# Challenges in Using Search-Based Test Generation to Identify Real Faults in Mockito

Gregory Gay

University of South Carolina, Columbia, SC, USA,
greg@greggay.com

**Abstract.** The cost of test creation can potentially be reduced through automated generation. However, to impact testing practice, automated tools must be at least as effective as manual generation. The Mockito project—a framework for mocking portions of a system—offers an opportunity to assess the capabilities of test generation tools on a complex real-world system. We have identified 17 faults in the Mockito project, added those to the Defects4J database, and assessed the ability of the EvoSuite tool to detect these faults. In our study, EvoSuite was only able to detect one of the 17 faults. Analysis of the 16 undetected faults yields lessons in how to improve generation tools. We offer these faults to the community to assist in benchmarking future test generation advances.

**Keywords:** Search-based testing, automated unit test generation, real faults

## 1 Introduction

Software testing is a notoriously expensive and difficult activity. With the exponential growth in the complexity of software, the cost of testing has only continued to rise. Much of the cost of testing can be traced directly to the human effort required to conduct most testing activities—such as producing test input and expected outputs. However, such effort is often in service of goals that can be framed as *search* problems, and automated through the use of optimization algorithms [1].

Test case generation can naturally be seen as a search problem. There are hundreds of thousands of test cases that could be generated for any particular SUT. From that pool, we want to select—systematically and at a reasonable cost—those that meet our goals and are expected to be fault-revealing [1]. Automated unit test generation tools have become very effective—even covering more code than tests manually constructed by developers [4]. However, to make an impact on testing practice, automated test generation techniques must be *as effective*, if not more so, at detecting faults as human-created test cases [7].

The Mockito project[1] offers an opportunity to assess the capabilities of test generation tools. Mockito is a *mocking framework* for Java unit testing, allowing users to create customized stand-ins (*mock objects*) for classes in a system, permitting testers to isolate units of a system from their dependencies. Rather than performing the functions

[1] http://mockito.org/

of the mocked object, the mock instead issues preprogrammed output. Mockito is an essential tool of modern development, and is one of the most used Java libraries [8].

Mockito serves as an interesting benchmark for two reasons. First, it is a complex project. Much of its functionality is, naturally, related to the creation and manipulation of mock objects. The inputs required by many Mockito functions are complex objects—which are difficult for many test case generators to produce [2]. Second, Mockito is a mature project, having undergone eight years of active development. Recent Mockito faults are unlikely to be the simple syntactic mistakes modeled by mutation coverage. Faults that emerge in a mature project are more likely to require specific, difficult to trigger, combinations of input and method calls. If a test generation tool can detect such faults, then it is likely ready for real-world use. If not, then by studying these faults—and others like them—we may be able to learn lessons that will improve these tools.

We have identified 17 real faults in the Mockito project, and have added them to the Defects4J fault library [6]. We generated test suites using the search-based EvoSuite generation framework [3], and measured the suites' ability to cover the affected classes and detect each fault. EvoSuite was only able to detect one of the 17 faults discovered in the project. Some of the issues preventing fault detection include poor guidance for the fitness function, the need for complex input to methods and object constructors, specific environmental configurations and factors, uncertainty in which classes to generate tests for, and simplistic handling of interface changes between software versions. We have made this set of Mockito faults available to provide data and examples for benchmarking future test generation advances.

## 2 Study

Recent studies have assessed the capabilities of test generation tools on faults in open-source projects [7], but more data is needed to understand where such tools excel and where they need to be improved. In this study, we have generated tests using the search-based EvoSuite framework [3] on classes of the Mockito project. In doing so, we wish to answer the following research questions:

1. Can EvoSuite detect faults found in Mockito?
2. What factors prevented EvoSuite from detecting faults?

In order to answer these questions, we have performed the following experiment:

1. **Derived Faults:** We have identified 17 real faults in the Mockito project, and added them to the Defects4J fault database (See Section 2.1).
2. **Generated Test Cases:** For each fault, we generated tests on the fixed version of fault-affected classes. (See Section 2.2).
3. **Removed Non-Compiling Tests:** Any tests that do not compile or that fail on the fixed system are automatically removed (See Section 2.2).
4. **Assessed Fault-finding and Coverage:** For each suite and fault, we measure the number of tests that pass on the fixed version and fail on the faulty version. We also record the achieved code coverage.
5. **Analyzed Faults That Were Not Detected:** For each undetected fault, we examined the report and source code to identify possible detection-preventing factors.

