Integration of Automatic Test Generators For Maximizing the Code Coverage Level

Ivan Jakovljević and Aleksander Radovan *

Posted Date: 23 July 2024

doi: 10.20944/preprints202407.1767.v1

Keywords: automatic test generation; jUnit; Spring framework; JaCoCo; EvoMaster; EvoSuite

 doi:10.20944/preprints202407.1767.v1

Article

Integration of Automatic Test Generators For Maximizing the Code Coverage Level

Ivan Jakovljević and Aleksander Radovan *

Algebra University College; jakovljevic.ivan@protonmail.com
* Correspondence: aleksander.radovan@algebra.hr

Abstract: This paper explores the development of a sophisticated system for automatic test generation tailored specifically for Java web applications utilizing REST interfaces. Writing tests ensures the quality of the software solution even after the introduction of subsequent changes, because automated regression testing can detect potential bugs in the code even before the application is resealed into production. By utilizing and comparing various automatic test generators, the system described provides users with a visual representation of code structure and coverage, enhancing insights through call graph visualization. The paper describes the concepts of theoretical automatic test generators, gives an overview of generators, and details the design and implementation of the implemented solution, which integrates the system test generator and the unit test generator based on open source to achieve the maximum code coverage level.

Keywords: automatic test generation; jUnit; Spring framework; JaCoCo; EvoMaster; EvoSuite;

1. Introduction

State-of-the-art automated test generators utilize code instrumentation to gather feedback from test execution, using advanced search-based techniques and symbolic analysis in order to generate tests which are able to exercise code paths that are unlikely to be reached via simpler methods such as random input mutation. Search strategies implemented by such generators analyze conditional statement logic and guide the test generation towards a state that satisfies particular logical predicate needed to reach a particular execution branch. However, there are some types of conditions that do not lend themselves to such analysis. Examples are finding an input that satisfies a particular cryptographic hash function, finding Base64 encoding result of an mock image that does not exist, or a scenario where a particular value is sent to an external service and returned value is used in a conditional statement. Tests for such parts of the code can be written manually, but another possibility is to also use automated unit test generator to generate tests for such hard-to-reach code segments.

The goal of this application is to provide visualization of code structure and code coverage to give additional insight into the result of a testing process, identify areas of low code coverage that are connected to critical functionalities, and facilitate combining multiple testing tools (such as system and unit test generators).

Automated test generators were analyzed with an industrial software component [1], focusing on effectiveness of a set-of-the research test generator on a family of industrial programs with nontrivial domain-specific peculiarities. The described study applied a set of test generators based on random testing, dynamic symbolic execution, search-based testing, and an ensemble of these techniques, against of subject programs developed by developers. It was shown that the test generators can be effective on industrial software, up to exposing bug that had escaped detection during the testing of a prototype deployment of the safety critical system. In comparison to this paper, our research focused more on visualization of code structure and improving the code coverage by using two different open-source tools, EvoMaster and EvoSuite to achieve maximum code coverage.

 doi:10.20944/preprints202407.1767.v1

A comprehensive comparison of code-based test input generator tools was done in paper [2]. The source code features that were considered were code coverage, clarity, well-organized structure, compactness and minimizing the dependencies. The research included rather simple code snippets that were the subject of generated tests. The results related to the EvoSuite tool used in our research shown that can be applied to more complex cases, can tolerate internal errors, supports logging, can be configured with a lot of configuration parameters, supports test suite minimization, and nontrivial assertions. In our research we were using the more recent version of EvoSuite and focuses on both unit and system testing of a real-world based web application that implement features of a web-shop and based on programming language Java with Spring framework, not only simple code snippets.

Combining black-box model-based testing with white-box parameterized unit testing to maximize the source code coverage with automated test generation [3]. It was described that model-based tools allow expressing end-to-end functional properties in a rather partial fashion. The combination of model-based symbolic exploration and parameterized unit testing results in a means to force the execution of code-coverage maximizing, specification-relevant behavior by synthesizing the appropriate data values. In comparison to this research, in our research we analyze the Visitor pattern implemented in EvoMaster and EvoSuite tools for discovering the test cases and methods that will be tested with automatically generated tests.

Besides generating separate test methods, it's possible to generate the whole test suites [4] that uses the EvoSuite tool as well as in our research. Test Suite optimization was achieved by using genetic algorithms. It was proved that the whole test suite generation achieves higher coverage than single branch test case generation. In comparison to our research, we focus on generating tests of a whole web application based on Spring framework based on several SOLID principles and design patterns implemented in the test cases, which is more challenging rather than simple methods with few lines of code.

An alternative way of unit test generation that uses a test data generation paradigm commonly known as dynamic test data generation [5]. The source code of a program is instrumented to collect information about the program as it executes. The resulting information, collected during each test execution of the program, is used to heuristically determine how close the test came to satisfying a specified test requirement. The paper investigates the use of genetic search to generate test cases by function minimization by using GADGET (the Genetic Algorithm Data GEneration Tool) written in C and C++. Test data were generated for programs of various sizes. The results proved that the performance of random test generation deteriorates for larger programs, and as program complexity increases, nonrandom test generation techniques become increasingly desirable. In comparison to our research, the presented approach does not include the generation of tests related to REST API interfaces which represent state of the art interface for integration of information systems.

A concept of Certainty Boolean values, which encode how certain a "true" or "false" value proved that an optimization in test generation can be achieved where complex interprocedural calculation in conditional statements need to be calculated [6,7]. Evaluation on a set of complex Java classes and the EvoSuite test generator showed that testability transformation substantially alters the fitness landscape for Boolean branches, and the altered fitness landscape leads to performance improvements. The fitness landscape describes the topological structure of the search space. Transformed branches show a change in the fitness landscape which increases the success rate from 68% to 71%, but the results depend on the number of branches that depend on Boolean values.

