Detecting Real Faults in the Gson Library Through Search-Based Unit Test Generation

Gregory Gay

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

Abstract. An important benchmark for test generation tools is their ability to detect *real faults*. We have identified 16 real faults in Gson—a Java library for manipulating JSON data—and added them to the Defects4J fault database. Tests generated using the EvoSuite framework are able to detect seven faults. Analysis of the remaining faults offers lessons in how to improve generation. We offer these faults to the community to assist future research.

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

1 Introduction

Automation of unit test creation can assist in controlling the cost of testing. One promising form of automated generation is *search-based* generation. Given a measurable testing goal, powerful optimization algorithms can select test inputs meeting that goal [6].

To impact practice, automated generation techniques must be effective at detecting the complex faults that manifest in real-world software projects [2]. "Detecting faults" is not a goal that can be measured. Instead, search-based generation relies on *fitness functions*—based on coverage of code structures, synthetic faults, and other targeted aspects—that are believed to increase the probability of fault detection. It is important to identify which functions produce tests that detect real faults.

By offering case examples, fault databases—such as Defects4J [5]—allow us to explore questions like those above. The Google Gson library [1] offers an excellent opportunity for expanding Defects4J. Gson is an open-source library for serializing and deserializing JSON input that is an essential tool of Java and Android development and is one of the most popular Java libraries [4].

Gson serves as an interesting benchmark because much of its functionality is related to the parsing of JSON input and creation and manipulation of complex objects. Manipulation of complex input and non-primitive objects is challenging for automated generation. Gson is also a mature project. Its faults will generally be more complex than the simple syntactic mistakes modeled by mutation testing [2]. Rather, detecting faults will require specific, contextual, combinations of input and method calls. By studying these faults, we may be able to learn lessons that will improve test generation tools.

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

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

We have identified 16 real faults in the Gson project, and added them to Defects4J. We generated test suites using the EvoSuite framework [6]—focusing on eight fitness functions and three combinations of functions—and assessed the ability of these suites to detect the faults. Ultimately, EvoSuite is able to detect seven faults. Some of the issues preventing detection include a need for stronger coverage criteria, the need for specific data types or values as input, and faults that only emerge through class interactions—requiring system testing to detect. We offer these faults and this analysis to the community to assist future research and improve test generation efforts.

2 Study

In this study, we have extracted faults from the Gson project, gathering faulty and fixed versions of the code and developer-written test cases that expose each fault. For each fault, we have generated tests for each affected class-under-test (CUT) with the EvoSuite framework [6]—using eight fitness functions and three combinations of functions—and assessed the efficacy of generated suites. We wish to answer the following research questions: (1) *can suites optimizing any function detect the extracted faults?*, (2) *which fitness function or combination of functions generates suites with the highest overall likelihood of fault detection?* and (3), *what factors prevented fault detection?*

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

1. **Extracted Faults:** We have identified 16 real faults in the Gson project, and added them to the Defects4J fault database (See Section 2.1).
2. **Generated Test Cases:** For each fault, we generated 10 suites per fitness function and combination of functions, using the fixed version of each CUT. We repeat this step 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, function, and fault, we measure the likelihood of fault detection. For each undetected fault, we examined gathered data and the source code to identify possible detection-preventing factors.

2.1 Fault Extraction

Defects4J is an extensible database of real faults extracted from Java projects [5]. Currently, the core dataset consists of 395 faults from six projects, with an experimental release containing 597 faults from fifteen projects[2]. 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 Gson to Defects4J. This process consisted of developing build scripts that would compile and execute all tested project versions, extracting candidate faults using Gson'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 until it only contains fault-related code. Following this process, we extracted 16 faults from a pool of 132 candidate faults that met all requirements.

[2] Core: `http://defects4j.org`; Experimental: `http://github.com/Greg4cr/defects4j`

Each fault is required to meet three properties. First, the fault must be related to the source code. The "fixed" version must be explicitly labeled as a fix to an issue[3], 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 refactoring.

The faults used in this study can be accessed through the experimental version of Defects4J[4]. Additional data about each fault can be found at `http://greggay.com/data/gson/GsonFaults.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 [6]. In this study, we used EvoSuite version 1.0.5 with eight fitness functions: Branch Coverage, Direct Branch Coverage, Line Coverage, Exception Coverage, Method Coverage, Method (Top-Level, No Exception) Coverage, Output Coverage, and Weak Mutation Coverage. Rojas et al. provide a primer on each [6]. We have also used three combinations of fitness functions: all eight of the above, Branch/Exception Coverage, and Branch/Exception/Method Coverage. The first is EvoSuite's default configuration, and the other two were found to be generally effective at detecting faults [3]. When a combination is used to generate tests, the individual fitness functions are calculated and added to obtain a single fitness score.

Tests are generated from the fixed version of the system and applied to the faulty version in order to eliminate the oracle problem. Given the potential difficulty in achieving coverage over Gson classes, two search budgets were used—two and ten minutes, a typical and an extended budget [2]. As results may vary, we performed 10 trials for each fault, fitness function, and budget. Generation tools may generate flaky (unstable) tests [2]. We automatically remove non-compiling test cases. Then, each test is executed on the fixed CUT five times. If results are inconsistent, the test case is removed. On average, less than 1% of tests are removed from each suite.