### 2.1 Fault Extraction

Using Mockito's version control and issue tracking systems, we have identified 17 faults. Each fault is required to meet three properties. First, the fault must be related to the source code. For each reported issue, we attempted to identify a pair of code versions that differ only by the minimum changes required to address the fault. The "fixed" version must be explicitly labeled as a fix to an issue, and changes imposed by the fix must be to source code, not to other project artifacts such as the build system. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix to the fault must be isolated from unrelated code changes such as refactorings.

In order to focus on the faults typical of a mature project, we limited our extraction to the GitHub-based issue tracking system that Mockito began using in July 2014 (previously, Google Code was used). To help identify candidate faults, we used automation provided by Defects4J [6]—a library of faults from five open-source Java programs and tools for assessing tests intended to find such faults.

We have added Mockito as a sixth Defects4J project. This consisted of developing build files that work across project versions, extracting candidate faults, ensuring that each candidate could be reliable reproduced, and minimizing the "patch" used to distinguish fixed and faulty classes. Following this process, we extracted 17 faults from a pool of 89 candidate faults. Six of the 17 faults were "false-positives", fixes to issues reported in the old issue tracker that shared an issue ID with issues in the newer tracking system. As these six faults met reasonable system maturity and complexity thresholds, we also added them to Defects4J.

The faults used in this study can be accessed by cloning the `bug-mining` branch of `https://github.com/Greg4cr/defects4j`. Additional data about each fault can be found at `http://greggay.com/data/mockito/mockitofaults.csv`, including commit IDs, fault descriptions, and a list of triggering tests. We plan to add additional faults and improvements in the future.

### 2.2 Test Generation and Removal

EvoSuite applies a genetic algorithm in order to evolve test suites over several generations, forming a new population by retaining, mutating, and combining the strongest solutions. It is actively maintained and has been successfully applied to a variety of projects [7]. In this study, we used EvoSuite version 1.0.3 with the default fitness function—a combination of branch, context branch, line, exception, weak mutation, method-output, top-level method, and no-exception top-level method coverage. Given the potential difficulty in achieving coverage over Mockito classes, the search budget was set to 10 minutes. To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 30 trials for each fault by generating tests for the classes patched to fix the fault.

Tests are generated from the fixed version of the system and applied to the faulty version in order to eliminate the oracle problem. In practice, this translates to a regression testing scenario. Due to changes introduced to fix faults, such as altered method

| ID | Fault Detected | # Tests Generated | # Tests Removed | % LC | % BC | % EC | % WMC | % OC | % MC | % MNEC | % CBC | Resulting % Coverage |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | X | 4.23 | 0.00 | 10.00 | 7.00 | 100.00 | 2.00 | 2.00 | 25.00 | 8.00 | 3.00 | 20.00 |
| 2 | ✓ | 92.00 | 1.00 | 86.97 | 87.95 | 100.00 | 62.85 | 50.00 | 100.00 | 50.50 | 87.95 | 72.47 |
| 3 | X | 4.31 | 0.00 | 10.00 | 8.00 | 100.00 | 2.00 | 2.00 | 25.00 | 8.00 | 3.00 | 20.00 |
| 4 | X | 84.70 | 0.00 | 73.67 | 85.33 | 100.00 | 24.50 | 0.00 | 100.00 | 1.00 | 85.33 | 46.67 |
| 5 | X | 15.03 | 0.00 | 61.80 | 77.63 | 98.90 | 77.00 | 100.00 | 100.00 | 100.00 | 77.63 | 87.00 |
| 6 | X | 60.13 | 0.00 | 100.00 | 100.00 | 100.00 | 100.00 | 44.50 | 100.00 | 100.00 | 100.00 | 93.00 |
| 7 | X | 14.82 | 0.00 | 12.86 | 20.86 | 92.86 | 12.82 | 0.00 | 10.00 | 0.00 | 20.86 | 26.00 |
| 8 | X | 14.97 | 0.00 | 12.97 | 20.93 | 100.00 | 12.93 | 0.00 | 10.00 | 0.00 | 20.93 | 26.00 |
| 9 | X | 1.00 | 0.00 | 33.00 | 33.00 | 100.00 | 0.00 | 0.00 | 100.00 | 50.00 | 33.00 | 38.00 |
| 10 | X | 2.00 | 0.00 | 8.00 | 10.00 | 100.00 | 0.00 | 0.00 | 100.00 | 33.00 | 10.00 | 24.00 |
| 11 | X | 1.00 | 0.00 | 6.00 | 12.00 | 100.00 | 20.00 | 0.00 | 10.00 | 0.00 | 12.00 | 20.67 |
| 12 | X | 1.00 | 0.00 | 12.00 | 11.00 | 100.00 | 0.00 | 0.00 | 100.00 | 50.00 | 11.00 | 29.00 |
| 13 | X | 10.07 | 0.00 | 45.90 | 59.78 | 100.00 | 25.00 | 67.00 | 100.00 | 75.00 | 59.77 | 62.93 |
| 14 | X | 41.89 | 4.63 | 81.59 | 83.52 | 93.48 | 67.81 | 62.63 | 99.84 | 83.96 | 83.26 | 80.02 |
| 15 | X | 16.70 | 7.37 | 65.36 | 64.09 | 92.48 | 55.42 | 50.56 | 85.56 | 80.97 | 64.09 | 64.00 |
| 16 | X | 73.90 | 7.57 | 86.43 | 84.91 | 80.68 | 83.33 | 38.68 | 100.00 | 77.43 | 84.41 | 71.36 |
| 17 | X | 35.50 | 3.43 | 99.04 | 97.43 | 95.21 | 94.91 | 57.50 | 100.00 | 100.00 | 97.43 | 91.03 |

