Using Search-Based Test Generation to Discover Real Faults in Guava

Hussein Almulla, Alireza Salahirad, Gregory Gay

University of South Carolina, Columbia, SC, USA,[**]
[halmulla, alireza]@email.sc.edu, greg@greggay.com

Abstract. Testing costs can be reduced through automated unit test generation. An important benchmark for such tools is their ability to detect *real faults*. Fault databases, such as Defects4J, assist in this task. The Guava project—a collection of Java libraries from Google—offers an opportunity to expand such databases with additional complex faults. We have identified 11 faults in the Guava project, added them to Defects4J, and assessed the ability of the EvoSuite framework to detect these faults. Ultimately, EvoSuite was able to detect three faults. Analysis of the remaining faults offers lessons in how to improve generation tools. We offer these faults to the community to assist future benchmarking efforts.

Keywords: Search-based test generation, automated test generation, software faults

1 Introduction

With the growing complexity of software, the cost of testing has grown as well. Automation of tasks such as unit test creation can assist in controlling that cost. One promising form of automated test generation is *search-based* generation. Given a measurable testing goal, and a fitness function capable of guiding the search towards that goal, powerful optimization algorithms can select test inputs able to meet that goal [3].

When testing, developers ultimately wish to detect faults. Therefore, to impact testing practice, automated generation techniques must be effective at detecting the complex faults that manifest in real-world software projects [7]. By offering examples of such faults, fault databases—such as Defects4J [6]—allow us to benchmark generation tools against realistic case examples. Importantly, Defects4J can be expanded to include additional systems and example faults.

The Guava project [1] offers an excellent expansion opportunity. Guava is an open-source set of core libraries for Java, developed by Google, that include collection types, graph libraries, functional types, in-memory caching, and numerous other utilities. Guava is an essential tool of modern development, and is one of the most used libraries [8].

Guava serves as an interesting benchmark subject for two reasons. First, much of its functionality is, naturally, related to the creation and manipulation of complex objects. Guava defines a variety of new data structures, and functionality related to those structures. Generation and initialization of complex input is an outstanding challenge area

[**] This work is supported by National Science Foundation grant CCF-1657299.

[1] https://github.com/google/guava

for automated generation [1]. Second, Guava is a mature project. Faults in Guava—particularly recent faults—are unlikely to resemble the simple syntactic mistakes modeled by mutations. Rather, we expect to see faults that require specific, difficult to trigger, combinations of input and method calls. Generation tools that can detect such faults are likely to be effective on other real-world projects. If not, then by studying these faults, we may be able to learn lessons that will improve these tools.

We have identified 11 real faults in the Guava project, and added them to Defects4J. We generated test suites using the EvoSuite framework [3], and assessed the ability of these suites to detect nine of the faults[2]. Ultimately, EvoSuite is able to detect three of the nine studied faults. Some of the issues preventing fault detection include the need for specific input values, data types, or sequences of method calls—generally factors that cannot be addressed through code coverage alone. We have made these faults available to provide data and examples that could benefit future test generation research.

2 Study

In this study, we have extracted faults from the Guava project. We have generated tests for the fixed version of each class using the EvoSuite framework [3], and applied those tests to the faulty version in order to assess the efficacy of generated suites. In doing so, we wish to answer the following research questions: (1) *can EvoSuite detect the extracted faults?*, and (2), *what factors prevented fault detection?*

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

1. **Extracted Faults:** We have identified 11 real faults in the Guava project, and added them to the Defects4J fault database (See Section 2.1).
2. **Generated Test Cases:** For nine of the faults, we generated 10 suites per fault using the fixed version of each class-under-test (CUT). We repeat this process with a two-minute and a ten-minute search budget per CUT (See Section 2.2).
3. **Removed Non-Compiling Tests:** Any tests that do not compile, or that return inconsistent results, are automatically removed (See Section 2.2).
4. **Assessed Fault-finding Efficacy:** For each budget and fault, we measure the likelihood of fault detection. For each undetected fault, we examined the report and source code to identify possible detection-preventing factors.

2.1 Fault Extraction

Defects4J is an extensible database of real faults extracted from Java projects [6][3]. Currently, it consists of 395 faults from six projects. For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose each fault, and a list of classes and lines of code modified to fix the fault.

We have added Guava to Defects4J. This consisted of developing build files that work across project versions, extracting candidate faults using Guava's version control and issue tracking systems, ensuring that each candidate could be reliable reproduced, and minimizing the ”patch” used to distinguish fixed and faulty classes.

[2] Two faults were omitted from the case study as they require the use of JDK 7 (see Section 2).

[3] Available from `http://defects4j.org`

For inclusion in the final dataset, 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.

One property of all Defects4J faults is that the commit message for the "fixed" version reference a reported issue in the project's tracking system (i.e., "fixes #2345"). Fulfilling this property has not been a problem for the six existing projects, as the developers of those projects have used a standard commit message format. However, Guava commits do not follow a standard format—many "fixes" do not reference a reported issue. To maintain continuity with the other Defects4J projects, we restricted our search to fixes that do make an explicit reference. Following this process, we extracted 11 faults from a pool of 63 candidate faults that reference an explicit issue. In the future, we may allow commits without explicit references in order to mine additional faults.

