SICPA SA
Department of Digital Research & Innovation
At the Platform of Trust - Foundations Service

and

Ecole Polytechnique Fédérale de Lausanne
Department of Computer Sciences
At the HexHive Lab

Automatic generation of mock functions for unit tests

by Julien DI TRIA

Supervisor	Michal KAPALKA
Manager	Damien SNOECK
Professor	Mathias PAYER

Submitted in partial fulfilment of the requirements for the degree of

Master of Science

April 19, 2024

Abstract

Testing of software is an important part of software development. It helps ensure that the software behaves as expected and has as few bugs as possible, as bugs could lead to a crash of the software or security issues. Testing is also used to increase confidence in the correctness of the software. Users are more comfortable using a library that is properly tested than using one that has no tests. Testing regularly allows the detection of potential issues faster in the process and helps ensure that previous features are still working while other parts of the software are under development, which is called regression testing.

Automatically generating test scenarios could make testing more efficient, more useful, and less straining on developers. It could reduce the time needed to create tests and increase the coverage by testing corner cases while providing a maintainable test suite. This in turn would allow developers to keep focusing on developing the software features while knowing that their code is being tested systematically. By assuming that the current state of the program is the desired behaviour, one could generate a regression test suite to capture the working features. Any changes to the program that would break the regression test would show a behaviour change. Multiple tools that can generate tests already exist (e.g. Randoop and EvoSuite for Java, IntelliTest for C#, UTBotCpp for C/C++) and can be improved in many ways.

In this thesis, the EvoSuite tool is augmented to handle the generation of unit tests for Java applications that are using the Spring framework, mainly focusing on the Spring Web MVC module. A unit test is a test of the smallest functional unit of code of a program. The work focuses on generating unit tests for Web controllers that require Mock MVC, a test framework for Spring Web. To correctly execute with Mock MVC, preloading the Spring context before the virtual test execution phase of EvoSuite is required so that the autowired injected mockMVC object can be used and propagate the Web request from the test to the actual controller.

The work contribution is the modification of the tool EvoSuite to detect Spring MVC controllers and allow EvoSuite to generate tests using the Spring Mock MVC framework. A new base class called DeclarationStatement was added to support Java object injection in test cases, and the automatic guided generation of Spring MockMVC calls was implemented to reach faster the methods of the controller. As EvoSuite is a generic tool, there is no support for Spring itself, and the work done is therefore made as generic as possible to include proper support for Spring while not reducing EvoSuite's original capabilities.

Acknowledgements

I would like to thank all the people who made this project achievable. Firstly, I extend my sincere appreciation to SICPA SA for providing me with the opportunity to do applied research in the company, as well as to my professor Mathias Payer, who followed the progress all along. Special thanks to Damien Snoeck, my manager, who provided me guidance in times of doubt, Adrian Baudat and Michal Kapalka, my supervisors, whose guidance and expertise were more than appreciated during this work. I am also grateful to the entire Platform of Trust team at SICPA, especially Loïc for his wit moments, Dahn for his joy, Smiljka and Luca for the help on Spring, and finally Colin for the coffee.

Contents

List of Figures

Listings

Acronyms

CBT Constraint-Based Testing. 7, 8, 10, 15

CFG Control Flow Graph. 6–8

CI Continuous Integration. 1

CUT Class Under Test. 3, 4, 6–8, 10, 11, 14–16, 21, 22, 25, 29, 30, 38, 47, 51–53

DI Dependency Injection. 9, 18, 26, 27, 42, 54

DSE Dynamic Symbolic Execution. 7, 10, 13, 15, 32

E2E End-to-End. 53

GA Genetic Algorithm. 6, 13, 15, 16

HTTP Hypertext Transfer Protocol. 9, 20, 22, 32

IoC Inversion of Control. 9, 18, 19, 26

JAR Java ARchive. iv, 45, 46, 53, 55

JDK Java Development Kit. 11

JEE Java Entreprise Edition. 11

JPA Jakarta Persistence API. 18

JVM Java Virtual Machine. 9, 19

LLM Large Language Model. 8

LTS Long Term Support. 9

MVC Model View Controller. 9, 10

NPE Null-Pointer Exception. 44

OCG Object Construction Graph. 13, 14, 16, 17, 19, 22, 31

OOP Object Oriented Programming. 9

PoC Proof of Concept. 32, 54

SBST Search-Based Software Testing. 5–8, 10, 13, 15

SE Symbolic Execution. 7, 8, 32

SMockMvc Simplified MockMvc. 26

SUT Software Under Test. 1, 2, 4, 11, 28, 29, 35, 39, 55

TDD Test Driven Development. 10

UI User Interface. 26

URI Uniform Resource Identifier. 32

URL Uniform Resource Locator. 32

UUT Unit Under Test. 2

VFS Virtual File System. 10

VM Virtual Machine. 15, 16, 23, 28, 35, 37, 38, 51

VNET Virtual Network. 10

Glossary

@Autowired Spring annotation to "automatically wire the required beans (dependencies) into your classes, eliminating the need for manual configuration" [3]. 26, 27, 35, 38, 39, 54

@MockBean Spring annotation "that can be used to add mocks to a Spring ApplicationContext" [4]. 26, 27, 37–39, 54

@PathVariable Spring annotation "which indicates that a method parameter should be bound to a URI template variable" [5]. 32

@RequestMapping Spring annotation "for mapping web requests onto methods in request-handling classes with flexible method signatures." [6]. 28

@RunWith JUnit annotation "used to tell JUnit to run the test using a specific test runner class instead of its default runner" [7]. 19, 26

@WebMvcTest Spring annotation "that can be used for a Spring MVC test that focuses only on Spring MVC components." [8]. 26

bean Java object instantiated by Spring in the context of Dependency Injection. "A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container." [9]. 9, 18, 26

Chromosome EvoSuite abstraction to represent a TestCase or a TestSuite which can be executed and evolved to improve its fitness value. EvoSuite uses a lot the concept of Genetic Algorithm to evolve the TestSuite, and the term chromosome comes from there.. 30

DeclarationStatement A new statement added to EvoSuite statements, composing a TestCases. It declares a variable of some type, holding a variable reference, similar to an assignment, but with no value. It allows to do injection easily.. 34, 35

downcast Downcasting means typecasting of a parent object to a child object, to regain the ability to call methods that are specific to the child. Downcasting cannot be implicit. 37

HandlerMethod "Encapsulates information about a handler method consisting of a method and a bean. Provides convenient access to method parameters, the method return value, method annotations, etc." [10]. 28

JUnit "JUnit is a unit testing framework for the Java programming language" [11, 12]. 28

MockMvc "The Spring MVC Test framework, also known as MockMvc, provides support for testing Spring MVC applications. It performs full Spring MVC request handling but via mock request and response objects instead of a running server." [13]. 10, 26–28

Repository "The goal of the repository abstraction of Spring Data is to reduce the effort to implement data access layers for various persistence stores significantly" [14]. 27

SAT Boolean satisfiability problem, or SATisfiability, is the computer science problem of "determining if there exists an interpretation that satisfies a given Boolean formula" [15]. It is the first known NP-complete problem [16]. 7

Todo list

1 Introduction

1.1 Motivations

Testing of software is an important part of software development. It helps ensure that the software behaves as expected and has as few bugs as possible, as bugs could lead to a crash of the software or security issues. The correctness of software can be certified using formal verification methods that apply mathematical models to prove that a certain property holds in all possible cases of a program. As complexity grows, the specification required for formal verification becomes too complicated, and applying proper proof becomes unsound. Testing is therefore used to increase confidence in the correctness of the software. Users are more comfortable using a library that is properly tested than using one that has no tests. Testing regularly allows the detection of potential issues faster in the process and helps ensure that previous features are still working while other parts of the software are under development, which is called regression testing.

To achieve this, tests are written alongside the program, maintained up to date all together, and executed regularly, on multiple platforms if possible, by developers or Continuous Integration (CI) servers. A modification of the program should not break previously written tests. Due to the complexity of software, it is not possible to completely test a program to ensure correctness [17]. Testing will never prove that no bugs exist in the program, it might only reveal that some exist. Even with a lot of tests, only the reliability of the software increases, but not necessarily its correctness. Writing tests is also time-consuming, such that testing usually implies a tradeoff between cost, time and quality level. "Typically, more than 50% of the development time is spent in testing" [17].

Automatically generating test scenarios could make testing more efficient, more useful, and less straining on developers. It would reduce the time needed to create tests and increase the coverage by testing corner cases while providing a maintainable test suite, allowing developers to keep focusing on developing the software features while knowing that their code is being tested systematically. By assuming that the current state of the program is the desired behaviour, one could generate a regression test suite to capture the working features. Any changes to the program that would break a regression test would show a behaviour change. Multiple tools that can generate tests already exist (e.g. Randoop and EvoSuite for Java, IntelliTest for C#, UTBotCpp for C/C++) and can be improved in many ways (see subsection 2.3). Two recurrent problems with the automatic generation of tests are the "choice of next call" and the "input data value". The choice of the next call to add to a test can make a test useful or wasted. For example, calling `i = i - 1` right after `i = i + 1` might be a waste of the two calls, unless changing `i` triggers events. A poor choice of the next call for a test could also lead to a non-compilable test, for example calling `file.open()` before creating the `file`, wasting the time to generate that test. A poor choice of input value (for example, choosing 2^{63} as an end-of-loop counter) can lead to a very long test time, wasting execution time.

One improvement needed concerns the generation of mock objects and function stubs. Mocking is sometimes required during the process of unit testing. A unit test is a test of the smallest functional unit of code of a program, in isolation from the other components. Usually, a small functional unit is a function or procedure, and therefore each unit test aims to test one specific function. Each function can have multiple associated unit tests to cover different paths of the function. Flaky calls (API, time-dependent) or external system (database, filesystem) interactions need to be mocked to keep the Software Under Test (SUT) isolated from the other components in a controlled environment. While flaky calls could be detected by executing multiple times the same functions over a time span, mocking external system interactions can be quite tricky for the testers as there are many factors to take into account [18]. Mocks are often used during tests, between 23% and 30% for Java projects [18, 19]. Having auto-generated tests using mocks would greatly improve the number of systems that could be tested.

In order to automatically generate tests for more cases, support for mocking should be added in the process of test generation. The problem is:

How to automatically detect the need for mocking and implement the stub functions while generating unit tests?

Answering this question would allow the automation of the generation of mocks and stub functions when needed, allowing the test generation tool to create tests for more complex software, and increasing the percentage of code covered by the tests.

The goal of this research is to improve the automatic generation of unit test for software requiring mocking, specifically in the context of Spring Web MVC Java application (see subsection 2.2.2 for details on Spring). While basic mocking support exists in some tools, specific support for Spring MVC has not been found, and the current status would be improved by extending the automatic test generation to Spring, which is widely used in industry, improving test generation for more complex software in general.

1.2 Challenges

Numerous challenges appear when dealing with auto-generated tests and mocks. In simple cases, as in Listing 1, tools can process the code and generate tests for straightforward functions or methods quite easily. However, when complex logic comes into place, like in Listing 2, the tool needs to start understanding the code's behaviour to write meaningful tests. This is the case when conditional loops, dependencies or complex if-else switches are used. Increasing the coverage of programs with complex logic can be quite hard, as reaching all the different paths necessitates figuring out all the conditions for reaching that path. Additionally, supporting different frameworks and libraries is not an easy task as the logic might be completely obscured and hidden over multiple layers of complexity.

Mocking allows for avoiding interactions with external systems. However, the correct interaction flow needs to be learned first in order to be mocked correctly, and the injected values to generate meaningful mocking tests are hard to extract. This requires an understanding of the SUT in much depth, usually meaning that the test-generation tool needs to be tailored to the SUT, preventing the creation of a generic tool. In complex scenarios, it is hard even for developers to correctly write tests using a mocking framework. In such cases, they usually use custom-made support libraries to ease the usage of the mocking and other external systems.

Mocking should properly isolate the Unit Under Test (UUT) from its dependencies. For this, it requires a proper configuration which can be complex, especially when dealing with dynamic behaviours. Setting up a flexible yet effective configuration of the mocking library is non-trivial, as it can also lead to side effects. When the mocking library might have state variables, it becomes necessary to make sure that the mocks behave in a predictable and consistent manner across multiple tests.

An example of where mocking could be useful is given in Listing 3, where the `paintReceptDatabase` or the `paintingMachine.paint` call can be mocked to not depend on an external database or physical machines nor on unreliable connection or actual implementation of the painting functionality as the behaviour can be tested without actually painting a Car. One challenge here is that there is no "a priori" need to mock the `paintingMachine` by looking at the code only (without the comment). Determining that this must be mocked in an automatic way requires a deeper understanding of the SUT.

Listing 1: Example of low complexity code

```
boolean testParity(int i) {
    if(i mod 2 == 0) {
        return true;
    } else {
        return false;
    }
}
```

Listing 2: Example of higher complexity code (simplified from [2])

```
protected List<TestChromosome> breedNextGeneration() {
    List<TestChromosome> offspringPopulation = new ArrayList<>(POPULATION);
    for (int i = 0; i < POPULATION / 2 && !this.isFinished(); i++) {
        TestChromosome parent1 = this.selectionFunction.select(this.population);
        TestChromosome parent2 = this.selectionFunction.select(this.population);
        TestChromosome offspring1 = parent1.clone();
        TestChromosome offspring2 = parent2.clone();
        if (Randomness.nextDouble() ≤ Properties.CROSSOVER_RATE) {
            this.crossoverFunction.crossOver(offspring1, offspring2);
        }

        this.removeUnusedVariables(offspring1);
        this.removeUnusedVariables(offspring2);

        this.mutate(offspring1, parent1);
        this.mutate(offspring2, parent2);

        offspringPopulation.add(offspring1);
        offspringPopulation.add(offspring2);
    }
    return offspringPopulation;
}
```

Listing 3: Example of mocks possibility

```
Color paintObject(int colorId, Object objectToPaint) {
    // database access over unreliable bluetooth connection
    Paint paint = paintRecipeDatabase.get(colorId);
    if (objectToPaint isInstanceOf Car){
        // takes 10 min to actually paint
        Color painted = paintingMachine.paint(paint, (Car)objectToPaint);
    } else {
        throw new IllegalArgumentException("Only Car object can be painted");
    }
    return painted;
}
```

payer: can you highlight and enumerate the challenges, e.g., challenge 1, challenge 2 ... and then refer back to them later on in the design?

In summary, a couple of challenges can be identified when generating tests with mocking.

1.2.1 Dependency identification

Accurately finding and choosing the dependencies to mock requires an in-depth analysis of the Class Under Test (CUT) to understand whether a dependency should be mocked or not.

1.2.2 Mocking behaviour

Determining how the mock object should behave and interact with other components of the system to correctly behave when integrating with it is crucial for a proper mock, otherwise, it might be completely useless.

1.2.3 Mocking configuration

Providing a flexible configuration of the mock objects so that they can be used in multiple scenarios when auto-generating them is not a trivial task.

1.2.4 Isolation and side effects

Ensuring consistent behaviour across tests is a must for auto-generated mocks, as breaking encapsulation could have a massive impact during test generation due to inconsistent test execution.

1.3 Research objectives

This project aims at improving the mocking integration into the process of automatic test generation. While basic support already exists, handling more frameworks that require mocking will increase the usage of existing tools and reduce the workload on developers.

One important point to note here is that SUT is assumed to be correct, so that auto-generated tests can be used to capture the behaviour for a certain version, and can be used as regression tests for the next versions. Detecting that the SUT does or does not behave according to the specification is out of the scope of this project, as it requires another set of analysis tools and prerequisites to match the specification to the behaviour. Finding bugs while auto-generating tests is merely a side effect, as those bugs appear while executing some corner cases that were not tested before. However, it is more likely that those bugs will be considered "desired behaviour" and tests triggering those bugs will be wrapped as "test expected to fail". Manually going through the tests is required afterwards to categorize the normal behaviour or actual bugs. The example in Listing 4 shows a code creating a file reader to read a file that does not exist. This is expected to fail or throw an error which should be handled. The test generation will capture that failure as a desired behaviour.

The main objective is therefore to increase the coverage of a CUT, as this will better capture the current state and behaviour of the program. The project focuses on increasing the coverage of cases where mocking is required in Java applications, by detecting when it is needed and implementing the integration correctly.

Listing 4: Example of expected failure

```java
String filename = "notExistingFile.txt";
BufferedReader reader = new BufferedReader(new FileReader(filename)); // ...
    throws an Exception
String currentLine = reader.readLine();
reader.close();
```

1.4 Statement of Originality

The work presented in this project is based on the research done by the author during his time at the Department of Digital Research & Innovation at SICPA SA and the Department of Computer Sciences at EPFL. It is the author's own work, and the work of other authors included in this project is referenced and cited accordingly. No part of this work was used or submitted to another place to obtain a degree.

2 State of the Art

In this section, we present an overview of existing technologies that are already available and closely related to the project. A study of the current state of the art in the domain of unit test generation is made, gathering information about what already exists, what is missing, what is still evolving, and what is currently being developed. Details about relevant tools and technologies are given.

2.1 Unit test generation techniques

To survey the field, we have explored 17 journal articles, 30 conference papers, 6 project reports and thesis, and about 25 software tools, plugins, web pages and forum posts, to gather base knowledge about **unit test generation**, dating from 1976 (Automatic Generation of Floating-Point Test Data [20]) to 2023 (How well does LLM generate security tests? [21]). The following Figure 1 is a histogram of the documents found by year.

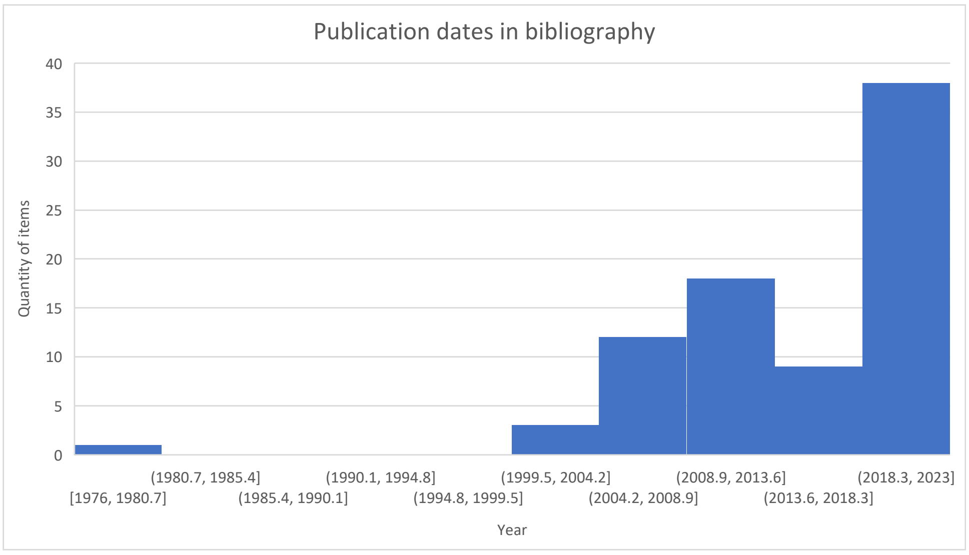


Figure 1: Histogram of publication year of the bibliography

The search focuses on "Unit test generation", "Search-Based Software Testing", "automatic test generation", "random testing", "AI test generation", "LLM test generation" to gather general information.

Different ways to do unit test generation were found, and the following sections detail the relevant methods.

2.1.1 Random testing

A definition is given by [22]: "Random testing simply executes the program with random inputs and then observes the program structures executed." This works by choosing random input (which can be data or program calls) and executing the new program. The idea is to let randomness generate a sequence that will lead to unexplored regions of the source code. While it works well for simple programs, structures that have a low probability of being reached are often not covered by random testing. This original definition transformed into fuzzing overtime: "an automatic testing technique that covers numerous boundary cases using invalid data [...] as application input to better ensure the absence of exploitable vulnerabilities" [23, 24]. Fuzzing mainly focuses on finding input data that can lead to a crash of the application, randomly, or with some guidance or mutation of the input data seed.

In this project, it is considered that fuzzing does not generate test cases, but only "data input" provided to the developer test case in order to detect bugs in the program. The developer has to write the call to the program functions. In comparison, random testing generates the whole test case (object instantiation, method call sequence, input data values to those methods and objects). Sequences of statements are selected randomly, with random input data, and incrementally built and mutated (sometimes guided mutation) usually based on execution feedback of the test case, until the test covers some desired goals [25].

The main idea is to create a statement by randomly selecting a call and randomly selecting its parameters' values. Once the statement is created, it is added to an existing sequence of statements, and executed. If the execution is satisfying, i.e. compiling and actually executable, the test is kept as a seed to be built upon. This process is repeated indefinitely until some criterion (time in the most simple case) is reached.

Due to their low setup cost and great autonomy, random testing and fuzzing are used to generate a lot of different inputs to reach corner cases that could trigger bugs, crashes, or any other unhandled behaviour [26, 27, 28]. Still, pure random testing is known to provide poor code coverage [29]. For example, a simple branch like `if(x==5){...}` is unlikely to be reached by random choice of `x` amongst its 2^{32} options if x is an integer.

In order to improve the poor coverage of the fully randomly generated tests, random testing is usually augmented with some search capability [29], dynamic analysis [30], execution feedback [25], or with method-order recommendation [26] to better choose the next call to generate in the tests.

2.1.2 Search-Based Software Testing

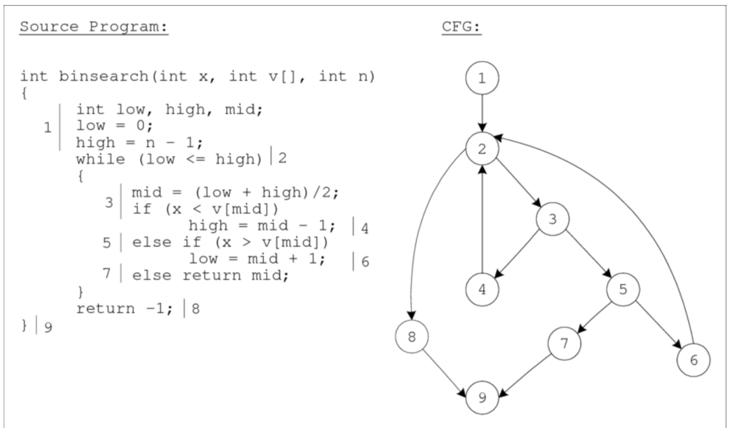
In Search-Based Software Testing (SBST), the problem of generating a test case is transformed into an optimisation problem. The solution is searched by evolving tests in order to optimise against a fitness function expressing how good the test suite is. A fitness function is a function which takes a candidate object to evaluate and quickly returns a score of that candidate, representing how good that candidate is with respect to the problem that we are currently solving. For example, if some quadcopters were getting evolved to transport goods using Genetic Algorithm (GA), a fitness function could be the flight duration of that drone, so a drone with a flight duration of one hour would have a better fitness value than a drone with 10 minutes. Another fitness function could be the payload capacity of the drone. Finding the combination of fitness functions that better represents the problem can be a research topic in itself.

