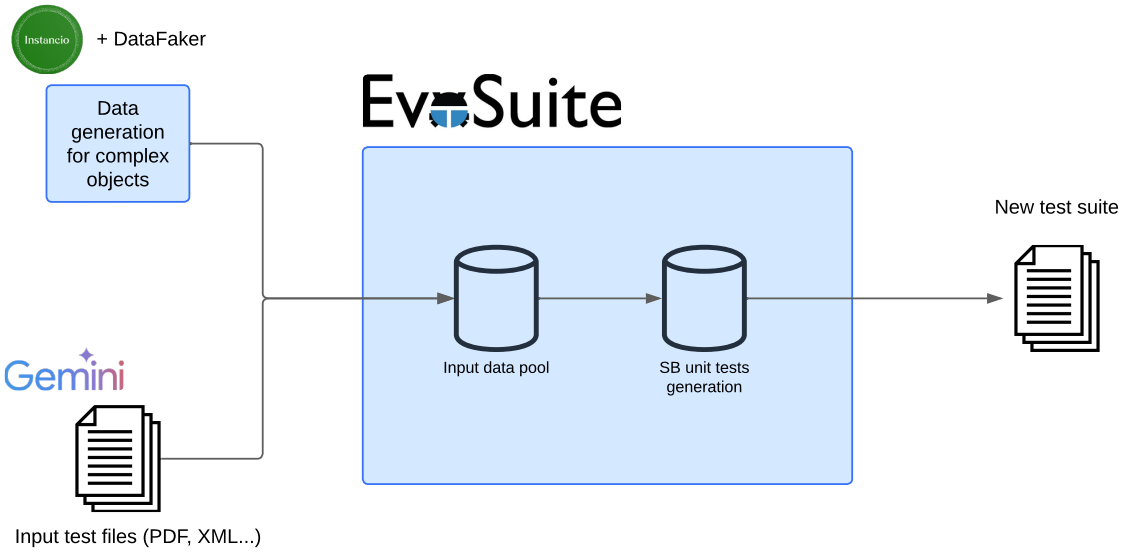


Figure 6.1: Architecture of the novel approach, aggregating LLMs, fake data generators, and EvoSuite.

### 6.2.3 Combination of LLMs with fake data generators

In the current days, it is visible the impact LLMs in several software areas, especially in the software testing one. Moreover, due to the generative capabilities of several generative LLMs, it has also been studied the utilization of these tools for data generation in EvoSuite [13, 53]. Furthermore, to combine with the previous ideas discussed a valid approach, we have also reflected about the utilization of fake data generators and Google Gemini's API [41], for complex data generation to be utilized as input in EvoSuite's unit tests.

As illustrated in Figure 6.1, the idea consists of a plug-in for EvoSuite, which integrates two fake data instatiators, i.e., `Instancio` and `Datafaker`. Moreover, the plug-in would also access Gemini's capabilities via API. The first two instatiators are responsible for generating and populating structured data using primitives (strings, numbers, and collections), as well as the population of more complex classes and objects. Furthermore, the Gemini's LLM would conduct the generation of the required files, according to their type, using the respective predefined prompts in Listing 6.3.

With the integration of these tools, the input pool is augmented, and hence EvoSuite would have more available data to populate the unit tests generated. Moreover, this increase of available data is straightforward, by calling the respective fake data generators at the beginning of the test suites. Hence, it is expected that EvoSuite is capable of dealing with more complex inputs, leading to more effective results in their test suites, as *Pool*++ not only deals with simple data input (such as primitives), but also creates properly populated objects or files for the unit tests.

When it comes to file generation with Gemini, *Pool*++ would use its API [1] whenever it is needed a test file as input. To do that, the API would embed prompts capable to generate files of four different types: PDF, XML, audio, and image files. The audio files types supported are .WAV

[1]Google Gemini API: `https://ai.google.dev/api/generate-content`, accessed 10 June 2025.

and .MP3, while the supported file types for images are .PNG and .JPEG. The following snippet of code provides an example of a generic prompt used by this solution.

Listing 6.3: Example of Geminis's prompts by *Pool++*, to generate a sample of an image to be used as input in an unit test.

```
curl -s -X POST \
  "https://generativelanguage.googleapis.com/v1beta/models/gemini-2.0-flash-preview-image-generation:
  generateContent?key=$GEMINI_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "contents": [{
      "parts": [
        {"text": "Generate a sample png image for test purposes with 16x16 pixels."}
      ]
    }],
    "generationConfig":{"responseModalities":["TEXT","IMAGE"]}
  }' \
  | grep -o '"data": "[^"]*"' \
  | cut -d'"' -f4 \
  | base64 --decode > gemini-native-image.png
```

The Listing 6.3 shows the structure of the Gemini's prompts to be used in the aforementioned idea. As EvoSuite may need different types of files when generating unit tests, the prompts must also be dynamic, to cover all the different file types supported. As a result, when EvoSuite generates the data pool, it would catch the values needed for test file generation process, inserting them into the prompts.

Moreover, it could be used the free version of the API, preventing from adding an additional cost to the solution, despite having more limited resources for content generation [2]. However, the simulation of file creation is already implemented, with EvoSuite's mock files [7, 9, 11, 54]. Even if this solution would result in an extension of EvoSuite's `MockFile` class, the generation of the files would face the same challenge found in Section 6.2.2, as it would also require the machine's file system usage, being dependent on this external component.

### 6.2.4 Discussion

In the previous section, there have been introduced several approaches to improve the work of this thesis. In fact, it has been presented novel ideas to improve the work done on this thesis, such as the expansion of the content supported by the mocks, as well as the reduction of the overheads generated by them. Those improvements would lead *Pool++* to a wider range of the complex it could cover, and enhance its performance.

Furthermore, novel ideas to address the problem in this work have also been presented, with advantages and disadvantages associated with them. In the first approach, the usage of fake data generators can be useful to automate the generation of complex objects present with more meaningful values in the CUT, removing the instation's responsibility from EvoSuite. Moreover, there is also evidence of the benefits of that approach, on a class from SF110 dataset.

[2]Google Gemini API pricing: https://ai.google.dev/gemini-api/docs/pricing, accessed 10 June 2025

Nevertheless, the generation of those complex classes would be cumbersome, as it would be needed to not only to analyze the generated code and perceive whether it is necessary to invoke the `Settings` instance, to cover interfaces and abstract attributes for the class under test, when using `Instancio` for the effect. Moreover, if there is no class in the project the CUT which implements, or extends the interface or abstract class to be populated, there is no viable alternative for `Instancio` to cover that class, losing its purpose to cover complex classes. Allied to this, as `Instancio` consists of a third-party tool, it would generate a dependency for EvoSuite, with the eventual risk of the fake data generator having some vulnerability or not being available for usage anymore, which may compromise the viability of the aforementioned solution.