One additional limiting factor is that particular Java Development Kit versions must be installed and used to build certain versions of Guava. Due to language changes, JDK 7 must be used to build faults 10 and 11. Faults 1-9 can be built using JDK 8. Recently, the decision was made to require that all new additions to Defects4J be compatible with JDK 8. Faults 10 and 11 will still be made available, but will not be included in the core Defects4J database or used in our case study.

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/guava/guavafaults.csv`, including commit IDs, fault descriptions, and a list of triggering tests. Later, these faults will be migrated into the `master` branch at `http://defects4j.org`. 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 [7]. In this study, we used EvoSuite version 1.0.5 with a combination of three fitness functions—Branch, Exception, and Method Coverage—a combination recently found to be generally effective at detecting faults [5].

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. Given the potential difficulty in achieving coverage over Guava classes, two search budgets were used—two and ten minutes, a typical and an extended budget [4]. 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 10 trials for each fault and search budget.

Generation tools may generate flaky (unstable) tests [7]. For example, a test case that makes assertions about the system time will only pass during generation. We auto-

Fault	Budget	Fault Detected	Likelihood of Detection	Branch Coverage (Covered/Total Goals)	Suite Size	Suite Length	Number of Tests Removed
1	2 min	X	0.00%	11.83% (22.60/191.00)	6.90	26.10	0.90
	10 min	X	0.00%	73.25% (139.90/191.00)	38.70	180.80	11.10
2	2 min	X	0.00%	92.47% (82.30/89.00)	26.10	65.60	0.00
	10 min	X	0.00%	93.60% (83.30/89.00)	26.90	70.00	0.00
3	2 min	✓	90.00%	96.64% (132.40/137.00)	65.90	114.90	0.00
	10 min	✓	90.00%	97.52% (133.60/137.00)	67.70	117.00	0.00
4	2 min	✓	60.00%	67.56% (83.10/123.00)	91.40	270.40	1.20
	10 min	✓	100.00%	92.03% (113.20/123.00)	130.40	424.20	1.60
5	2 min	X	0.00%	32.38% (13.60/42.00)	4.70	49.40	0.00
	10 min	X	0.00%	76.31% (30.70/40.20)	14.50	96.90	0.00
6	2 min	X	0.00%	3.10% (30.90/1008.00)	3.00	11.60	0.00
	10 min	X	0.00%	3.10% (31.10/1008.00)	3.40	13.00	0.10
7	2 min	X	0.00%	2.32% (23.30/1005.00)	3.40	20.50	0.00
	10 min	X	0.00%	1.91% (19.20/1005.00)	2.20	15.50	0.30
8	2 min	✓	10.00%	21.51% (11.40/53.00)	7.30	35.10	0.00
	10 min	✓	60.00%	91.89% (48.70/53.00)	42.40	214.90	1.20
9	2 min	X	0.00%	3.54% (5.60/158.00)	3.40	8.40	0.00
	10 min	X	0.00%	57.47% (90.80/158.00)	74.30	175.30	0.20

Table 1. Test generation results for each fault and search budget—likelihood of fault detection, average achieved branch coverage (covered/total branches), average number of tests, average suite length, and average number of tests removed.

matically remove flaky tests[4]. First, non-compiling test cases are removed. Then, each test is executed on the fixed CUT five times. If results are inconsistent, the test case is removed. On average, 0.92 tests are removed from each suite.

3 Results and Discussion

In Table 1, we list—for each search budget—whether EvoSuite was able to detect each fault and, if so, the likelihood of detection (the proportion of suites that detected the fault). We also list the average Branch Coverage attained over the ten trials, the average number of tests in the generated suites, the average suite length (number of test steps), and the average number of tests removed.

From Table 1, we can see the three of the nine faults were detected (Faults 3, 4, and 8). Of these, Fault 3 was detected the most reliably (90% likelihood for both budgets). This fault, dealing with incorrect rounding[5], is a classic example of the types of faults that automated generation excels at. Branch Coverage drives the search towards the affected code and towards differing output between versions.

Fault 4[6] was also detected reliably. The faulty version uses a non-standard ASCII character in the `toString()` function for class `Range`. This is a relatively easy fault to catch—any call to *toString()* with a valid `Range` object will result in differing output between faulty and fixed versions. With the shorter search budget, EvoSuite is somewhat less likely to call the function and somewhat more likely to set up an invalid range. However, the longer budget ensures that the fault is caught by all suites.

Fault 8 involves the computation of the intersection of `RegularContiguousSet` objects when one is a singleton[7]. One of the changes made to fix the fault is a shift from

[4] This process is documented in more detail in [7] and [4].