**Table 1.** Average test generation results for each fault—whether the fault was detected, number of generated tests, number of non-compiling tests, line coverage (LC), branch coverage(BC), exception coverage (EC), weak mutation coverage (WMC), method-output coverage (OC), method coverage (MC), no-exception top-level method coverage (MNEC), context branch coverage (CBC), and the resulting average across all coverage metrics.

signatures or new classes, some tests may not compile on the faulty version of the system. We have automatically removed such tests. We have also removed tests that fail on the fixed version of the system, as these do not assist in identifying faults. On average, 4.48% of the tests are removed from each suite. More statistics are included in Table 1.

## 3 Results and Discussion

The results of our experiment can be seen in Table 1. In our study, only one of the 17 faults was detected—Fault 2. This particular fault—revolving around incorrect handling of negative time values—is an excellent example of the kind of fault that automated test generation is able to handle. The code fix adds conditional behavior to handle time input. By covering the new branches, the tests are guided to detect the fault in all 30 trials. However, EvoSuite failed to detect the other 16 faults. Therefore, our next step was to examine these faults to identify factors preventing detection. These factors include:

**Poor Guidance for the Fitness Function:** While EvoSuite is often able to achieve reasonable levels of coverage across Mockito classes, coverage is sometimes quite low. While coverage does not guarantee fault detection, unexecuted code cannot reveal faults [5]. One reason coverage may not be achieved is that the code offers no guidance to the search tool in selecting *better* test suites.

Many fitness functions are designed to measure the distance from optimality of generated test cases. However, it is not always obvious how to calculate this distance. The code that must be covered to detect Fault 12[2] provides a good example. Both branches use the `instanceof` operator. Without a method of determining the "distance" between class types, the search devolves into a random search.

[2] `https://github.com/mockito/mockito/commit/7a647a702c8af81ccf5d37b09c11529c6c0cb1b7`

**Complex Input is Required to Trigger a Fault:** A challenge for test generation techniques is generating inputs of complex data types [2]. As Mockito generates objects that mimic other objects, many of its methods require complex objects as input. Even in cases where coverage is high, test generators may have difficulty producing the intricate, highly-specific, input required to detect that fault.

Consider Fault 13[3], which occurs when Mockito's verification capabilities are invoked on a method call that, itself, has an embedded method call within it. Triggering this fault requires generating two different mock objects, then embedding a call to one object within a call to the second. Coverage alone is unlikely to suggest such input. Rather, fitness functions that incorporate domain expertise may be needed to help generate more complex input scenarios. Promising work has been conducted using grammars to produce complex input [2].