3 Results and Discussion

In Table 1, we list—for each search budget and fitness function—the likelihood of fault detection (the proportion of suites that detected the fault). Seven of the sixteen faults were detected. EvoSuite failed to generate test suites for Fault 12. At the two minute budget, the most effective fitness function is a combination of Branch/Exception/Method Coverage, with an average likelihood of fault detection of 40.67%—closely followed by the Branch/Exception combination and Branch Coverage alone. At the ten minute

[3] The commit message for the "fixed" version must reference either a reported issue or a pull request that describes and fixes a fault (that is, it must not add new functionality).

[4] These faults will be migrated into the core dataset following additional testing and study.

Fault	Budget	BC	DBC	EC	LC	MC	M(TLNE)	OC	WMC	C-All	C-BE	C-BEM
2	2m	100.00%	100.00%	70.00%	70.00%	-	-	-	100.00%	100.00%	100.00%	100.00%
	10m	100.00%	100.00%	40.00%	90.00%	-	-	-	100.00%	100.00%	100.00%	100.00%
3	2m	70.00%	60.00%	-	80.00%	-	-	-	60.00%	30.00%	90.00%	70.00%
	10m	100.00%	80.00%	-	100.00%	-	-	-	100.00%	70.00%	90.00%	100.00%
6	2m	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	10m	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
8	2m	20.00	30.00%	-	50.00%	-	-	-	10.00%	10.00%	10.00%	40.00%
	10m	90.00%	60.00%	-	100.00%	-	-	-	80.00%	80.00%	100.00%	90.00%
10	2m	100.00%	100.00%	30.00%	100.00%	20.00%	10.00%	40.00%	50.00%	100.00%	100.00%	100.00%
	10m	100.00%	100.00%	10.00%	100.00%	30.00%	10.00%	40.00%	70.00%	100.00%	100.00%	100.00%
13	2m	100.00%	100.00%	20.00%	30.00%	100.00%	100.00%	-	90.00%	100.00%	100.00%	100.00%
	10m	100.00%	100.00%	10.00%	-	100.00%	100.00%	-	100.00%	100.00%	100.00%	100.00%
16	2m	100.00%	100.00%	60.00%	100.00%	40.00%	10.00%	-	40.00%	100.00%	100.00%	100.00%
	10m	100.00%	100.00%	30.00%	100.00%	-	30.00%	30.00%	10.00%	100.00%	100.00%	100.00%
Average	**2m**	**39.33%**	**39.33%**	**18.67%**	**35.33%**	**17.33%**	**14.67%**	**9.33%**	**30.00%**	**36.00%**	**40.00%**	**40.67%**
	10m	**46.00%**	**42.67%**	**12.67%**	**39.33%**	**15.33%**	**16.00%**	**11.33%**	**37.33%**	**43.33%**	**46.00%**	**46.00%**

Table 1. Likelihood of fault detection for each fitness function (two-minute/ten-minute budget). (D)BC = (Direct) Branch Coverage, EC = Exception Coverage, LC = Line Coverage, M(TLNE)C = Method (Top-Level, No Exception) Coverage, OC = Output Coverage, WMC = Weak Mutation Coverage, C-All = combination of all criteria, C-BE = combination of BC/EC, C-BEM = combination of BC/EC/MC. Undetected faults (1, 4, 5, 7, 9, 11, 14, and 15) are omitted.

budget, these three configurations perform equally, with an average detection likelihood of 46.00%. Unlike in other Defects4J systems [3], Exception Coverage does not add significant value. Specialized metrics, like Output Coverage, also do not seem to have much situational applicability.

Fault 6 was detected the most reliably, regardless of search budget or fitness function. This fault updates Gson to be compliant with the 2014 JSON RFC 7159 standard, and adds a leniency check to enable backwards compatibility[5]. Compliance checks are spread throughout the code, resulting in fault detection if even a small amount of coverage is attained. Fault 13[6], dealing with an index out of bounds error, is a classic example of what automated generation excels at. The fix adds boundary checks, which are efficiently covered by Branch Coverage—ensuring differing output between versions.

Fault 8[7] was detected the least reliably. This fault causes issues with deserializing map structures when a key is an unquoted long or integer. The generated tests arguably expose the fault—they produce differing behavior and result in the same exception as the human-written test cases. Yet, these failing tests also point out an issue with test suite generation. The test cases fail in the same manner as the human-written cases, but not for the same reason. The failing tests pass strings to methods meant to handle long or integer values and expect a `NumberFormatException`—which is not thrown by the faulty version. The exception thrown instead—a complaint about a string—makes sense, given the input used. Rather than helping a human tester identify actual issues, these test cases only show that the two versions of the code behave differently.