Using GA, Hill Climbing or Simulated Annealing, the tests (methods and input data) are evolved, until the hopefully global maximum of the fitness function is reached. The main problem is defining an efficient fitness function, which is critical to solving an optimisation problem in a short time-frame[31].

In software testing, the fitness is usually chosen to represent the coverage of the CUT by the test, because increasing the coverage means that the CUT is better tested. It can be the coverage percentage in terms of lines, methods, or branches, usually found by instrumenting the CUT to track which part is executed. A branch of code, in this context, is a node of the Control Flow Graph (CFG). The CFG is "a representation, using graph notation, of all paths that might be traversed through a program during its execution" [32, 33]. It represents the different cases of execution of a program and can be used to know whether all parts of the program have been tested. One example is shown in Figure 2.

One fitness function, call *approach-level*, is used to measure how close the execution is to reaching a branch. For example, let's try to reach the branch number 6 in the Figure 2. If the execution goes 1-2-8-9, then it was not close to reaching node 6. If the execution is 1-2-3-5-7-9, it was closer to node 6. The actual computation of the *approach-level* is more complex [34].

Another fitness function called *branch-distance*, evaluates how close the test is to covering a single goal or branch of the CUT. In its basic form, this distance to the branch estimates whether the condition to reach the branch is satisfied or not from the previous branch. If satisfied, the value is 1, otherwise 0, which is similar to either the branch is executed (1), or it is not (0). This makes it difficult to know whether a solution is close to that branch or not because there is no continuous gradient in between 0 and 1. This is referred to as the plateau problem in the search landscape, where neighbours of a solution lead to the same fitness value [35]. The predicate of the branch can be converted to different functions that estimate how far the execution is from satisfying that condition [36], and some work has been done to improve the boolean flag conditions for reaching a branch, where the condition is a function returning a boolean [37].



```
int binsearch(int x, int v[], int n)
{
   1 | int low, high, mid;
     | low = 0;
       high = n - 1;
       while (low <= high) | 2
       {
         3 | mid = (low + high)/2;
           | if (x < v[mid])
                 high = mid - 1;  | 4
         5 | else if (x > v[mid])
                 low = mid + 1;   | 6
         7 | else return mid;
       }
       return -1; | 8
} | 9
```

Figure 2: Example of Control Flow Graph [38]

SBST is continuously researched as it brings a "generic approach to solve software testing problems automatically, using optimisation algorithms" [31]. It is still widely used and researched due to the amount of tests needed and its proficiency in generating them [22, 39, 40, 41, 1].

2.1.3 Constraint-Based Testing

Constraint-Based Testing (CBT) is based on solving constraints of Symbolic Execution (SE) [35]. SE goes through all paths of the program, gathering all conditions (=constraints) to go down those paths in an abstract (=symbolic) manner. To reach a path through the CFG, the constraints that apply to this specific path can be solved, providing the conditions and concrete input data values that make the program execute into that path. SE can be used to enter only a single path and reach a target block defined as the search objective.

However, solving all the constraints for a path might become too hard because the number of constraints grows exponentially. It is shown that the class of constraints-solving problem falls into the NP-hard category [42], but in the case of logical path constraints, they can be reduced to the SAT problem [43, 15], which is NP-complete. However, even with SAT solvers in the loop, which are developed to solve as efficiently as possible boolean-satisfaction problem [43], solving SE remains exponential in time. To reduce the time to solve the problem, Dynamic Symbolic Execution (DSE) can be used (also called concolic execution, for concrete symbolic execution). Some variables are chosen and fixed at runtime to satisfy some constraints, reducing their number, and therefore reducing the search space of the problem. This will give a better chance to solve the remaining constraints and to find a solution.

As based on SE, CBT techniques suffer from the same affliction as SE: they are unable to cope with dynamic aspects of the program as well as unavailable source code [44]. However, they are also efficient on the same points: finding a solution to a problem where the search space is immense, and reaching precisely some targetted path of the program [45]. This makes them useful for reaching deep code regions that would not be reached by using random input.

The main difference with SBST is that CBT is targetting a specific branch or path of the program and generates a test case just for that one in a single constraint resolution, while SBST runs an optimisation procedure to evolve multiple solutions until reaching an optimum which might not reach the path.

For example, using Figure 2, the constraints for the path 1-2-3-5-7-9 are the following:

$$\begin{aligned}
low_0 &= 0 && (node1)\\
high_0 &= n_0 - 1 && (node1)\\
low_0 &\leq high_0 && (node2)\\
mid_0 &= \lfloor (low_0 + high_0)/2 \rfloor && (node3)\\
x_0 &\geq v_0[mid_0] && (node3)\\
x_0 &\leq v_0[mid_0] && (node5)
\end{aligned}$$

2.1.4 AI testing

With the big advancement of AI in the last years, particularly Large Language Model (LLM), research has been showing that LLM can generate tests when given a CUT [21]. It proved its effectiveness in generating security tests when given prompts that described a vulnerability.

While particularly efficient when used with prompts, AI capabilities are limited to simple code structures, as it fails to understand the more complex and in-depth of more elaborate code. The generation is usually based on a context approach, and what the next word would be the most probable in this context. Having intricate structures means that the most probable words are staying on the surface of the complexity, as going in depth is unlikely.

2.1.5 Combination of methods

Each of the existing techniques has advantages and drawbacks, which can surprisingly complement each other quite well. This is why a lot of research is dedicated to combining multiple methods together in order to make the best of each.

For example, a combination of SBST with CBT was proposed by introducing a fitness function that uses symbolic analysis of the executed branches or hardness to satisfy the branch condition [46, 44]. These combinations can be seen in a tool called EvoSuite (see subsection 2.3.1), which generates tests using SBST along the CFG until some concrete values are needed, then switches to CBT and use SE to solve the constraints and get the correct values.

Another combination is done to improve random testing. To reduce the search space, or guide the generation towards better sequences, a tool called DART was created [29], using random value generation and symbolic execution to solve constraints starting from the random values in order to direct the search to the targetted branch. This technique was further extended and was given the name of concolic execution [47]. Another tool, Randoop (see subsection 2.3.2), was developed [25], generating random method call sequences using feedback from execution results to prune the search space of the next method to choose.

2.2 Context and Framework

Due to the project being done in industry, some context is already in place. While it should be taken into consideration, the industrial context should not limit the options of the research project. However, it could balance the final choice.

The frameworks detailed here are not compared with each other but given as context because they are used in industry. Some relevant details are provided to better understand the choices made later on.

2.2.1 Java

Java is a high-level programming language whose purpose is to run anywhere thanks to the abstraction of the Java Virtual Machine (JVM). It is a class-based Object Oriented Programming (OOP) language, implementing the concepts of class inheritance and polymorphism, abstract and interface classes, method overloading, class constructors, and other OOP concept [48, 49]. To be system or architecture-independent, Java code is converted into bytecode, which is then used by the JVM to run on the different OS architectures. The JVM abstraction allows having a portable bytecode that can be used seamlessly wherever a JVM is available. Scala, Kotlin and Groovy are other languages compiled using the same bytecode instructions, but using different programming paradigms.

Java has now many different versions, with a few Long Term Support (LTS) ones: 8 (2014), 11 (2018), 17 (2021) and 21 (2023). Each version has improvements and removal from the previous one. For example, a major difference between Java 8 and Java 11 is a restriction on Java reflection to only available API (private and nested API called through reflection are denied), Java 17 introduces pattern matching in switches, and sealed classes (which restrict which other classes can extend it), Java 21 adds virtual threads and string templates. Many other implementation details and breaking changes are made between each version and therefore, an implementation in one version might not work in another one.

The two important points to take from this are the OOP paradigm of Java, and the reflection. OOP gives a structure to the code, dependencies between objects, and a programming paradigm that imposes some rules that can be expected to be followed by Java. The Java reflection is used extensively to manipulate internal properties of the program, allowing it to know and modify the behaviour of a class dynamically. For example, at runtime, a class k could find a list of its own methods, and choose one to call based on some parameters, without the developer having to specify that list of methods.

2.2.2 Spring framework

Spring is a framework for Java used in industry to create fast and secure applications. Spring provides easy data access, transaction management, security and natively integrates with some other frameworks and tools such as Hibernate or Kafka. Spring also brings in RESTful services and allows for the development of web applications [50, 51]. As its core, Spring provides a mechanism called Dependency Injection (DI) used for Inversion of Control (IoC), to manage Java objects through reflection, which are called beans in this context. The main idea is to give the management of those beans to Spring so that the developer does not have to do it himself. For example, a web controller that would need a database does not need to care that the database is created before using it. Those dependencies are entirely managed by Spring.

DI is used to provide the needed Java objects to a bean. It can provide other beans, but also methods, by using reflection on the bean currently being configured.

Spring also comes with its own Model View Controller (MVC) framework for developing web applications, called Spring MVC [52]. It gives a proper separation between the model (abstraction of the data used), the controller (logic about how the data is used or modified), and the view (how to display the data), which makes a lot of sense for web applications, where the webpage is the view of some data from the model, and the different Hypertext Transfer Protocol (HTTP) requests are reaching controllers to request data modification or new data to display.

Spring needs a massive boilerplate setup, so to cope with that and allow fast development of applications, an overlay framework called SpringBoot is commonly used [53]. It hides the complexity of Spring and allows starting a Spring application with minimal configuration by providing some autoconfiguration features that reduce the amount of manual configuration that a developer has to do to set up Spring. While they are two different frameworks and have different roles and concepts, Spring and SpringBoot are used together most of the time. In this project, as SpringBoot basically offers a simpler configuration of Spring, no distinction is made concerning the origin of concepts coming from either Spring or SpringBoot. All will be referred to as coming Spring or SpringBoot without any differences.

2.2.3 Testing frameworks

Many different testing frameworks exist and are used for different scenarios. In this project, the focus is put on unit testing and mocking, so only relevant frameworks are noted here.

JUnit is a testing framework for Java used mainly for unit testing [11]. Other testing frameworks exist [54], such as TestNG [55, 56], very similar to JUnit, but JUnit is by far the most commonly used in open-source projects [19]. JUnit is widely used in Test Driven Development (TDD) and allows testing of under-development code, detecting bugs as early as possible.

Mockito [57, 58], EasyMock [59], jMock [60], JMockit [61] or GMock [62] are mocking frameworks, used to inject mock objects when required, with Mockito being mostly used [19]. Each of them has their special work cases, and usually inspired by each other. A mock object is used in unit test to provide isolation of the test components. When testing a given component, which requires some other objects whose behaviour is not completely implemented, requires external interaction, or might fail, then we can mock those dependent objects. Their behaviour needs to be "fake" but similar to the real one, so that the CUT can execute correctly. For example, we can mock a database so that when a "get data" request is done, some data (that the test developer chooses) is returned, without having to actually implement and set up a complete database connection in the test. This is done to test each component individually, without relying on others.

In each of these mocking frameworks, the mocking is done on a few objects used in the tests, so that the implementation of the object itself does not matter. It is quite useful for TDD where the behaviour is specified and tested before the implementation.

In some cases, there might arise a need to mock a lot more of objects or to provide a context with only a few mocked objects in it. In Spring for example, the Spring MVC Test framework, or simply MockMvc, was introduced to support the testing of MVC application [13]. It can be used to mock web requests and their responses instead of running a server.

2.3 Existing Tools

Tools that implement some unit test generation techniques already exist. Their examination is undertaken to see their capabilities and limitations, both theoretical (research-oriented) and practical (implementation details).

2.3.1 EvoSuite

EvoSuite [63] is a tool that uses a hybrid approach of SBST and CBT to generate and optimize test suites for Java in JUnit format. It tries to optimize the test suite to satisfy coverage criteria that can be provided by the user. The program's behaviour is captured by including assertions in the unit test cases that are generated. DSE and testability transformation (the transformation of a program to make it easier to test) [64] are implemented in EvoSuite [65], and important features such as the Virtual File System (VFS), the Virtual Network (VNET) and the sandbox allow the generation and execution of test cases in a controlled and isolated environment. Additionally, the test suite generated is reduced to only tests that contribute to the global satisfaction of the coverage criterion. Independent tests are created so that each test execution does not impact the result of any other tests. This is important to make sure to properly capture a program's behaviour by unit testing.

It implements an algorithm for a whole test suite minimization [66] which improves the generation of test cases by considering a fitness function for a whole suite, and not only for a single test case. Moreover, it is capable of handling Java's generics and handling type erasure, which is frequently overlooked when generating test cases [67], and implements functional mocking, allowing the testing of non-instantiable dependencies or complex object-oriented encapsulation objects [68].

It was tested in recent SBST Tool competitions, in 2020 [69], 2021 [70] and 2022 [41], in which it ranked amongst the best tools.

While it implements state-of-the-art techniques, EvoSuite has a few drawbacks. It is built with Maven on Java 8, which is quite an old version of Java now, and can only run for Java 8 and 11 programs. This is mainly due to the fact that some parts of the internal java package, which are packages coming directly from the Java Development Kit (JDK), are mocked by EvoSuite. Updating for a newer Java version would require updating all the mocks that were implemented. Furthermore, only limited capabilities for generating mocks are implemented using Mockito. EvoSuite is not capable of searching for bugs or crashes. As its main purpose is to generate tests that capture the current behaviour of the program, if a path that is covered results in a bug, it will simply be considered as a "normal behaviour" and be captured by this test.

A major limitation is the lack of support for injections that depend on Spring or Java Entreprise Edition (JEE) containers. This means that whenever an object is dependent on a dependency coming from within an external framework that injects it, no test can be generated because that dependency cannot be injected into the CUT.

2.3.2 Randoop

Randoop [71] is another tool that can generate unit tests for Java in JUnit format. It is based on feedback-directed random testing [72, 25] and generates many random tests, most of which are redundant to each other. It can be used to find bugs in the program, similar to fuzzing, or to create regression test suites, which capture the behaviour of the SUT. Generating a new test is based on a simple *extend* operator which takes 2 sequences of statements, S_1 and S_2 and a method m and creates a new sequence S_e which is the concatenation of S_1 and S_2 in either order with the call to the method m using parameters coming from S_1 or S_2.

It was tested in recent SBST Tool competitions, in 2021 [70] and 2022 [41], in which it showed good performance, similar to EvoSuite.

As for fuzzing, generating a lot of random tests is quite easy but generates a lot of rubbish. About 100 tests per second are generated, and while the total coverage is high, it is hard to tell which tests are useful. Executing them on the fly allows filtering the correctly behaving tests, and discarding the others. However, executing with poorly chosen arguments might make the test execute for a long time or never-ending. Choosing those values could be done by using symbolic execution or constant mining.

2.3.3 Diffblue AI Cover

Diffblue cover AI [73] uses reinforcement learning to generate test cases based on the provided CUT and by learning a model representing the structure of the SUT. It can generate Java unit tests in JUnit or TestNG format that can not be distinguishable from developers'. Each method is analysed one by one, tests are generated to cover each pathway and assertions are added to capture the program's behaviour. Each test is executed and given a reward value, to then choose a subset of the tests to put in a suite with a high coverage and low number of tests.

It also comes with a sandbox to execute the tests in isolation from the network, file system or system changes [74]. Diffblue Cover is developed to work with Mockito and supports Spring natively.

As an AI tool, the main drawback is the lack of search and randomness in generated tests. Hard-to-reach paths (example in Listing 5), or cases that require dynamic execution of the program, are not reached by generative AI.

Listing 5: Hard to reach path

```java
boolean testFunction(Color color) {
    if(hashCode(new Car(color)) == 0xcafebabe) {
        return true; // hard
    } else {
        return false; // easy
    }
}
```

2.3.4 Tool comparison table

Characteristic\Tool	**EvoSuite**	**Randoop**	**Diffblue Cover Ai**
Opensource	yes, LGPL-3.0 license	yes, MIT license	no, not free (30k/year)
Build	mvn j8	gradle j11	as a plugin
Java/JUnit	Java 8 JUnit 3, 4, 5	Java 8, 11, 17 JUnit 4	Java 8, 11, 17, 21 JUnit 4, 5
Spring Support	mock when classes are public, no support for Spring annotations	no mocking, no bean, no Spring support	full support, mocking of spring (one of their salespoint)
How it works	SBST approach integrating state-of-the-art techniques such as for example hybrid search, dynamic symbolic execution and testability transformation.	Random Testing approach to create method sequences incrementally, by randomly selecting a method call to apply and selecting arguments from previously constructed sequences. As soon as it is created, a new sequence is executed and checked against a set of contracts.	reinforcement learning modelling the program structure
Search guidance	Try to cover those goals: Line Coverage Branch Coverage Exception Mutation testing (weak) Method-Output Coverage Top-Level Method Coverage No-Exception Top-Level Method Coverage Context Branch Coverage	Next method choice: Randomly Bloodhound (least covered method).	Reinforcement Learning model generated
TestCase generation strategy	OneBranch Random ENTBUG MultiObjectiveSUITE DSE, NOVELTY MAP_ELITES	extension of previous sequence by concatenation of 2 parents tests sequences	N/A
Advantages	search based, GA, DSE deterministic run each test isolated (EvoSuite runner) object construction graph	randomness (based on seed) pure unit tests	AI but deterministic
Limitations	Run with a complex EvoSuite runner which makes some test fail locally. Doesn't switch midway to another strategy to maximise their strength.	lots of tests generated (100tests/sec), most of which do not improve coverage, as they are simple extension of the previous one. tests are hard to maintain. no usage of external library otherwise increases search space too much. no mocking at all (not even basic interface or abstract)	hard to cover branches AI harder to improve no test case execution feedback
Improvement?	Add Spring support add Java 17 support improve Object Construction Graph (OCG)	improve choice of input value mocking minimise test suite implement object construction graph	N/A

2.3.5 Tool choice

After analysis of the tools, EvoSuite was chosen as a base to improve upon. Its capacity to generate a TestSuite with high coverage using a few TestCases and its diversity in terms of unit test generation techniques are the motivations to choose it. Being open source also plays an important role in the decision.

Two major drawbacks are the fact that it runs only with Java 8 and is therefore a quite old project, and the internal complexity of the tool, which makes it difficult to add modifications to it. However, the fact that EvoSuite uses the concept of OCG (see subsection 3.2) means that adding the support for mock objects should be feasible once the need arises.

Using EvoSuite, the different challenges can be addressed:

- EvoSuite has a great analysis capability, which can be used to help identify dependencies of a CUT, allowing to introduce detection of mock in the process.
- The behaviour of the mock object can be evolved and tried multiple times during the process of test generation, which can help reach a correct behaviour without a priori knowledge of the mock exact behaviour.
- The configuration can be done in the initialization stage of the CUT analysis
- EvoSuite provides an important setup to isolate the tests and to detect their independence, which will be very helpful for making sure the generated mocks do not break isolation.

2.4 Clarification of the thesis

Finally, as the state of the art is completed and several technologies have been learned, the objectives for the project can be established.

The goal is to improve EvoSuite capabilities to include the generation of Mock MVC tests for Spring.

This will require a very specific implementation of generating mocks during unit test generation while bringing an important value to unit test generation in industry, as the current automated generation tools are not capable of doing so properly. It includes detecting the need for mocking by detecting which class can be tested with Spring Mock MVC, as well as automatically generating the sequence of statements to call MockMVC, searching for the test variable values, to create TestCases that can become complex to test very specific functions.

While the pure research novelty is low, the technical improvements are major for industrial automation of testing.

3 Design

In this section, an overview of the design of the project is given, along with the choices that resulted from them. Firstly, some explanations about the tool EvoSuite, its structure and concepts, then about the Spring framework, and finally a few details on JUnit and Java itself. This leads to design choices that are implemented in section 4.

3.1 EvoSuite details

EvoSuite is a program used to generate test suites in JUnit format, written to Java files. To do this, it analyses a given class, called the CUT, with a given classpath, and searches to generate sequences of Java calls that will reach the functions of the CUT, and therefore cover it. EvoSuite uses evolution of tests (GA from SBST) and local search on variable values (DSE from CBT) to reach many paths of the CUT. Each Java call is called a Statement in EvoSuite, and a sequence of Statements is called a TestCase. Additionally, each Statement is composed of VariableReference, that are used by EvoSuite to hold the values of the variables used in the TestCase.

VariableReference A VariableReference represents a variable in a TestCase. It holds the position of the variable the first time it is assigned, and at TestCase internal-execution time, gets assigned a concrete value. It has a few specific implementations for FieldReference (holds the reference of a field in an object), for ConstantValue (simply holds a constant), or special Array references (holds either a reference to an array, or an index in the array).

Statement A Statement is a building block for EvoSuite TestCases. It represents a value of some type, held in its return value VariableReference. Different Statements are implemented, like MethodStatement, ConstructorStatement, FieldStatement or PrimitiveStatement. They are all holding a VariableReference returnValue, can be executed in EvoSuite Virtual Machine (VM) and finally can be converted to Java code representation, which we call rendering. Most of the Statements are of the form varRefY = f([varRefx]), with f being a simple function of some variables. Java streams, lambda functions and combinations of multiple standard statements, like chained access to objects or nested calls, are not implemented in EvoSuite, so they cannot be generated. Examples are given in Listing 6.

Listing 6: Statement examples

```
// ok in EvoSuite
this.name = "Julien"
this.specialties = new HashSet<String>();
this.specialties.add("Java");

// not implemented: nested, chained calls, lambda
PropertyComparator.sort(sortedSpecs, new MutableSortDefinition("name", true));
it.getAllClasses().stream().filter(c -> c.contains("BaseEntity"))
    .limit(30)
    .map(c -> it.subclasses(c) + "->" + c + "->" + it.superclasses(c))
    .forEach(c -> System.out.println(c));

// instead of
vets.getVetList().addAll(this.vets.findAll());

// in EvoSuite
Collection<Vet> localVets = vets.getVetList();
Collection<Vet> currentVets = this.vets.findAll();
localsVers.addAll(currentVets);
```

TestCase A TestCase is a sequence of Statements, which can be executed or transformed into code, and a list of assertions that can be added later in the process. Whenever a Statement is added to a TestCase, it is verified. A Statement is valid in a TestCase when all the VariableReferences it is using have a position prior to the one of the statement itself. When internally-executed, the TestCase executes all its statements in order within EvoSuite VM environment.

TestSuite A TestSuite is simply a list of TestCases. However, it is per se not implemented in EvoSuite. The concept is implemented in a TestSuiteChromome which holds much more information than just the list of TestCases.

Chromosome A Chromosome in EvoSuite can be a TestCase or a TestSuite. It is used in the Genetic Algorithm (GA) to be evolved and improve its fitness value by mutation with other Chromosome.

Fitness Value The Fitness Value of a Chromosome is a real number that represents how good the Chromosome is for some Fitness Function.