Another approach called Search-based software testing (SBST) [8] that uses test seeds, i.e., test code skeletons of legitimate objects which enable the use of classical measurements, and overcomes the problem related to the case when the function under test takes object inputs, as classical measurements. Given a target branch in function under test, the described approach first statically analyzes the function to build an object construction graph that captures the relation between the operands of the target methods and the states of their relevant object inputs. Based on the graph, the test template code is created where each "slot" is a mutation point for the search algorithm. This approach can be integrated with existing SBST algorithms with EvoObj on top of the EvoSuite unit test generation tool and proved that it outperformed results that EvoSuite generates. In comparison

 doi:10.20944/preprints202407.1767.v1

to our research, we combined another open-source tool, EvoMaster, to compensate weaknesses of EvoSuite generated to achieve maximum code coverage.

White-box fuzzers like EvoMaster supports testing of RESTful APIs [9]. With extensions of original techniques like defining a new branch distance, using various data structures like Lists, Sets and Maps, and additional methods replacements based on programming language Java, improved coverage levels are achieved. In comparison to our research, we use more recent version v.3.0.0. in comparison to v.1.6.1.

Other fuzzing infrastructures like OSS-Fuzz and ClusterFuzz were used to find bugs with various configurations related coverage guidance [10]. The research determined the relationship between efficacy of coverage guidance and multiple factors in closed-source software. The research proved that the quality of initial seed (initial seeds were provided given the test case data for mutation) has a strong correlation to coverage feedback efficacy.

A novel vulnerability-mining model proposal, called HotCFuzz, that is centered on the coverage of hotspot code blocks, especially focusing on software vulnerabilities related of security incidents was described in this research [11]. Path-coverage-based fuzzing test techniques used, exemplified by AFL, emphasize the exploration of new paths and the minimization of test input for seed selection. Introduced algorithm to detect and pinpoint hotspot codes during the static analysis stage proved that HotCFuzz surpasses other alternative solutions like AFL.

Another fuzzer implementation, called ObFuzzer, increases object operation complexity during execution from 29% do 40%, which enhances the ability to discover vulnerabilities related to object operations [12]. The described proposal solves the disadvantages of traditional procedure-oriented vulnerability-discovery approaches in vulnerabilities caused by object operations.

Evolutionary algorithms include limiting factors related to genetic operators (crossover and mutation) are fully randomized, potentially breaking promising patterns in the sequence of API requests discovered during the search. Breaking these patterns has a negative impact on the effectiveness of the test case generation process. To address this limitation, a new approach that uses agglomerative hierarchical clustering (AHC) is proposed [13]. The goal was to infer a linkage tree model, which captures, replicates, and preserves these patterns in new test cases for REST API testing. The results show that the solution proposal achieves a statistically significant increase in test target coverage compared to other options. Those findings and conclusions could be used in future studies related to solutions based on evolutionary algorithms with the limitations reflected in our research as well.

A proposal for a fast format-aware fuzzing approach to recognize dependencies from the specified input to the corresponding comparison instruction described how the divide dependencies into Input-to-State (I2S) and indirect dependencies [14]. The advantages of this approach are based on recognizing I2S dependencies more completely and swiftly using the input based on the de Bruijn sequence and its mapping structure and obtaining indirect dependencies with a light dependency existence analysis on the input fragments. The results proved that with this approach it was possible to reduce the average time overhead and code coverage in comparison to other solutions.

Our research includes analyzing how it is possible to achieve maximum coverage with tests that are automatically generated by EvoSuite or EvoMaster open-source solutions. The thesis to be confirmed is based on combining different tools used to generate tests to achieve greater coverage than when using a single tool separately because they focus on different segments of testing. Tested application consists of several common modules such as controller classes that expose REST API endpoints, service classes that enable the integration of controller classes and repository classes, and classes that are related to mapping objects from domain objects to DTO objects, security configurations, various events and listeners, and custom classes that represent exceptions that may occur. Such a multi-layer application is a very common choice of architecture when developing a backend system that exposes REST API interface, in the Java programming language and other technologies, so the results presented in this research show how useful it can be for development teams and speed up the development of tests that ensure maximum coverage of the implemented solution.

 doi:10.20944/preprints202407.1767.v1

In following section, the materials and methods used during the experiment were described. After that, the achieved results were presented that proved better code coverage. At the end, in discussion section we reflect on importance of automation testing of software and using generated tests that significantly cover the code with test to avoid unnecessary problems in production environments.

2. Materials and Methods

This application is developed using Spring Boot [15], a framework which helps accelerate Java application development by providing default configurations for commonly used tools such as servlet containers (Spring MVC) or data persistence (Spring Data). The applications consists of several parts: code parser for generating call graph, embedded NoSQL database, embedded web server/servlet container, manager/wrapper for an external code coverage tool and a frontend application for visualizing the graph and accepting user input.

2.1. Methodology

The methodology used in this described in the flow chart of actions shown on **Figure 1**. The initialization phase defines the location of the code that will te tested, that is packaged in JAR (Java ARchive file) that contains compiled source code of the tested system. The following phase is related to unpacking the JAR file and extracting files that contain bytecode resulted from compiling source code. The classes and methods defined in bytecode are then analyzed and the complexity of the code is determined. After that, embedded HTTP server needs to be started to service query request during the generation of tests and generates the call graph. A client application needs to start to send requests that specifies root node and generates JSON results. With JaCoCo command line interface (CLI) the coverage results are stored to OrientDB database.

2.2. Bytecode Parsing and Graph Generation

The application needs to be started from a command line and location of the SUT (System Under Test) JAR passed as an argument. The application will then parse the class files using the ASM library [16]. ASM uses the Visitor pattern – developer is expected to implement ClassVisitor and MethodVisitor abstract classes and provide implementations of functions that will get executed every time the bytecode reader reaches a particular code construct (module, class, method, annotation, field or a specific instruction). In this case, the responsibility of those functions is to store each parsed class (or an interface) as a class vertex in OrientDB's graph database. Inheritance relationships between class vertices are stored as graph edges. Visited methods are stored as method vertices and each method vertex is linked to its respective class vertex. Bytecode instructions are visited next and if the visited instruction is a method invocation, then edge between the two method vertices will be created. Additionally, cyclomatic complexity is measured on a class and method level. Whenever parsing an instruction which leads to branching of execution paths (such as an IF instruction), cyclomatic complexity counter is incremented and eventually stored in the database as a part of a method vertex. Once all methods have been visited, their cyclomatic complexity is aggregated and stored in a class vertex.

