# On the automated unit tests generation for Java applications using Spring

Kirill Shishin, Ilia Muravev, Egor Kulikov
*Department of Software Engineering*
*St. Petersburg State University*
{kirill.a.shishin, muravjovilya, egor.k.kulikov}@gmail.com

***Abstract*—This paper considers the automated unit tests generation for programs written in Java using the Spring framework. Although several test generation tools for "pure" Java applications have been developed in recent decades, the features of this framework are mostly not taken into account. However, Spring is used to develop many industrial Java applications. At the same time, the presence of Spring components in the application for which the tests are generated imposes additional requirements not only on the code analysis approaches, but also on the structure of the generated tests.**

**The main source of the information about object types and their properties is the Spring application context. The paper proposes an instrument for analyzing the application context, that in some cases allows generating test scenarios corresponding to real program executions and avoiding excessive mocking. The full initialization of the application context does not occur during this analysis. It makes the test generation safe for user data.**

**The proposed instrument for analyzing the Spring context has been integrated into the UnitTestBot Java automatic test generation tool. We also provide examples of tests generated for real open-source projects.**

***Index Terms*—Software Testing, Automated Unit Test Generation, Spring, Mocking, UnitTestBot Java.**

## I. Introduction

Software testing is an important and essential part of any project. While manually written tests often cover only a small percentage of program execution paths, their development usually requires significant effort and time from developers and testers. That's why solutions for the automated test generation have been actively developed over the last decades. They are intended to help their users to significantly increase the test coverage of a program, reducing many times the efforts spent on writing tests. There is a well-known experiment [1] on the *Coreutils*[1] project, showing how effective and useful automated test generation can be. Although this project has been developed for many years and tests have been written manually all the time, the test automation tool created by the authors of the experiment managed to significantly increase code coverage with tests in just a few hours and found some defects that had been unknown for more than 15 years. The relevance of automated test generation is also confirmed by the annual competitions of test generation tools [2]. At these competitions, participating tools test real projects (e.g., *Guava*[2], *Seata*[3], *Spoon*[4], etc.). As a result, winners are identified in several categories, including the efficiency of code defect detection, the percentage of program lines covered, and the human-readability of the generated tests.

Among the automated test generators showing good results at competitions, we would like to mention the open-source project *UnitTestBot Java*[5], which is being developed at Huawei Russian Research Institute. In 2022, the tool performed well in the competition [3], and in 2023, it was ranked first in the human-readability of generated tests and second overall in all categories [4]. *UnitTestBot Java* is a command-line tool and a plugin for *IntelliJ IDEA*[6], which aims to generate unit tests for Java applications. The methodological basis of the project is two code analysis techniques: symbolic execution and fuzzing.

*UnitTestBot Java* is good at generating tests for code in "pure" Java, that is, without the use of special frameworks. However, such frameworks are used quite often and impose additional requirements on the analysis of user code and the format of generated tests. One of the most popular frameworks today [5] is *Spring*[7]. It is used in the development of many Java projects. Therefore, it is important for *UnitTestBot Java* to be capable of generating specialized tests for applications that rely on this framework.

Spring is a diverse framework. However, one of its main features are *Dependency Injection (DI)* and *Inversion of Control (IoC)*. Spring has a *DI/IoC* container[8] that stores managed objects — *beans*. The configuration of this container can be defined in various ways: through annotations in user code, special configuration classes, or XML files. In addition, the user can also choose a *profile* that defines how the application will be configured. More than that, the framework offers a mechanism for implementing the MVC pattern, distinguishing between services and controllers, each of which requires a distinct testing approach as agreed by human testers. It is important that the generated tests must be not only correct, but should also follow the commonly accepted guidelines

[1]Coreutils project. Available: https://www.gnu.org/software/coreutils/

[2]Guava: Google Core Libraries for Java. Available: https://github.com/google/guava

[3]Seata: Simple Extensible Autonomous Transaction Architecture. Available: https://github.com/apache/incubator-seata

[4]Spoon. Available: https://github.com/INRIA/spoon

[5]UnitTestBot. Available: https://www.utbot.org

[6]IntelliJ IDEA — the Leading Java and Kotlin IDE. Available: https://www.jetbrains.com/idea/

[7]Spring framework. Available: https://spring.io

[8]The IoC container. Available: https://docs.spring.io/spring-framework/reference/core/beans.html

for testing Spring applications [6], [7], [8]. These and other features of the framework make automated testing of Spring applications a very challenging task.