Moreover, the power of `Datafaker` is also limited. In fact, `Instancio` can also produce values for basic Java attributes, besides the random generation of object instances. Moreover, EvoSuite has the *Randomness* class, which is also capable of creating random values for primitive types and collections. As a result, the usage of `Datafaker`'s benefit is reduced.

In the second approach, generation of complex files is another approach that provides a beneficial solution to the problem presented in this work. In fact, the extension of the EvoSuite's `MockFile` to generate a file containing fake data, to simulate a valid input, would maintain the majority of the bytecode equal to the one created by executing the CUT, as it would only replace the `File` objects instatiation. Moreover, the novel mock files would provide the data needed by EvoSuite to cover the CUT's methods which process the file's content, if they are implemented in a proper manner, enhancing the coverage in those methods.

However, there are three aspects that might compromise the generation of a valid document: its extension, its data content, and whether the file is or is not in the file system.

As the file generation process would be affected by GA's randomness, EvoSuite might not generate a mock file with the certain extension. However, as the tool has the ability to the detect in the CUT target values, such as strings, sub-strings, and integers that might be important to test the functionalities under test. As a result, if the target extension for the file needed is specified in the their source code, there is a significant probability for EvoSuite insert in the mock file's name the correct extension, and hence making the mock file more adequate for testing.

Another major challenge for that approach is the case it is necessary the necessary file have a specific content on that data, as the case of the problem presented in the first chapter. Audio files, for instance, have a specific format and the bytes present on them are also encoded in a proper manner. Since EvoSuite does not know how to generate that content for the mock file, it causes exceptions and errors on the Java sound's libraries, which decreases the coverage of the test, or not testing SUT at all, in the worst scenario. A possible approach to overcome the previous challenge is provide EvoSuite the ability to generate sample bytes that can be inserted in the mock files, in a similar approach described to this thesis. As a result, mock files having appropriate content for the context of the test, which will also make them appropriate for the tests, and allowing EvoSuite to test the functionalities under test.

However, it would also be needed to create the respective generator for each type of file. Since there are several types of documents that can be processed, a selection to a strict group of them

would be required, to limit the range of that solution. That approach would be able to generate for three kinds of documents: PDF, XML, and Waveform Audio File Format (WAV) files, since they are the most common types to be processed in Java applications, by analyzing SF110 dataset.

The last challenge to overcome is whether the mock file is or is not in the file system. It is a fact mock files are not stored in the actual machine's file system, but in a virtual one. Moreover, if they are stored in the actual system, that would be dangerous for the machine, as it would generate files each time a test that assesses those functionalities. As a result, in a scenario a test is run several times, it would consume a significant amount of resources, impacting EvoSuite's performance.

To overcome this problem, one possible solution is to generate a temporary file, which is stored in the temporary folder, making it available for the file system but it is eliminated, once the machine is rebooted. Jointing this solution for the challenge found, it results in the new mock file class, extending the EvoSuite's MockFile one. The novel mock will generate the corresponding file's sample content, as well as the correct extension.

Furthermore, the combination of fake data generators with LLMs also brings advantages to the aforementioned approaches. In a similar way to the previous solutions, the aforementioned approach also removes the responsibility from EvoSuite to generating the needed input to simulate the intended behavior from the CUT. Moreover, the aggregation with the generators robusts the solution, providing to the automatic test generation tool another source of data for mock creation.

Nevertheless, there are risks associated with this solution, due to the inherited debilities from the tools utilized. Besides the risks related to the usage of the fake data generators, the major threat from this solution threatens the determinism of the tests produced, due to the inconsistency of the LLMs. As the generative models may not always produce the intended output, due to their generative abilities, it will lead to inconsistent generation of test suites. That inconsistency may result in flaky tests, reducing the coverage scores for the test suites, for a certain CUT.

To mitigate the inconsistency threat, it is mandatory to restrict the generative abilities of the LLM, so that its response may be more controlled. An idea to materialize that solution consists of defining a standarized structure of the prompts to request the files, where it is solely variying the file size, as well as the file type requested, according to the requirements of the functionality to be tested. As a result, the contention of the valid prompts may also contain the LLM responses, generating more consistent inputs.

Another paramount detail is the choice of the tools utilized of this approach: in fact, there is no obligation to utilized the aforementioned tools, and can be utilized other that have other advantages to the approach. For instance, the utilization of local LLM, as the case of Meta's Llama 3.1[3], would remove the need to make Web requests, and obtain similar responses as the ones provided by the Google Gemini. Hence, it is necessary to make a careful choice of the tools to be utilized, when advancing with the aforementioned idea in the future.

[3]Meta Llama 3.1 version, https://ai.meta.com/blog/meta-llama-3-1/, accessed 26 June 2025

# References


[1] Sujoy Acharya. *Mastering unit testing using Mockito and JUnit*. Packt Publishing Ltd, 2014. Cited on page 7.

[2] Nasser Albunian, Gordon Fraser, and Dirk Sudholt. Causes and effects of fitness landscapes in unit test generation. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 1204–1212, Cancún Mexico, June 2020. ACM. ISBN 978-1-4503-7128-5. doi: 10.1145/3377930.3390194. URL https://dl.acm.org/doi/10.1145/3377930.3390194. Cited on page 16.

[3] Shaukat Ali, Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel C. Briand. Generating test data from ocl constraints with search techniques. *IEEE Transactions on Software Engineering*, 39(10):1376–1402, 2013. doi: 10.1109/TSE.2013.17. Cited on pages 20 and 21.

[4] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272, 2017. doi: 10.1109/ICSE-SEIP.2017.27. Cited on pages 1 and 2.

[5] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 1–10, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1985795. URL https://doi.org/10.1145/1985793.1985795. Cited on page 48.

[6] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3): 594–623, June 2013. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-013-9249-9. URL http://link.springer.com/10.1007/s10664-013-9249-9. Cited on page 21.

[7] Andrea Arcuri and Juan P. Galeotti. Sql data generation to enhance search-based system testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 1390–1398, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361118. doi: 10.1145/3321707.3321732. URL https://doi.org/10.1145/3321707.3321732. Cited on pages 1, 5, 16, 35, 36, and 71.

[8] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 79–90, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2642986. URL https://doi.org/10.1145/2642937.2642986. event-place: Vasteras, Sweden. Cited on pages 5, 11, and 49.

[9] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Generating tcp/udp network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 155–165, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786828. URL https://doi.org/10.1145/2786805.2786828. event-place: Bergamo, Italy. Cited on pages 7, 19, 36, and 71.

[10] Andrea Arcuri, José Campos, and Gordon Fraser. Unit test generation during software development: Evosuite plugins for maven, intellij and jenkins. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 401–408, April 2016. doi: 10.1109/ICST.2016.44. Cited on pages 1 and 6.