Fitness Function A Fitness Function is a way of assessing the quality of a Chromosome, usually used in GAs. In EvoSuite, many Fitness Functions are implemented, and they measure if a Chromosome covers some predefined goals. Those goals are usually branches, lines, methods, or exception coverage, either for the TestCase or for the TestSuite globally.

Constant Pool The Constant Pool is a container of constants found during static analysis of the CUT. It is used in 2 different steps in EvoSuite when choosing a random value for a ConstantValue variable. The first time is when a random ConstantValue is added to a TestCase, and the second time is when a TestCase is mutated. This is used to guide the search by providing values that are used in the CUT, increasing the probability of reaching a specific path. Indeed, giving only random values is more likely to reach either always the same branch or go in an infinite loop.

Time allocation A small note on how the time resource is allocated in EvoSuite. To not run EvoSuite for an indefinite time, which could happen if the result of an optimisation takes too long to be found, or if the execution of a test goes into an infinite loop, a search-budget is usually given to EvoSuite as a start parameter for the SEARCH phase, usually 30 or 120 seconds. Each other phase (minimization, assertions generation, JUnit checks, ...) receives a dedicated time-budget, which will limit their maximum execution time. This means that in some cases, some phases might not be complete but are forced to stop, and the next phase should assume that the previous phase might not have finished.

Callgraph A callgraph is a graph of direct calls from one function F to the other functions f_i called in that function F. The nodes are functions, and the edges are pairs of function (F, f_i) where F calls f_i. It is used as a backbone of the OCG.

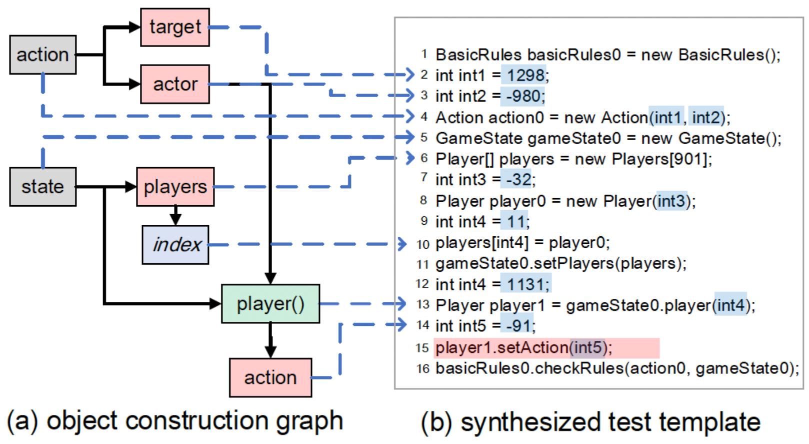
3.2 Object Construction Graph

EvoSuite tries to be as generic as possible and does not specialize in using some specific framework except for some mocks. To generate a TestCase from scratch, it uses the concept of Object Construction Graph (OCG) [1] while randomly choosing some Java calls to use as the next Statement. It first chooses a method, constructor, or any other basic instruction available in Java (coming from the CUT or from preloaded Java libraries like `java.lang` or `java.net`), and then tries to solve all the requirements needed for that call to be reached. Taking a method for example, it might need parameters, which need to be generated or reused from the TestCase, and it might need a callee if the method is not static. The parameters, which are Java objects, can be created from Java calls which are generated recursively. Similarly, the callee is satisfied recursively. This way of going backwards from the "wished" object towards satisfying its requirement implements the OCG concept at runtime.

OCG can be extended not only to Java objects, but also at a logical level, to if-else branches, while loop, or any other control structures that would create a path. In this case, EvoSuite calls it a dependency graph. This allows specifying a path as a target to be generated, and its dependencies will be generated as well in a backward recursive way at the logical level. Each dependency will then be converted into a set of Java instructions, on which the OCG can be applied as before. This will generate a TestCase which is capable of reaching the target branch, by means of mutating the input variables.

The construction of the graph is not done beforehand in EvoSuite, but only at search time in a recursive way, storing the graph dynamically in an ObjectPool in the form of a sequence of statements. Each sequence can afterwards be reused to reach the same object if needed without having to redo the search. This is implemented in the `TestFactory` class, using the `satisfyParameters` functions to recursively generate the required parameters.

An example of an OCG and of a dependency graph are shown in Figure 3 and Figure 4:



```
BasicRules basicRules0 = new BasicRules();
int int1 = 1298;
int int2 = -980;
Action action0 = new Action(int1, int2);
GameState gameState0 = new GameState();
Player[] players = new Players[901];
int int3 = -32;
Player player0 = new Player(int3);
int int4 = 11;
players[int4] = player0;
gameState0.setPlayers(players);
int int4 = 1131;
Player player1 = gameState0.player(int4);
int int5 = -91;
player1.setAction(int5);
basicRules0.checkRules(action0, gameState0);
```

Figure 3: Object construction graph (taken from [1])

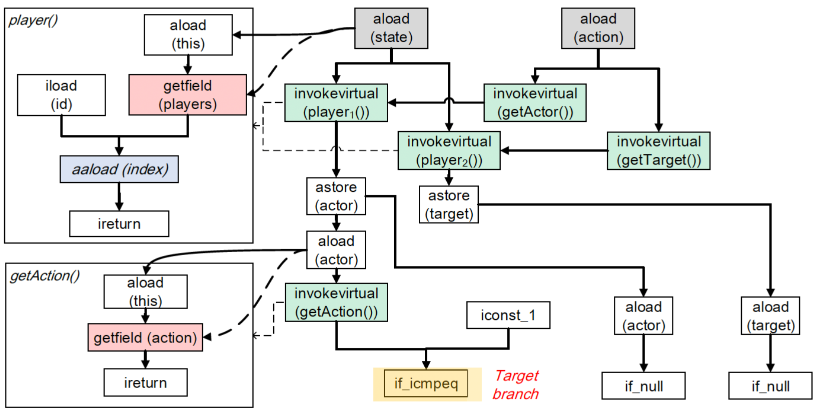


Figure 4: Dependency graph (taken from [1])

Using the OCG and callgraph is a starting point towards the identification of dependencies that can be mocked (challenge of subsection 1.2.1). Indeed, using the OCG, one can determine the dependencies needed to create an object and analyse them to know if they can be mocked or not.

3.3 Spring details

SpringBoot uses Java annotations to make use of the Spring framework concepts. The primary emphasis goes to the MVC concepts and DI as they are the core of Spring Web App on which this thesis focuses, and the usage of IoC used to inject beans when needed is important to make automated testing work.

Amongst those annotations, those shown here are important to understand the way Spring works. The definitions are coming from the Spring documentation [53], and summarized:

- @SpringBootApplication: it defines a Spring application by enabling autoconfiguration, component scan and allows extra configuration definition on the "application class".
- @EnableAutoConfiguration: attempts to guess and configure beans that are likely to be needed, based on the classpath and what other beans have been defined.
- @ComponentScan and @Component: with @ComponentScan, the IoC Spring container will scan the classpath for beans, and those beans can be @Component classes, which are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.
- @Autowired: Marks a constructor, field, setter method, or config method to be autowired by Spring's dependency injection facilities. Autowired means that Spring will try to inject that bean automatically. For a constructor, it defines that this constructor will be used when doing an injection of that class object. Fields are injected right after the construction of a bean.
- @Controller and @RestController: Spring considers the class when handling incoming web requests, and a @RestController = @Controller where @RequestMapping methods assume @ResponseBody.
- @RequestMapping: (put on a method or a class) provides "routing" information for web requests, mapping web requests onto methods. Usually, the child variants @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, or @PatchMapping are used instead of @RequestMapping.
- @ResponseBody: indicates that the method return value should be bound to the web response body.
- @Repository and @Entity: used for Jakarta Persistence API (JPA), @Entity tells JPA that instances of this class can be stored in a database, by default in a table of the name of the class. @Table allows customizing the table to save the entity. @Repository "defines" a @Component that sets an API to create, persist data and do queries to retrieve them. @Entity usually extends @MappedSuperclass which are objects without tables, like "interfaces".

3.3.1 Spring Annotation Processing

Java annotations have 3 modes of being used based on their retention policy [75]. If the retention is SOURCE, they are kept only in the source file; for CLASS, into the compiled bytecode file, with extension ".class"; and for RUNTIME, into the bytecode and are loaded at runtime.

With SOURCE, they can be processed at compile time by annotation processors, and bytecode can be generated from them, creating new classes or functionalities, so that the compiled bytecode of the annotated object can be executed by the JVM without needing the annotations. With CLASS, at runtime, they are not loaded by the JVM, so they can not impact the behaviour of the bytecode when it is executed. With RUNTIME, the annotations are loaded at runtime by the JVM, so they can be accessed by reflection, and could influence the program at runtime if used.

This last option is used with Spring, where annotations retention is set to RUNTIME, to be analysed at runtime by the Spring framework. When starting the Spring application, or when executing some functionality, runtime annotation processors look for annotations and change the behaviour of the program based on the annotations processed. This is done in the Application Context to find beans, spawn them and inject them where needed, or to verify some parameters when executing a Request.

In order to generate meaningful tests for Spring, those annotations need to be taken into account the same way Spring does. However, each framework uses its annotations differently, and handling in EvoSuite all possible annotations would not make sense. Only the relevant one should be handled, and the rest delegated to the framework itself.

This is required to solve the challenge of subsection 1.2.3 as most of the Spring configuration is done through annotation handling, and the challenge of subsection 1.2.1 because some mock can be figured out through their annotations.

3.3.2 Spring Unit Testing

At a pure unit test level, when a single method of the controller is tested, a Spring controller is like every other class. It has its definition, internal fields, constructors and methods which can be used to create the OCG and the dependency graph in EvoSuite. Those methods could therefore be called in EvoSuite as if they were not specifically from a Spring controller. However, in the context of Spring, a controller behaves quite differently, because it is manipulated by the IoC container as the annotations are taken into account at runtime by Spring. As a controller behaves differently in and out of the Spring context, generating a unit test for a controller should be done in its appropriate context, which is using the Spring context.

This is why most of the unit tests for Spring Web App are run with a special JUnit SpringRunner, and use MockMvc to be able to perform web requests in the context of the Spring app. From a categorisation point of view, using the application context is more integration testing, as it is not only the class that is tested, but also the context of the class in the Spring MVC environment. However, for Spring controllers, this can still be categorized in unit testing as those classes are never going to be used outside this context, and as the context makes them work or not. Testing a controller purely in unit testing would test only its logic, which should be decoupled if too complex, and could likely fail as the parameters are not the same as when running within the Spring context.

3.3.3 MockMvc capabilities

As MockMvc is the primary entry point for the unit testing for Spring controllers, an explanation of how it works is given here. Definitions are coming from the Spring documentation [13], and summarized:

- @RunWith(SpringRunner.class): Using the @RunWith annotation in front of a TestSuite class tells JUnit to use the SpringRunnner instead of the default BlockJUnit4ClassRunner. This runner is an alias for SpringJUnit4ClassRunner which "provides the functionality of the Spring test context framework" [76] in the test, allowing the loading of Spring context for example.

- @WebMvcTest: This is used to specify the controller class to test. SpringBoot provides the @WebMvcTest annotation to test Spring MVC Controllers. @WebMvcTest-based tests run faster because only the specified controller and its dependencies will be loaded without loading the entire application, in contrast with @SpringBootTest which loads the complete application. When using @WebMvcTest, SpringBoot instantiates only the web layer, rather than the whole application context. In an application with multiple controllers, you can even ask for only one of them to be instantiated by using, for example, @WebMvcTest(HomeController.class). It also allows adding some Component discovery, as by default, only the ones in the class of test will be discovered.

- The MockMvc instance is autowired by Spring when starting the test, with a correct bean setup and configuration. To be constructed, the MockMvc object needs a servlet and a list of filters which filter the request or the response. Filters can be empty in simple cases. The FilterChain is responsible for executing the request as the last filter, and gathering the result (see challenge subsection 1.2.3).
- The only entry point of MockMvc is a function `ResultActions=perform(RequestBuilder)`. It builds the request, executes it and gets a response (sync or async), then wrap it into a MvcResult, and returns a ResultActions used to match (call with `andExpect`), handle (call with `andDo`) or return (call with `andReturn`) the MvcResult of the request.
- To perform the request, the RequestBuilder is needed. Spring provides the protected class `MockHttpServletRequestBuilder` and the factory in `MockHttpServletRequestBuilders` get post, put, head, etc... to create basic HTTP requests with the method and URL. Then `MockHttpServletRequestBuilder` provides functions to set the parameters, contents, sessionAttribute, cookies, etc of the web request.
- Spring provides many ResultMatchers (`model(), view(), status(), header(), ...`) to help check the expected values of the request. A few predefined ResultHandlers (mainly printing) are also provided.

3.4 JUnit details

JUnit uses the concept of runner to execute TestSuite. When a TestSuite is executed, the TestSuite class is first loaded and instrumented (to add breakpoints or to track coverage for example), and then an instance of that class is constructed, instantiating all the static variables and fields. Methods annotated with @BeforeAll and @AfterAll are executed once before all the tests start and once after all the tests are done. Similarly for methods annotated with @Before and @After, but before and after each Test. Finally, each method annotated with @Test is executed and the result of the test (pass or fail) is returned to the runner to keep track of the progress. Developers are used to write setup and clean-up code in the Before and After methods, to avoid duplication of code in the tests themselves.

3.5 Design choices

To satisfy the generation of tests for Spring using MockMvc in EvoSuite, both frameworks need to be interacting together. Three options are possible, explained below.

1. Modify Spring to integrate parts of EvoSuite

   This option implies that EvoSuite would become a dependency of Spring. A Spring library would be used to provide an interface to call EvoSuite test generation procedures. Having EvoSuite in Spring would prevent most problems with Spring configuration, as Spring would start as usual, similar to starting the Spring application, and then could call EvoSuite to generate tests. Any objects coming from Spring (like the MockMvc object) as well as the full Spring capabilities would be available for EvoSuite. The modification needed for EvoSuite to use Spring would be implemented in the Spring framework, such that EvoSuite would behave as a plugin.

However, the amount of cherry-pick-like entry points that would act as "injection" of Spring into EvoSuite would be numerous. EvoSuite is indeed splitting the responsibility of tasks into many different singleton manager classes that would need to be overridden by Spring to handle Spring correctly.

2. Modify EvoSuite to integrate parts of Spring

   In this case, modifications are done to EvoSuite to support a specific framework, Spring. Support for just-needed functionalities would be added to have EvoSuite be able to understand the required part of Spring. At least annotations handling and processing, dependency injection, and Spring context loading would be added to EvoSuite, as well as a specific call graph to reach the handler methods so that EvoSuite can generate those without searching through the full call hierarchy with reflection that Spring does when handling a web request.

3. Create a new program that acts as a binder between EvoSuite and Spring

   This solution acts as another application that uses EvoSuite and Spring as dependencies, and calls the required methods from one or the other library on need basis. EvoSuite would be modified as little as possible, and Spring not modified at all. The application would handle specific flows to load a Spring context, create wrapper classes around the controller, call EvoSuite to generate some tests on the wrapper, and update those tests to revert to proper MockMvc tests.

   This was evaluated during the project as it seems to offer less modification to EvoSuite, but the wrapper needed was getting too big and copying too much of the Spring codebase, without the full capabilities.

Based on this evaluation, the 2nd option is chosen: EvoSuite will be modified to support the minimum of Spring features required to generate a MockMvc test. EvoSuite's source code is hosted on GitHub [77], and a public fork [78] is created to hold the work in progress of this thesis, as well as to allow the contribution back into EvoSuite's original repository. The main steps of the process are presented in Figure 5. The first step is to modify the flow of the class analysis to check whether the CUT is a Spring controller. If it is, then the Spring context should be loaded to create a MockMvc object in order to inject it into the TestCases. During the TestCase generation, some statements are added pseudo-randomly and at this stage, a SpringCall, which is a sequence of statements that will call methods of the controller through the Spring framework, can be generated. It will use the MockMvc object created before so that during the TestCase execution, the mocked web request will go through the Spring context correctly until reaching the handler method in the controller.

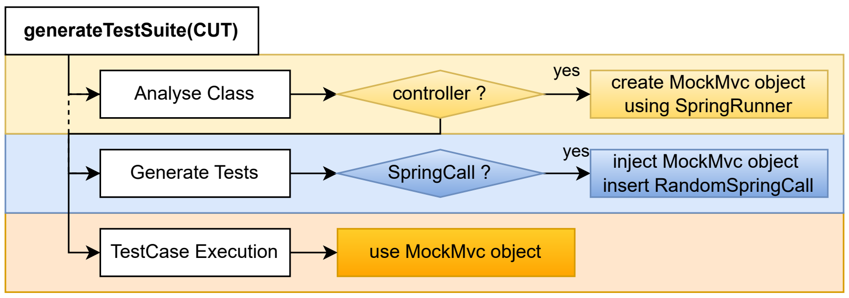


Figure 5: Schema of Design to be implemented in this project. This shows the modifications needed to make EvoSuite generate Spring call using MockMvc during the generateTestSuite call.

Class analysis The stage where the CUT is analysed will take care of handling the Spring annotations, of figuring out if the class is actually a controller, and of doing the necessary steps to load a MockMvc object hold into a Spring context that will be executable later on. It will also load the "call graph" allowing to reach the handler methods from the MockMvc calls, allowing to know which call will lead to which method.

SpringCall generation The stage responsible for generating a Spring call randomly will be integrated into the EvoSuite process that generates statements using the OCG methodology. A random switch should be added to select with some probability whether to generate a SpringCall or not. Two options are feasible, shown in Figure 6.

The switch can be at the same level as the choice of generating a call directly to the Controller (for example when generating a method call to the controller, or a field access call), or a level lower, at the choice of generating a method call. Both cases result in a call to the handler method but with different probabilities.

The main difference is when the choice is made. In the first case, the choice is made between fields, methods, constructors, reflections, SpringCalls, at the same level as the actual method that could be called. In the second place, the choice is made between method, SpringCall. Once the method has already been chosen to be generated, instead of generating directly the method, we choose to generate a SpringCall that will reach that method.

The second way would keep the same probability of generation that the original method has. This means that if the method was being generated, it would either be called directly with the OCG, or with MockMvc.

However, there is a chance that this might never be chosen in EvoSuite due to access-level modifiers. EvoSuite, by default, respects the access-level modifier put in place by Java, so a private method would not be selected at this stage. To make this work, it would require modifications to the EvoSuite call Generator class which is used in multiple other places, and the impact would be too big. It was therefore chosen to use option 1 as it offers fewer modifications to EvoSuite with the same result.

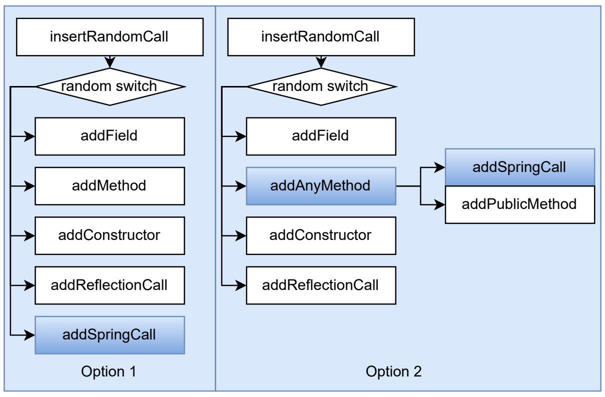


Figure 6: 2 options for SpringCall generation. Option 1 with SpringCall at the same level as CUT call choice. Option 2 with SpringCall below the level of Method call choice.

When generating a SpringCall, multiple Statements will be generated. Most values should be chosen randomly or guided by EvoSuite ConstantPool, while some other values need to be set in a deterministic way (like the HTTP route of the request). This is the role of the SpringTestFactory class to handle and delegate these choices to the responsible Spring generator helpers: one for the MockMvc, also responsible for doing the MockMvc injection, one for the RequestBuilder, and one for the ResultMatcher.

Test execution Finally the module for the TestCase execution will be responsible for executing the TestCase in EvoSuite VM, to make sure that the MockMvc object was injected, and finally to compile the TestCase into TestSuite in order to write only TestCases that are working. This step includes the rendering of the TestSuite, including the search of imports, the choice of the TestSuite runner, and the rendering of the injection of the MockMvc.

4 Implementation

4.1 Simplified Algorithm

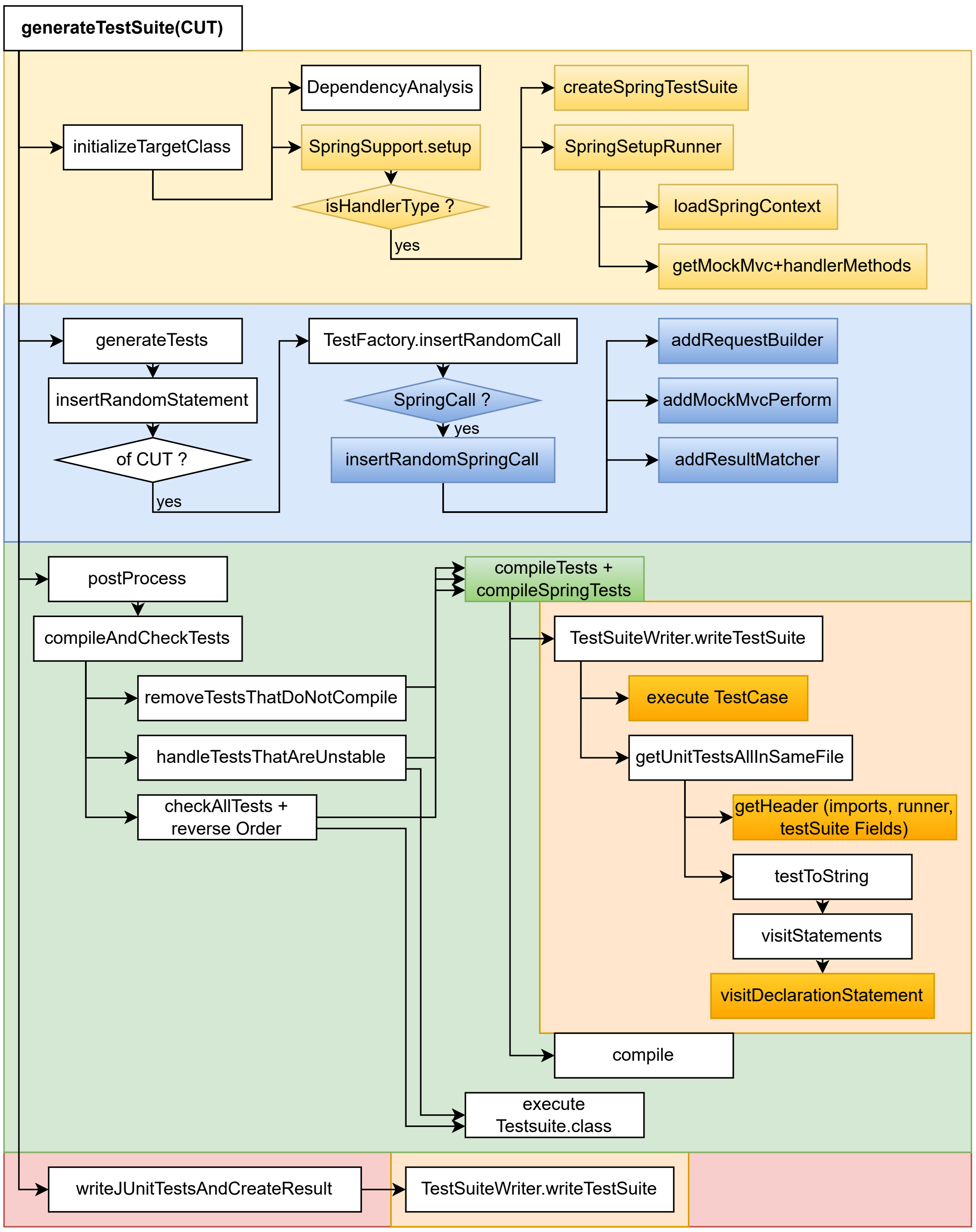


Figure 7: Algorithm of EvoSuite - generateTestSuite. The initialisation is in yellow, the test statements generation in blue, the post-processing in green, and the export in red. The stage of "writing TestSuite" in the orange block is the same, but detailed only once. White boxes are original from EvoSuite, and strongly coloured boxes are modifications made to support Spring. Decision blocks in diamond shape only show the happy path for simplification.