When creating unit tests, we usually test a component in isolation from the external environment (other microservices, databases, authentication mechanisms). Therefore, many complicated features of the Spring framework do not have a significant impact on the unit test generation mechanisms and will be important only in the case of creating integration or end-to-end tests, meaning that "pure" Java testing techniques can be used. However, when it comes to virtual calls, in a "pure" Java program, there is no reliable way to determine which implementation should be called in tests. In contrast, in a Spring application, there is a configuration that determines which concrete classes will be used in place of abstract types at runtime. Using this type substitution information, it is possible to generate tests that are better aligned with the actual behavior of the program and test those scenarios that can occur during application use.

It is also important to note that since unit testing does not involve launching an application and initializing its context, and usually means testing a component in isolation, it is expected that full context initialization will not take place during the automated test generation, which includes analysis of the user configuration. Otherwise, test generation may be dangerous for user data: for example, during context initialization, environment variables may be unexpectedly set, some data may be loaded from third-party services, etc. This creates additional challenges when solving the task of test generation for applications written with Spring.

## II. Spring-based test generation

Let's consider a minimalistic example in which the type information from the application configuration allows generating tests better representing the real behavior of the program.

Assume the task is to generate tests for the `getSpecies()` method from the `SpeciesService` class,

```
@Service
public class SpeciesService {

    @Autowired
    @Setter
    private Animal animal;

    public String getSpecies() {
        return animal.getSpecies();
    }
}
```

where `Animal` is the interface having the following form:

```
public interface Animal {
    String getSpecies();
}
```

Suppose that we have two implementations of this interface in the project.

```
public class Cat implements Animal {
    public String getSpecies() {
       return "cat";
    }
}

public class Dog implements Animal {
    public String getSpecies() {
       return "dog";
    }
}
```

However, in the application configuration, only one of them — `Cat` — is used to create the `Animal` bean.

```
public class AnimalConfiguration {

    @Bean
    public Animal animal(){
        return new Cat();
    }
}
```

Assume that we develop tests for this method manually. One possible approach is mocking virtual call of `getSpecies()` method.

```
public void testGetSpecies() {
    SpeciesService speciesService
        = new SpeciesService();

    Animal animalMock = mock(Animal.class);
    when(animalMock.getSpecies())
        .thenReturn("mouse");

    speciesService.setAnimal(animalMock);

    String actual = speciesService.getSpecies();
    assertEquals("mouse", actual);
}
```

Such a test is formally correct, but far from checking the actual behavior of the program and therefore is hardly valuable in practice.

At the same time, when developing tests, we have an opportunity to investigate the application configuration and to find out that only `Cat` implementation is used for a given interface and to write a test that much more closely resembles the real behavior.

```
public void testGetSpecies() {
    SpeciesService speciesService
        = new SpeciesService();

    Cat animal = new Cat();
    speciesService.setAnimal(animal);

    String actual = speciesService.getSpecies();
    assertEquals("cat", actual);
}
```

Now, let's consider the scenario of automated test generation. If we do not extract type information from the Spring application configuration indicating which of the `Animal` implementations is preferred, we can either generate the already mentioned test with a mock or choose any of the `Animal` interface implementations arbitrarily. In this way, another formally correct but valueless test using the `Dog` implementation can be generated. However, if the application configuration is analyzed and the test generation tool is provided with the information that only the `Cat` implementation is used for the `Animal` interface, then the generated test will accurately represent the actual behavior of the program.