The next step after bytecode parsing is to refine the call graph. Java supports polymorphism, meaning that a method call can execute different target methods, depending on the object's type. This poses a problem for static call graph analysis as object's type might not be known before runtime. The most basic way of dealing with this issue is by assuming that a polymorphic call will invoke all of the candidate functions. For instance, if an interface method is invoked, then the call graph constructor will create edges between the caller method and each callee method in every class that implements the interface. This approach is called the Class Hierarchy Analysis (CHA) [17]. Although simple, its main drawback is tendency towards false positives. For instance, if a caller method calls another method of a Collection object, then the CHA generator will record that the caller method has called all of the methods of a specific name that are members of classes implementing the Collection

 doi:10.20944/preprints202407.1767.v1

interface (HashMap, Stack, LinkedList, etc.). Easy way to prune the number of false positives is by excluding methods belonging to objects that haven't been instantiated. This approach is called Rapid Type Analysis (RTA) [18] and is used by the call graph generator in this application.

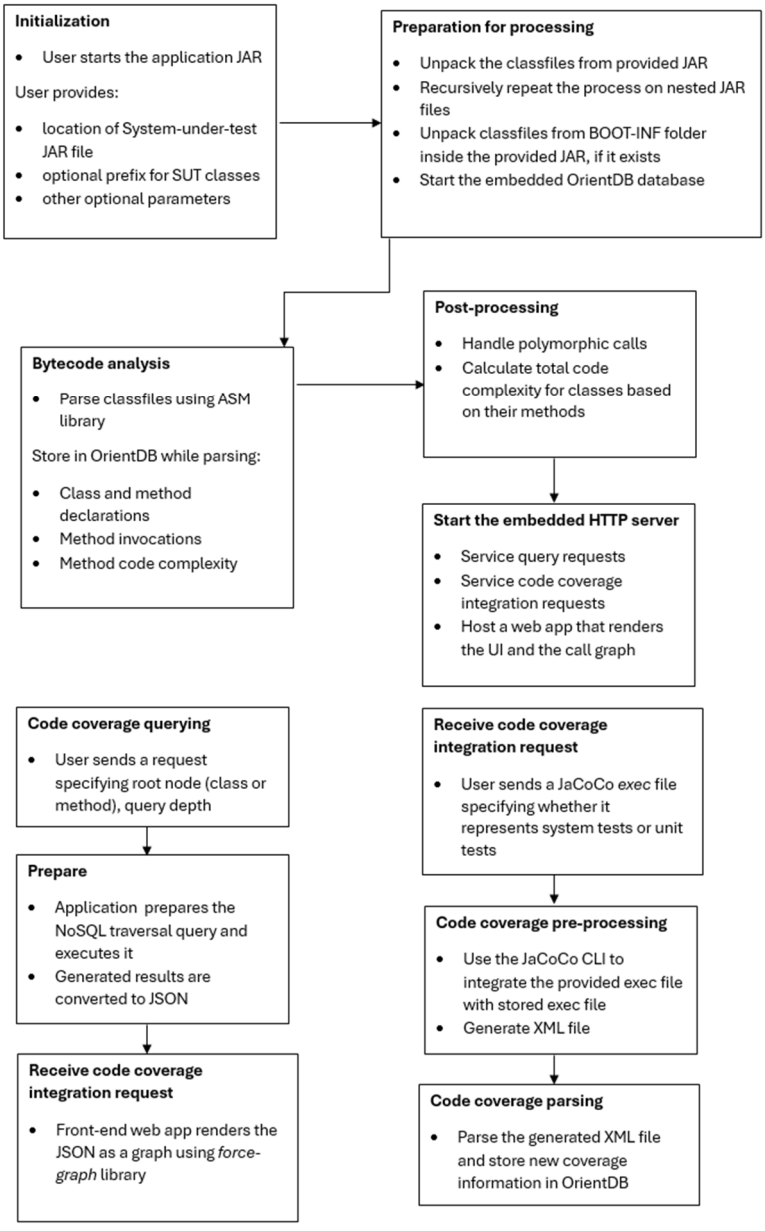


Figure 1. Flow chart of actions.

2.3. Handling JaCoCo

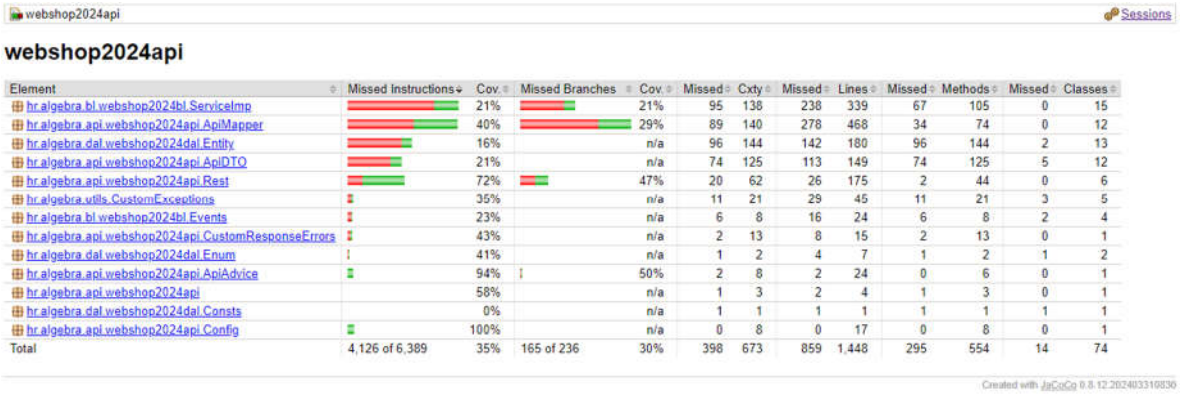
After the class files have been parsed and call graph generated, the application is ready to accept the code coverage data. JaCoCo [19] is one of the most popular code coverage libraries for Java. It can be combined with any test generators simply by instrumenting the SUT JAR file and then running the generated tests. After SUT terminates, JaCoCo will automatically generate a file containing coverage report that can be uploaded to this application using the web interface provided by the React application. The application will then run a JaCoCo CLI tool that transforms the code coverage report into an XML form which is then parsed and coverage data is stored inside the database. JaCoCo has a useful feature of merging multiple code coverage reports. This allows the user to run another

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 July 2024 doi:10.20944/preprints202407.1767.v1

testing campaign and upload the JaCoCo report file to the application which will then integrate the code coverage with previous testing results.