[11] Andrea Arcuri, Gordon Fraser, and René Just. Private api access and functional mocking in automated unit test generation. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 126–137, 2017. doi: 10.1109/ICST.2017.19. Cited on pages 7, 22, 23, 36, and 71.

[12] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5): 507–525, 2015. doi: 10.1109/TSE.2014.2372785. Cited on page 5.

[13] Benoit Baudry, Khashayar Etemadi, Sen Fang, Yogya Gamage, Yi Liu, Yuxin Liu, Martin Monperrus, Javier Ron, André Silva, and Deepika Tiwari. Generative ai to generate test data generators, June 2024. URL http://arxiv.org/abs/2401.17626. arXiv:2401.17626. Cited on pages 28 and 70.

[14] Matteo Biagiola, Gianluca Ghislotti, and Paolo Tonella. Improving the readability of automatically generated tests using large language models, 2024. URL https://arxiv.org/abs/2412.18843. Cited on pages 34 and 67.

[15] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. Sushi: a test generator for programs with complex structured inputs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 21–24, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356633. doi: 10.1145/3183440.3183472. URL https://doi.org/10.1145/3183440.3183472. Cited on pages 16, 17, 34, and 36.

[16] Marcel Böhme, Charaka Geethal, and Van-Thuan Pham. Human-in-the-loop automatic program repair. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 274–285, 2020. doi: 10.1109/ICST46399.2020.00036. Cited on page 34.

[17] Jose Campos, Annibale Panichella, and Gordon Fraser. Evosuite at the sbst 2019 tool competition. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, pages 29–32. IEEE, May 2019. ISBN 978-1-72812-233-5. doi: 10.1109/SBST.2019.00017. URL https://ieeexplore.ieee.org/document/8812194/. Cited on pages 2 and 6.

[18] José Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. Entropy-based test generation for improved fault localization. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 257–267, 2013. doi: 10.1109/ASE.2013.6693085. Cited on page 1.

[19] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. Continuous test generation: enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 55–66, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330138. doi: 10.1145/2642937.2643002. URL https://doi.org/10.1145/2642937.2643002. Cited on pages 1, 31, and 50.

[20] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 104:207–235, 2018. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2018.08.010. URL https://www.sciencedirect.com/science/article/pii/S0950584917304858. Cited on pages 1 and 50.

[21] Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. Search-based test data generation for sql queries. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1220–1230, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180202. URL https://doi.org/10.1145/3180155.3180202. Cited on page 10.

[22] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000. ISSN 0362-1340. doi: 10.1145/357766.351266. URL https://doi.org/10.1145/357766.351266. Cited on page 31.

[23] Avito Alexandre Costa da Silva. Enhancing the readability of automatically generated unit tests with large language models. Master's thesis, Faculty of Engineering of University of Porto, 2024. Cited on pages 34 and 67.

[24] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211, 2014. doi: 10.1109/ISSRE.2014.11. Cited on page 6.

[25] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 57–67, 2017. Cited on page 33.

[26] Datafaker.net. Getting started - datafaker. https://www.datafaker.net/documentation/getting-started/, 2016. Accessed 16-02-2025. Cited on pages 22 and 69.

[27] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. Leveraging large language models for enhancing the understandability of generated unit tests. *arXiv preprint arXiv:2408.11710*, 2024. URL https://arxiv.org/abs/2408.11710. Cited on pages 33, 34, and 35.

[28] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. Fuzzing deep-learning libraries via automated relational api inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 44–56, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9413-0. doi: 10.1145/3540250.3549085. URL

https://doi.org/10.1145/3540250.3549085. event-place: Singapore, Singapore. Cited on pages 24 and 25.

[29] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie Van Deursen. Search-based crash reproduction using behavioural model seeding. *Software Testing, Verification and Reliability*, 30(3):e1733, May 2020. ISSN 0960-0833, 1099-1689. doi: 10.1002/stvr.1733. URL https://onlinelibrary.wiley.com/doi/10.1002/stvr.1733. Cited on page 15.

[30] Xuanwen Ding, Qingshun Wang, Dan Liu, Lihua Xu, Jun Xiao, Bojun Zhang, Xue Li, Liang Dou, Liang He, and Tao Xie. Finhunter: Improved search-based test generation for structural testing of fintech systems. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 10–20, Porto de Galinhas Brazil, July 2024. ACM. ISBN 9798400706585. doi: 10.1145/3663529.3663823. URL https://dl.acm.org/doi/10.1145/3663529.3663823. Cited on pages 12 and 13.

[31] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025179. URL http://doi.acm.org/10.1145/2025113.2025179. Cited on pages 1 and 6.

[32] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012. doi: 10.1109/TSE.2012.14. Cited on pages 1 and 65.

[33] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2), dec 2014. ISSN 1049-331X. doi: 10.1145/2685612. URL https://doi.org/10.1145/2685612. Cited on pages 1, 2, 48, and 67.

[34] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical Softw. Engg.*, 20(3):611–639, jun 2015. ISSN 1382-3256. doi: 10.1007/s10664-013-9288-2. URL https://doi.org/10.1007/s10664-013-9288-2. Cited on pages 1 and 49.

[35] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012. doi: 10.1109/TSE.2011.93. Cited on pages 1 and 5.

[36] Gordon Fraser, Andrea Arcuri, and Phil McMinn. A memetic algorithm for whole test suite generation. *Journal of Systems and Software*, 103:311–327, 2015. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2014.05.032. URL https://www.sciencedirect.com/science/article/pii/S0164121214001216. Cited on page 20.

[37] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers? a controlled empirical study, September 2015. ISSN 1049-331X. URL https://doi.org/10.1145/2699688. Place: New York, NY, USA Publisher: Association for Computing Machinery. Cited on pages 2 and 49.

[38] Gordon Fraser, José Miguel Rojas, and Andrea Arcuri. Evosuite at the sbst 2018 tool competition. In *Proceedings of the 11th International Workshop on Search-Based Software Testing*, pages 34–37. ACM, May 2018. ISBN 978-1-4503-5741-8. doi: 10.1145/3194718.3194729. URL https://dl.acm.org/doi/10.1145/3194718.3194729. Cited on pages 2 and 6.

[39] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 360–369, Pasadena, CA, USA, November 2013. IEEE. ISBN 978-1-4799-2366-3. doi: 10.1109/ISSRE.2013.6698889. URL http://ieeexplore.ieee.org/document/6698889/. Cited on pages 21 and 30.

[40] Javier Godoy, Juan Pablo Galeotti, Diego Garbervetsky, and Sebastián Uchitel. Enabledness-based testing of object protocols. *ACM Transactions on Software Engineering and Methodology*, 30(2):1–36, April 2021. ISSN 1049-331X, 1557-7392. doi: 10.1145/3415153. URL https://dl.acm.org/doi/10.1145/3415153. Cited on page 22.