Despite the fact that the type concretization described above cannot always be done (due to possible ambiguity of the possible types choice) and it is not always necessary to do it, in some cases, concretization allows for generating more expressive tests for Spring applications that verify real execution scenarios. Therefore, the ability to generate tests with type concretization is a desirable option for the test generator.

## III. Overview

### A. Existing tools

There are a number of tools that to some extent solve the problem of automated testing of programs in Java. All of them use one or several basic code analysis techniques: symbolic execution, fuzzing or machine learning [9]. The most famous open-source solutions are *EvoSuite*[9], *UnitTestBot Java*[10], and *Randoop*[11]. Among the commercial tools let us mention *Parasoft Jtest*[12], *Diffblue Cover*[13] and *Machinet*[14].

Among these tools, only a small subset can generate tests for Spring applications. For example, *Parasoft Jtest* generates only test method templates. Of course, this reduces the total time required to write tests, but the scope of the covered code depends on the user, who needs to substitute arguments with values in the code of the generated test method templates.

The code snippet below shows an example of a generated test template for a Spring application using *Parasoft Jtest*. It is taken from the official *Parasoft* website[15].

```
@Test
public void testGetPerson() throws Throwable {
    MockedStatic<ExternalPersonService> mocked =
    mockStatic(ExternalPersonService.class);
    mocks.add(mocked);

    Person getPersonResult = null; // UTA: default
    value
    mocked.when(() ->
        ExternalPersonService.getPerson(anyInt())
    ).thenReturn(getPersonResult);

    // Given
    PeopleController underTest = new
    PeopleController();

    // When
    int id = 1;
    Model model = mock(Model.class);
    ResponseEntity<Person> result = underTest.
    getPerson(id, model);

    // Then
    assertNotNull(result);
    assertNotNull(result.getBody());
}
```

Another example of a tool that can generate tests for Spring applications is *Diffblue Cover*. It is able to generate tests that take into account the Spring application specifics.

Below is an example of the test generated for a Spring application using *Diffblue Cover*.

```
@ContextConfiguration(classes = {SpeciesService.
    class})
@ExtendWith(SpringExtension.class)
class SpeciesServiceUnitTests {
    @MockBean
    private Animal animal;

    @Autowired
    private SpeciesService speciesService;

    @Test
    void testGetSpecies() {
        when(animal.getSpecies())
            .thenReturn("");
        assertEquals("",
            speciesService.getSpecies());
        verify(animal, atLeast(1)).getSpecies();
    }
}
```

However, such tests do not follow the test writing guidelines for Spring applications mentioned in the introduction (for example, it is rather unconventional to use a class under test for context configuration) and the tool does not offer a mechanisms to deal with excessive mocking.

Thus, none of the test automation tools we are aware of offer mechanisms for generating tests based on the application configuration.

### B. UnitTestBot Java

*UnitTestBot Java* is a part of the *UnitTestBot* tool lineup for automated unit test generation. The tool uses two mechanisms to generate test scenarios: a symbolic engine and a fuzzer.

[9]What is EvoSuite? Available: https://github.com/EvoSuite/evosuite

[10]UnitTestBot Java: Automated unit test generation and precise code analysis for Java. Available: https://github.com/UnitTestBot/UTBotJava

[11]Randoop: Automatic unit test generation for Java. Available: https://randoop.github.io/randoop/

[12]Parasoft Jtest for Java Unit Testing. Available: https://www.parasoft.com/products/parasoft-jtest/java-unit-testing/

[13]What is Diffblue Cover? — Diffblue. Available: https://www.diffblue.com

[14]Machinet: AI Assistant for Developers. Available: https://www.machinet.net

[15]Accelerate Unit Testing of Spring Applications With Parasoft Jtest & Unit Test Assistant. Available: https://alm.parasoft.com/hubfs/New_Pages/Whitepaper:%20Accelerate%20Unit%20Testing%20of%20Spring%20Applications%20with%20Parasoft%20Jtest%20and%20Unit%20Test%20Assistant.pdf

The symbolic engine is one of the implementations of the symbolic execution paradigm [10]. It performs an analysis of the possible execution paths of a program by mapping a set of path constraints to each branch of execution. These constraints are expressed in terms of the logic of predicates, and then using the SMT solver *Z3*[16] their satisfiability is determined. Obtaining the possible paths of program execution, as well as their prioritization, comes from the control flow graph that is constructed from the byte code of the program. The byte code is preliminarily transformed into *Jimple* representation using the *Soot*[17]. With this transformation, the byte code takes a simpler representation having fewer instructions.