The JaCoCo manager is invoked by the Controller class, and its role is to start the JaCoCo CLI, merge the provided JaCoCo report with previous reports, generate XML report of code coverage, parse it and persist the extract data to the call graph stored in the OrientDB database.

An example of JaCoCo report is shown on **Figure 2**. In the "Element" column, a list of all packages is listed, and in other columns the number of missed instructions, branches in the code, cyclomatic complexity, missed lines, methods, and classes are shown.


webshop2024api

Sessions

webshop2024api

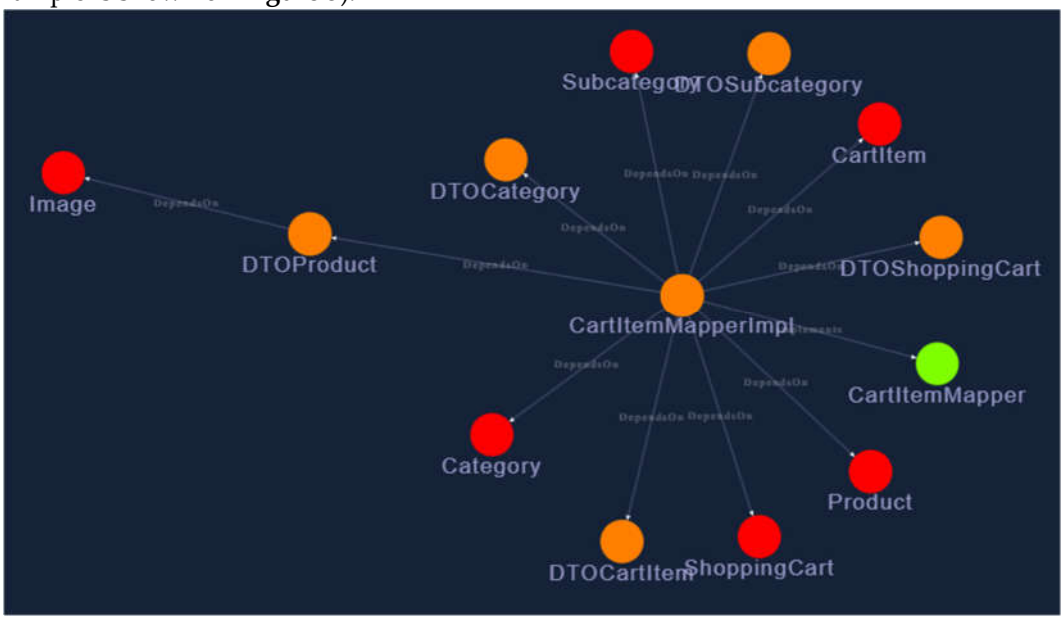
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
hr.algebra.bl.webshop2024bl.ServiceImp		21%		21%	95	138	238	339	67	105	0	15
hr.algebra.api.webshop2024api.ApiMapper		40%		29%	89	140	278	468	34	74	0	12
hr.algebra.dal.webshop2024dal.Entity		16%		n/a	96	144	142	180	96	144	2	13
hr.algebra.api.webshop2024api.ApiDTO		21%		n/a	74	125	113	149	74	125	5	12
hr.algebra.api.webshop2024api.Rest		72%		47%	20	62	26	175	2	44	0	6
hr.algebra.utils.CustomExceptions		35%		n/a	11	21	29	45	11	21	3	5
hr.algebra.bl.webshop2024bl.Events		23%		n/a	6	8	16	24	6	8	2	4
hr.algebra.api.webshop2024api.CustomResponseErrors		43%		n/a	2	13	8	15	2	13	0	1
hr.algebra.dal.webshop2024dal.Enum		41%		n/a	1	2	4	7	1	2	1	2
hr.algebra.api.webshop2024api.ApiAdvice		94%		50%	2	8	2	24	0	6	0	1
hr.algebra.api.webshop2024api		58%		n/a	1	3	2	4	1	3	0	1
hr.algebra.dal.webshop2024dal.Consts		0%		n/a	1	1	1	1	1	1	1	1
hr.algebra.api.webshop2024api.Config		100%		n/a	0	8	0	17	0	8	0	1
Total	4,126 of 6,389	35%	165 of 236	30%	398	673	859	1,448	295	554	14	74

Created with JaCoCo 0.8.12.202403310830

Figure 2. Example of JaCoCo code coverage report.

2.4. Visualization

The embedded web server has two roles: it serves the frontend application to the user and it contains a controller which accepts the JaCoCo file and passes is to other components. The frontend has been developed using React [20], a JavaScript library for building user interfaces based on components. The main component is the react-force-graph [21], a library which, when provided with graph structure in form of JSON, can automatically arrange the graph nodes in a way that is visually appealing. In the top bar, user can select test source (automated system test generator, automated unit test generator or manually written tests) and upload the generated JaCoCo file. After the integration is complete the graph nodes will become colored with a green, yellow or red tint depending on achieved coverage. Size of the node will be proportional to its cyclomatic complexity (an example is shown on **Figure 3**).



 doi:10.20944/preprints202407.1767.v1

Figure 3. Example of call graph code coverage visualization generated on a test open-source REST application.