Figure 7 is showing a simplified algorithm to generate a TestSuite written into a Java file for a given Java class (here called the CUT). A quick overview is given here to set the context of the complete workflow, and each step will be detailed in its respective sections.

1. Initialize Target Class

   This first step works as a setup, by doing some analysis of the CUT, of its dependency and inheritance. A step is added to check whether tests using Spring should be generated, and if so, the necessary steps are taken to set up Spring into EvoSuite.

2. Generate Tests

   This is the most important step of the four. All different strategies to generate and evolve tests implemented in EvoSuite end up inserting a random statement in the test. That statement is chosen either from the CUT methods, the environment, or existing variables in the tests. When generating a statement from the CUT, a random choice might happen to generate a Spring call if allowed.

3. Post Process

   This step regroups all the details to validate that the tests are executable, independent, and that they can be written as TestSuite files. There are not many modifications done in this part except the separation of normal tests and tests using Spring (springTests hereafter), one step of the execution of a TestCase, and the transformation of a springTest into a springTestSuite.

4. Write Tests

   The final step is to write all the valid tests into Java files. It does not bring any new value per se, as it simply writes the tests that were already written before. Due to EvoSuite's implementation choices, this step executes and visits one more time all the valid tests to generate their content into a TestSuite.

4.2 Initialize Target Class

The stage of initialization is the first that happens to generate the tests. Its purpose is to generate an inheritance tree and a call graph, load the CUT and instrument it to track the coverage. All these steps are done by EvoSuite originally to prepare the different graphs and tools that will help to guide the test generation as shown in Figure 8.

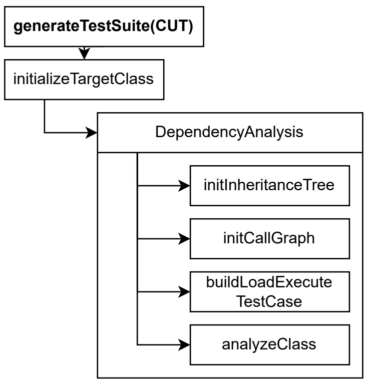


Figure 8: Original flow of initializeTargetClass

The first modification added to EvoSuite is to do the Spring support setup as shown in Figure 9. This step checks that the CUT is a Spring controller, in which case the setup actually happens to create and load the Spring context.

4.2.1 Need for Spring context

Using Spring and SpringBoot frameworks makes it easy to have a robust and complex web application running, while only coding the interesting parts: the web controllers and the User Interface (UI). Spring facilitates this by using IoC and DI, and to have this working correctly, when starting an application, Spring load a complex context based on application properties and Java annotations, creating the needed beans and injecting them where needed.

This is also done when running tests. An example of a typical (simplified) TestSuite for a Spring controller is given in Listing 7. The @RunWith annotation will make the test run with SpringRunner, which will load the Spring context for this specific test. In particular, it will load a test context based on the @WebMvcTest annotation, using the given controller, and inject a MockMvc bean (from the @Autowired) and a @MockBean for the owners' repository. Those beans are needed to run the test correctly in a similar context to the one created when running the actual Spring application.

The Spring context is used to generate a correct configuration of the MockMvc, which is solving the main challenge of "Mock configuration" (see subsection 1.2.3).

Listing 7: Spring simple test

```java
package packagename.spring.petclinic.owner;
import ...

@RunWith(SpringRunner.class)
@WebMvcTest(OwnerController.class)
public class OwnerControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private OwnerRepository owners;

    @Test
    public void testStatusOk() throws Exception {
        mockMvc.perform(get("/owners").param("lastName", "Smith"))
               .andExpect(status().isOk());
    }
}
```

Looking at the test, the code, and the documentation, the MockMvc is a wrapper to perform the request, and check the result afterwards. In order to have this capability in EvoSuite, the first idea was to implement a Simplified MockMvc (SMockMvc) in EvoSuite codebase itself, so that when generating tests and choosing a call, EvoSuite would be able to choose that SMockMvc to call the related method of the controller. After adding this SMockMvc, it was needed to add a lot of simplified classes from Spring (like SRequestBuilder, SResultMatcher, SHandlerMethod, SMappingRegistry...) which were getting simplified too much to make any useful work.

Instead of doing such simplification, which would also require a lot of refactoring for each version of Spring, the idea came to let Spring itself create the context, and get from it the MockMvc bean to inject it into EvoSuite. This requires more work during the setup, but also makes it use Spring directly instead of rewriting simplified code, so the full functionality of Spring can be used.

4.2.2 How to get MockMvc

To get the MockMvc bean, we need a Spring context, which is created by running the SpringRunner. To run, it needs a compiled class of a TestSuite for a controller. And to have that TestSuite and the beans it contains, we need to have the Spring context. This chicken-and-egg problem is not easily solved, even in classical applications, where engineers try to run the Spring application and based on the error, add the required beans and try again. In our case, relying on errors thrown by the Spring context loader would be quite complex, as it would require analysing the error and automatically acting based on it. Instead, it was decided to copy some part of the analysis that Spring does to know which beans will be needed.

Needed beans Spring detects the beans' dependency by analysing annotations or some specific class or interface inheritance. It can use Constructor-based, Setter-based, @Autowired injection, and a few other ways based on XML configuration [79] to instantiate beans. With Constructor-based DI, Spring checks the parameters of the bean's constructor, and if those parameters are beans, it will provide them. This is the case in Listing 8, where the OwnerController needs an OwnerRepository which is itself a bean managed by Spring. This works similarly for Setter-base and @Autowired injections.

While all beans will be created by Spring when loading the context during a real application startup, some will not be when running in a test context. This is the case for interfaces, or Repository.class. Those are the beans that will need to be set as @MockBean in the TestSuite.

For our purpose, a reduced but working Spring analysis is re-implemented in EvoSuite to detect Constructor-based interfaces and Repository. This is done in the step "analyzeClass" in Figure 8, and used to create the necessary @MockBeans in the TestSuite.

The Spring TestSuite is then completed with the imports (detected automatically based on the added @MockBeans), the @Autowired MockMvc, and a simple assert(true) in a @Test so that the SpringRunner can start. The TestSuite is then compiled and loaded to be used by the SpringRunner afterwards.

Figuring out the beans helps to solve the challenge of "dependency identification" (see subsection 1.2.1) as some of those beans are required as mock when using MockMvc.

Listing 8: simple Spring OwnerController

```java
package packagename.spring.petclinic.owner;
import ...

@Controller
public class OwnerController {

    private final OwnerRepository owners;

    public OwnerController(OwnerRepository clinicService) {
        this.owners = clinicService;
    }

    @GetMapping("/owners/{ownerId}/edit")
    public String initUpdateOwnerForm(@PathVariable("ownerId") int ownerId, ...
        Model model) {
        Owner owner = this.owners.findById(ownerId);
        model.addAttribute(owner);
        return "owners/createOrUpdateOwnerForm";
    }
}
```

Extend SpringRunner and Override Now that the SpringTestSuite.class is loaded, it can be used by other classes, and particularly the SpringRunner. This runner provided by Spring extends JUnitRunner to execute tests within a TestSuite that requires a Spring context. This is a very high entry point into Spring and can be used almost as is to load the context.

When `run` is called, the pipeline of JUnit to run tests in the TestSuite is started. It calls the `runChild` method whose purpose is to execute a @Test, by first creating a JUnit statement using the `methodBlock` method, and then executing it with the `runLeaf` method. The `methodBlock` calls the `createTest` method which is responsible for creating the Spring context.

Here, executing the test is not needed, as the only thing needed is the Spring context. To change the flow, the SpringRunner is extended as a SpringSetupRunner, which overrides the `runChild` to not call the `runLeaf`, as well as overrides the `methodBlock` to intercept the MockMvc object. The MockMvc object is held by the `testInstance` created by the `createTest`. Additionally, that MockMvc also holds a `Map<RequestMappingInfo, HandlerMethod>` that will be used later, so it is gathered here as well. The @RequestMapping annotation, parsed into a `RequestMappingInfo`, annotates the HandlerMethod, describing how to handle the web request and how to transform it into a correct Java method call.

New flow of initializeTargetClass These additions are summarized in the new flow of the `initializeTargetClass` method shown in Figure 9.

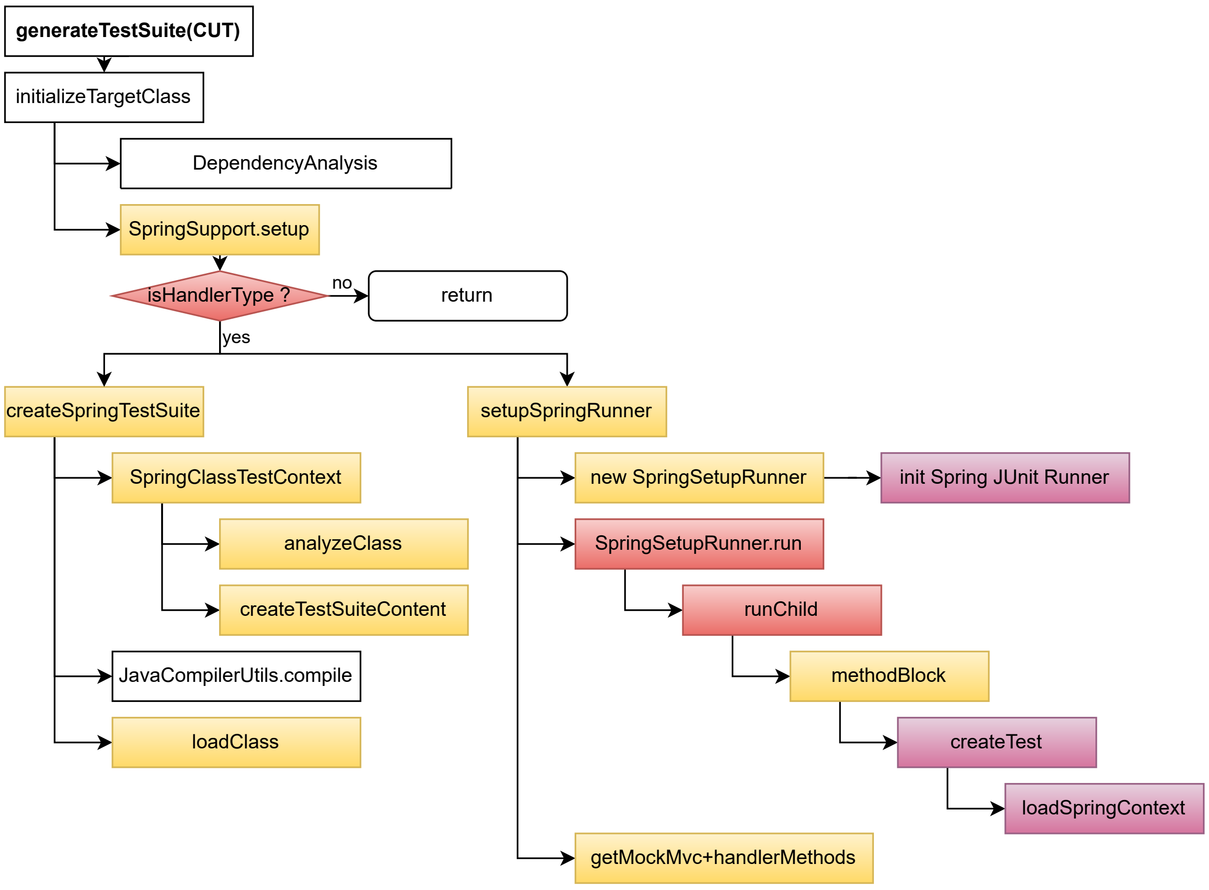


Figure 9: Flow of initializeTargetClass after addition of the Spring setup. White boxes are code blocks from EvoSuite, yellow are new implementations, red are overloaded implementations from Spring, and purple are calls to Spring.

4.2.3 Technical difficulties

A few technical difficulties have been encountered during the steps to get the MockMvc object.

The first issue is the loading of a class generated at runtime by EvoSuite with the same ClassLoader as the SUT is loaded with. It needs to be the same ClassLoader, so that when executed in the EvoSuite VM, and that MockMvc will call a method that should reach into the controller, it will be done in the same ClassLoader context. Two classes loaded by two different ClassLoaders will be considered different and cast exceptions will happen, so an application running with a specific ClassLoader should have all its classes loaded with this. EvoSuite handles this by creating a delegated ClassLoader that should be used anytime a class from the SUT is loaded. However, due to some classpath problem, this is not working when loading the generated SpringTestSuite. A fix to solve this is to move the compiled class into a folder where the bootstrap ClassLoader can expect it to find the class, because the bootstrap ClassLoader in Java is the parent to all ClassLoaders. This is done by moving the SpringTestSuite into the folder of the compiled CUT, which by construction, is in the classpath.

This is a small hack that can have a major impact on the correctness of the program, as it works inside EvoSuite, but is quite hard to see if it works when using EvoSuite as a standalone program. However, when testing with the PetClinic application, no issues were coming from this.

Another issue that was found and expected during the development of this part, is that the loading of the Spring context could not be tested at the unit test level of EvoSuite. It can only be done at the system test level, because it needs a full Spring application that can not be put in EvoSuite. This was additionally tested against the PetClinic application and resulted in not correctly loading the Spring application due to class shadowing introduced by multiple versions of the same class between EvoSuite and PetClinic.

A third issue was coming for the conflict of logger configuration between EvoSuite and Spring, as both used the same framework with different configurations. Loading Spring after EvoSuite was overriding the initial configuration and preventing any log from appearing. Restoring this configuration was mostly done. As EvoSuite uses a complex configuration due to its multiprocess architecture, the loggers could not be completely restored.

4.3 Generate Tests

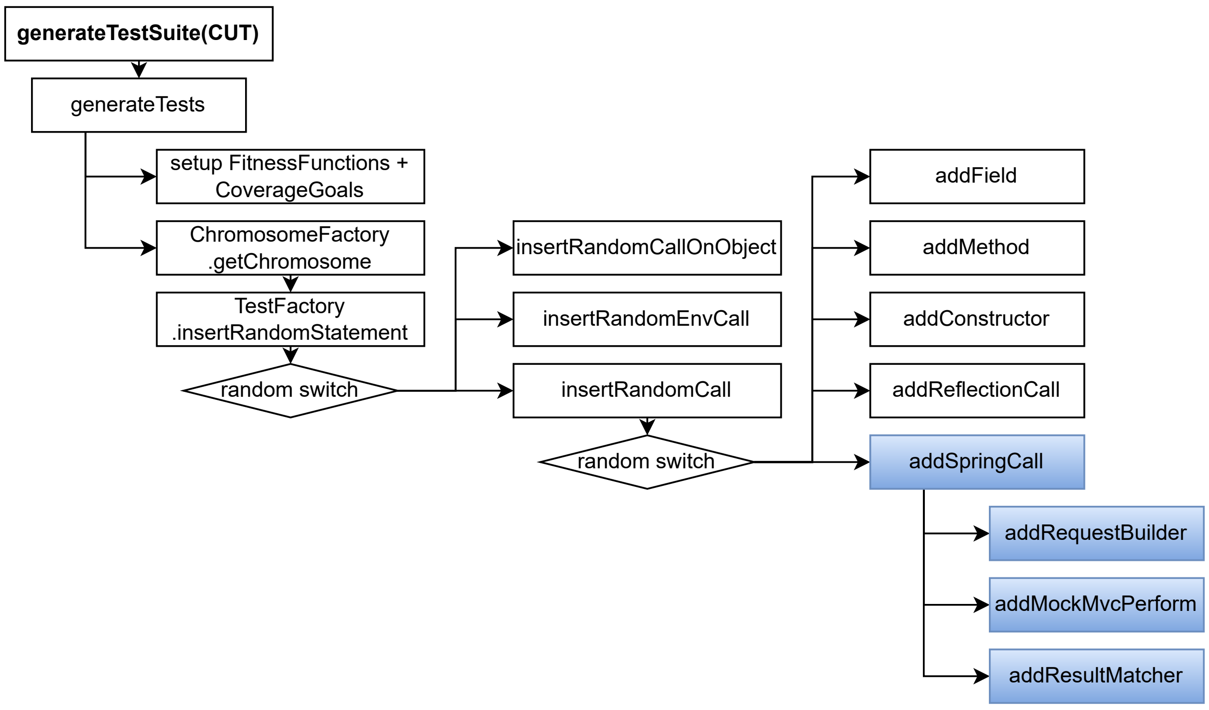


Figure 10: Flow of generateTests after addition of the Spring setup. White boxes are EvoSuite original code, blue are new implementations added for Spring support.

Generating a TestSuite is done with different strategies in EvoSuite. 10 concrete classes extending the abstract `TestGenerationStrategy` are implemented, with basically different objectives and search algorithms to evolve the Chromosomes. However, all of them end up calling the abstract `ChromosomeFactory.getChromosome`, implemented by about 20 classes. A TestSuite Chromosome will generate TestCase Chromosomes, and a TestCase Chromosome will generate a single test always in a similar manner, by calling `TestFactory.insertRandomStatement` a certain amount of time, to add a statement to an initially empty TestCase.

The insertion then switches between inserting a call from the CUT, from the environment, or from a variable that already exists in the test. "The environment of a test case are external resources for the test case such as handles to files on the file system, sockets that open network connections, etc" [80]. When the choice falls on a call from the CUT, another switch happens between 4 options: adding a constructor, a method call, a public field access, or a call by reflection (to bypass the access-level protection system).

4.3.1 Adding a Spring call

A Spring call is basically a call to `MockMvc.perform` alongside the (multiple) `.andExpect` that serve as assertion, as shown in the Listing 7. However, EvoSuite is not capable of chaining calls in streams, as the basic blocks of TestCases are Statements in the form of simple assignments. Therefore, the oneliner must be split into multiple statements, as shown in Listing 9.

Listing 9: Spring Test split oneliner

```
@Test
public void testOneliner() throws Exception {
    mockMvc.perform(get("/owners").param("lastName", "Smith"))
           .andExpect(status().isOk());
}

@Test
public void testSplitOption1() throws Exception {
    MockHttpServletRequestBuilder requestBuilder = ...
        MockMvcRequestBuilders.get("/owners");
    requestBuilder = requestBuilder.param("lastName", "Smith");

    ResultActions resultActions = mockMvc.perform(requestBuilder);

    StatusResultMatchers matcher = MockMvcResultMatchers.status();
    ResultMatcher resultMatcher = matcher.isOk();
    resultActions.andExpect(resultMatcher);
}

@Test
public void testSplitOption2() throws Exception {
    MockHttpServletRequestBuilder requestBuilder = ...
        MockMvcRequestBuilders.get("/owners");
    requestBuilder = requestBuilder.param("lastName", "Smith");

    ResultActions resultActions = mockMvc.perform(requestBuilder);
    MvcResult mvcResult = resultActions.andReturn()

    StatusResultMatchers matcher = MockMvcResultMatchers.status();
    ResultMatcher resultMatcher = matcher.isOk();
    resultMatcher.match(result);
}
```

How to generate the Spring call The EvoSuite way to generate the Spring call would be to request the generation of a call to `.andExpect` onto an object of type ResultActions. If everything was analysed correctly beforehand and working a usual manner, as EvoSuite uses the concept of OCG, it would be able to work recursively, creating first a callee object (ResultActions) by looking in the callgraphs, where it would find that `MockMvc.perform(MockMvcRequestBuilder)` returns a ResultActions, recursively create the MockMvc object, then satisfy the parameters of each method call recursively as well. An example of such a recursively generated callgraph is shown in Figure 11.

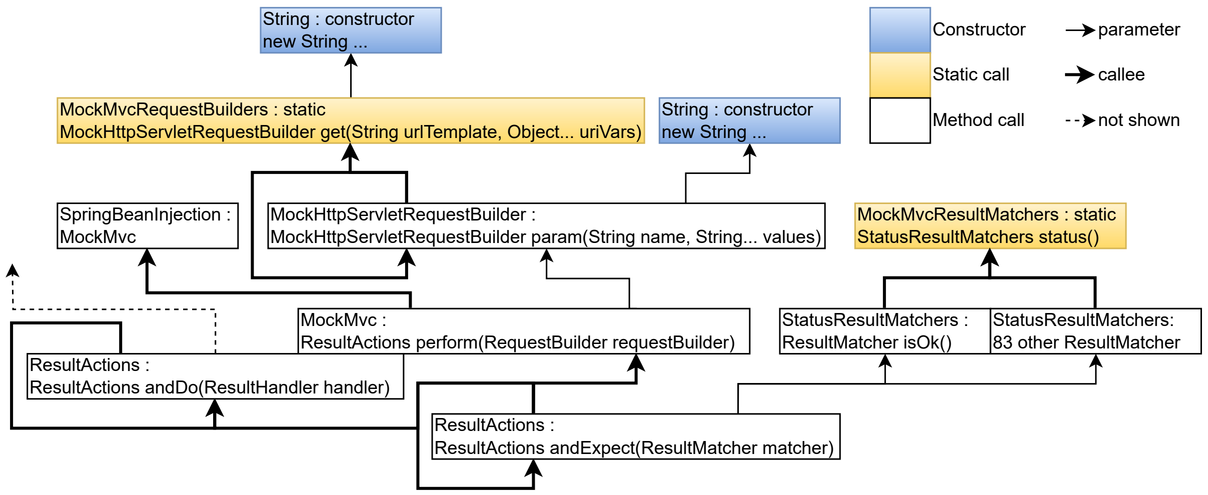

String : constructor
new String ...
MockMvcRequestBuilders : static
MockHttpServletRequestBuilder get(String urlTemplate, Object... uriVars)
String : constructor
new String ...
Constructor
Static call
Method call
parameter
callee
not shown
SpringBeanInjection :
MockMvc
MockHttpServletRequestBuilder :
MockHttpServletRequestBuilder param(String name, String... values)
MockMvcResultMatchers : static
StatusResultMatchers status()
MockMvc :
ResultActions perform(RequestBuilder requestBuilder)
StatusResultMatchers :
ResultMatcher isOk()
StatusResultMatchers:
83 other ResultMatcher
ResultActions :
ResultActions andDo(ResultHandler handler)
ResultActions :
ResultActions andExpect(ResultMatcher matcher)

Figure 11: Callgraph of the MockMvc perform method: Yellow blocks are static methods, blue are constructors, white are methods calls on objects of the given class. The class is given in the first line in the block, before the ":".