[41] Google. Gemini api | google ai for developers. https://ai.google.dev/gemini-api/docs, 2023. Accessed 17-03-2025. Cited on page 70.

[42] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, page 67–77, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450314541. doi: 10.1145/2338965.2336762. URL https://doi.org/10.1145/2338965.2336762. Cited on page 1.

[43] Xiujing Guo, Hiroyuki Okamura, and Tadashi Dohi. Automated software test data generation with generative adversarial networks. *IEEE Access*, 10:20690–20700, 2022. Cited on pages 28 and 29.

[44] Youssef Hassoun, Phil McMinn, Kiran Lakhotia, Mark Harman, and Joachim Wegener. Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. *IEEE Transactions on Software Engineering*, 38(02):453–477, March 2012. ISSN 1939-3520. doi: 10.1109/TSE.2011.18. URL https://doi.ieeecomputersociety.org/10.1109/TSE.2011.18. Place: Los Alamitos, CA, USA Publisher: IEEE Computer Society. Cited on page 18.

[45] Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007. Cited on page 6.

[46] Intancio.org. User guide - instancio. https://www.instancio.org/user-guide/#subtype-mapping, 2022. Accessed 16-02-2025. Cited on pages 22, 67, and 68.

[47] Sungmin Kang, Robert Feldt, and Shin Yoo. Sinvad: Search-based image space navigation for dnn image classifier test input generation. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 521–528, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379632. doi: 10.1145/3387940.3391456. URL https://doi.org/10.1145/3387940.3391456. Cited on page 24.

[48] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie Van Deursen. Effective and efficient api misuse detection via exception propagation and search-based testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium*

*on Software Testing and Analysis*, pages 192–203, Beijing China, July 2019. ACM. ISBN 978-1-4503-6224-5. doi: 10.1145/3293882.3330552. URL https://dl.acm.org/doi/10.1145/3293882.3330552. Cited on page 25.

[49] Wanida Khamprapai, Cheng-Fa Tsai, Paohsi Wang, and Chi-En Tsai. Multiple-searching genetic algorithm for whole test suites. *Electronics*, 10(16):2011, August 2021. ISSN 2079-9292. doi: 10.3390/electronics10162011. URL https://www.mdpi.com/2079-9292/10/16/2011. Cited on pages 17 and 18.

[50] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. Automated test generation for rest apis: No time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 289–301, Virtual South Korea, July 2022. ACM. ISBN 978-1-4503-9379-9. doi: 10.1145/3533767.3534401. URL https://dl.acm.org/doi/10.1145/3533767.3534401. Cited on pages 25 and 35.

[51] Barbara Kitchenham and Pearl Brereton. A systematic review of systematic review process research in software engineering. *Information and Software Technology*, 55(12):2049–2075, 2013. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2013.07.010. URL https://www.sciencedirect.com/science/article/pii/S0950584913001560. Cited on page 8.

[52] Annu Lambora, Kunal Gupta, and Kriti Chopra. Genetic algorithm- a literature review. In *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, pages 380–384, 2019. doi: 10.1109/COMITCon.2019.8862255. Cited on page 6.

[53] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931, Melbourne, Australia, May 2023. IEEE. ISBN 978-1-66545-701-9. doi: 10.1109/ICSE48619.2023.00085. URL https://ieeexplore.ieee.org/document/10172800/. Cited on pages 26 and 70.

[54] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. Graph-based seed object synthesis for search-based unit testing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1068–1080, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3468619. URL https://doi.org/10.1145/3468264.3468619. Cited on pages 1, 5, 14, and 71.

[55] Stephan Lukasczyk and Gordon Fraser. Pynguin: automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, page 168–172, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392235. doi: 10.1145/3510454.3516829. URL https://doi.org/10.1145/3510454.3516829. Cited on pages 1 and 6.

[56] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of automated unit test generation for python. *Empirical Softw. Engg.*, 28(2), jan 2023. ISSN 1382-3256. doi: 10.1007/s10664-022-10248-w. URL https://doi.org/10.1007/s10664-022-10248-w. Cited on pages 1 and 50.

[57] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004. doi: https://doi.org/10.1002/stvr.294. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.294. Cited on pages 1, 5, 27, and 28.

[58] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, 2011. doi: 10.1109/ICSTW.2011.100. Cited on page 5.

[59] Phil McMinn, Muzammil Shahbaz, and Mark Stevenson. Search-based test input generation for string data types using the results of web queries. In *2012 IEEE Fifth international conference on software testing, verification and validation*, pages 141–150. IEEE, 2012. Cited on pages 1, 22, and 35.

[60] Phil Mcminn, Chris J. Wright, and Gregory M. Kapfhammer. The effectiveness of test coverage criteria for relational database schema integrity constraints. *ACM Trans. Softw. Eng. Methodol.*, 25(1), December 2015. ISSN 1049-331X. doi: 10.1145/2818639. URL https://doi.org/10.1145/2818639. Cited on pages 20 and 34.

[61] OpenAI. Chatgpt. https://chatgpt.com, 2022. Accessed 17-03-2025. Cited on page 34.

[62] Oracle.com. Java sound technology. https://docs.oracle.com/javase/8/docs/technotes/guides/sound/, 2025. Accessed 16-02-2025. Cited on page 41.

[63] Wendkûuni C. Ouédraogo, Laura Plein, Kader Kaboré, Andrew Habib, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. Enriching automatic test case generation by extracting relevant test inputs from bug reports, December 2023. URL http://arxiv.org/abs/2312.14898. arXiv:2312.14898 [cs]. Cited on pages 1, 29, 35, 36, and 49.

[64] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-865-7. doi: 10.1145/1297846.1297902. URL http://doi.acm.org/10.1145/1297846.1297902. Cited on page 1.

[65] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: 10.1109/ICSE.2007.37. URL http://dx.doi.org/10.1109/ICSE.2007.37. Cited on page 1.

[66] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015. doi: 10.1109/ICST.2015.7102604. Cited on page 11.

[67] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017. Publisher: IEEE. Cited on page 11.

[68] Annibale Panichella, José Campos, and Gordon Fraser. Evosuite at the sbst 2020 tool competition. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 549–552. ACM, June 2020. ISBN 978-1-4503-7963-2. doi: 10.1145/3387940.3392266. URL https://dl.acm.org/doi/10.1145/3387940.3392266. Cited on pages 2 and 6.

[69] Anjana Perera, Aldeida Aleti, Marcel Böhme, and Burak Turhan. Defect prediction guided search-based software testing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 448–460, Virtual Event Australia, December 2020. ACM. ISBN 978-1-4503-6768-4. doi: 10.1145/3324884.3416612. URL https://dl.acm.org/doi/10.1145/3324884.3416612. Cited on page 19.

[70] Long H. Pham, Quang Loc Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin. Enhancing symbolic execution of heap-based programs with separation logic for test input generation. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis*, pages 209–227, Cham, 2019. Springer International Publishing. ISBN 978-3-030-31784-3. Cited on page 29.