The fuzzer used in *UnitTestBot Java* applies the greybox fuzzing technique, which involves generating random input values for a concrete execution of the methods under test. After each concrete execution, the fuzzer obtains feedback about the change in execution path. Based on this feedback, it mutates the input values for the next iteration of its work.

More detailed information about the implementation of the symbolic engine and fuzzer in *UnitTestBot Java* can be found on the official website of the project or in the documentation in the repository on GitHub[18].

Both of these code analysis techniques are usually combined for maximum efficiency. In *UnitTestBot Java*, by default, 95% of the time allocated for test generation is given to the symbolic engine, and the fuzzer is used as an additional auxiliary tool.

## IV. Implementation

This section provides a Spring configuration analyzer implementation and describes a modernization of the *UnitTestBot Java* symbolic engine, which allows types to be concretized during test generation based on the information obtained from the configuration analyzer.

### A. Spring configuration analyzer

The engine obtains Spring-specific information for type concretization from the user application configuration analyzer. This information is collected using Spring's own instruments during the initialization of the application context.

Spring context initialization consists of several steps:

1) Collecting bean definitions. During this phase application configurations are parsed and analyzed. As a result, in particular, bean definitions are created. They include information about the class of the bean, its properties and its relationships with other beans.
2) Configuration of the bean definitions (*BeanFactoryPostProcessor*[19]). Once the information about beans has been collected, Spring can modify these definitions before they are used to create the beans themselves. Configuring bean definitions involves setting dependencies, specifying scope, configuring lifecycle and other parameters specific to the bean.
3) Creating the beans and configuring them. This stage involves creating and further configuring the beans instances based on their definitions. Class instances are created and initialization methods are called.

While the steps of collecting and configuring bean definitions are safe for the user application, the step of creating the beans may cause changes to user data. For this reason, we decided to embed ourselves in the Spring context initialization process and implement our own *BeanFactoryPostProcessor*. It collects all necessary and available to the analyzer information about beans at the stage of setting up the bean definitions, destroys bean definitions, and then stops any further application initialization. Thus, the creation of the beans is prevented. The general pipeline of type information collecting during Spring application context initialization is shown in the Figure 1.

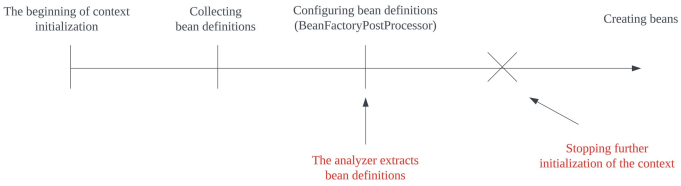


Fig. 1. The initialization stages of the Spring context

To analyze beans from a user application with our post processor, we start a "hybrid" Spring application whose classpath combines the classpaths of both the original user application and our Spring analyzer module. It is important to note that the Spring analyzer module has minimal dependencies, which helps avoid dependency conflicts with the user application. In particular, Spring analyzer does not depend on a specific version of Spring and utilizes reflection to handle any popular Spring version bundled with the user application. When starting such "hybrid" Spring application, we first determine whether Spring Boot is used and, based on that, choose an appropriate application starter class. Furthermore, we dynamically patch annotations to make the started application use desired Spring configuration (Java- or XML- based) and profiles.

In this way, we get an algorithm for configuring our own Spring application, shown in the Figure 2, while the entire process of collecting Spring-specific information for type concretization by the configuration analyzer is represented with the chain of actions shown in the Figure 3.