However, a few problems arise here. The first problem is that the MockMvc object is injected by Spring, which is not supported in EvoSuite's original source code. Secondly, the values of the strings would be randomly selected from the ConstantPool that EvoSuite maintains, and would unlikely work at all for the Spring request because of the specific format used. Thirdly, there would be no way to guide the generation to help craft meaningful Spring statements.

Those 3 problems are the incentive to generate a pre-determined ordered sequence of statements: addRequestBuilder, addMockMvcPerform, and addResultMatcher. Going against the principle of randomness and OCG in EvoSuite. Nevertheless, since the Spring calls are always going to be of the same form, it does not impact much the genericness of OCG as long as the randomness can still be applied. In addition, introducing the Spring bean injection in the middle of the process would be rather complicated in the OCG, as it would need to be added at the same level as calling the constructor for every object while making sure that the injection can happen correctly from Spring.

4.3.1.1 add RequestBuilder

Spring provides a few wrappers to create `MockHttpServletRequestBuilder` depending on the type of request (GET, POST, HEAD, PUT, ...), but all of them can be created by calling the static method `MockHttpServletRequestBuilder request(HttpMethod httpMethod, URI ... uri)` from the helper class `MockMvcRequestBuilders`. Once the RequestBuilder is created, it can be enhanced with parameters, content, accept type, header, cookie etc, until finally being given to the `MockMvc.perform()` method.

To help guide the generation of the tests, the `Map<RequestMappingInfo, HandlerMethod>`, previously loaded during the Spring setup stage in subsection 4.2.2, is used. An entry is chosen at random in the map, the RequestMappingInfo is used to provide the HttpMethod value, as well as the Uniform Resource Identifier (URI) template as a string, and the HandlerMethod is used to create a valid URI from the URI template.

Valid URI The URI given by the RequestMappingInfo is the route of the HTTP request as a URI template. This means that it can have some variables in it that will need to be substituted by real values to get a valid URI. For example, "/owners/{ownerId}/edit" is a Uniform Resource Locator (URL) template, while "/owners/2/edit" is a valid URI. Here, the validity of the URI is simply its syntactic validity.

In order to craft a potentially valid URI, knowing the types of the substitutes can help guide the choice of the value. This can be done thanks to the HandlerMethod, which is basically a wrapper for the method of a class, holding the parameters and their annotations. Indeed, the @PathVariable annotation can be used to map the URI variable to the correct parameter type. The value can be chosen randomly from the ConstantPool of EvoSuite based on that type.

We can assume that any variable in the URI template will have a matching @PathVariable parameter, as required by the Spring framework [81]. An example is given in Listing 10.

A note on implementation: in order to prevent the creation of the URI by EvoSuite, and as the URI is validated during this step as a String, a helper function `f_1` is implemented in a helper class `SmockRequestBuilder` (different from the one in subsection 4.2.1), which simply wraps the second parameter of `f_2`.

```
f_1 = MockHttpServletRequestBuilder request(HttpMethod httpMethod, String validUri)
f_2 = MockHttpServletRequestBuilder request(HttpMethod httpMethod, URI uri)
```

Listing 10: @PathVariable example

```
@GetMapping("/owners/{ownerId}/edit")
public String initUpdateOwnerForm(@PathVariable("ownerId") int ownerId, ...
    Model model) {...}
```

Enhancing A RequestBuilder without a header or a cookie can sometimes not match the security checks done by Spring before delegating the request to the controller. Without parameters, the functionality of the controller would sometimes not work properly. Adding this information is a needed step in the automatic generation of the RequestBuilder.

Due to the complexity of this task and lack of time and incentive, only a Proof of Concept (PoC) was done for the parameters using the RequestMappingInfo, which could hold the parameters. At runtime however, the parameters of the RequestMappingInfo parsed by Spring are empty.

A proper way to do this would be to generate the necessary parameters template based on analysis of the HandlerMethod parameter, and the value would be found by evolution of the tests later on. A quite effective solution here would be evolution using SE or DSE to figure out which value would lead to which code branch of the method. For example, in the Listing 11, the analysis would need to figure out that the Owner's first name is important, and that it can be given in the request parameters, even if not the method parameters. Indeed, Spring can prepare those values beforehand by doing injections on the method parameter values. It will look if a request parameter (paramName with value paramValue) matches a method parameter directly or one of its fields. In this example, the request parameter "firstName" would match the `Owner.firstName` field of the method parameter and therefore be injected by Spring with the value "Alice".

Doing this analysis would greatly improve the generated RequestBuilder, but would require an amount of work that is slightly out of the scope of the project.

Listing 11: RequestBuilder parameter example

```java
@GetMapping("/owners")
public String processFindByName(Owner owner, BindingResult result, ...
    Map<String, Object> model) {

    // find owners by first name
    Collection<Owner> results = owners.findByFirstName(owner.getFirstName());
    ...
}

// associated RequestBuilder template
String paramName = "firstName";
String paramValue = "Alice";
RequestBuilder requestBuilder = MockMvcRequestBuilders.get("/owners");
requestBuilder = requestBuilder.param(paramName, paramValue);
```

4.3.1.2 add Result Matcher

Similarly to the RequestBuilder, Spring provides helpers to create ResultMatcher in the package `org.springframework.test.web.servlet.result` which can be instantiated usually in a 2-level way. The abstract `MockMvcResultMatchers` is used as a factory to build the 1st level TopicRelatedResultMatcher, which holds a list of ResultMatchers specialized in their topic. Examples are given in Listing 12.

The 2-level structure is kept the same when generating the ResultMatcher. One of the 1st level ResultMatcher is chosen at random, and amongst its 2nd level, one is chosen at random. Its parameters are then generated with the `satisfyParameter` of the `TestFactory`, as explained in subsection 3.2.

In order to guide the generation into more relevant tests, the status, model and view 1st level ResultMatcher could be handled specifically. Indeed, the return value of the HandlerMethod is usually a View or a ModelAndView, while the model and the result can be given as the HandlerMethod, so their expected modification can be tracked. The EvoSuite way to do this is to analyse after executing the TestCase, fixing the values to the current execution trace. When adding assertions, ResultMatchers could be added as well, as they act as assertions.

Listing 12: ResultMatcher example

```java
StatusResultMatchers statusResultMatchers = MockMvcResultMatchers.status();
ResultMatcher resultMatcher0 = statusResultMatchers.is(200);
ResultMatcher resultMatcher1 = statusResultMatchers.isOk();
ResultMatcher resultMatcher2 = statusResultMatchers.isNotFound();

ModelResultMatchers modelResultMatchers = MockMvcResultMatchers.model();
ResultMatcher resultMatcher3 = modelResultMatcher.attributeHasNoErrors();
ResultMatcher resultMatcher4 = modelResultMatcher.attributeExists("owner");

ViewResultMatchers viewResultMatchers = MockMvcResultMatchers.view();
ResultMatcher resultMatcher5 = viewResultMatchers.name("owner/find");
```

For now, only a single status ResultMatcher is put in place: the is(200) ResultMatcher, because of prioritization. TestCase execution with Spring context was more important than implementing this.

4.3.1.3 add MockMvc Perform

The MockMvc perform generation step is an injection of the MockMvc object, a call to its "perform" method, and either nothing or a call to the "andReturn" on the ResultActions to get the MvcResult, as shown in Listing 9. The second option was actually implemented because it was found earlier in the decomposition of the oneliner, and no refactoring was done afterwards in that. This could be refactored without loss of generality.

MockMvc injection This is the place to inject the bean of the MockMvc that was found during the Spring Context Loading in subsection 4.2.2. EvoSuite has no support for injection, mainly because an injection means that a TestCase would be dependent on another entity, a TestSuite context for example, or would not be independent of the other TestCases, opening the door to TestCase interactions which would be hard to automatically handle.

We refer here to the subsection 3.1, where TestCase execution, Statement by Statement, is explained. In order to execute the `public ResultActions perform(RequestBuilder requestBuilder))` from the class MockMvc, the statement would be a MethodStatement in EvoSuite, constructed using the "perform" method, and when executed, taking the MockMvc object as callee and the RequestBuilder as method parameter. However, the MockMvc object is very specific and not created by EvoSuite during the tests, and thus could not be given as a parameter because it does not exist in the TestCase scope (holding only VariableReferences used in the TestCase).

To solve this, a first try was done to create a MockMvcStatement, very similar to a MethodStatement, but without the callee parameter. The callee would be set up in a global static way, so that when the statement executes, it would use the MockMvc object created by the SpringRunner before that (which would also be static). This idea was working but with 2 major drawbacks: firstly the lack of visibility on what adding that new statement would break in the automatic generation of tests, and secondly the lack of modularity or genericness by creating a statement just for performing one method call, while using a static context which would not necessarily be set up all the time.

Instead of setting the MockMvc down encapsulated in the MockMvcStatement, without allowing it to change, a new statement DeclarationStatement, and a wrapper Injection, were added. They are detailed in subsection 4.3.2.

4.3.2 DeclarationStatement

A DeclarationStatement is simply declaring a variable of some given type, the JVM would initialize it to its default value if none is given. Its rendering as text would be `ClassName ... variableName;`. While very generic, it was probably never implemented in EvoSuite because it does not bring anything to a TestCase per se, as executing the statement does not do anything. However, this DeclarationStatement allows doing Injection, which is basically having an external controller or manager setting the value of that variable.

The injection is done as a wrapper around that DeclarationStatement, by simply creating a new VariableReference of the type of the object to inject, and by setting the internal value of the DeclarationStatement to the object to inject itself. This is shown in Listing 13.

Listing 13: Injection wrapper example

```java
public void appendInjection(Object inject, TestCase testCase) {
    VariableReference varInject = new VariableReferenceImpl(testCase, ...
        inject.getClass());
    DeclarationStatement stmt = new DeclarationStatement(testCase, varInject);
    stmt.setValue(inject);
    testCase.addStatement(stmt);
}
```

This step of creating a Statement simply to hold a VariableReference and the objectToInject is needed as the value of the VariableReference cannot be set outside the scope of a TestCase execution in EvoSuite. Whenever this statement is executed, the value of the VariableReference will be set to the objectToInject. Seamlessly, the VariableReference can be used afterwards as a normal variable, without needing EvoSuite to manage it, and without having the developer (of EvoSuite) to struggle to know whether this VariableReference is valid or not.

When generating a TestCase that will need to use a Spring MockMvc object, it will only need to do that injection, and then can use all the methods of the class. This is generic in the sense that another part of the EvoSuite source code, or another project that needs to do injection can also do it.

Warning This is to be used with care because it might break the encapsulation that was put in place by EvoSuite, as each of the tests could use that same object that is injected. Here, the only call possible is MockMvc#perform, which seems to let the MockMvc object exactly as before, so calling perform multiple times would not change the object between TestCases, and the encapsulation is not breaking because of this new object. This refers to the challenge of "Isolation" (see subsection 1.2.4).

4.3.2.1 Rendering

A note about the rendering of the DeclarationStatement, which is the step of transforming the statement into a Java text representation. As a simple declaration, it will simply be "`ClassName variableName;`", and while serving as an injection "`// injection of: ClassName ... variableName;`". In both cases, the representation does not execute something functional. It could serve as a pre-declaration for a try-catch variable or if-else, but those cases are already specifically handled by EvoSuite differently. The idea is to still render the injection, but as a commentary to help understand the automatic generation process. This can be changed very easily when this solution is stable.

Additionally, EvoSuite gives names to the variables when writing them on the TestCase. The usual strategy is the name of the class in Lower Camel Case plus a number (start with a lowercase letter, then Camel Case, then a number), for example, `mockMvc0`, or `requestMatcher1`. The number increment for each variable of the same class instantiated. This is important to give a correct name to the @Autowired MockMvc bean when rendering the final TestSuite.

4.4 Post Process

Once a TestSuite is generated, it contains a list of TestCases that are executable in the EvoSuite VM. They can be still optimized, and need to be written out to a Java file. To reach that last step, a Post Process step is run, following the summarized steps shown in Figure 12 in order to only output a TestSuite, containing only the most relevant tests, ready to be compiled and used by JUnit in the SUT application.

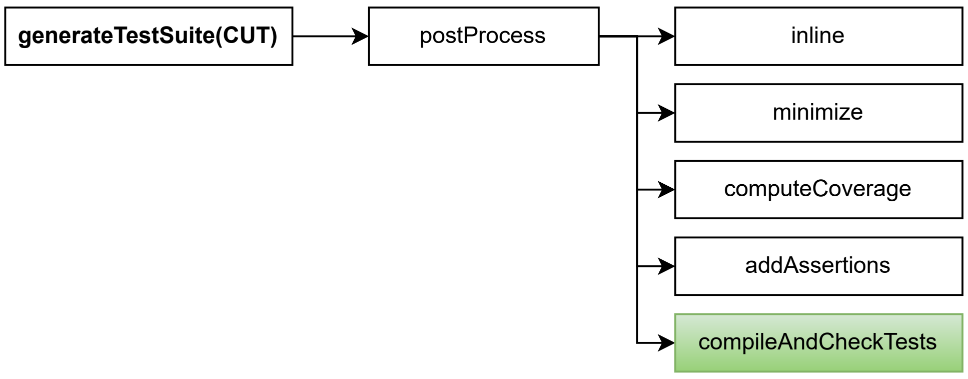


Figure 12: Overview of the Post Process flow. The white boxes are untouched, the green box is a process that was modified.

The inlining, minimizing, coverage and assertions are quickly explained in subsection 4.4.1 to give examples of the post-processing capabilities of EvoSuite, and the step with more modifications and refactoring, compileAndCheckTests, detailed in subsection 4.4.2.

4.4.1 EvoSuite Optimisations and Coverage

Inline This step is run to inline the constants and null references if possible. This is done to make the test more readable, by removing variables that simply hold an int and or a string that is used only one time later.

Listing 14: Inline processing example

```
// before inline processing
ClassUsingString classUsingString0 = new ClassUsingString();
String string0 = "EXAMPLE";
classUsingString0.foo(string0);

// after inline processing
ClassUsingString classUsingString0 = new ClassUsingString();
classUsingString0.foo("EXAMPLE");
```

Minimize When given a TestSuite with 300 TestCases, it would be hard for a developer to maintain all those tests during a regression step. Probably, some tests are covering the exact same lines of code, and are therefore not needed. A TestSuite Minimization step can be undertaken here to reduce the number of TestCases in the TestSuite without reducing the global coverage. While an optimal TestSuite Minimization is an NP-complete problem [82, 83], a good enough greedy solution exists, and is implemented in EvoSuite by checking which TestCase cover which goals, keeping the essential ones and removing redundant tests covering the same goals.

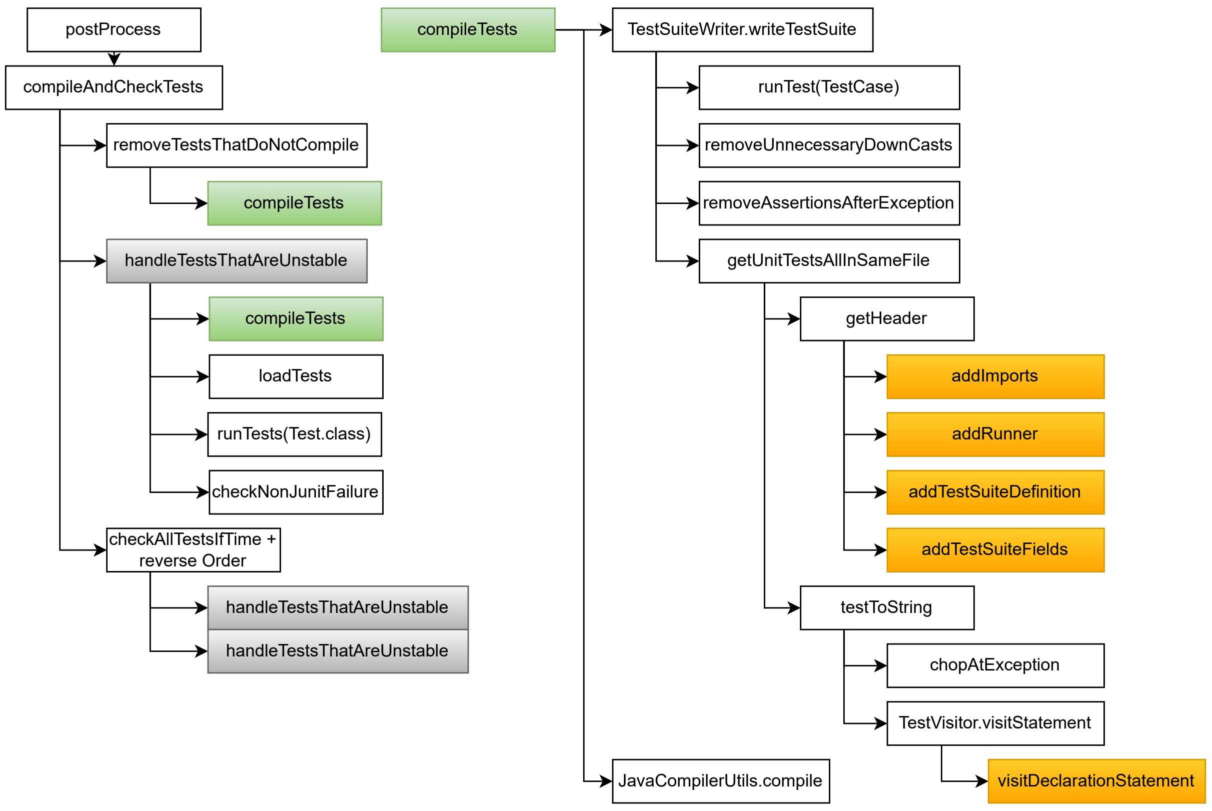
Compute Coverage As the stages of inlining and minimization might not be completed based on the time-budget allocated to their phase, or arguments-parameters given to the program, a final computation of the coverage is done to gather statistics on the covered goals of the TestSuite.

Assertions During this stage, each TestCase is executed in EvoSuite with different observers to gather information and execution traces about the evolution of the value of each variable during the test. The process is poorly written in EvoSuite source code (1 function of 700 lines), and too long to understand which modification would be needed to include a Spring Assertions Generator that would be able to reuse this framework to add ResultMatcher automatically.

One solution to generate the matchers would be to execute a Spring TestCase until getting the MvcResult, and then generate randomly many possible matchers, execute them one by one to see which one asserts correctly, and keep the ones that do not trigger exceptions.

4.4.2 Compilation and Check

As shown in Figure 13, the steps of compiling and checking the tests is quite repetitive. Indeed, a single test that is correct will be executed 4 times as a TestCase, compiled 4 times and executed by JUnit 3 times.


postProcess
compileAndCheckTests
removeTestsThatDoNotCompile
compileTests
handleTestsThatAreUnstable
compileTests
loadTests
runTests(Test.class)
checkNonJunitFailure
checkAllTestsIfTime + reverse Order
handleTestsThatAreUnstable
handleTestsThatAreUnstable
compileTests
TestSuiteWriter.writeTestSuite
runTest(TestCase)
removeUnnecessaryDownCasts
removeAssertionsAfterException
getUnitTestsAllInSameFile
getHeader
addImports
addRunner
addTestSuiteDefinition
addTestSuiteFields
testToString
chopAtException
TestVisitor.visitStatement
visitDeclarationStatement
JavaCompilerUtils.compile
Figure 13: Detailed flow of compileAndCheckTests stage. The white and grey boxes are untouched, the grey and green are used to show the same method, and the orange blocks are modified.

The first step is to remove TestCases that do not compile. If for some reason, a TestSuite containing a single TestCase cannot be compiled, then it is removed from the list of tests to add. This should not happen very often, and could be due to an undetected needed import or missing field for the TestSuite (@MockBean for example).

Secondly, the compiled tests are run one by one. They are loaded and executed by a JUnit runner and the ones that fail during execution related to an assertion of the test itself are marked as unstable. Failing due to any other exceptions makes the test being removed entirely. Those are runtime exceptions that are not planned by the TestCase itself, and therefore not acceptable as TestCases. After this stage, all TestCases are individually compilable and either unstable or not.

Finally, the TestCases are put all together in a single TestSuite and again compiled, run, and checked for instability. As each test is individually compilable, putting them all together is not a problem for compilation. However, there is a need to separate the normal TestCases from the ones using Spring as they do not use the same Runner. If an EvoSuiteSpringRunner supporting normal and Spring TestCases were implemented, this separation would not be needed. This is explained in the subsection 4.5.

The order of the TestCases in the TestSuite is then reversed to try to detect test dependencies.

4.5 Write Tests

This stage details the steps needed for writing a TestSuite to a correct Java file, including the modifications brought to support Spring.

The first thing to do is to run the TestCase in EvoSuite VM. This is used to gather the execution trace that stores exceptions, stack traces, or other information needed later on. Then 2 steps of cleaning are done to prevent some issues: unnecessary downcasts are removed as they could break ("Avoid downcasts that could break" [84]), and assertions after the first exception are removed ("Sometimes some timeouts lead to assertions being attached to statements related to exceptions. This is not currently handled, so as a workaround let's try to remove any remaining assertions" [85]).

Afterwards, the header of the TestSuite can be created. The header contains the package, the imports, the runner definition as well as the TestSuite class declaration including the fields of the TestSuite. The body of the TestSuite will be each TestCase one by one put as a text, called the rendering of the TestCase.

4.5.1 Imports

The imports are gathered in 2 different ways: the pre-build imports and the runtime imports. Pre-build imports are known beforehand by EvoSuite based on the Runner that is going to be used. JUnit 3, 4 or 5 or Spring runners have different required imports, but they are known and therefore stored in the source code of EvoSuite. Runtime imports are the ones that require more work, as they are not known before and depend exclusively on the Variables used in the Statements that compose the TestCase. EvoSuite finds them by checking the type of each Variable used in the TestCase and checking if they are in the current package. If not, they are imported.

The modification for the imports to support Spring is to add the necessary imports for the SpringRunner, as well as the @Autowired annotation and check the fields of the TestSuite class for any @MockBean. One runtime import was not found by EvoSuite, related to an Enum coming from Spring. EvoSuite checks the parameter types of the methods, as well as the exact type provided at runtime. During the visit of the MethodStatement parameters, one case was not adding the exact type of the variable used, and therefore, it was not put in the list of classes used in the test, so not imported afterwards. A fix was done for this specific case in the `TestCodeVisitor.getParameterString` function is called to render the parameters of a method and, at the same time, record the parameters' type.