[5] `https://github.com/google/guava/commit/1b1163b7e2c121d4a5b25b8966714201551976c4`

[6] `https://github.com/google/guava/commit/c6e21a35f3113a7a952a9615a0e92dcf1dd4bfb3`

[7] `https://github.com/google/guava/commit/44a2592b04490ad26d2bc874f9dbd4c1146cc5de`

$<$ in the return statement of the `isEmpty()` method to $<=$. Any test case where the two compared variables are the same will now detect the fault. A longer search budget increases the number of suites that detect the fault (10% to 30%), but this is still a clear case of a fault that requires not just coverage, but *picking specific input*.

EvoSuite failed to detect the other six faults. Therefore, our next step was to examine these faults to identify factors preventing detection. These factors include:

Specific Input Values are Required: As seen in Fault 8, it is not enough to simply cover a line. At times, specific input values are required to trigger and detect a fault. In the case of Fault 8, the generator is able to stumble on these inputs given enough time. In other cases, such as with Fault 2[8], not enough context is offered to the generator. Because of this fault, splitting a string with a zero-width regular expression pattern would result in single-character strings on either end of the split being dropped. The fix to the code changes a $>=$ to a $>$, but unless input matching this particular corner case is used, the fault will not be discovered.

Specific Data Types are Required for Input: Guava includes functionality for iterating over lists that is intended to function regardless of the *type* of list used. Fault 9[9] illustrates the difficulty of verifying such functionality. Unlike sets, lists typically allow duplicate elements. This is not universally true, however. Therefore, if a list type is used that does not allow duplicates, then the affected code in Guava will throw an exception. This is another case that coverage cannot handle, as coverage can be obtained using any type of list. Detecting the fault requires choosing a specialized data type.

Inputs are Instances of Complex Data Types: Generating input for complex data types is still an open challenge for automated generation [1]. If the generator cannot produce and manipulate input of such types, it may not be able to cover code, reducing the possibility of triggering faults. Fault 6[10] is one such example. This fault revolves around the wrong cause of removal being listed for items in a cache. To discover this fault, EvoSuite must generate and initialize an instance of the class `LocalCache`. In addition, this class is a generic type, further complicating automated generation [2].

A Specific Series of Method Calls Must Be Generated: Each unit test consists of a series of one or more calls to methods in the CUT. Rather than specific input, at times, triggering a fault requires a specific sequence of calls. Fault 5[11] is one such example. In this case, a long sequence of nested `Futures.transform(...)` calls on the same object will indefinitely hang because a `StackOverflowException` is thrown and swallowed. Detecting this fault requires not only input that triggers an exception, but a sequence of transformation calls on that input.

Fault 1[12] offers a second example of this factor. `MinMaxPriorityQueue` fails to remove the correct object after a sequence of multiple `add` and `remove` calls—specifically, certain elements may be iterated more than once if elements are removed during iteration. It would not be unusual to see a sequence of calls in a generated test case. However, the example tests created by humans to reproduce this fault include

8 https://github.com/google/guava/commit/55524c66de8db4c2e44727b69421c7d0e4f30be0
9 https://github.com/google/guava/commit/1a1b97ee1f065d0bc52c91eeeb6407bfaa6cbea1
10 https://github.com/google/guava/commit/0a686a644ca5cefb9e7bf4a38b34bf4ede9e75aa
11 https://github.com/google/guava/commit/52b5ee640da780e0fd2502ec995436fcdc93e03e
12 https://github.com/google/guava/commit/2ef955163b3d43e7849c1929ef4e5d714b93da96

relatively long sequences of calls. The suite minimization and bloat control mechanisms used to control suite size in automated generation are designed to avoid a long series of calls that do not contribute to code coverage—actively discouraging the generation of the very type of test cases that would detect this fault.

Many of these factors cannot be solved through increasing code coverage. Rather, they require context from the project. Methods of gleaning that context, either through seeding from existing test cases or data mining of project elements, may assist in improving the efficacy of test generation.

4 Conclusion

We have identified 11 real faults in the Guava project, and added them to the Defects4J fault database. To study the capabilities of modern test generation tools, we generated test suites using the EvoSuite framework. Ultimately, EvoSuite is able to detect three of the nine studied faults. Some of the issues preventing fault detection include the need for specific input values, data types, or sequences of method calls—generally factors that cannot be addressed through code coverage alone. We have made these faults available to provide data and examples that could benefit future test generation research.

References

1. 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)
2. Fraser, G., Arcuri, A.: Automated Test Generation for Java Generics, pp. 185–198. Springer International Publishing, Cham (2014), http://dx.doi.org/10.1007/978-3-319-03602-1_12
3. 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, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2483760.2483774
4. Gay, G.: The fitness function for the job: Search-based generation of test suites that detect real faults. In: Proceedings of the International Conference on Software Testing. ICST 2017, IEEE (2017)
5. Gay, G.: Generating effective test suites by combining coverage criteria. In: Proceedings of the Symposium on Search-Based Software Engineering. SSBSE 2017, Springer Verlag (2017)
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/