### B. Modernization of symbolic engine

The core idea of the symbolic engine modernization is to change the mechanism used for selecting symbolic object types. Whereas the symbolic engine previously selected an arbitrary type, determined by the SMT solver as satisfying the

[16]Z3. Available: https://github.com/Z3Prover/z3

[17]Soot. Available: https://github.com/soot-oss/soot

[18]UnitTestBot Java documentation. Available: https://github.com/UnitTestBot/UTBotJava/tree/main/docs

[19]BeanFactoryPostProcessor. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/BeanFactoryPostProcessor.html

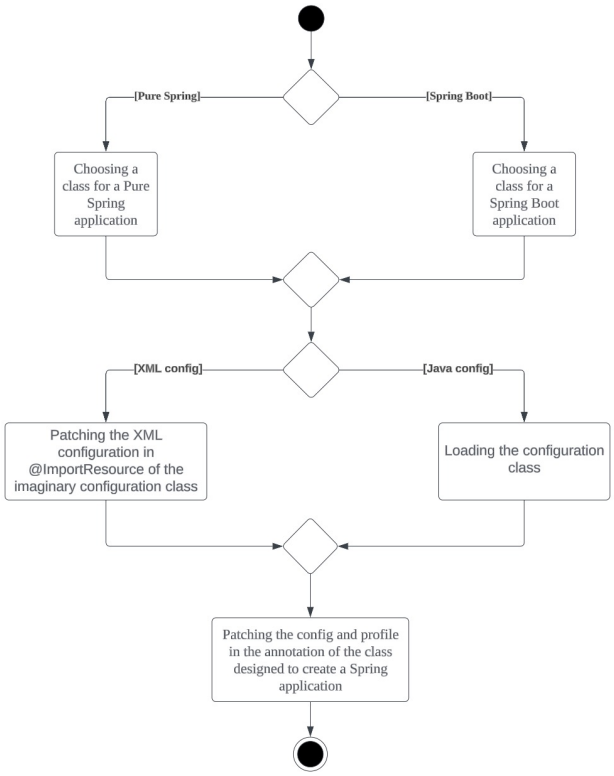


Fig. 2. The activity diagram of creating a Spring application on the *UnitTestBot Java* side

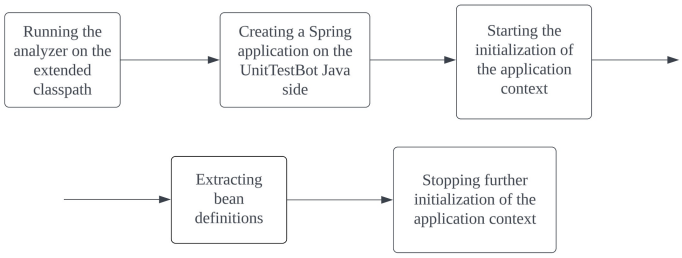


Fig. 3. The Spring-analyzer's work scheme

symbolic path constraints, it now tries to consider only those types that are used in the application configuration chosen for test generation.

Let's discuss the example of test generation for the `getSpecies()` method from the `SpeciesService` class, presented in Section II. In the past, when generating tests, the `Dog` implementation of the `Animal` interface could be chosen because it satisfied the symbolic path constraints. However, tests using the `Dog` class were not particularly useful because they did not test the actual behavior of the program. In contrast, now, the `Cat` class is deterministically chosen as it is specified in the application configuration.

### C. Integration of symbolic engine and configuration analyzer

The Spring configuration analyzer is launched in a separate process from the symbolic engine. Firstly, this allows running the configuration analyzer on the extended classpath without any difficulties. After that, it prevents a possible crash of the entire test generation in case of an error during the analysis of custom configurations. For example, running a Java Spring application based on a custom one on the *UnitTestBot* side may cause the JVM crash if the custom application is poorly designed.

The symbolic engine and user configuration analyzer processes, like all other processes in *UnitTestBot Java*, communicate using the *RD*[20] framework.

## V. Results

As a result of this research, we managed to propose an approach to the analysis of Spring application configurations, which sometimes allows generating tests that correspond better to the actual behavior of the program. The developed configuration analyzer was integrated into the well-known tool of automated test generation *UnitTestBot Java*.