4.5.2 Runner

The TestSuite can be implemented with 2 runners: the EvoSuiteRunner or the SpringRunner. The EvoSuiteRunner takes a few parameters coming from the starting parameter settings of EvoSuite. It then also needs to have the ES_Scaffolding (see subsection 4.5.3) put in place at that point. Making a SpringRunner supporting the same parameters would take more knowledge of EvoSuite behaviour to be sure to make it work.

As the tests using Spring are mostly handled by the Spring framework anyway, it was decided to only put the SpringRunner for now, and not the ES_Scaffolding. This introduces some uncertainty about the executability of the TestCases because they were created and run in the EvoSuite VM which takes into account the same parameters that the ES_Scaffolding put in place for the execution by JUnit. They could be added back, by making a special EvoSuiteSpringRunner that would merge the EvoSuiteRunner and SpringRunner and support the ES_Scaffolding.

When using the SpringRunner, it is defined along with the annotation @WebMvcTest(CUT.class), and the imports are added as well.

4.5.3 TestSuite Definition

Here the class for the TestSuite is defined with its extensions. If using the EvoSuiteRunner, then usually the ES_Scaffolding is added as an extension, otherwise it is defined as a normal public class.

ES_Scaffolding The Scaffolding defines a base class for the TestSuite, implementing @Before, @After, @BeforeAll and @AfterAll methods, used to initialize EvoSuite runtime, set up a sandbox, set some system property settings, load and instrument some classes of the SUT, and reset them after each test.

4.5.4 TestSuite fields

A major change is the addition of fields to the TestSuite. Those are usually quite common in any handmade JUnit Test, written by developers. They are used as placeholders for setup before the TestSuite starts or before each TestCase, in order to not repeat the definition of the variable.

In EvoSuite, this is the opposite, there are no fields in the TestSuite. Each TestCase being independent, those setup variables are declared in the TestCases themselves, preventing any leakage from test to test. It looks a bit more cumbersome, but maintains independence.

Using Spring however, we need to add a few fields. The first one which is always added is the @Autowired MockMvc, as the whole application works with it. After that, the necessary @MockBeans are also added. Those are found as explained in subsection 4.2.2.

Adding fields to the TestSuite might introduce dependencies between the tests, and should therefore be done with care.

4.5.5 TestToString

The part of writing a TestCase to string, the rendering, is done mainly by EvoSuite.

The first step is to cut the TestCase after the 1st exception encountered, as that part is not going to be executed anyway. This is a small optimisation done by EvoSuite for the rendering, requiring more than simply the list of Statements to render, but also their execution result.

The second step is to actually transform each Statement into its code textual representation. This is done using the visitor design pattern, splitting the rendering details from the Statement class itself. The TestCodeVisitor class is used to implement all the rendering of each Statement, and does the rendering of each statement one by one. The complex work of preventing name clashes for the variables but also for the classes is done here, as well as the generation of try-catch blocks for exceptions.

As the DeclarationStatement was added to the list of Statements, its rendering should also be defined here. This is implemented as explained in subsection 4.3.2.1.

4.6 Optimisation on Spring call

Reuse RequestBuilder and ResultMatcher When generating the Spring statements as explained in subsection 4.3.1, a RequestBuilder, a MockMvc and ResultMatcher are always instantiated. When creating a second call to MockMvc perform, the 3 objects would be recreated. However, the ResultMatcher could be reused as is to match another result of the MockMvc#perform, while the RequestBuilder could be reused to call again the same request. The MockMvc object itself must be instantiated only once for now, due to the nature of the injection.

Only one MockMvc Instantiating more than one MockMvc (let's say i objects, $i > 1$) objects would result in "mockMvc0" and "mockMvci" variable names used, but only a single "mockMvc0" injected in the TestSuite. The easy solution here is to force using only a single MockMvc object in a TestCase, which will result in a single "mockMvc0" variable used. This is checked easily as adding a spring call is done in a deterministic way, and only a single part of the code adds that injection.

A better solution to manage injections would be to have a TestSuiteManager in EvoSuite that could record effectively the injected fields, and therefore provide them to a TestCase. In doing so, the injection in the TestCase could simply be an assignment from the TestSuite field to a local TestCase variable (`ClassName testCaseVarName = testSuiteVarName;`) which would solve the problem of multiple declarations of the same object under different variable names.

The TestSuiteManager is not implemented at all by EvoSuite as, in theory, each TestCase should be self-sufficient. Adding this would take a lot of time and refactoring to support the concept of TestSuite itself, but it would make the tool more easily readable and customizable. This would be an important feature for future work on EvoSuite.

An example of a generated TestSuite is given below (without the imports).

Listing 15: Generated TestSuite example

```
@RunWith(SpringRunner.class)
@WebMvcTest(OwnerController.class)
public class OwnerControllerTest {

    @Autowired
    private MockMvc mockMvc0;

    @MockBean
    private OwnerRepository owners;

    @Test
    public void testGeneratedSuccess(){
        // first call
        MockHttpServletRequestBuilder mockHttpServletRequestBuilder0 = ...
            SmockRequestBuilder.request(HttpMethod.GET, "/owners/0");
        // injection of: MockMvc mockMvc0;
        ResultActions resultActions0 = ...
            mockMvc0.perform(mockHttpServletRequestBuilder0);
        MvcResult mvcResult0 = resultActions0.andReturn();
        StatusResultMatchers statusResultMatchers0 = ...
            MockMvcResultMatchers.status();
        ResultMatcher resultMatcher0 = statusResultMatchers0.is(200);
        resultMatcher0.match(mvcResult0);

        // second call
        MockHttpServletRequestBuilder mockHttpServletRequestBuilder1 = ...
            SmockRequestBuilder.request(HttpMethod.GET, "/owners/1");
        // reuse of mockMvc0
        ResultActions resultActions1 = ...
            mockMvc0.perform(mockHttpServletRequestBuilder1);
        MvcResult mvcResult1 = resultActions1.andReturn();
        // reuse of resultMatcher0
        resultMatcher0.match(mvcResult1);
    }
}
```

5 Evaluation

The results are comparing two versions of EvoSuite:

1. EvoSuite version 1.2.1-SNAPSHOT, hereafter called **EvoSuiteV1**.

   This is the latest non-released version of EvoSuite, coming from the master branch on GitHub. The latest commit taken into account is 1948d7639 (short hash), dating from the 11 of July 2023. Source code on GitHub [77].

2. EvoSuite version 1.2.2-SNAPSHOT, hereafter called **EvoSpring**.

   This version includes all the modifications related to this project, on the GitHub fork. The latest commit taken into account is e6fb5cfd0 (short hash), dating from the 10 of April 2024. Source code on GitHub [78].

The term EvoSuite is used for topics related to both versions, that apply without loss of generality to EvoSuiteV1 and to EvoSpring.

The first examined results are individual TestCases. They are the first and most important blocs of the EvoSuite generation. Then, generated TestSuites are reviewed as they will be the final output of the tool. Finally, coverage increase is evaluated.

The evaluation was done on a MacBook Pro, with an Apple M1 Pro chip, 32 GB of RAM, running macOS Sonoma version 14.3.1, using IntelliJ IDEA 2023.3.5 as an IDE and the build system JDK Zulu 8.74.0.17-CA-macos-aarch64 for Java 1.8.0_392.

5.1 PetClinic application context

For most of the evaluation, the Spring application called PetClinic is used. "The Spring PetClinic is a sample application designed to show how the Spring stack can be used to build simple, but powerful database-oriented applications." [86]. PetClinic is composed of multiple entities, controllers and views to handle the management of a clinic for pets. Each Pet has an Owner, and can go to the Vet to do a medical Visit.

Here is a part of the code that is of interest to provide context about the application, mainly the OwnerController to help understand how it works with Spring.

Listing 16: OwnerController sample

```java
package org.ditria.samples.petclinic.owner;

public class Owner {
    private Integer id;
    private String firstName;
    private String lastName;
    private String address;
    private String city;
    private Set<Pet> pets;

    // getters and setters
}

@Controller
public class OwnerController {
    private OwnerRepository owners;

    public OwnerController(OwnerRepository clinicService) {
        this.owners = clinicService;
    }

    @InitBinder
    public void setAllowedFields(WebDataBinder dataBinder) {
        dataBinder.setDisallowedFields("id");
    }

    @GetMapping("/owners/new")
    private String initCreationForm(Map<String, Object> model) {
        model.put("owner", new Owner());
        return "owners/createOrUpdateOwnerForm";
    }

    @GetMapping("/owners")
    public String processFindForm(Owner owner, BindingResult result, ...
        Map<String, Object> model) {
        // allow parameterless GET request for /owners to return all records
        if (owner.getLastName() == null) {
            owner.setLastName(""); // empty string == broadest search
        }

        // find owners by last name
        Collection<Owner> results = owners.findByLastName(owner.getLastName());
        if (results.isEmpty()) { // no owners found
            result.rejectValue("lastName", "notFound", "not found");
            return "owners/findOwners";
        } else if (results.size() == 1) { // 1 owner found
            owner = results.iterator().next();
            return "redirect:/owners/" + owner.getId();
        } else { // multiple owners found
            model.put("selections", results);
            return "owners/ownersList";
        }
    }

    @GetMapping("/owners/{ownerId}/edit")
    public String initUpdateOwnerForm(@PathVariable("ownerId") int ownerId, ...
        Model model) {
        Owner owner = this.owners.findById(ownerId);
        model.addAttribute(owner);
        return "owners/createOrUpdateOwnerForm";
    }

    @PostMapping("/owners/{ownerId}/edit")
    public String processUpdateOwnerForm(@Valid Owner owner, BindingResult ...
        result, @PathVariable("ownerId") int ownerId) {
        if (result.hasErrors()) {
            return "owners/createOrUpdateOwnerForm";
        } else {
            owner.setId(ownerId);
            this.owners.save(owner);
            return "redirect:/owners/{ownerId}";
        }
    }
}
```

The Owner class is simply a data class, holding information about the owner, like its first and last name, address, and a list of Pets that he owns.

The OwnerController class is the web controller (as shown by the @Controller annotation) that will handle the requests. It has a constructor that simply sets the internal repository of owners, which is an abstraction of the database. The constructor shows a typical Constructor-based DI pattern used in Spring for Controller with Repository. Four methods are HandlerMethods, three of them "get requests", annotated by the `@GetMapping`, and one is a "post request". Even if the `initCreationForm` is private, it can be used by Spring, but not by EvoSuite without reflection. The method `initUpdateOwnerForm` is used with the route `"/owners/{ownerId}/edit"`, which shows that URL can have variable replacement using the @PathVariable annotation. The method `processFindForm` shows that the request can get a complex object (`Owner`) as a parameter, on which the `lastName` can be set for example. At the same time, the method shows a complexity of 2*3=6 paths, 2 paths for the first if with hidden else, and 3 paths for the if-elsif-else. All those paths should be covered by tests, depending on the value of `lastName` and the `owners` repository.

One thing that can be seen is that those functions use two special parameters from Spring: the bindingResult and the modelMap. Both those variables are actually provided by Spring framework, and not by the user. The result comes from the server-side validation process, during which the parameters of the request are checked. For example, the handler `processUpdateOwnerForm` requests a @Valid Owner, which is therefore checked beforehand by Spring. Any error is reported in the bindingResult. The modelMap is an output variable, holding the model that the handler has to set and to give to the view to be displayed. The important point is that those two parameters are provided by Spring, so when generating tests, it would be hard to emulate them correctly without running Spring directly. This situation can be seen in Listing 17 for example, where the result and model map are generated with poor meaning and no proper functionality.

5.2 Examples of generated TestCase

TestCases can be generated as part of the unit testing process of EvoSuite, by providing in the test folder of EvoSuite a basic Spring Application without its configuration. If the configuration was provided, it would be applied to every test in EvoSuite and conflict with EvoSuite's own configuration. Without the complete configuration, the Spring App cannot be loaded properly, but it can be used to see whether generated TestCases make sense or not.

5.2.1 Tests generated by EvoSuiteV1

Listing 17: EvoSuiteV1: test of OwnerController.processFindForm

```
OwnerRepository ownerRepository0 = mock(OwnerRepository.class, new ...
    ViolatedAssumptionAnswer());
doReturn(Collections.EMPTY_LIST).when(ownerRepository0).findByLastName(anyString());
OwnerController ownerController0 = new OwnerController(ownerRepository0);
Owner owner0 = new Owner();
DirectFieldBindingResult directFieldBindingResult0 = new ...
    DirectFieldBindingResult((Object) null, "Q2b`*j6Q");
ownerController0.processFindForm(owner0, directFieldBindingResult0, ...
    (Map<String, Object>) null);
```

Listing 18: EvoSuiteV1: test of OwnerController.processCreationForm

```
OwnerRepository ownerRepository0 = mock(OwnerRepository.class, new ...
    ViolatedAssumptionAnswer());
OwnerController ownerController0 = new OwnerController(ownerRepository0);
Object object0 = new Object();
BeanPropertyBindingResult beanPropertyBindingResult0 = new ...
    BeanPropertyBindingResult(object0, "==================");
ownerController0.processCreationForm((Owner) null, beanPropertyBindingResult0);
```

Listing 19: EvoSuiteV1: test of OwnerController.initCreationForm

```
OwnerRepository ownerRepository0 = mock(OwnerRepository.class, new ...
    ViolatedAssumptionAnswer());
OwnerController ownerController0 = new OwnerController(ownerRepository0);
ExtendedModelMap extendedModelMap0 = new ExtendedModelMap();
String string0 = ownerController0.initCreationForm(extendedModelMap0);
```

Listings 17, 18 and 19 are examples of TestCases generated by EvoSuiteV1. They clearly show the mocked repository, the construction of the controller and of the parameters needed for the functions called on the controller. No assertion is made at this stage of the generation. The constant values that are used are quite random and could trigger some unexpected behaviour. However, they mainly trigger Null-Pointer Exceptions (NPEs) which is not very useful for the coverage of the class.

5.2.2 Tests generated by EvoSpring

Listing 20: EvoSpring: test of OwnerController.initCreationForm

```
MockHttpServletRequestBuilder mockHttpServletRequestBuilder0 = ...
    SmockRequestBuilder.request(HttpMethod.GET, "/owners/new");
// injection: MockMvc mockMvc0;
ResultActions resultActions0 = mockMvc0.perform(mockHttpServletRequestBuilder0);
MvcResult mvcResult0 = resultActions0.andReturn();
StatusResultMatchers statusResultMatchers0 = MockMvcResultMatchers.status();
ResultMatcher resultMatcher0 = statusResultMatchers0.is(200);
resultMatcher0.match(mvcResult0);
```

Listing 21: EvoSpring: test of OwnerController.processCreationForm

```
MockHttpServletRequestBuilder mockHttpServletRequestBuilder0 = ...
    SmockRequestBuilder.request(HttpMethod.POST, "/owners/new");
// injection: MockMvc mockMvc0;
ResultActions resultActions0 = mockMvc0.perform(mockHttpServletRequestBuilder0);
MvcResult mvcResult0 = resultActions0.andReturn();
StatusResultMatchers statusResultMatchers0 = MockMvcResultMatchers.status();
ResultMatcher resultMatcher0 = statusResultMatchers0.is(200);
resultMatcher0.match(mvcResult0);
```

Listing 22: EvoSpring: test of OwnerController.initFindForm

```
MockHttpServletRequestBuilder mockHttpServletRequestBuilder0 = ...
    SmockRequestBuilder.request(HttpMethod.GET, "/owners/find");
// injection: MockMvc mockMvc0;
ResultActions resultActions0 = mockMvc0.perform(mockHttpServletRequestBuilder0);
MvcResult mvcResult0 = resultActions0.andReturn();
StatusResultMatchers statusResultMatchers0 = MockMvcResultMatchers.status();
ResultMatcher resultMatcher0 = statusResultMatchers0.is(200);
resultMatcher0.match(mvcResult0);
```

Listing 23: EvoSpring: test of OwnerController.processFindForm

```
MockHttpServletRequestBuilder mockHttpServletRequestBuilder0 = ...
    SmockRequestBuilder.request(HttpMethod.GET, "/owners");
// injection: MockMvc mockMvc0;
ResultActions resultActions0 = mockMvc0.perform(mockHttpServletRequestBuilder0);
MvcResult mvcResult0 = resultActions0.andReturn();
StatusResultMatchers statusResultMatchers0 = MockMvcResultMatchers.status();
```

```
ResultMatcher resultMatcher0 = statusResultMatchers0.is(200);
resultMatcher0.match(mvcResult0);
```

Listings 20, 21, 22 and 23 show examples of TestCases generated by EvoSpring for some request handlers of the OwnerController.

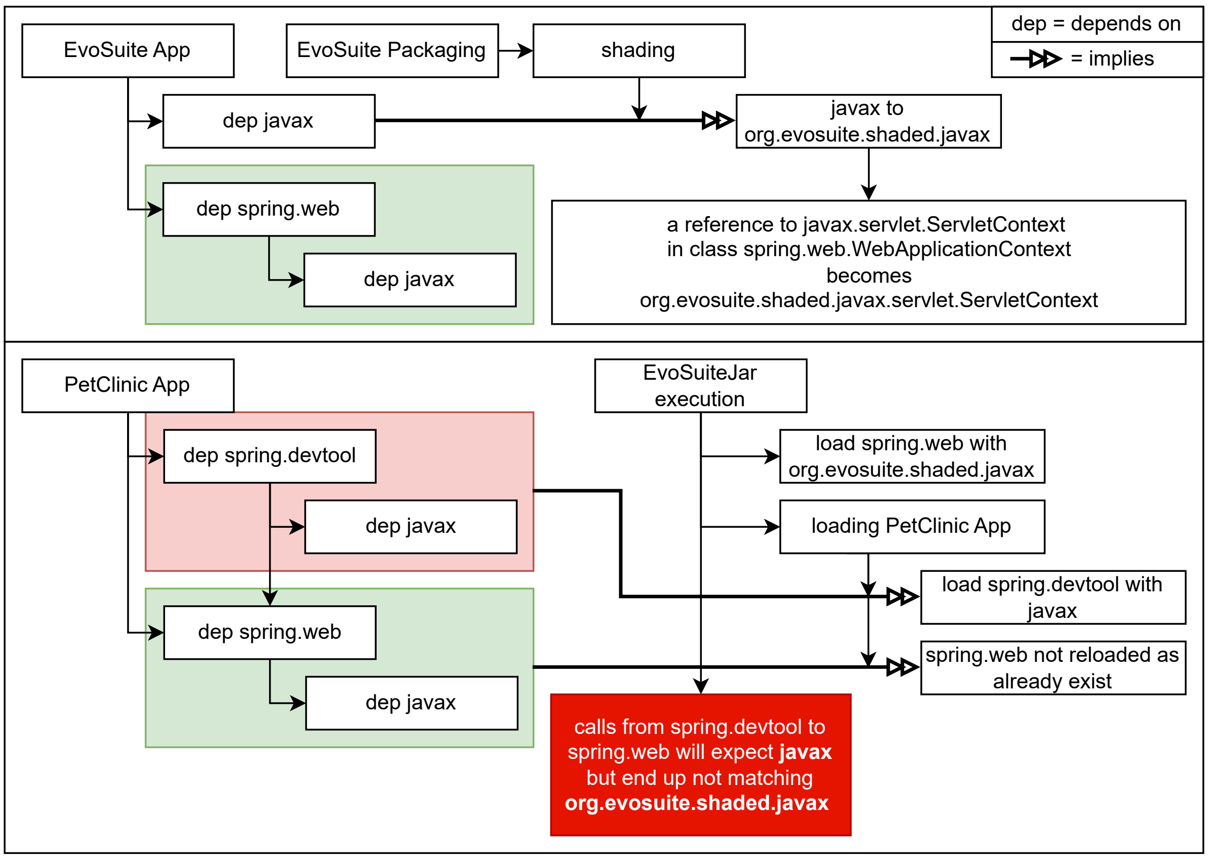
5.3 Examples of generated TestSuite

When executing a Spring TestCase as part of the unit testing process in EvoSuite, it is likely to fail because the Spring context can not entirely be loaded, and executing the MockMvc perform call is unlikely to succeed. When this happens, the TestCase will be annotated as unstable, and when its rendering is done in a TestSuite, it will be chopped at the first exception, and put in a try-catch to compare the expected exception with the actual one.

5.3.1 EvoSuite as a JAR application

To correctly generate a TestSuite, EvoSuite should be used as a JAR application and run as an external tool on a Spring Application. However, due to EvoSuite's complexity, the packaged JAR app needs to include part of the Spring library, but not its original version, but with a different name. Doing so avoids conflict of multi-version declaration of the same class, but at the same time, might prevent their correct usage. Indeed, during the packaging process, some packages are replaced by org.evosuite.shaded.package, and when Spring from the PetClinic application try to execute some Spring functionalities that are calling inside EvoSuite shaded library, they will fail to locate the correct class.

A concrete example is given in Figure 14. EvoSuite does a renaming of the javax package to org.evosuite.shaded.javax, on which spring.web also depends. At execution time, EvoSuite is loaded with the shaded javax, and spring.web. When loading the PetClinic application, the spring.devtool, a dependency not used in EvoSuite but which depends on spring.web, will be loaded with javax, but not reloading spring.web, as already loaded by EvoSuite. A call from spring.devtool to spring.web, expecting a javax, will fail because spring.web will use the shaded javax, making a mismatch of class.



EvoSuite App
EvoSuite Packaging
shading
dep = depends on
= implies
dep javax
javax to org.evosuite.shaded.javax
dep spring.web
dep javax
a reference to javax.servlet.ServletContext in class spring.web.WebApplicationContext becomes org.evosuite.shaded.javax.servlet.ServletContext
PetClinic App
EvoSuiteJar execution
dep spring.devtool
load spring.web with org.evosuite.shaded.javax
dep javax
loading PetClinic App
load spring.devtool with javax
dep spring.web
spring.web not reloaded as already exist
dep javax
calls from spring.devtool to spring.web will expect javax but end up not matching org.evosuite.shaded.javax

Figure 14: Shading override by EvoSuite JAR in Spring PetClinic.

While this is failing the full operation of EvoSuite as a standalone JAR, correctly handling this is out of the scope of the project, as it requires a much deeper understanding of the choice of EvoSuite shading. This issue has not been solved and poses a big limitation to the project application. To try to bypass this issue, EvoSpring is not packaged (so no shading is applied), and then used as a Java class and run on the command line, without much success either.