User can select to view code coverage generated by one of the aforementioned sources, or he can chose a view which integrates all of the test sources (combined code coverage). Call graphs are usually too large to be shown on screen at once, which is why the application requires that the user enter a root node and depth parameter. For each caller method, starting from the root method, all of its callees are fetched. This fetching process is repeated the number of times specified by the depth parameter. User can also choose to invert the relationship and fetch the caller methods instead of callees. It is also possible to switch between method and class nodes. Visualization of class nodes functions in almost the same way except that inheritance relationship is also shown together with method invocations. Hovering over a node will display coverage, complexity and a class/method name.

2.5. Running Automated Test Generators

The motivation behind code coverage and graph visualization is to allow better insight into the result of system test generation campaign and identify areas of poor coverage that wouldrequire unit test generation testing or even manual testing. This application can visualize code coverage results for tests generated by any automated test generator for Java Virtual Machine, as long as the resulting JaCoCo file is provided. We will describe the usage of EvoMaster and Evosuite, the two whitebox test generators for Java whose effectiveness is supported by empirical data.

2.4.1. EvoMaster

EvoMaster is an automated system test generator for Java applications that are using REST, GraphQL or RPC APIs [22]. It supports blackbox and whitebox modes. Both modes require that a schema is provided (OpenAPI/Swagger in case of REST API). The whitebox mode only works for applications that run on JVM. It instruments the SUT so that feedback from code execution can be used to guide the test generation. The blackbox mode isn't limited to JVM language but it has worse performance in achieving code coverage.

EvoMaster consists of two parts: a client process that runs the SUT and a server/master process that receives feedback from client, generates test cases and sends them to client for execution. First step in using EvoMaster is to write a driver. The driver class needs to extend the EvoMaster's ExternalSutController or EmbeddedSutController. It needs to implement a set of methods that define how the SUT is to be started and stopped, and how the state is to be cleared between tests. The driver also needs to provide a method which returns the location of an OpenAPI schema (usually through an endpoint exposed by the application) and, optionally, the relational database schema.

After the driver class has been built and started, it will open a TCP port awaiting commands from the server process. User needs to start the evomaster.jar which will then connect to the port and begin the test generation process.

Once the testing process has been completed, it is necessary to run the generated JUnit tests with JaCoCo instrumented SUT so that the report file gets produced. Most reliable way of integrating JaCoCo code coverage library is to use JaCoCo CLI to offline instrument the SUT class files or JAR, and add the instrumented files to JUnit runner's classpath. After JUnit tests finish running, the report file will be generated which can then be uploaded to the visualization application.

An example of EvoMaster test generation report is shown on **Figure 4**.

 doi:10.20944/preprints202407.1767.v1

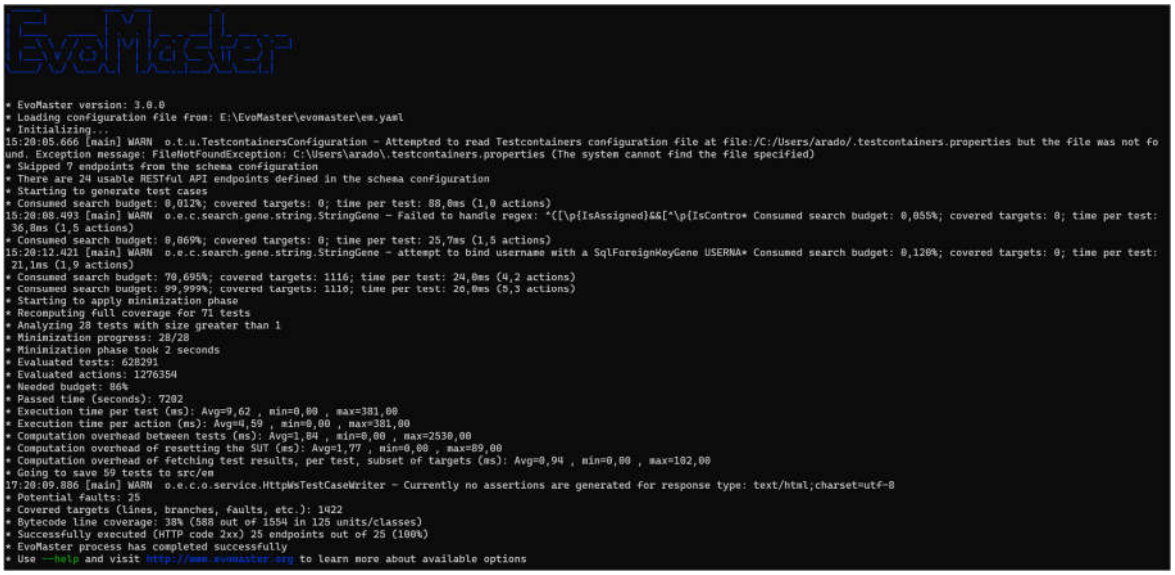


Figure 4. Example of EvoMaster generation process report.

2.4.2. EvoSuite

EvoSuite is a an automated unit test generator for Java [23]. It can generate tests for individual classes or an entire project. It is important to keep in mind that tests need to be independent from each other which is why EvoSuite replaces file system and network calls with mocks. This simplifies the test creation but on the other hand it disqualifies EvoSuite as a proper system test generator.

EvoSuite can either be used from a command line or as a Eclipse, IntelliJ or a Maven plugin [24]. If it is run from command line, then SUT classpath needs to be given as an argument. EvoSuite can target particular classes using the -class option or even particular methods using the -target_method or -target_methods setting.

SUT class files need to be instrumented with JaCoCo so that the report gets created, which then can be uploaded to the visualization app.

2.4.3. Architecture of Analyzed Application

The Java web application implemented with Spring boot that was analyzed was divided into four packages:

1) *API module* – contains classes that represent custom exception handler, 12 DTO (*Data Translfer Object*) classes for every domain class, mapper interfaces that convert a domain object to DTO and vice versa, security configuration class, error response class, and Controller classes that implement endpoints for exposing CRUD operations (CREATE, READ, UPDATE, and DELETE)
2) *Business layer module* – contains classes that represent events and listeners of custom events that happen asynchronously while the users are using the application, interfaces for service classes that contain method definition for business logic implementation and classes that implement those interfaces.
3) *Data access layer module* – contains a class that contains constant definitions,

classes that represent entity classes and use annotations like @Entity and @Table (related to Object-Relational Mapping framework), and other Lombok [25] annotations like @NoArgsContructor, @AllArgsConstructor, @Getter, @Setter, and @ToString, enumeration classes, and implementation of repository interfaces that extend JpaRepository interface supported by Spring Data JPA framework [26].