In particular, the modernized tool is able to generate a test that checks the actual execution of the `getSpecies()` method from the `SpeciesService` class of the running example given in Section II. To go further, we also provide tests generated with the modernized tool on real open-source projects *Java Blog Aggregator: Boot*[21] and *Mall*[22].

### A. Java Blog Aggregator: Boot

We paid attention to the *Java Blog Aggregator: Boot* project because it is mentioned, for example, in the article [11] as one of the recommended projects to study for beginners in the Spring framework.

Consider the `AllCategoriesService` class that has the autowired `CategoryService` field.

```
@Service
public class AllCategoriesService {

  @Autowired
  private CategoryService categoryService;

  public Integer[] getAllCategoryIds() {
    List<Category> categories
            = categoryService.findAll();

    Integer[] result
            = new Integer[categories.size()];

    for (int i = 0; i < categories.size(); i++) {
      result[i] = categories.get(i).getId();
    }

    return result;
  }
}
```

The signature of the `CategoryService` class has the form:

[20]RD: Reactive Distributed communication framework for .NET, Kotlin and C++ (experimental). Inspired by JetBrains Rider IDE. Available: https://github.com/JetBrains/rd

[21]Java Blog Aggregator: Boot. Available: https://github.com/jirkapinkas/java-blog-aggregator-boot

[22]Mall. Available: https://github.com/macrozheng/mall

```
@Service
public class CategoryService
```

The `CategoryService` class has the annotation `@Service`, which means that this class is used to define a bean.

As a result of test generation using the modernized *UnitTestBot Java* for the `getAllCategoryIds()` method from the `AllCategoriesService` class, we get a test:

```
@Test
public void testGetAllCategoryIds() throws Exception
{
    AllCategoriesService allCategoriesService
        = new AllCategoriesService();

    CategoryService categoryService
        = new CategoryService();

    CategoryRepository categoryRepositoryMock
        = mock(CategoryRepository.class);

    when(categoryRepositoryMock.findAll())
        .thenReturn(new ArrayList<>());

    setField(categoryService,
             "cz.jiripinkas.jba.service
                .CategoryService",
             "categoryRepository",
             categoryRepositoryMock);

    setField(allCategoriesService,
             "cz.jiripinkas.jba.service
                .AllCategoriesService",
             "categoryService",
             categoryService);

    Integer[] actual =
        allCategoriesService.getAllCategoryIds();

    assertEquals(0, actual.length);
    assertEquals(new Integer[0], actual);
}
```

Instead of mocking the `CategoryService` class, its concrete implementation is used in this test. It makes the test more expressive. In particular, we can observe that the `CategoryService` interacts with the database. Also, the user can adjust the behavior of the mock related to database access if necessary.

### B. *Mall*

We also generated tests for the *Mall* project, which is very popular, having over one hundred thousand forks and stars on GitHub. Let's discuss the test for the `delAdmin()` method in the `UmsAdminCacheServiceImpl` class, that the modernized *UnitTestBot Java* has generated based on Spring application configuration analysis.

```
@Service
public class UmsAdminCacheServiceImpl implements
    UmsAdminCacheService
{
    @Autowired
    private UmsAdminService adminService;

    @Autowired
    private RedisService redisService;

    ...

    @Override
    public void delAdmin(Long adminId) {
        UmsAdmin admin
            = adminService.getItem(adminId);

        if (admin != null) {
            String key = REDIS_DATABASE + ":" +
                         REDIS_KEY_ADMIN + ":" +
                         admin.getUsername();

            redisService.del(key);
        }
    }

    ...

}
```

This class has two autowired fields: `UmsAdminService` and `RedisService`, which have corresponding beans in the application configuration.