5.3.2 TestSuite generated by EvoSuiteV1

Listing 24: EvoSuiteV1: TestSuite of OwnerController

```
@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism = true, ...
    useVFS = true, useVNET = true, resetStaticState = true, ...
    separateClassLoader = true)
public class OwnerController_ESTest extends ...
    OwnerController_ESTest_scaffolding {
  @Test(timeout = 4000)
  public void test0()  throws Throwable  {
    OwnerRepository ownerRepository0 = mock(OwnerRepository.class, new ...
        ViolatedAssumptionAnswer());
    doReturn(Collections.EMPTY_LIST).when(ownerRepository0).findByLastName(anyString());
    OwnerController ownerController0 = new OwnerController(ownerRepository0);
    Owner owner0 = new Owner();
    DirectFieldBindingResult directFieldBindingResult0 = new ...
        DirectFieldBindingResult((Object) null, "Q2b`*j6Q");
    ownerController0.processFindForm(owner0, directFieldBindingResult0, ...
        (Map<String, Object>) null);
  }

  @Test(timeout = 4000)
  public void test1()  throws Throwable  {
    OwnerRepository ownerRepository0 = mock(OwnerRepository.class, new ...
        ViolatedAssumptionAnswer());
    OwnerController ownerController0 = new OwnerController(ownerRepository0);
    Object object0 = new Object();
    BeanPropertyBindingResult beanPropertyBindingResult0 = new ...
        BeanPropertyBindingResult(object0, "==================");
    // Undeclared exception!
    try {
      ownerController0.processCreationForm((Owner) null, ...
          beanPropertyBindingResult0);
      fail("Expecting exception: NullPointerException");
    } catch(NullPointerException e) {
      // no message in exception (getMessage() returned null)
      verifyException("org.ditria.samples.petclinic.owner.OwnerController", e);
    }
  }

  @Test(timeout = 4000)
  public void test2() throws Throwable  {
    OwnerRepository ownerRepository0 = mock(OwnerRepository.class, new ...
        ViolatedAssumptionAnswer());
    OwnerController ownerController0 = new OwnerController(ownerRepository0);
    ExtendedModelMap extendedModelMap0 = new ExtendedModelMap();
    String string0 = ownerController0.initCreationForm(extendedModelMap0);
    assertEquals("owners/createOrUpdateOwnerForm", string0);
  }
}
```

Listing 24 shows a full TestSuite (without the imports) generated by EvoSuiteV1. As expected, it is set up to use the EvoRunner and extends the ES_scaffolding. Additionally, each test is annotated with a timeout of four seconds, which is more than expected for unit tests. In test1, the try-catch is added by EvoSuite during the rendering due to an execution error of the call to the CUT method.

5.3.3 TestSuite generated by EvoSpring

Listing 25: EvoSpring: TestSuite of OwnerController

```
@RunWith(SpringRunner.class)
@WebMvcTest(OwnerController.class)
public class OwnerControllerSpringESTest {
    @Autowired
    private MockMvc mockMvc0;

    @MockBean
    private OwnerRepository ownerRepository;

    @Test(timeout = 4000)
    public void test0() throws Throwable  {
        MockHttpServletRequestBuilder mockHttpServletRequestBuilder0 = ...
            SmockRequestBuilder.request(HttpMethod.POST, "/owners/0/edit");
        // injection: MockMvc mockMvc0;
        try {
            mockMvc0.perform(mockHttpServletRequestBuilder0);
            fail("Expecting exception: NestedServletException");
        } catch(NestedServletException e) {
            // Handler processing failed; nested exception is ...
                java.lang.ExceptionInInitializerError
19              verifyException("org.springframework.web.servlet.DispatcherServlet", ...
                    e);
20          }
21      }
22
23      ...
24  }
```

Listing 25 gives a part of a TestSuite generated by EvoSpring for the OwnerController. As it can be seen in lines 18-19, the test is expected to fail with a NestedServletException, which is actually raised by the Spring framework when executing a call while the Spring context is not correctly loaded.

While this test might increase the coverage because the function is called, it does not provide any insight on the correctness of the function behaviour. To make it properly work, it requires the complete Spring context to be set up as in a real application, so this cannot be done during EvoSuite unit testing process. Assuming that EvoSpring works, we can compile it, and then run it against the real PetClinic application. This would allow the Spring setup to be loaded correctly and the call to MockMvc perform to work as expected in EvoSpring.

A fixed TestSuite is given in Listing 26. It was fixed by hand after being generated by EvoSuite, to simply comprises the TestCases generated above without additional checks, along with the same class definition and runners. It is used in subsection 5.4 to compute the coverage improvement of EvoSpring.

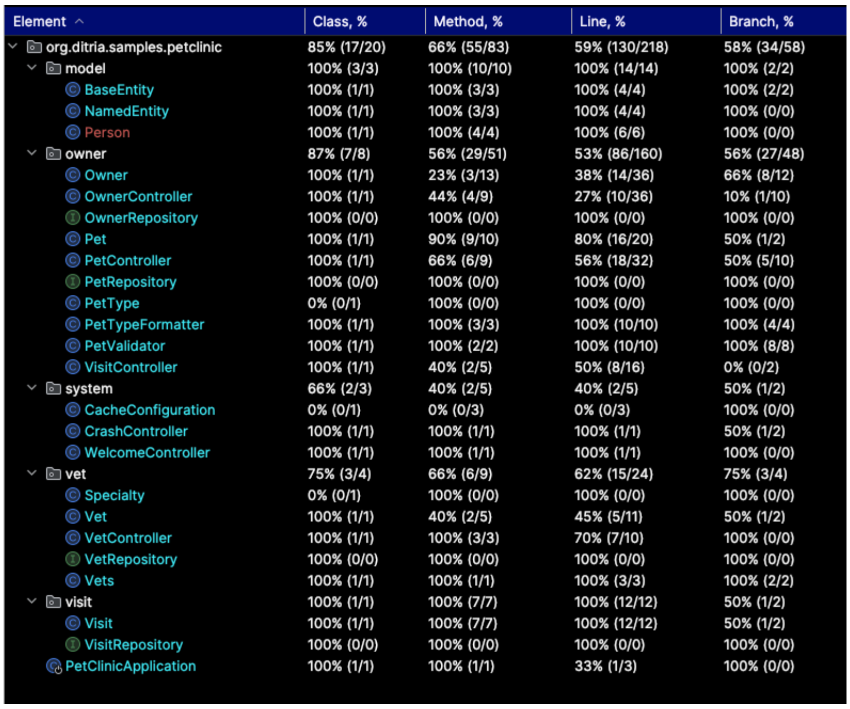
Listing 26: EvoSpring Fixed: TestSuite of OwnerController

```
@RunWith(SpringRunner.class)
@WebMvcTest(OwnerController.class)
public class OwnerController_SpringESTest {
    @Autowired
    private MockMvc mockMvc0;

    @MockBean
    private OwnerRepository ownerRepository;

    @Test(timeout = 4000)
    public void test0() throws Throwable {
        MockHttpServletRequestBuilder mockHttpServletRequestBuilder0 = ...
            SmockRequestBuilder.request(HttpMethod.GET, "/owners/new");
        // injection: MockMvc mockMvc0;
        ResultActions resultActions0 = ...
            mockMvc0.perform(mockHttpServletRequestBuilder0);
        MvcResult mvcResult0 = resultActions0.andReturn();
        StatusResultMatchers statusResultMatchers0 = ...
            MockMvcResultMatchers.status();
        ResultMatcher resultMatcher0 = statusResultMatchers0.is(200);
        resultMatcher0.match(mvcResult0);
    }

    @Test(timeout = 4000)
    public void test1() throws Throwable {
        MockHttpServletRequestBuilder mockHttpServletRequestBuilder0 = ...
            SmockRequestBuilder.request(HttpMethod.POST, "/owners/new");
        // injection: MockMvc mockMvc0;
        ResultActions resultActions0 = ...
            mockMvc0.perform(mockHttpServletRequestBuilder0);
        MvcResult mvcResult0 = resultActions0.andReturn();
        StatusResultMatchers statusResultMatchers0 = ...
            MockMvcResultMatchers.status();
        ResultMatcher resultMatcher0 = statusResultMatchers0.is(200);
        resultMatcher0.match(mvcResult0);
    }

    @Test(timeout = 4000)
    public void test2() throws Throwable {
        MockHttpServletRequestBuilder mockHttpServletRequestBuilder0 = ...
            SmockRequestBuilder.request(HttpMethod.GET, "/owners/find");
        // injection: MockMvc mockMvc0;
        ResultActions resultActions0 = ...
            mockMvc0.perform(mockHttpServletRequestBuilder0);
        MvcResult mvcResult0 = resultActions0.andReturn();
        StatusResultMatchers statusResultMatchers0 = ...
            MockMvcResultMatchers.status();
        ResultMatcher resultMatcher0 = statusResultMatchers0.is(200);
        resultMatcher0.match(mvcResult0);
    }

    @Test(timeout = 4000)
    public void test3() throws Throwable {
        MockHttpServletRequestBuilder mockHttpServletRequestBuilder0 = ...
            SmockRequestBuilder.request(HttpMethod.GET, "/owners");
        // injection: MockMvc mockMvc0;
        ResultActions resultActions0 = ...
            mockMvc0.perform(mockHttpServletRequestBuilder0);
        MvcResult mvcResult0 = resultActions0.andReturn();
        StatusResultMatchers statusResultMatchers0 = ...
            MockMvcResultMatchers.status();
        ResultMatcher resultMatcher0 = statusResultMatchers0.is(200);
        resultMatcher0.match(mvcResult0);
    }
}
```

5.4 Coverage comparison

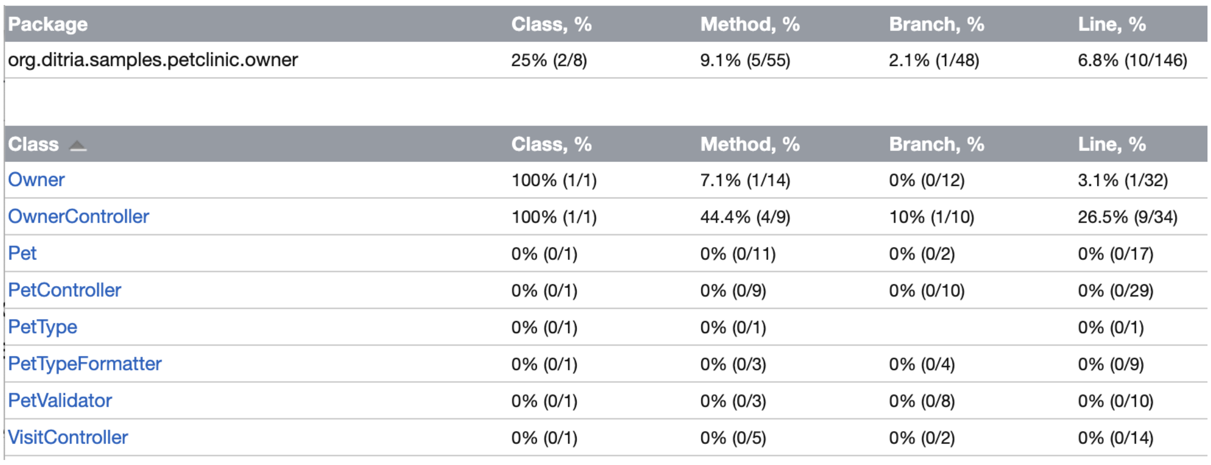
EvoSuiteV1 was run with coverage on the PetClinic application, and the coverage results are shown in Figure 15. We can see that some classes already have a very good coverage (the model and the entities like Owner or Pet) but that the controllers' coverage could be improved.

Figure 16 and Figure 17 show the coverage impact of the tests generated only from the OwnerController using the TestSuite from subsection 5.3.2 and subsection 5.3.3. Even with the limited number of TestCases in the TestSuite, the coverage of methods increased from 44% to 66%, and of lines from 25% to 47%. Using all the RequestMapping methods in the TestSuite would increase the coverage even more for no cost. Indeed, all the RequestMapping methods are given in the `Map<RequestMappingInfo, HandlerMethod>`, loaded during the Spring setup stage in subsection 4.2.2. EvoSpring did not use all of them because of the failure of execution. While the tests' assertions are not complete (only one matcher is used), it provides a base for the generated test by ensuring proper execution of Spring until that point.



Element	Class, %	Method, %	Line, %	Branch, %
org.ditria.samples.petclinic	85% (17/20)	66% (55/83)	59% (130/218)	58% (34/58)
model	100% (3/3)	100% (10/10)	100% (14/14)	100% (2/2)
BaseEntity	100% (1/1)	100% (3/3)	100% (4/4)	100% (2/2)
NamedEntity	100% (1/1)	100% (3/3)	100% (4/4)	100% (0/0)
Person	100% (1/1)	100% (4/4)	100% (6/6)	100% (0/0)
owner	87% (7/8)	56% (29/51)	53% (86/160)	56% (27/48)
Owner	100% (1/1)	23% (3/13)	38% (14/36)	66% (8/12)
OwnerController	100% (1/1)	44% (4/9)	27% (10/36)	10% (1/10)
OwnerRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
Pet	100% (1/1)	90% (9/10)	80% (16/20)	50% (1/2)
PetController	100% (1/1)	66% (6/9)	56% (18/32)	50% (5/10)
PetRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
PetType	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
PetTypeFormatter	100% (1/1)	100% (3/3)	100% (10/10)	100% (4/4)
PetValidator	100% (1/1)	100% (2/2)	100% (10/10)	100% (8/8)
VisitController	100% (1/1)	40% (2/5)	50% (8/16)	0% (0/2)
system	66% (2/3)	40% (2/5)	40% (2/5)	50% (1/2)
CacheConfiguration	0% (0/1)	0% (0/3)	0% (0/3)	100% (0/0)
CrashController	100% (1/1)	100% (1/1)	100% (1/1)	50% (1/2)
WelcomeController	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
vet	75% (3/4)	66% (6/9)	62% (15/24)	75% (3/4)
Specialty	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
Vet	100% (1/1)	40% (2/5)	45% (5/11)	50% (1/2)
VetController	100% (1/1)	100% (3/3)	70% (7/10)	100% (0/0)
VetRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
Vets	100% (1/1)	100% (1/1)	100% (3/3)	100% (2/2)
visit	100% (1/1)	100% (7/7)	100% (12/12)	50% (1/2)
Visit	100% (1/1)	100% (7/7)	100% (12/12)	50% (1/2)
VisitRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
PetClinicApplication	100% (1/1)	100% (1/1)	33% (1/3)	100% (0/0)

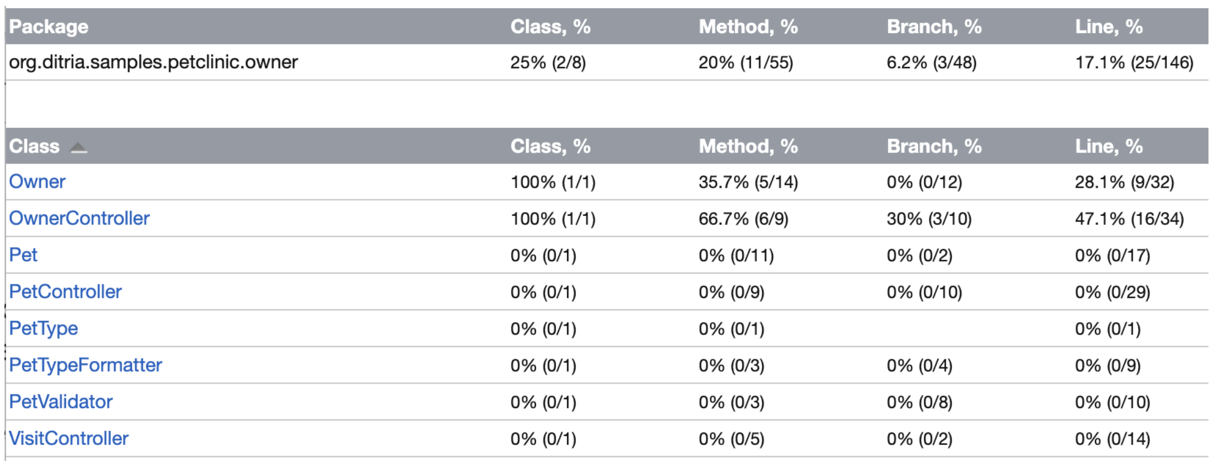
Figure 15: Coverage report of the PetClinic application using EvoSuiteV1



Package	Class, %	Method, %	Branch, %	Line, %
org.ditria.samples.petclinic.owner	25% (2/8)	9.1% (5/55)	2.1% (1/48)	6.8% (10/146)

Class	Class, %	Method, %	Branch, %	Line, %
Owner	100% (1/1)	7.1% (1/14)	0% (0/12)	3.1% (1/32)
OwnerController	100% (1/1)	44.4% (4/9)	10% (1/10)	26.5% (9/34)
Pet	0% (0/1)	0% (0/11)	0% (0/2)	0% (0/17)
PetController	0% (0/1)	0% (0/9)	0% (0/10)	0% (0/29)
PetType	0% (0/1)	0% (0/1)		0% (0/1)
PetTypeFormatter	0% (0/1)	0% (0/3)	0% (0/4)	0% (0/9)
PetValidator	0% (0/1)	0% (0/3)	0% (0/8)	0% (0/10)
VisitController	0% (0/1)	0% (0/5)	0% (0/2)	0% (0/14)

Figure 16: Coverage report of the OwnerController class using EvoSuiteV1



Package	Class, %	Method, %	Branch, %	Line, %
org.ditria.samples.petclinic.owner	25% (2/8)	20% (11/55)	6.2% (3/48)	17.1% (25/146)

Class	Class, %	Method, %	Branch, %	Line, %
Owner	100% (1/1)	35.7% (5/14)	0% (0/12)	28.1% (9/32)
OwnerController	100% (1/1)	66.7% (6/9)	30% (3/10)	47.1% (16/34)
Pet	0% (0/1)	0% (0/11)	0% (0/2)	0% (0/17)
PetController	0% (0/1)	0% (0/9)	0% (0/10)	0% (0/29)
PetType	0% (0/1)	0% (0/1)		0% (0/1)
PetTypeFormatter	0% (0/1)	0% (0/3)	0% (0/4)	0% (0/9)
PetValidator	0% (0/1)	0% (0/3)	0% (0/8)	0% (0/10)
VisitController	0% (0/1)	0% (0/5)	0% (0/2)	0% (0/14)

Figure 17: Coverage report of the OwnerController class using EvoSpring

5.5 Qualitative complexity and runtime analysis

An analysis of the complexity and runtime cost is done here. As seen in Figure 7, the major part of the modification is done in linear ways, at the same level as other EvoSuite calls.

Spring Support setup The Spring Support setup is done only once, as to load a single time the Spring context for a given controller. This is done to avoid reloading the context each time a TestCase is created, as this happens very often in EvoSuite. Every TestSuite generation strategy used starts with an initial population of around 50 TestCases, which are then mutated and copied. The mutation stage often counts 100 of parent-child generation, in which every TestCase implies the creation of a new TestCase. Reloading the Spring context each time, which takes about 5ms for a simple project, but up to 10-15s for more complex Spring applications, would have a massive slowdown impact on the mutation process, reducing the speed of search.

The generation of the Spring calls happens during the generation of calls on the CUT, at the same place it would generate a field, method or constructor call. As the process is linear, only the 3 objects (RequestBuilder, MockMvc perform, and a ResultMatcher) being created without recursion nor search loops, the impact of this modification is similar to the one of adding the other CUT calls.

Many ResultMatchers The addition of multiple ResultMatchers, as talked in subsection 4.3.1.2 has to be analysed a bit more carefully in order not to take too much time when not needed. The fact that EvoSuite adds the assertions only after the TestCases are generated shows that this has been thought about. Indeed, instead of searching for every possible assertion at TestCase generation time, it is done only once, when we know that the test is meaningful and covers some important part of the CUT. As the assertions do not bring any additional coverage whatsoever, they can entirely be omitted during the TestCase generation stage, and added after when a TestSuite has been generated. This is even more optimal as the generation of assertions happens only on a minimized TestSuite, with a low number of TestCases.

For now, only one ResultMatcher is added per TestCase at the same time the MockMvc perform call is done, in a deterministic way, so the generation complexity is similar to the one of adding a constructor method.

Post Process During the Post Process stage, as explained in subsection 4.4, and particularly shown in Figure 13, the TestCases executed in EvoSuite VM, and then compiled and executed as JUnit tests. Before the compilation, the search for the imports is linear with the number of variables added to the TestCase, which is almost constant by adding only 4-5 method calls from

Spring. Each Statement is analysed to gather the class used, and that needs to be imported. However, due to the way EvoSuite handles those needed imports, and as the test can be modified at each compilation pass, this search is done every time. For a single test, this analysis depends linearly on the number of statements and is done 4 times during the Post Process stage.

While this is linear in the number of statements, it could be optimised to reduce the repetition of the search with a better handling of searching multiple times in the same statement.

Another important cost that could appear is the execution time of the MockMvc perform call. While in pure unit testing, the call directly goes to methods of the CUT, using Spring MockMvc induces a cost before reaching the method of the controller. However, this is usually as fast as possible, as any delay here will make the use of Spring framework less favourable in the industrial context. Handling HTTP request dispatch should be as fast as possible.

An additional cost is incurred at runtime due to the execution of the compiled TestCases: the setup of Spring context for each compiled Test. Indeed, when the TestSuite is generated at the end of the Post Process stage, it is compiled and executed, at which point the Spring setup is completely reloaded in a new JUnit process to be executed out-of-context of EvoSuite. This might be an important cost, especially if the loading takes time. This increases the runtime by about $T * p$, with T being the Spring context load time, and p the number of paths of the controller. p can be computed as the sum of the number of branches in every controller handler method, which is the minimum number of tests that will be required to reach all the branches in the controller independently.

6 Discussion

6.1 Status of the program

The current state of the program does not allow running EvoSpring as a standalone application to properly generate tests on a Spring application. The Spring tests are generated at the unit test level into EvoSuite by copying the Spring application into EvoSuite in order to test its functionality. However, the following functionalities have been implemented:

- setup stage
  - identification if the CUT is a Spring controller
  - generation of Spring simpleTestSuite Java file, with its compilation and loading
  - loading of the Spring context using the simpleTestSuite
  - retaining MockMvc and the map of <RequestMappingInfo, HandlerMethod>
- TestCase generation stage
  - random generation of Spring statements based on RequestMappingInfo
  - guided generation of the RequestBuilder
  - call of MockMvc perform with MockMvc injection
  - simple generation of ResultMatcher
  - execution of the test locally as an EvoSuite TestCase
- TestSuite export stage
  - separate normal and Spring tests
  - writing to file

All the aforementioned steps are working and have been tested independently. They would also work together as long as one thing is correct: the Spring context loading. If the Spring context was loaded correctly, either in integration tests, or in the actual usage of EvoSuite onto a Spring application, then the generation of MockMvc tests would work. During unit tests, Spring can load a default base context with minimal configuration, making the execution of the generated test fail after the execution of the handler, due to a Spring context misconfiguration. This shows that with a minimal Spring context, the process of generating the MockMvc test works. However, while running with the full EvoSpring JAR application when testing in End-to-End (E2E) mode, the Spring context loading conflicts with the EvoSpring JAR file and fails to load correctly.