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

Stronger Adequacy Criteria are Required: Fault 14[8] causes all instances of -0 ("negative 0") to be converted to 0. Catching this fault would require the generation

[5] `https://github.com/google/gson/commit/af68d70cd55826fa7149effd7397d64667ca264c`
[6] `https://github.com/google/gson/commit/9e6f2bab20257b6823a5b753739f047d79e9dcbd`
[7] `https://github.com/google/gson/commit/2b08c88c09d14e0b1a68a982bab0bb18206df76b`
[8] `https://github.com/google/gson/commit/9a2421997e83ec803c88ea370a2d102052699d3b`

framework to produce -0 as input—an unlikely choice. However, the test generator could be guided towards this input. The fixed version of the class has a complex `if-condition` that includes this corner case. Branch Coverage simply requires the full predicate to evaluate to `true` and `false`, so coverage can be achieved without -0 input. However, a stronger criterion such as Modified Condition/Decision Coverage [1] would require -0 to attain full coverage.

Specific Data Types are Required as Input: Fault 9[9] causes an error when Gson attempts to initialize an interface or abstract class. This fault can only be detected if a test case attempts to instantiate either type of object. Most generation frameworks will not attempt this, and the feedback provided by criteria like Branch Coverage is not sufficient to suggest such an action.

Fault Emerges Through Class Interactions and System Testing: Fault 1[10] causes Gson to fail to serialize or deserialize a class when its super class has a type parameter. Like Fault 9, this is a case where tests would need to attempt to generate a highly specific object. In addition, the developer-written test exposing this fault is a *system-level* test, not a unit test—working through Gson's top-level serialization and deserialization functions. It is possible that unit testing could expose the fault, but this is code that—like Fault 9 above—that would be hard to cover. System testing is more likely to expose the fault, but external context would still be needed to guide data type selection.

By default, Gson converts application classes to JSON using its built-in type adapters. If Gson's default JSON conversion isn't appropriate for a type, users can specify their own adapter using an annotation. Fault 5[11] deals with ensuring that custom type adapters safely handle null objects. However, performing unit testing of the modified class will not expose the fault. Rather, one needs to define a type adapter for a null class, then use Gson's top-level API. Fault 7[12] modifies the same class, fixing a null pointer exception when a null object is returned instead of a proper `TypeAdapter`. A similar scenario exists for Fault 11[13], where custom adapters are ignored for primitive fields. In all three cases—as long as the right input is chosen—system testing will expose this fault while unit testing may not be able to replicate the same example. However, system testing alone will still not be sufficient. Each of these scenarios requires external context to create the specific conditions called for to detect the fault.

Gson is a complex system designed to be accessed through a simple API. Human-written tests tend to use that API, even when testing specific classes. Unit test generation may not be suited to detecting some of the faults that emerge from this type of system, and even if it can, the generated test suites may not be easily understood by human developers. Many of the most mature test generation approaches are based on unit testing, and more work clearly needs to be conducted in the system testing realm.

Regardless of the form of testing, better means are needed of extracting *context* from the system and its associated artifacts. Automation requires information to guide test creation. Often, this is some form of code coverage. However, code coverage doesn't

[9] https://github.com/google/gson/commit/0f66f4fac441f7d7d7bc4afc907454f3fe4c0faa
[10] https://github.com/google/gson/commit/c6a4f55d1a9b191dbbd958c366091e567191ccab
[11] https://github.com/google/gson/commit/57b08bbc31421653481762507cc88ee3eb373563
[12] https://github.com/google/gson/commit/dea305503ad8827121e8212248c271f1f2f90048
[13] https://github.com/google/gson/commit/bb451eac43313ae08b30ac0916718ca00c39656d

provide the same type of information developers use during test creation, and many of the studied faults were detected by *almost any* coverage criterion or *no* criterion. Rather, information from the project is needed to guide input generation. Methods of gleaning that information, either through seeding from existing test cases or data mining of project elements, may assist in improving the efficacy of test generation. Approaches to mining of requirements information or bug reports, for instance, might suggest using particular data types or values as input.

4 Conclusion

Testing costs can be reduced through automated unit test generation. An important benchmark for such tools is their ability to detect *real faults*. We have identified 16 real faults in Gson, and added them to Defects4J. We generated test suites and found that EvoSuite is able to detect seven faults. Some of the issues preventing fault detection include a lack of fitness functions for stronger coverage criteria, the need for specific data types or values as input, and faults that only emerge through class interactions—requiring system testing rather than unit testing to detect. We offer these faults to the community to assist future research.

References

1. Chilenski, J.: An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Tech. Rep. DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C. (April 2001)
2. 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)
3. 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)
4. Idan, H.: The top 100 java libraries in 2017 - based on 259,885 source files (2017), https://blog.takipi.com/the-top-100-java-libraries-in-2017-based-on-259885-source-files/
5. 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
6. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining multiple coverage criteria in search-based unit test generation. In: Barros, M., Labiche, Y. (eds.) Search-Based Software Engineering, Lecture Notes in Computer Science, vol. 9275, pp. 93–108. Springer International Publishing (2015), http://dx.doi.org/10.1007/978-3-319-22183-0_7