**Complex Input is Required to *Generate Any Tests*:** Unit tests instantiate an object and call the methods offered by that object. At times, objects must be provided with input when they are instantiated (there is no "default" constructor). Many of the code changes made to fix Fault 3[4] are contained within one method. EvoSuite not only fails to fully cover this method, it fails to invoke this method at all. In this case, EvoSuite attempts to instantiate the `InvocationMaster` class, but many of these attempts fail due to invalid input. EvoSuite cannot cover the methods of an object that it cannot instantiate.

**Faults Require Specific Environmental Factors:** Fault 5[5] revolves around an undesired dependency on the JUnit framework. Fixing this fault requires code changes—yet, coverage of this code will not reveal this fault. Rather, the fault is detected when JUnit is removed from the local classpath. This is an example of a fault that depends on environmental factors—in this case, the classpath used to compile code. EvoSuite does manipulate certain environmental factors, such as file system access, but more examination of such factors is needed in future test generation research.

**Fault Detection Requires Generating Tests for Related Classes:** The classes affected by Fault 6[6] offer another interesting example. Mock objects can be configured to return different values based on the type of function input. Due to this fault, a mock can produce a value intended for certain data types when a `null` object is passed instead of the intended type. The fault-fixing changes are primarily in methods that do not require input—methods that are called by Mockito's argument matchers. Because these methods do not require input, this fault cannot be detected without generating tests for the argument matcher classes that, in turn, call these methods. Under normal circumstances, EvoSuite could produce the required `null` input, but tests would need to be generated for classes that do not contain faulty code, and instead depend on faulty code. Some consideration should be given to which classes are used when generating tests, and the dependencies between those classes.

**Changes to Code Invalidate Test Cases:** When tests are generated on one version of a system and applied to another, code changes such as the addition of new classes or altered method signatures can result in tests that do not compile on one version. In

[3] `https://code.google.com/archive/p/mockito/issues/138`
[4] `https://github.com/mockito/mockito/commit/3eec7451d6c83c280743c39b39c77a179abb30f9`
[5] `https://github.com/mockito/mockito/issues/152`
[6] `https://github.com/mockito/mockito/commit/dc205824dbc289acbcde919e430176ad72da847f`

this study, we removed those tests. This may prevent fault detection. Fault 17[7] affects the ability to set mock objects as serializable. EvoSuite is correctly guided to create serializable mock objects. However, any time this occurs, interactions take place with a new class. These tests are removed, as they do not compile on the faulty version of the system. In normal practice, this is not an issue, as tests are generated on the version they are applied to, but during regression testing, similar issues may occur. Intelligent strategies are needed to generate tests that compile across multiple versions of systems.

## 4 Conclusion

The capabilities of test generation techniques have increased. Yet, from the examples extracted from the Mockito project, we can see that there are still fault-detection hurdles to overcome. EvoSuite was only able to detect one of the 17 faults. Some of the issues preventing fault detection include poor guidance for the fitness function, the need for complex input to methods and object constructors, environmental factors, uncertainty in which classes to generate tests for, and simplistic handling of interface changes between multiple software versions. We hope that the set of faults extracted from Mockito will provide data and examples for benchmarking new test generation advances.

## References

1. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. Software Engineering, IEEE Transactions on 36(6), 742–762 (2010)
2. Feldt, R., Poulding, S.: Finding test data with specific properties via metaheuristic search. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). pp. 350–359 (Nov 2013)
3. Fraser, G., Arcuri, A.: Whole test suite generation. Software Engineering, IEEE Transactions on 39(2), 276–291 (Feb 2013)
4. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated white-box test generation really help software testers? In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. pp. 291–301. ISSTA 2013, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2483760.2483774
5. Gay, G., Staats, M., Whalen, M., Heimdahl, M.: The risks of coverage-directed test case generation. Software Engineering, IEEE Transactions on PP(99) (2015)
6. Just, R., Jalali, D., Ernst, M.D.: Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 437–440. ISSTA 2014, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2610384.2628055
7. Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A.: Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). ASE 2015, ACM, New York, NY, USA (2015)
8. Weiss, T.: We analyzed 30,000 GitHub projects - here are the top 100 libraries in Java, JS and Ruby (2013), http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/

[7] https://github.com/mockito/mockito/commit/77cb2037314dd024eb53ffe2e9e06304088a2d53