4) *Utils module* - contains custom exception classes and ImageConverter class with an example of irreversible method.

The source code is available online: https://github.com/ivancerosi/WebShopFuzz/tree/main/src/test/java/hr/algebra/api/webshop2024api.

 doi:10.20944/preprints202407.1767.v1

3. Results

Generation of tests by EvoMaster are started with defining the search budget, or time the available for the algorithm to analyze the code. It was set to 1h which was adequate to generate significant results comparable to test generated by EvoSuite, that generated tests for around 70 classes with duration of 5 minutes per class. EvoMaster generated 51 tests and EvoSuite 518 tests in total. The coverage was focused on classes, methods, branches and lines. In Table 1. the coverage results are shows, first separated by EvoMaster and EvoSuite coverage, and then combined coverage, Table 2 shows the coverage breakdown achieved by EvoMaster, Table 3 the coverage breakdown achieved by EvoSuite and Table 4 coverage breakdown of combined.

Table 1. Overall Coverage Summary.

	Class (%)	Method (%)	Branch (%)	Line (%)
EvoMaster	76,7 (%) (56/73)	41.6% (186/447)	20.4% (43/211)	32.7% (419/1280)
EvoSuite	63% (46/73)	70.5% (315/447)	49% (117/239)	63.8% (816/1280)
EvoMaster and EvoSuite combined	94.5% (69/73)	82.8% 4(370/447)	57% (134/235)	76.2% (975/1280)

Table 2. Coverage Breakdown – EvoMaster.

Package name	Class (%)	Method (%)	Branch (%)	Line (%)
hr.algebra.api.webshop2024api	100% (1/1)	66.7% (2/3)		66.7% (2/3)
hr.algebra.api.webshop2024api.ApiAdvice	100% (1/1)	100% (6/6)	50% (2/4)	91.7% (22/24)
hr.algebra.api.webshop2024api.ApiDTO	50% (5/10)	34.9% (29/83)		22.5% (29/127)
hr.algebra.api.webshop2024api.ApiMapper	100% (10/10)	41% (25/61)	17.5% (20/114)	26.4% (103/390)
hr.algebra.api.webshop2024api.Config	100% (1/1)	100% (6/6)		100% (16/16)
hr.algebra.api.webshop2024api.CustomResponseErrors	100% (1/1)	77.8% (7/9)		50% (7/14)
hr.algebra.api.webshop2024api.Rest	100% (6/6)	100% (30/30)	37% (10/27)	80.1% (117/146)
hr.algebra.bl.webshop2024bl.Events	50% (2/4)	25% (2/8)		33.3% (6/18)
hr.algebra.bl.webshop2024bl.ServiceImp	100% (14/14)	34.8% (32/92)	19% (11/58)	24.2% (69/285)
hr.algebra.dal.webshop2024dal.Consts	0% (0/1)	0% (0/1)		0% (0/1)
hr.algebra.dal.webshop2024dal.Entity	83.3% (10/12)	30.7% (31/101)		20% (31/155)
hr.algebra.dal.webshop2024dal.Enum	50% (1/2)	50% (2/4)		42.9% (3/7)
hr.algebra.utils.CustomExceptions	50% (3/6)	50% (11/22)		37.9% (11/29)
hr.algebra.utils.ImageConverter	100% (1/1)	75% (3/4)		37.5% (3/8)
hr.algebra.webshop2024.Advice	0% (0/1)	0% (0/2)	0% (0/4)	0% (0/9)
hr.algebra.webshop2024.DTO	0% (0/1)	0% (0/12)		0% (0/25)
hr.algebra.webshop2024.Mapper	0% (0/1)	0% (0/3)	0% (0/4)	0% (0/23)

Table 3. Coverage Breakdown – EvoSuite.

Package name	Class (%)	Method (%)	Branch (%)	Line (%)
hr.algebra.api.webshop2024api	0% (0/1)	0% (0/3)		0% (0/3)
hr.algebra.api.webshop2024api.ApiAdvice	0% (0/1)	0% (0/6)	0% (0/4)	0% (0/24)
hr.algebra.api.webshop2024api.ApiDTO	100% (10/10)	100% (83/83)		100% (127/127)
hr.algebra.api.webshop2024api.ApiMapper	90% (9/10)	88.5% (54/61)	85.1.5% (97/114)	88.2% (344/390)
hr.algebra.api.webshop2024api.Config	0% (0/1)	0% (0/6)		0% (0/16)
hr.algebra.api.webshop2024api.CustomResponseErrors	100% (1/1)	100% (9/9)		100% (14/14)
hr.algebra.api.webshop2024api.Rest	66.7% (4/6)	80% (24/30)	55.6% (15/27)	63% (92/146)
hr.algebra.bl.webshop2024bl.Events	0% (0/4)	0% (0/8)	0% (0/4)	0% (0/18)
hr.algebra.bl.webshop2024bl.ServiceImp	0% (0/14)	0% (0/92)	0% (0/78)	0% (0/285)
hr.algebra.dal.webshop2024dal.Consts	100% (1/1)	100% (1/1)		100% (1/1)

 doi:10.20944/preprints202407.1767.v1

hr.algebra.dal.webshop2024dal.Entity	100% (12/12)	100% (101/101)		100% (155/155)
hr.algebra.dal.webshop2024dal.Enum	0% (0/2)	0% (2/4)		0% (0/7)
hr.algebra.utils.CustomExceptions	100% (6/6)	100% (22/22)		100% (29/29)
hr.algebra.utils.ImageConverter	100% (1/1)	100% (4/4)	50% (1/2)	50% (4/8)
hr.algebra.webshop2024.Advice	0% (0/1)	0% (0/2)	0% (0/6)	0% (0/9)
hr.algebra.webshop2024.DTO	100% (1/1)	100% (12/12)		100% (25/25)
hr.algebra.webshop2024.Mapper	100% (1/1)	100% (3/3)	100% (4/4)	100% (23/23)

Table 4. Coverage Breakdown – Cumulative coverage (EvoMaster and EvoSuite).