As explained in subsection 5.3.1, running the EvoSuite JAR for a given Spring Application (e.g. PetClinic) is more complex than expected due to shading, classpath shadowing, logger conflicts, and runtime class instrumentation. Fixing this conflicting configuration would allow to correct the loading of the Spring context, which grants a correct execution of the MockMvc perform call, a proper coverage fitness value of the TestCase when evaluating it, allowing its evolution with the others TestCases, and finally correctly adding the different matchers based on ExecutionResult during the TestSuite export, similarly to adding the assertions.

This task is quite complex and would require deeper knowledge of EvoSuite that could be given by the original authors of the tools.

6.2 On the simplicity of generated Spring TestCases

In subsection 5.2.2, the pattern of [RequestBuilder, MockMvc injection and call to perform, ResultMatcher and matching] is clearly visible. Additionally, there is no randomness nor complex recursive object generation in the tests generated and only a single matcher with the expected status result 200 is generated. This makes the tests look similar to each other and not complicated.

One could argue that there would be no need to use such a complex tool such as EvoSuite to generate those tests, and that, by simply reading the controller class, one could write those tests very easily. While this is true for those simple requests, having EvoSuite in the loop means that the dynamic search for coverage optimisation, the search for not-thought corner cases, the addition of resultMatchers based on concrete execution of the test, the test suite minimization and the execution within the loaded Spring context can also be added thanks to the integration within EvoSuite.

Some integration steps are still required to unlock the full potential of EvoSuite on Spring controllers. The @MockBeans used for the repository need to be configured, the handler parameters need to be used when generating the RequestBuilder to reach different paths in the handler method. Setting the correct values in those parameters and configurations can be done for free by EvoSuite once the support for those cases is implemented. Finding those values in complex cases could not be done without a more powerful analysis and search-engine that EvoSuite provides.

For example, the method `processFindForm` in Listing 16 has 6 individual paths. Generating the 6 different tests to cover each of the paths, and to be sure that they are covered, without using EvoSuite, would require analysing the code of the function, understanding the if-else branches, figuring out the objects that are important for a specific branch, searching for their values and setting them, generating those objects and their dependency, evaluating the return of the function and map it to Spring MvcResult, adding the assertions of the result value, and then putting all 6 tests into a test suite that can be compiled and executed without a developer fixing typos or missing imports.

All these steps are provided by EvoSuite, and reimplementing them by hand would be a misuse of performant tools that already exist. Implementing this support for Spring in EvoSuite allows extensions to support other REST API frameworks which work similarly to Spring MVC, like Quarkus [87] or Blade [88].

6.3 Improvements and Future work

6.3.1 Improve Spring dependency

The implementation done in subsection 4.2.2 to set up the Spring context and get a MockMvc object is minimal as it serves as a PoC. Some code is copied from Spring to break the chicken-and-egg problem of SpringContext-TestSuite. This should be kept up to date with the Spring framework in the future by directly calling some of the Spring methods implemented for the beans' detection setup. Even if this might seem like a hack as it might require injections, it will rely on Spring directly instead of copied code that might not work in future versions of Spring. This will as a side effect also include the support for @Autowired and Setter-based DI that was left out, as only the Constructor-based DI was copied into EvoSuite.

While this means that EvoSuite relies more on Spring, making it less generic, it also provides a much-improved support for that framework. Supporting a specific framework like this comes at the cost of less generic behaviour. However, all the concepts of genericness of EvoSuite are left untouched, and only the addition of Spring support is less generic. An idea would be to let the core of EvoSuite client as a project, and support for different frameworks could be added as EvoSuite-plugins that could simply be added at runtime depending on the detected framework.

6.3.2 TestSuite support in EvoSuite

The modifications added in 4.5.4 about the TestSuite imports, runner, definition and field of a TestSuite, are big incentives to create proper support for TestSuite in EvoSuite, including rendering, imports tracking, and private field usage in TestCases. While this seems to go against the principle of independence of TestCases in EvoSuite, this will improve a lot the separation of responsibility in the source code of EvoSuite and still allow the TestCases to be run independently by generating one TestSuite per TestCase if needed.

This future work is mostly concerning EvoSuite development, and less the work of this project itself. However, supporting more frameworks that needs mocking means that EvoSuite must be even more generic than its current state, and allow for more cases that happen in real-life unit testing of Java applications. This is a concern directly related to the support of mocking frameworks needed for generating tests using those frameworks.

6.3.3 EvoSuite as a JAR file

As explained in subsection 5.3.1, EvoSuite is packaged as a JAR application including its dependencies and shading some of them. However, adding the dependencies related to Spring made this packaged app fail when used on a Spring application. An important fix for the continuity of this project would be to resolve this shading problem. This should be done by deciding which package to shade according to the loading of the Spring context. Indeed, classes reused by Spring should be handled with care, as they might be reused outside EvoSuite, in the SUT. A better understanding of the Spring loading process could help as well to test EvoSuite in an internal way.

This would also allow the executions of the Spring TestCases in EvoSuite directly, so that the MockMvc#perform calls execute correctly. Once this is done, the ResultMatcher can also be generated during the same steps as Assertions, making the generated TestCase much more robust and meaningful for Spring.

7 Conclusion

The primary objective of the project was the increase of coverage of classes that required mocking. This was more refined to Spring controllers that need the MockMvc framework to be controlled in a meaningful way. The detection of the need for the mock for Spring as well as the implementation inducing an increase in coverage was done in this project.

The support for Spring MockMvc was successfully implemented in EvoSuite up to the TestCase generation and execution. The mockMvc object is correctly created and injected in the TestCases using the new implementation of `DeclarationStatement`. The rendering of TestCases as TestSuite using the Spring framework was achieved correctly as well. At the unit test level in EvoSuite, all these functionalities were tested correctly.

While the deployment of the new EvoSpring program as a JAR file worked, its execution on Spring applications failed due to a conflict of libraries during Spring context loading. The standalone mode failed at the Spring context loading stage, the JAR of EvoSpring was therefore not usable on Spring applications.

The results show an increase in coverage given a TestSuite generated while taking into account Spring, even if a few manual corrections had to be undertaken to execute the TestSuite.

The complexity of generating the tests did not increase more than linearly with the number of the few Statements added per test, while the execution runtime could increase drastically, based on the Spring context load time and the number of branches in the controller.

In conclusion, this project offers an implementation of Spring MockMVC support in the EvoSuite tool to automatically generate tests for systems that require mocking, showing an increase in the coverage of the controller and more meaningful tests, making them more reliable for use in real projects.

References

[1] Y. Lin, Y. S. Ong, J. Sun, G. Fraser, and J. S. 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*. Athens Greece: ACM, Aug. 2021, pp. 1068–1080. [Online]. Available: https://dl.acm.org/doi/10.1145/3468264.3468619

[2] EvoSuite, "AbstractMOSA - evosuite/client at master · EvoSuite/evosuite," Oct. 2021. [Online]. Available: https://github.com/EvoSuite/evosuite/blob/master/client/src/main/java/org/evosuite/ga/metaheuristics/mosa/AbstractMOSA.java#L139

[3] W. Haywood, "Autowired Annotation in Spring," Jul. 2023. [Online]. Available: https://www.linkedin.com/pulse/autowired-annotation-spring-william-haywood

[4] Spring, "MockBean (Spring Boot 3.2.4 API)," Apr. 2024. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/mock/mockito/MockBean.html

[5] ——, "PathVariable (Spring Framework 6.1.5 API)," Apr. 2024. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PathVariable.html

[6] ——, "RequestMapping (Spring Framework 6.1.5 API)," Apr. 2024. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html

[7] A. Obregon, "Using @RunWith and @SpringBootTest for Effective Unit Testing in Spring," Sep. 2023. [Online]. Available: https://medium.com/@AlexanderObregon/using-runwith-and-springboottest-for-effective-unit-testing-in-spring-856ba42fb4d

[8] Spring, "WebMvcTest (Spring Boot 3.2.4 API)," Apr. 2024. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/web/servlet/WebMvcTest.html

[9] ——, "Introduction to the Spring IoC Container and Beans :: Spring Framework," Nov. 2023. [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans/introduction.html

[10] ——, "HandlerMethod (Spring Framework 6.1.5 API)," Apr. 2024. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/method/HandlerMethod.html

[11] Wikipedia, "JUnit," Sep. 2023, page Version ID: 1176085079. [Online]. Available: https://en.wikipedia.org/w/index.php?title=JUnit&oldid=1176085079

[12] JUnit, "JUnit 5," Apr. 2024. [Online]. Available: https://junit.org/junit5/

[13] Spring, "MockMvcTest Framework," Apr. 2024. [Online]. Available: https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework.html

[14] ——, "Working with Spring Data Repositories :: Spring Data Commons," Apr. 2024. [Online]. Available: https://docs.spring.io/spring-data/commons/reference/repositories.html

[15] Wikipedia, "Boolean satisfiability problem," Dec. 2023, page Version ID: 1190847212. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Boolean_satisfiability_problem&oldid=1190847212

[16] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, ser. STOC '71. New York, NY, USA: Association for Computing Machinery, May 1971, pp. 151–158. [Online]. Available: https://dl.acm.org/doi/10.1145/800157.805047

[17] J. Pan, "Software Testing," *Dependable Embedded Systems*, Mar. 1999. [Online]. Available: http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/

[18] H. Zhu, L. Wei, M. Wen, Y. Liu, S.-C. Cheung, Q. Sheng, and C. Zhou, "MockSniffer: characterizing and recommending mocking decisions for unit tests," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, Jan. 2021, pp. 436–447. [Online]. Available: https://dl.acm.org/doi/10.1145/3324884.3416539

[19] A. Zerouali and T. Mens, "Analyzing the evolution of testing library usage in open source Java projects," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2017, pp. 417–421. [Online]. Available: https://ieeexplore.ieee.org/document/7884645

[20] W. Miller and D. Spooner, "Automatic Generation of Floating-Point Test Data," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 223–226, Sep. 1976, conference Name: IEEE Transactions on Software Engineering. [Online]. Available: https://ieeexplore.ieee.org/document/1702369

[21] Y. Zhang, W. Song, Z. Ji, Danfeng, Yao, and N. Meng, "How well does LLM generate security tests?" Oct. 2023, arXiv:2310.00710 [cs]. [Online]. Available: http://arxiv.org/abs/2310.00710

[22] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.294. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.294

[23] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security & Privacy*, vol. 3, no. 2, pp. 58–62, Mar. 2005, conference Name: IEEE Security & Privacy. [Online]. Available: https://ieeexplore.ieee.org/document/1423963

[24] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the Art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018, conference Name: IEEE Transactions on Reliability. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8371326

[25] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in *29th International Conference on Software Engineering (ICSE'07)*. Minneapolis, MN, USA: IEEE, May 2007, pp. 75–84, iSSN: 0270-5257. [Online]. Available: http://ieeexplore.ieee.org/document/4222570/

[26] W. Zheng, Q. Zhang, M. Lyu, and T. Xie, "Random unit-test generation with MUT-aware sequence recommendation," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: Association for Computing Machinery, Sep. 2010, pp. 293–296. [Online]. Available: https://dl.acm.org/doi/10.1145/1858996.1859054

[27] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, “GRT: Program-Analysis-Guided Random Testing (T),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 212–223. [Online]. Available: https://ieeexplore.ieee.org/document/7372010

[28] J. Satyabrata, “Random Testing in Software Testing,” Dec. 2021, section: Computer Subject. [Online]. Available: https://www.geeksforgeeks.org/random-testing-in-software-testing/

[29] P. Godefroid, N. Klarlund, and K. Sen, “DART: directed automated random testing,” in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI ’05. New York, NY, USA: Association for Computing Machinery, Jun. 2005, pp. 213–223. [Online]. Available: https://dl.acm.org/doi/10.1145/1065010.1065036

[30] S. Artzi, M. D. Ernst, A. Kiezun, C. Pacheco, and J. H. Perkins, “Finding the needles in the haystack: Generating legal test inputs for object-oriented programs,” *CSAIL Technical Reports*, vol. 056, Aug. 2006, accepted: 2006-09-05T16:31:22Z. [Online]. Available: https://dspace.mit.edu/handle/1721.1/33959

[31] P. McMinn, “Search-Based Software Testing: Past, Present and Future,” in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, Mar. 2011, pp. 153–163. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5954405

[32] Wikipedia, “Control-flow graph,” Mar. 2024, page Version ID: 1213624494. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Control-flow_graph&oldid=1213624494

[33] F. E. Allen, “Control flow analysis,” *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, Jul. 1970. [Online]. Available: https://dl.acm.org/doi/10.1145/390013.808479

[34] A. Arcuri, “It Does Matter How You Normalise the Branch Distance in Search Based Software Testing,” in *Verification and Validation 2010 Third International Conference on Software Testing*, Apr. 2010, pp. 205–214, iSSN: 2159-4848. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5477082

[35] J. Malburg and G. Fraser, “Combining search-based and constraint-based testing,” in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Lawrence, KS, USA: IEEE, Nov. 2011, pp. 436–439. [Online]. Available: http://ieeexplore.ieee.org/document/6100092/

[36] S. Vogl, S. Schweikl, and G. Fraser, “Encoding the certainty of boolean variables to improve the guidance for search-based test generation,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO ’21. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 1088–1096. [Online]. Available: https://dl.acm.org/doi/10.1145/3449639.3459339

[37] Y. Lin, J. Sun, G. Fraser, Z. Xiu, T. Liu, and J. S. Dong, “Recovering fitness gradients for interprocedural Boolean flags in search-based testing,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 440–451. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397358

[38] R. Al-Ekram and K. Kontogiannis, "Source code modularization using lattice of concept slices," in *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, Mar. 2004, pp. 195–203, iSSN: 1534-5351. [Online]. Available: https://www.researchgate.net/publication/4065402_Source_code_modularization_using_lattice_of_concept_slices

[39] A. Arcuri and X. Yao, "Search based software testing of object-oriented containers," *Information Sciences*, vol. 178, no. 15, pp. 3075–3095, Aug. 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025507005609

[40] Z. Zhou, Y. Zhou, C. Fang, Z. Chen, and Y. Tang, "Selectively Combining Multiple Coverage Goals in Search-Based Unit Test Generation," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3551349.3556902

[41] A. Gambi, G. Jahangirova, V. Riccio, and F. Zampetti, "SBST tool competition 2022," in *Proceedings of the 15th Workshop on Search-Based Software Testing*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 25–32. [Online]. Available: https://dl.acm.org/doi/10.1145/3526072.3527538

[42] J. Zhang, "Constraint Solving and Symbolic Execution," in *Verified Software: Theories, Tools, Experiments*, B. Meyer and J. Woodcock, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 4171, pp. 539–544, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-540-69149-5_59

[43] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011. [Online]. Available: https://dl.acm.org/doi/10.1145/1995376.1995394

[44] A. Sakti, Y.-G. Guéhéneuc, and G. Pesant, "Constraint-Based Fitness Function for Search-Based Software Testing," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, C. Gomes and M. Sellmann, Eds. Berlin, Heidelberg: Springer, 2013, pp. 378–385.

[45] A. Gotlieb, "Chapter Two - Constraint-Based Testing: An Emerging Trend in Software Testing," in *Advances in Computers*, A. Memon, Ed. Elsevier, Jan. 2015, vol. 99, pp. 67–101. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245815000340

[46] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos, "Symbolic search-based testing," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Nov. 2011, pp. 53–62, iSSN: 1938-4300. [Online]. Available: https://ieeexplore.ieee.org/document/6100119

[47] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, Sep. 2005. [Online]. Available: https://dl.acm.org/doi/10.1145/1095430.1081750

[48] J. Gosling, "Java (programming language)," Mar. 2024, page Version ID: 1216363238. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=1216363238

[49] ——, "What is Java and why do I need it?" Apr. 2024. [Online]. Available: https://www.java.com/en/download/help/whatis_java.html

[50] Spring, “SpringFramework,” Feb. 2024, page Version ID: 1206815437. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Spring_Framework&oldid=1206815437

[51] ——, “SpringHome,” Apr. 2024. [Online]. Available: https://spring.io/

[52] ——, “SpringWebMVC framework,” Apr. 2024. [Online]. Available: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html

[53] ——, “SpringBoot,” Apr. 2024. [Online]. Available: https://spring.io/projects/spring-boot

[54] Wikipedia, “List of unit testing frameworks,” Apr. 2024, page Version ID: 1216953665. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_unit_testing_frameworks&oldid=1216953665

[55] TestNG, “TestNG Documentation,” Apr. 2024. [Online]. Available: https://testng.org/

[56] Wikipedia, “TestNG,” Feb. 2024, page Version ID: 1206057792. [Online]. Available: https://en.wikipedia.org/w/index.php?title=TestNG&oldid=1206057792

[57] Mockito, “Mockito framework site,” Jul. 2023. [Online]. Available: https://site.mockito.org/

[58] Wikipedia, “Mockito,” Jul. 2023, page Version ID: 1167296787. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Mockito&oldid=1167296787

[59] EasyMock, “EasyMock,” Jul. 2023. [Online]. Available: https://easymock.org/

[60] jMock, “jMock - An Expressive Mock Object Library for Java,” Jul. 2023. [Online]. Available: http://jmock.org/

[61] JMockit, “The JMockit testing toolkit,” Jul. 2023. [Online]. Available: https://jmockit.github.io/

[62] Google, “gMock for Dummies,” Jul. 2023. [Online]. Available: http://google.github.io/googletest/gmock_for_dummies.html

[63] G. Fraser, “About EvoSuite,” Sep. 2021. [Online]. Available: https://www.evosuite.org/evosuite/

[64] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, “Testability transformation,” *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, Jan. 2004, conference Name: IEEE Transactions on Software Engineering. [Online]. Available: https://ieeexplore.ieee.org/document/1265732

[65] G. Fraser and A. Arcuri, “EvoSuite: Automatic test suite generation for object-oriented software,” in *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Szeged, Hungary, Sep. 2011, p. 5, journal Abbreviation: SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering Pages: 419 Publication Title: SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering. [Online]. Available: http://dx.doi.org/10.1145/2025113.2025179

[66] ——, “Whole Test Suite Generation,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, Feb. 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6152257/

[67] ——, “Automated Test Generation for Java Generics,” in *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering*, ser. Lecture Notes in Business Information Processing, D. Winkler, S. Biffl, and J. Bergsmann, Eds. Cham: Springer International Publishing, 2014, pp. 185–198.

[68] A. Arcuri, G. Fraser, and R. Just, “Private API Access and Functional Mocking in Automated Unit Test Generation,” in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Mar. 2017, pp. 126–137. [Online]. Available: https://ieeexplore.ieee.org/document/7927969

[69] X. Devroey, S. Panichella, and A. Gambi, “Java Unit Testing Tool Competition: Eighth Round,” in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. Seoul Republic of Korea: ACM, Jun. 2020, pp. 545–548. [Online]. Available: https://dl.acm.org/doi/10.1145/3387940.3392265

[70] S. Panichella, A. Gambi, F. Zampetti, and V. Riccio, “SBST Tool Competition 2021,” in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, May 2021, pp. 20–27. [Online]. Available: https://ieeexplore.ieee.org/document/9476243

[71] M. Ernst, “Randoop: Automatic unit test generation for Java,” Oct. 2023. [Online]. Available: https://randoop.github.io/randoop/

[72] C. Pacheco and M. D. Ernst, “Eclat : automatic generation and classification of test inputs,” Thesis, Massachusetts Institute of Technology, 2005, accepted: 2006-08-25T18:51:17Z Journal Abbreviation: Automatic generation and classification of test inputs. [Online]. Available: https://dspace.mit.edu/handle/1721.1/33855

[73] D. Ltd., “What is Diffblue Cover? Autonomous Java Unit Test Writing with AI for Code,” 2023. [Online]. Available: https://www.diffblue.com/products/

[74] ——, “Sandbox in Diffblue Documentation,” Mar. 2024. [Online]. Available: https://docs.diffblue.com/features/cover-plugin/writing-tests/diffblue-sandbox

[75] Oracle, “RetentionPolicy (Java Platform SE 8),” Apr. 2024. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/RetentionPolicy.html

[76] Spring, “SpringJUnit4ClassRunner (Spring Framework 6.1.5 API),” Apr. 2024. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/junit4/SpringJUnit4ClassRunner.html

[77] EvoSuite, “gitHub repository - EvoSuite/evosuite,” Apr. 2024, original-date: 2015-09-09T19:53:09Z. [Online]. Available: https://github.com/EvoSuite/evosuite

[78] J. Di Tria, “gitHub repository - JulienDiTria/evosuite,” Apr. 2024, original-date: 2023-12-13T13:38:49Z. [Online]. Available: https://github.com/JulienDiTria/evosuite

[79] Spring, “Dependencies :: Spring Framework,” May 2023. [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans/dependencies.html

[80] EvoSuite, “RandomInsertion evosuite/client/src/main/java/org/evosuite/testcase/mutation/RandomInsertion.java at master EvoSuite/evosuite,” Apr. 2024. [Online]. Available: https://github.com/EvoSuite/evosuite/blob/master/client/src/main/java/org/evosuite/testcase/mutation/RandomInsertion.java#L69

[81] Spring, “Mapping Requests :: Spring Framework,” Jan. 2024. [Online]. Available: https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html

[82] R. Singh and M. Santosh, “Test Case Minimization Techniques : A Review,” *International Journal of Engineering Research*, vol. 2, no. 12, Dec. 2013. [Online]. Available: https://www.researchgate.net/publication/265086128_Test_Case_Minimization_Techniques_A_Review

[83] A. Deneke, B. Gizachew Assefa, and S. Kumar Mohapatra, “Test suite minimization using particle swarm optimization,” *Materials Today: Proceedings*, vol. 60, pp. 229–233, Jan. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214785321082195

[84] G. Fraser, “TestSuiteWriter.java L234 - avoid downcast,” Sep. 2021. [Online]. Available: https://github.com/EvoSuite/evosuite/blob/master/client/src/main/java/org/evosuite/junit/writer/TestSuiteWriter.java#L234

[85] ——, “TestSuiteWriter.java L240 - add assertions,” Sep. 2021. [Online]. Available: https://github.com/EvoSuite/evosuite/blob/master/client/src/main/java/org/evosuite/junit/writer/TestSuiteWriter.java#L240

[86] Spring, “The SpringPetClinic Community,” Feb. 2024. [Online]. Available: https://spring-petclinic.github.io/

[87] Quarkus, “Quarkus: Supersonic Subatomic Java - Using the REST Client,” Apr. 2024. [Online]. Available: https://quarkus.io/guides/rest-client

[88] Baeldung, “Blade - A Complete Guidebook | Baeldung,” Jan. 2019. [Online]. Available: https://www.baeldung.com/blade

Signature

Lausanne, April 19, 2024

J.DiTria

Julien DI TRIA