[71] Xiaoxue Ren, Xinyuan Ye, Yun Lin, Zhenchang Xing, Shuqing Li, and Michael R. Lyu. Api-knowledge aware search-based software testing: Where, what, and how. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1320–1332, San Francisco CA USA, November 2023. ACM. ISBN 9798400703270. doi: 10.1145/3611643.3616269. URL https://dl.acm.org/doi/10.1145/3611643.3616269. Cited on page 23.

[72] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5):366–401, 2016. doi: https://doi.org/10.1002/stvr.1601. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1601. Cited on pages 1, 18, and 67.

[73] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 22: 852–893, 2016. Publisher: Springer. Cited on page 17.

[74] Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. An empirical comparison of evosuite and dspot for improving developer-written test suites with respect to mutation score. In Mike Papadakis and Silvia Regina Vergilio, editors, *Search-Based Software Engineering*, volume 13711, pages 19–34. Springer International Publishing, Cham, 2022. ISBN 978-3-031-21250-5 978-3-031-21251-2. doi: 10.1007/978-3-031-21251-2_2. URL https://link.springer.com/10.1007/978-3-031-21251-2_2. Series Title: Lecture Notes in Computer Science. Cited on pages 14 and 50.

[75] Abdelilah Sakti, Gilles Pesant, and Yann-Gael Gueheneuc. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering*, 41(3):294–313, March 2015. ISSN 0098-5589, 1939-3520. doi: 10.1109/TSE.2014.2363479. URL http://ieeexplore.ieee.org/document/6926828/. Cited on pages 26, 27, and 34.

[76] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *2015 30th IEEE/ACM International Conference on Automated*

*Software Engineering (ASE)*, pages 201–211, 2015. doi: 10.1109/ASE.2015.86. Cited on pages 1, 2, 6, 47, and 49.

[77] Sina Shamshiri, José Miguel Rojas, Luca Gazzola, Gordon Fraser, Phil McMinn, Leonardo Mariani, and Andrea Arcuri. Random or evolutionary search for object-oriented test suite generation? *Software Testing, Verification and Reliability*, 28(4):e1660, June 2018. ISSN 0960-0833, 1099-1689. doi: 10.1002/stvr.1660. URL https://onlinelibrary.wiley.com/doi/10.1002/stvr.1660. Cited on pages 12 and 67.

[78] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sérgio Lopes de Souza. A systematic review on search based mutation testing. *Information and Software Technology*, 81:19–35, 2017. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2016.01.017. URL https://www.sciencedirect.com/science/article/pii/S0950584916300167. Cited on page 8.

[79] Kunal Taneja, Yi Zhang, and Tao Xie. Moda: Automated test generation for database applications via mock objects. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 289–292, 2010. Cited on page 7.

[80] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering*, 50(6):1340–1359, June 2024. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2024.3382365. URL https://ieeexplore.ieee.org/document/10485640/. Cited on pages 23 and 24.

[81] Doan Thi Hoai Thu, Duc-Anh Nguyen, and Pham Ngoc Hung. Automated test data generation for typescript web applications. In *2021 13th International Conference on Knowledge and Systems Engineering (KSE)*, pages 1–6, 2021. doi: 10.1109/KSE53942.2021.9648782. Cited on pages 14 and 36.

[82] Luca Della Toffola, Cristian-Alexandru Staicu, and Michael Pradel. Saying 'hi!' is not enough: Mining inputs for effective test generation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 44–49, Urbana, IL, October 2017. IEEE. ISBN 978-1-5386-2684-9. doi: 10.1109/ASE.2017.8115617. URL http://ieeexplore.ieee.org/document/8115617/. Cited on pages 29 and 30.

[83] Javier Tuya, María José Suárez-Cabal, and Claudio de la Riva. Full predicate coverage for testing sql database queries. *Software Testing, Verification and Reliability*, 20(3):237–288, 2010. doi: https://doi.org/10.1002/stvr.424. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.424. Cited on page 10.

[84] Rachel Tzoref-Brill, Saurabh Sinha, Antonio Abu Nassar, Victoria Goldin, and Haim Kermany. Tackletest: A tool for amplifying test generation via type-based combinatorial coverage. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 444–455, Valencia, Spain, April 2022. IEEE. ISBN 978-1-66546-679-0. doi: 10.1109/ICST53961.2022.00050. URL https://ieeexplore.ieee.org/document/9787840/. Cited on pages 13, 34, and 36.

[85] András Vargha and Harold D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000. doi: 10.3102/10769986025002101. URL https://doi.org/10.3102/10769986025002101. Cited on page 49.

[86] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, Jose Campos, and Annibale Panichella. Evosuite at the sbst 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 28–29. IEEE, May 2021. ISBN 978-1-66544-571-9. doi: 10.1109/SBST52555.2021.00012. URL https://ieeexplore.ieee.org/document/9476230/. Cited on pages 2 and 6.

[87] Chengcheng Wan, Shicheng Liu, Sophie Xie, Yifan Liu, Henry Hoffmann, Michael Maire, and Shan Lu. Automated testing of software that uses machine learning apis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 212–224, Pittsburgh Pennsylvania, May 2022. ACM. ISBN 978-1-4503-9221-1. doi: 10.1145/3510003.3510068. URL https://dl.acm.org/doi/10.1145/3510003.3510068. Cited on pages 26 and 35.

[88] Dietmar Winkler, Pirmin Urbanke, and Rudolf Ramler. Investigating the readability of test code. *Empirical Software Engineering*, 29(2):53, February 2024. ISSN 1573-7616. doi: 10.1007/s10664-023-10390-z. URL https://doi.org/10.1007/s10664-023-10390-z. Cited on pages 1, 33, and 35.

[89] C. Wohlin, P. Runeson, M. Hst, M. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012. ISBN 3642290434. Cited on page 50.

[90] Congying Xu, Valerio Terragni, Hengcheng Zhu, Jiarong Wu, and Shing-Chi Cheung. Mr-scout: Automated synthesis of metamorphic relations from existing test cases. *ACM Transactions on Software Engineering and Methodology*, 33(6):1–28, July 2024. ISSN 1049-331X, 1557-7392. doi: 10.1145/3656340. URL https://dl.acm.org/doi/10.1145/3656340. Cited on page 32.

[91] Xiong Xu, Ziming Zhu, and Li Jiao. An adaptive fitness function based on branch hardness for search based testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 1335–1342, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-4920-8. doi: 10.1145/3071178.3071184. URL https://doi.org/10.1145/3071178.3071184. event-place: Berlin, Germany. Cited on page 13.

[92] S. Yoo and M. Harman. Test data regeneration: generating new test data from existing test data. *Software Testing, Verification and Reliability*, 22(3):171–201, 2012. doi: https://doi.org/10.1002/stvr.435. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.435. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.435. Cited on page 32.