Package name	Class (%)	Method (%)	Branch (%)	Line (%)
hr.algebra.api.webshop2024api	100% (1/1)	66.7% (2/3)		66.7% (2/3)
hr.algebra.api.webshop2024api.ApiAdvice	100% (1/1)	100% (6/6)	50% (2/4)	91.7% (22/24)
hr.algebra.api.webshop2024api.ApiDTO	100% (10/10)	100% (83/83)		100% (127/127)
hr.algebra.api.webshop2024api.ApiMapper	100% (10/10)	90.2% (55/61)	85.1% (97/114)	88.2% (344/390)
hr.algebra.api.webshop2024api.Config	100% (1/1)	100% (6/6)		100% (16/16)
hr.algebra.api.webshop2024api.CustomResponseErrors	100% (1/1)	100% (9/9)		100% (14/14)
hr.algebra.api.webshop2024api.Rest	100% (6/6)	100% (30/30)	70,4% (19/27)	92.5% (135/146)
hr.algebra.bl.webshop2024bl.Events	50% (2/4)	25% (2/8)		33.3% (6/18)
hr.algebra.bl.webshop2024bl.ServiceImp	100% (14/14)	34.8% (32/92)	14.1% (11/78)	24.2% (69/285)
hr.algebra.dal.webshop2024dal.Consts	100% (1/1)	100% (1/1)		100% (1/1)
hr.algebra.dal.webshop2024dal.Entity	100% (12/12)	100% (101/101)		100% (155/155)
hr.algebra.dal.webshop2024dal.Enum	50% (1/2)	50% (2/4)		42.9% (3/7)
hr.algebra.utils.CustomExceptions	100% (6/6)	100% (22/22)		100% (29/29)
hr.algebra.utils.ImageConverter	100% (1/1)	100% (4/4)	50% (1/2)	50% (4/8)
hr.algebra.webshop2024.Advice	0% (0/1)	0% (0/2)	0% (0/6)	0% (0/9)
hr.algebra.webshop2024.DTO	100% (1/1)	100% (12/12)		100% (25/25)
hr.algebra.webshop2024.Mapper	100% (1/1)	100% (3/3)	100% (4/4)	100% (23/23)

Empty cells represent situations where branches were detected in the source code. The result show that EvoMaster performed better than EvoSuite in class coverage, but EvoSuite showed better results in method, branch and line coverage. Combined, achieved coverage of classes was 94.5%, method coverage 82.8%, branch coverage of 57%, and line coverage 76,2%. The tests were generated on Windows 10 machine with AMD Ryzen 5 3600 6-Core Processor, 3593 Mhz, and 16 GB RAM.

4. Discussion

Implementation of unit and integration tests that will ensure the correct functioning of the software is critical to any project. In practice, it often happens that due to the short deadlines available to developers to complete functionality and ensure timely deliveries, implementation of tests is left for later stages of the project. This can have significant negative consequences on the quality of the delivered software. Solutions that enable automatic test generation can significantly contribute to the improvement of that segment of the entire software delivery industry and to maintaining the expected level of quality. In cases where tools like EvoMaster generate about 50 tests within an hour or two, or EvoSuite few hundred tests within few hours, this can contribute to achieving that goal. Although it is still not realistic for test generators to automate the writing of tests that will cover most of the implemented code, with the growth of the community using such tools, the additional popularization of test writing, the incorporation of automated testing into CI/CD (*Continuous Integration/Continuous Delivery*) cycles, and the increase of the community that supports this approach, with each new version we will be closer to such solutions.

 doi:10.20944/preprints202407.1767.v1

By combining different tools, approaches and methodologies, like shown in this research, it is possible to use the best properties of all domains and achieve even better results about the accuracy of individual tools, which confirms the thesis stated at the beginning of this research.

Given that we are witnessing more and more frequent cyberattacks [27], especially in case when installing an update can disrupt a lot of IT systems [28], it is crucial for programmers to be aware that they are writing secure code that will be resistant to such attacks and failures. In future work, we will focus on combining the results of the test generators to focus on writing secure code to mitigate the risk of potential cyberattacks.

5. Conclusion

Sophisticated white-box test generators rely on concolic execution and search-based methods that aim to find test cases which exercise new code paths and lead to increased code coverage of the generated test suite.

One of the major obstacles in this approach are logical expressions that cannot be reversed effectively i.e., initial values that satisfy the condition cannot be derived. Examples of this are conditions that involve functions which are mathematically irreversible, such as the cryptographic hash function, or functions that call services external to the program's process, such as file or network calls.

An example of the former case is a function which accepts the URL string and commences the processing of a downloaded file or returns an error message in case the URL is invalid. It is not possible to derive a string that satisfies the validity condition because string validity is decided externally by attempting to make a network request to fetch the given resource.

However, a possible recourse in such a situation is to extract the untestable code block and then re-apply the search-based methods to generate inputs that maximize code coverage in the extracted code block. This effectively means supplementing the system test generators with unit tests.

As we are not aware of a unit test generator for Java that can identify and extracting unreachable code blocks, we relied on EvoSuite, a whitebox test generator for individual classes, to generate unit tests for classes that were poorly covered by the EvoMaster, a system test generator for applications using REST API.

Additionally, we used a call graph visualizer for identifying classes that are closely functionally related to the application's REST controllers and have poor code coverage. Such classes have been tested with unit test generator and improvements in code coverage have been observed.

One of the limitations of this approach is that it is limited to pre-defined code structure. Lines of code that follow the irreversible condition evaluation cannot effectively be tested with EvoSuite unless they are in a separate class. A possible improvement to this approach could use code transformation to extract such code blocks into a form, such as a synthetic class or a method, that can be tested with a unit-test generator.