The test generated for the `delAdmin()` method of the `UmsAdminCacheServiceImpl` class is as follows:

```
@Test
public void testDelAdmin()
    throws Exception
{
    UmsAdminCacheServiceImpl
    umsAdminCacheServiceImpl
        = new UmsAdminCacheServiceImpl();

    UmsAdminServiceImpl adminService
        = new UmsAdminServiceImpl();

    UmsAdminMapper adminMapperMock
        = mock(UmsAdminMapper.class);
    when(
        adminMapperMock.selectByPrimaryKey(any())
    ).thenReturn(null);

    setField(adminService,
             "com.macro.mall.service
                .impl.UmsAdminServiceImpl",
             "adminMapper", adminMapperMock);

    setField(umsAdminCacheServiceImpl,
             "com.macro.mall.service.impl
                .UmsAdminCacheServiceImpl",
             "adminService", adminService);

    umsAdminCacheServiceImpl.delAdmin(null);
}
```

In this test, instead of mocking the abstract type `UmsAdminService`, its concrete implementation `UmsAdminServiceImpl` is substituted according to the application configuration. Initialization of the second

autowired field did not occur because it is not required in the tested program execution path. Although there are no assertions in this test because the method has `void` return type, it is still valuable. Since the tested method takes a nullable value as an argument, a scenario in which `adminId` is `null` is possible and is a kind of edge case that often causes `NullPointerException`. The generated test ensures that no such exception actually occurs in the method under test. When writing tests manually, similar scenarios are often not taken into account.

## VI. Future work

Unit tests are often used to verify the logic of Spring application components, so high-quality automatic generation of such tests is important. However, some bugs can only be detected by integration and end-to-end tests that interact with real data storage and other microservices, as well as take into account the diverse features of the Spring framework (e.g., authorization and authentication). For this reason, developing an integration test generation tool is a prominent direction for future work. Such a tool will likely also need to initialize a modified Spring application, meaning that the "hybrid" Spring application starter developed in this work may find additional uses.

## References


[1] C. Cristian, D. Daniel, and E. Dawson, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (OSDI'08), USA : USENIX Association, 2008, pp. 209–224.

[2] "Workshop on Search-Based and Fuzz Testing," Accessed: Apr. 5, 2024. [Online]. Available: https://sbft24.github.io

[3] D. Ivanov et al., "UTBot Java at the SBST2022 Tool Competition," *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing* (SBST), Pittsburgh, PA, USA, 2022, pp. 39–40.

[4] D. Ivanov, A. Menshutin, M. Pelevin et al., "UTBot at the SBFT 2023 Java Tool Competition," *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing* (SBFT), 2023, pp. 68–69.

[5] "Java Programming — The State of Developer Ecosystem in 2022 Infographic," *JetBrains: Developer Tools for Professionals and Teams*, Accessed: Apr. 5, 2024. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2022/

[6] "Spring Boot Unit Testing Service Layer using JUnit and Mockito," Accessed: Apr. 5, 2024. [Online]. Available: https://www.javaguides.net/2022/03/spring-boot-unit-testing-service-layer.html

[7] "Overview: Spring Framework," Accessed: Apr. 5, 2024. [Online]. Available: https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework/server.html

[8] S. Chathuranga, "Unit and Integration Testing in Spring Boot Micro Service," *Medium*, Accessed: Apr. 5, 2024. [Online]. Available: https://salithachathuranga94.medium.com/unit-and-integration-testing-in-spring-boot-micro-service-901fc53b0dff

[9] M. Kim, Q. Xin, S. Sinha, and A. Orso, "Automated test generation for REST APIs: no time to rest yet," *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2022), Association for Computing Machinery, New York, NY, USA, 2022, pp. 289–301, doi: https://doi.org/10.1145/3533767.3534401.

[10] R. Baldoni, E. Coppa, D. Cono D'elia, C. Demetrescu, and I. Finocchi, "2018. A Survey of Symbolic Execution Techniques," *ACM Comput. Surv. 51, 3, Article 50*, 2019, 39 pages, https://doi.org/10.1145/3182657.

[11] R. Fadatare, "10+ Free Open Source Projects Using Spring Boot," Accessed: Apr. 5, 2024. [Online]. Available: https://www.javaguides.net/2018/10/free-open-source-projects-using-spring-boot.html