[93] Shayan Zamani and Hadi Hemmati. A pragmatic approach for hyper-parameter tuning in search-based test case generation. *Empirical Softw. Engg.*, 26(6), November 2021. ISSN 1382-3256. doi: 10.1007/s10664-021-10024-2. URL https://doi.org/10.1007/s10664-021-10024-2. Place: USA Publisher: Kluwer Academic Publishers. Cited on pages 15 and 67.

[94] Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, Xiapu Luo, Jingzhu He, and Yutian Tang. Coverage goal selector for combining multiple criteria in search-based unit test generation. *IEEE Transactions on Software Engineering*, 50(4):854–883, April 2024. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2024.3366613. URL https://ieeexplore.ieee.org/document/10438901/. Cited on page 49.

# Appendix A

# *Pool++*

## A.1 Mock structure and organization

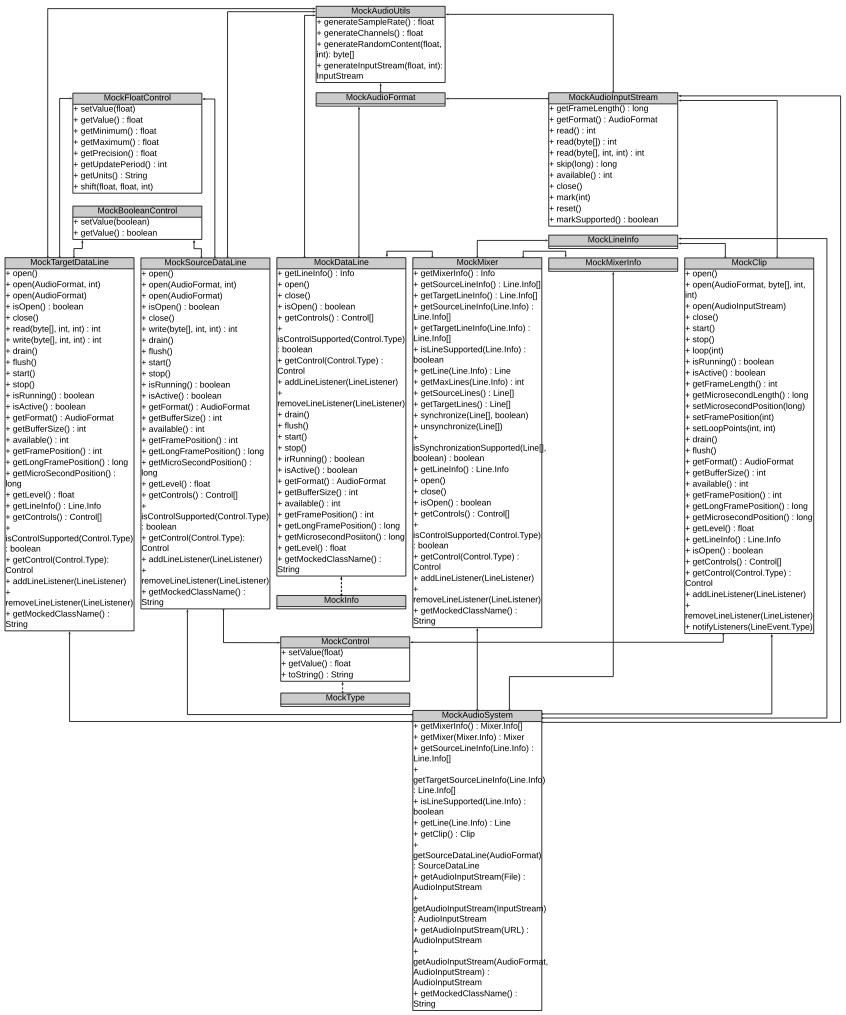


Figure A.1: *Pool++* novel mocks' structure and organization, where the *MockAudioUtils* class provided the necessary data for all the other ones. Cited on page 41.

## A.2 Subject programs

Listing A.1: Code snippet of MP3, used in the exploration phase. Cited on page 37, and on page 47.

```
1  import java.io.*;
2  import javax.sound.sampled.*;
3
4  public class MP3 extends Thread
5  {
6   AudioInputStream in = null;
7   AudioInputStream din = null;
8   String filename = "";
9   public MP3(String filename)
10  {
11   this.filename = filename;
12   this.start();
13  }
14  public void run()
15  {
16   AudioInputStream din = null;
17   try {
18    File file = new File(filename);
19    AudioInputStream in = AudioSystem.getAudioInputStream(file);
20    AudioFormat baseFormat = in.getFormat();
21    AudioFormat decodedFormat = new AudioFormat(
22       AudioFormat.Encoding.PCM_SIGNED,
23       baseFormat.getSampleRate(), 16, baseFormat.getChannels(),
24       baseFormat.getChannels() * 2, baseFormat.getSampleRate(),
25       false);
26    din = AudioSystem.getAudioInputStream(decodedFormat, in);
27    DataLine.Info info = new DataLine.Info(SourceDataLine.class, decodedFormat);
28    SourceDataLine line = (SourceDataLine) AudioSystem.getLine(info);
29    if(line != null) {
30
31     line.open(decodedFormat);
32     FloatControl volumeControl = (FloatControl) line
33           .getControl(FloatControl.Type.MASTER_GAIN);
34     volumeControl.setValue(-20);
35     byte[] data = new byte[4096];
36     // Start
37     line.start();
38
39     int nBytesRead;
40     while ((nBytesRead = din.read(data, 0, data.length)) != -1) {
41      line.write(data, 0, nBytesRead);
42     }
43     // Stop
44     line.drain();
45     line.stop();
46     line.close();
47     din.close();
48    }
49
50   }
51   catch(Exception e) {
52    e.printStackTrace();
53   }
```

```
54      finally {
55        if(din != null) {
56          try { din.close(); } catch(IOException e) { }
57        }
58      }
59    }
60  }
```