With the results obtained using the EvoMaster test generator, it can be concluded that most of the generated tests can be obtained within 1-2 hours, depending on the complexity of the application being analyzed. With EvoSuite, the number of generated tests, around 5 minutes per class can generate few hundred tests in few hours, which is a lot faster in comparison to manual writing tests by developers. The limitations of evolutionary (genetic) algorithms were obvious in this case where the mutation of initial test cases didn't change the target of the generated test adequately to cover majority of test cases for the same method being tested.

Author Contributions: Conceptualization, I.J.; methodology, I.J.; software, I.J.; validation, A.R.; formal analysis, A.R.; investigation, I.J.; resources, I.J.; data curation, I.J. and A.R.; writing—original draft preparation, A.R.; writing—review and editing, I.J. and A.R.; visualization, I.J. and A.R.; supervision, A.R.; project administration, I.J. and A.R. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Braione, P., Denaro, G., Mattavelli, A. et al. Software testing with code-based test generators: data and lessons learned from a case study with an industrial software component. *Software Qual J* **22**, 311–333 (2014). https://doi.org/10.1007/s11219-013-9207-1
2. Cseppentő, L., & Micskei, Z. (2017). Evaluating code-based test input generator tools. Software Testing, Verification and Reliability, 27(6), e1627. doi:10.1002/stvr.1627
3. Kicillof, Nicolás & Grieskamp, Wolfgang & Tillmann, Nikolai & Braberman, Víctor. (2007). Achieving both model and code coverage with automated gray-box testing. Proceedings of the 3rd International Workshop Advances in Model Based Testing, AMOST 2007. 1-11. 10.1145/1291535.1291536.
4. G. Fraser and A. Arcuri, "Whole Test Suite Generation," in IEEE Transactions on Software Engineering, vol. 39, no. 2, pp. 276-291, Feb. 2013, doi: 10.1109/TSE.2012.14.
5. C. C. Michael, G. McGraw and M. A. Schatz, "Generating software test data by evolution," in *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1085-1110, Dec. 2001, doi: 10.1109/32.988709.
6. Sebastian Vogl, Sebastian Schweikl, and Gordon Fraser. 2021. Encoding the certainty of boolean variables to improve the guidance for search-based test generation. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '21). Association for Computing Machinery, New York, NY, USA, 1088–1096. https://doi.org/10.1145/3449639.3459339.
7. Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-based seed object synthesis for search-based unit testing. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021). Association for Computing Machinery, New York, NY, USA, 1068–1080. https://doi.org/10.1145/3468264.3468619.
8. Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. 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 (ISSTA 2020). Association for Computing Machinery, New York, NY, USA, 440–451. https://doi.org/10.1145/3395363.3397358.
9. Andrea Arcuri, Man Zhang, and Juan Galeotti. 2024. Advanced White-Box Heuristics for Search-Based Fuzzing of REST APIs. ACM Trans. Softw. Eng. Methodol. 33, 6, Article 142 (July 2024), 36 pages. https://doi.org/10.1145/3652157.
10. Jang, D.; Kim, J.; Kim, J.; Im, W.; Jeong, M.; Choi, B.; Kil, C. On the Analysis of Coverage Feedback in a Fuzzing Proprietary System. *Appl. Sci.* **2024**, *14*, 5939. https://doi.org/10.3390/app14135939.
11. Du, C.; Guo, Y.; Feng, Y.; Zheng, S. HotCFuzz: Enhancing Vulnerability Detection through Fuzzing and Hotspot Code Coverage Analysis. Electronics 2024, 13, 1909. https://doi.org/10.3390/electronics13101909.
12. He, X.; Wang, P.; Lu, K.; Zhou, X. ObFuzzer: Object-Oriented Hybrid Fuzzer for Binaries. Appl. Sci. 2022, 12, 9782. https://doi.org/10.3390/app12199782.
13. Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2022. Improving test case generation for REST APIs through hierarchical clustering. In Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE '21). IEEE Press, 117–128. https://doi.org/10.1109/ASE51524.2021.9678586.
14. Chen, Z.; Lu, Y.; Zhu, K.; Yu, L.; Zhao, J. Fast Format-Aware Fuzzing for Structured Input Applications. *Appl. Sci.* **2022**, *12*, 9350. https://doi.org/10.3390/app12189350.
15. Spring Boot. Available online: https://spring.io/projects/spring-boot (accessed on 2nd June 2024).
16. ASM. Available online: https://asm.ow2.io/ (accessed on 2nd June 2024).
17. J. Dean, D. Grove and C. Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference*, Aarhus, Denmark.
18. D. F. Bacon and P. F. Sweeney. 1996. Fast static analysis of C++ virtual function calls. In Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications., San Jose, USA.
19. JaCoCo. Available online: https://www.eclemma.org/jacoco/ (accessed on 2nd June 2024).
20. React. Available online: https://react.dev/ (accessed on 3rd June 2024).
21. React-force-graph. Available online: https://github.com/vasturiano/react-force-graph (accessed on 4th June 2024).
22. EvoMaster. Available online: https://github.com/EMResearch/EvoMaster (accessed on 5th June 2024).
23. G. Fraser and A. Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *13th European conference on Foundations of software engineering*, Szeged, Hungary.
24. EvoSuite. Available online: https://www.evosuite.org (accessed on 6th June 2024).
25. Lombok. Available online: https://projectlombok.org/ (accessed on 14th July 2024).
26. Spring Data JPA. Available online: https://spring.io/projects/spring-data-jpa (accessed on 14th July 2024).

 doi:10.20944/preprints202407.1767.v1

27. Top Cybersecurity Statistics for 2024. Available online: https://www.cobalt.io/blog/cybersecurity-statistics-2024 (accessed on 14th July 2024).
28. CrowdStrike Windows Outage. Available online: https://www.forbes.com/sites/kateoflahertyuk/2024/07/19/crowdstrike-windows-outage-what-happened-and-what-to-do-next/ (access on 21st July 2024).