Listing A.2: Code snippet of `SoundPlayer`, with the modifications commited. The lines starting with red (-) corresponds to removal of code, while the ones start with green (+) are added lines. Cited on page 48.

```
1  package org.pdfsam.guiclient.commons.business;
2
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  import javax.sound.sampled.AudioInputStream;
7  import javax.sound.sampled.AudioSystem;
8  import javax.sound.sampled.Clip;
9  import javax.sound.sampled.DataLine;
10
11 - import org.apache.log4j.Logger;
12 - import org.pdfsam.guiclient.configuration.Configuration;
13 - import org.pdfsam.i18n.GettextResource;
14
15 /**
16  * Plays sounds
17  *
18  * @author Andrea Vacondio
19  *
20  */
21 public class SoundPlayer {
22
23  - private static final Logger log =
   Logger.getLogger(SoundPlayer.class.getPackage().getName());
24    private static final String SOUND = "/resources/sounds/ok_sound.wav";
25    private static final String ERROR_SOUND = "/resources/sounds/error_sound.wav";
26
27    private static SoundPlayer player = null;
28
29    private Clip errorClip;
30
31    private Clip soundClip;
32
33    private ExecutorService executor;
34
35    private SoundPlayer() {
36       executor = Executors.newSingleThreadExecutor();
37    }
38
39    public static synchronized SoundPlayer getInstance() {
40       if (player == null) {
41          player = new SoundPlayer();
42       }
43       return player;
44    }
45
```

```
46      /**
47       * Plays an error sound
48       */
49      public void playErrorSound() {
50        - if (Configuration.getInstance().isPlaySounds()) {
51            try {
52               if (errorClip == null) {
53                  AudioInputStream sound =
    AudioSystem.getAudioInputStream(this.getClass().getResource(ERROR_SOUND));
54                  DataLine.Info info = new DataLine.Info(Clip.class, sound.getFormat());
55                  errorClip = (Clip) AudioSystem.getLine(info);
56                  errorClip.open(sound);
57               }
58               executor.execute(new PlayThread(errorClip));
59            } catch (Exception e) {
60                 + e.printStackTrace();
61                 - log.warn(GettextResource.gettext(
62                 - Configuration.getInstance().getI18nResourceBundle(),
63                 - "Error playing sound") + ": " + e.getMessage());
64            }
65         }
66      }
67
68      /**
69       * Plays a sound
70       */
71      public void playSound() {
72          - if (Configuration.getInstance().isPlaySounds()) {
73               try {
74                  if (soundClip == null) {
75                     AudioInputStream sound = AudioSystem.getAudioInputStream(this.getClass()
76                     .getResource(SOUND));
77                     DataLine.Info info = new DataLine.Info(Clip.class, sound.getFormat());
78                     soundClip = (Clip) AudioSystem.getLine(info);
79                     soundClip.open(sound);
80                  }
81                  executor.execute(new PlayThread(soundClip));
82               } catch (Exception e) {
83                 + e.printStackTrace();
84                 - log.warn(GettextResource.gettext(
85                 - Configuration.getInstance().getI18nResourceBundle(),
86                 - "Error playing sound") + ":" + e.getMessage());
87                }
88            - }
89      }
90
91      /**
92       * Plays the sound
93       *
94       * @author Andrea Vacondio
95       *
96       */
97      private class PlayThread extends Thread {
98         private Clip clip;
99
100        /**
101         * @param clip
```

```
102         */
103        public PlayThread(Clip clip) {
104           this.clip = clip;
105        }
106
107        public void run() {
108           try {
109              clip.setFramePosition(0);
110              clip.stop();
111              clip.start();
112           } catch (Exception e) {
113             + e.printStackTrace();
114             - log.error(GettextResource.gettext(
115             - Configuration.getInstance().getI18nResourceBundle(),
116             - "Error playing sound"), e);
117           }
118        }
119     }
120  }
```

## A.3 Empirical Study

Table A.1: Results obtained from the experiments using class `MP3`, with the coverage scores, number of generations, and the size of the tests. Cited on page 51.

| Version | TARGET_CLASS | Seed | Generations | Size | Length | Total_Branches | Lines | BranchCoverage | LineCoverage | BranchCoverageBitString | LineCoverageBitString | Execution_time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pool++ | MP3 | 0 | 94 | 2 | 5 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 4144 |
| vanilla | MP3 | 0 | 3001 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 811 |
| pool++ | MP3 | 1 | 104 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 934 |
| vanilla | MP3 | 1 | 3220 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 869 |
| pool++ | MP3 | 2 | 88 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 3988 |
| vanilla | MP3 | 2 | 3248 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 327 |
| pool++ | MP3 | 3 | 80 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 4027 |
| vanilla | MP3 | 3 | 2981 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 778 |
| pool++ | MP3 | 4 | 105 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 895 |
| vanilla | MP3 | 4 | 2735 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 915 |
| pool++ | MP3 | 5 | 92 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 868 |
| vanilla | MP3 | 5 | 3012 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 498 |
| pool++ | MP3 | 6 | 167 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 4098 |
| vanilla | MP3 | 6 | 2908 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 775 |
| pool++ | MP3 | 7 | 94 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 816 |
| vanilla | MP3 | 7 | 3078 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 855 |
| pool++ | MP3 | 8 | 111 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 241 |
| vanilla | MP3 | 8 | 3182 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 227 |
| pool++ | MP3 | 9 | 74 | 2 | 5 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 816 |
| vanilla | MP3 | 9 | 2772 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 234 |
| pool++ | MP3 | 10 | 111 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 861 |
| vanilla | MP3 | 10 | 3564 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 795 |
| pool++ | MP3 | 11 | 102 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 3998 |
| vanilla | MP3 | 11 | 3533 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 872 |
| pool++ | MP3 | 12 | 92 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 3486 |
| vanilla | MP3 | 12 | 1243 | 5 | 18 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 6777 |
| pool++ | MP3 | 13 | 93 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 236 |
| vanilla | MP3 | 13 | 2846 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 967 |
| pool++ | MP3 | 14 | 124 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 853 |
| vanilla | MP3 | 14 | 3019 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 891 |
| pool++ | MP3 | 15 | 124 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 837 |
| vanilla | MP3 | 15 | 2799 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 244 |
| pool++ | MP3 | 16 | 94 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 898 |
| vanilla | MP3 | 16 | 3219 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 869 |
| pool++ | MP3 | 17 | 98 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 832 |
| vanilla | MP3 | 17 | 2962 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 804 |
| pool++ | MP3 | 18 | 89 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 1008 |
| vanilla | MP3 | 18 | 2876 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 297 |
| pool++ | MP3 | 19 | 97 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 969 |
| vanilla | MP3 | 19 | 3151 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 321 |
| pool++ | MP3 | 20 | 106 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 835 |
| vanilla | MP3 | 20 | 3352 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 922 |
| pool++ | MP3 | 21 | 78 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 952 |
| vanilla | MP3 | 21 | 3156 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 851 |
| pool++ | MP3 | 22 | 74 | 2 | 5 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 4046 |
| vanilla | MP3 | 22 | 2903 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 864 |
| pool++ | MP3 | 23 | 66 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 4048 |
| vanilla | MP3 | 23 | 2854 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 232 |
| pool++ | MP3 | 24 | 83 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 284 |
| vanilla | MP3 | 24 | 2600 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 1011 |
| pool++ | MP3 | 25 | 126 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 856 |
| vanilla | MP3 | 25 | 2742 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 246 |
| pool++ | MP3 | 26 | 104 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 4110 |
| vanilla | MP3 | 26 | 2790 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 756 |
| pool++ | MP3 | 27 | 92 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 844 |
| vanilla | MP3 | 27 | 2554 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 804 |
| pool++ | MP3 | 28 | 88 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 894 |
| vanilla | MP3 | 28 | 3170 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 836 |
| pool++ | MP3 | 29 | 87 | 2 | 4 | 10 | 2 | 0.5454545454545454 | 0.9714285714285714 | 10110110000 | 11111111111111111111111111111101 | 890 |
| vanilla | MP3 | 29 | 2971 | 1 | 2 | 10 | 2 | 0.18181818181818183 | 0.4000000000000000 | 10000001000 | 11111111110000000000000000000111001 | 238 |

Table A.2: Results obtained from the experiments using class `SoundPlayer`, with the coverage scores, number of generations, and the size of the tests. Cited on page 51.

| Version | Target Class | Seed | Generations | Size | Length | Total Branches | Lines | BranchCoverage | LineCoverage | BranchCoverageBitString | LineCoverageBitString | Execution time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pool++ | SoundPlayer | 0 | 7 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4003 |
| vanilla | SoundPlayer | 0 | 7125 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 818 |
| pool++ | SoundPlayer | 1 | 2 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4047 |
| vanilla | SoundPlayer | 1 | 7025 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 941 |
| pool++ | SoundPlayer | 2 | 3 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 7151 |
| vanilla | SoundPlayer | 2 | 7091 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 929 |
| pool++ | SoundPlayer | 3 | 2 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4106 |
| vanilla | SoundPlayer | 3 | 6980 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 985 |
| pool++ | SoundPlayer | 4 | 4 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4140 |
| vanilla | SoundPlayer | 4 | 7066 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 793 |
| pool++ | SoundPlayer | 5 | 6 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4150 |
| vanilla | SoundPlayer | 5 | 7049 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 845 |
| pool++ | SoundPlayer | 6 | 5 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4091 |
| vanilla | SoundPlayer | 6 | 7207 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 813 |
| pool++ | SoundPlayer | 7 | 2 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4111 |
| vanilla | SoundPlayer | 7 | 7080 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 829 |
| pool++ | SoundPlayer | 8 | 9 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 947 |
| vanilla | SoundPlayer | 8 | 7099 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 909 |
| pool++ | SoundPlayer | 9 | 4 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4074 |
| vanilla | SoundPlayer | 9 | 6990 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 788 |
| pool++ | SoundPlayer | 10 | 3 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4001 |
| vanilla | SoundPlayer | 10 | 6985 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 875 |
| pool++ | SoundPlayer | 11 | 5 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4146 |
| vanilla | SoundPlayer | 11 | 6915 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 880 |
| pool++ | SoundPlayer | 12 | 3 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 7028 |
| vanilla | SoundPlayer | 12 | 7244 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 809 |
| pool++ | SoundPlayer | 13 | 3 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 969 |
| vanilla | SoundPlayer | 13 | 6892 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 805 |
| pool++ | SoundPlayer | 14 | 4 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4012 |
| vanilla | SoundPlayer | 14 | 7116 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 836 |
| pool++ | SoundPlayer | 15 | 3 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4094 |
| vanilla | SoundPlayer | 15 | 7082 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 777 |
| pool++ | SoundPlayer | 16 | 8 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4059 |
| vanilla | SoundPlayer | 16 | 6998 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 886 |
| pool++ | SoundPlayer | 17 | 4 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4103 |
| vanilla | SoundPlayer | 17 | 7060 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 829 |
| pool++ | SoundPlayer | 18 | 6 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 7097 |
| vanilla | SoundPlayer | 18 | 6996 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 877 |
| pool++ | SoundPlayer | 19 | 5 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 6970 |
| vanilla | SoundPlayer | 19 | 7289 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 916 |
| pool++ | SoundPlayer | 20 | 5 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4160 |
| vanilla | SoundPlayer | 20 | 7068 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 932 |
| pool++ | SoundPlayer | 21 | 4 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4107 |
| vanilla | SoundPlayer | 21 | 7071 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 844 |
| pool++ | SoundPlayer | 22 | 3 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4151 |
| vanilla | SoundPlayer | 22 | 7048 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 748 |
| pool++ | SoundPlayer | 23 | 5 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 7016 |
| vanilla | SoundPlayer | 23 | 6981 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 755 |
| pool++ | SoundPlayer | 24 | 3 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 6967 |
| vanilla | SoundPlayer | 24 | 7003 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 823 |
| pool++ | SoundPlayer | 25 | 6 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4034 |
| vanilla | SoundPlayer | 25 | 7087 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 880 |
| pool++ | SoundPlayer | 26 | 2 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4173 |
| vanilla | SoundPlayer | 26 | 7116 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 817 |
| pool++ | SoundPlayer | 27 | 3 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4073 |
| vanilla | SoundPlayer | 27 | 6954 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 883 |
| pool++ | SoundPlayer | 28 | 3 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 3982 |
| vanilla | SoundPlayer | 28 | 6934 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 893 |
| pool++ | SoundPlayer | 29 | 5 | 3 | 8 | 6 | 6 | 0.8750000000000000 | 0.8055555555555556 | 10111111 | 11111111111111111111111111110000000 | 4037 |
| vanilla | SoundPlayer | 29 | 7014 | 3 | 6 | 6 | 6 | 0.5000000000000000 | 0.4444444444444444 | 00110101 | 11111111000011011100001101000000000 | 827 |

## A.4 Future Work

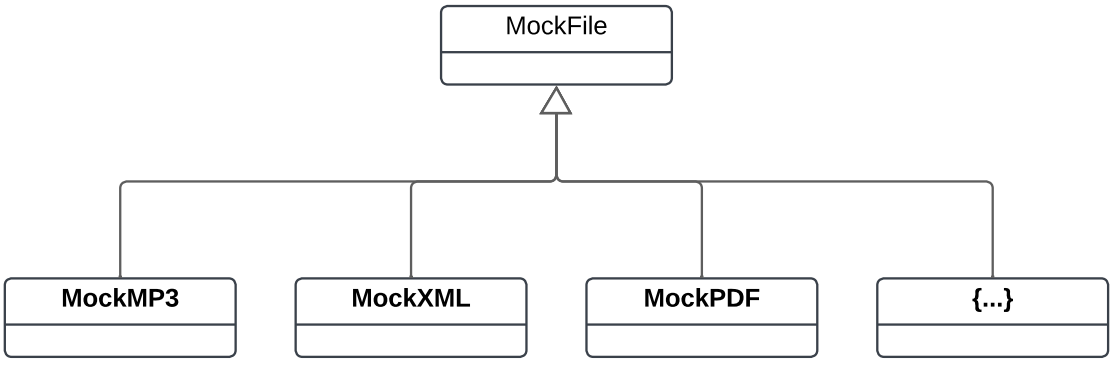


Figure A.2: Possible architecture's for the future `MockFile` extension idea. Cited on page 69.