Classification of Randomly Generated Test Cases

Cyrille Artho
AIST, Osaka, Japan
c.artho@aist.go.jp

Lei Ma
Chiba University, Chiba, Japan
malei.2005@gmail.com

***Abstract*—Random test case generation produces relatively diverse test sequences, but the validity of the test verdict is always uncertain. Because tests are generated without taking the specification and documentation into account, many tests are invalid. To understand the prevalent types of successful and invalid tests, we present a classification of 56 issues that were derived from 208 failed, randomly generated test cases. While the existing workflow successfully eliminated more than half of the tests as irrelevant, half of the remaining failed tests are false positives. We show that the new `@NonNull` annotation of Java 8 has the potential to eliminate most of the false positives, highlighting the importance of machine-readable documentation.**

I. Introduction

Random test case generation is useful because it can create diverse test cases without human intervention. In object-oriented software, a test case typically consists of a sequence of method calls. Each step calls a method m of object o with signature $o.m(T_{in_1} v_1, T_{in_2} v_2, \ldots, T_{in_n} v_n)$, where each parameter in the call is a value v_i of type T_{in_i}.

The problem of generating a random test is to set up o such that it can accept a method call with the right parameters, and constructing objects of type T_{in_i} as parameters to m. In feedback-directed random testing, input objects are derived by constructing sequences of calls that build higher-level values from primitive data types [11].

Successful calls to m return a value r that can be used as a parameter in subsequent tests. If an exception occurs, this is considered as a failure of m. Without domain-specific knowledge, automatically generated test cases typically end up generating some test cases with invalid parameters or method call sequences. For example, numbers are typically specified by their type (integer or floating point), which include a much larger range of values than what is typically permissible in a given context. Lacking this information, automated tools cannot distinguish which exceptions are legitimate, and which ones indicate an actual failure. This can result in *false positives* (tools reporting test failures that are not a problem with the system under test), as method calls with illegal parameter values are expected (indeed, required!) to fail.

False negatives (defects that are not detected by tests) are also a problem. Properties that are random testing covers, are limited to uncaught exceptions and a few generic properties, e. g., equality must be reflexive, symmetric, and transitive. However, false negatives are expected with fully automated approaches, as the oracle problem [3] cannot be solved by program analysis alone. This work focuses on false positives.

We report on a case study involving ten large open source libraries, for which we automatically generated over 208 failed tests. They relate to 56 distinct issues in the given software, out of which 23 test cases pertained to confirmed defects [9]. We analyze all issues and categorize them based on the failure type, whether they were true or false positives, and how they were fixed. Our contribution shows that NullPointerExceptions are a frequent cause of invalid bug reports. Machine-readable specifications allow a tool to distinguish between expected and unexpected failures; they could become essential for reducing the false-positive rate of automatic test generation tools.

This paper is organized as follows: Section II describes the case study we conducted on ten open-source real-world software libraries by using automated test generator GRT [9]. Section III discusses previous work on bug detection and classification by using automated test generators. Section IV concludes and discusses future work.

II. Open Source Case Study

We applied GRT, a random test case generator, to the most recent version of real-world popular software projects, with the goal of uncovering new, previously unknown defects [9].

A. GRT

GRT [9] is a fully automated test generator that uses knowledge extracted from software under test (SUT) to guide each step of test generation to achieve high code coverage and bug detection ability [10]. GRT uses random test generator Randoop [11] as its basic framework, which generates tests iteratively. In each iteration, GRT randomly selects a method and reuses previously generated objects as input to create new test sequences. If a test sequence executes successfully, it is considered as further test input for future tests. Each of the test generation steps is guided by the six program analysis components of GRT, which analyze the general properties of constant values, method side effects, type dependencies, missing input types, execution cost and code coverage from the SUT. All analysis steps are fully automated and rely solely on the SUT, without using any human input or specification.

B. Evaluation of generated tests

We chose ten popular open-source real-world projects (see Table I) as the target for GRT. The projects were chosen because they seemed under active development (the last update being less than a year before our analysis), and the number of failed test cases reported is not prohibitively high (fewer than 100 failed tests for GRT). We use GRT in the mode where it records failed tests for further debugging [8].

The initial number of failed test cases over all ten projects is still very high, with a total number of over 200 tests. To limit the manual evaluation effort, we first filter out tests that confirm a problem that is either known or not going to be fixed in the code [8]. These issues include:

- Deprecated methods. Methods that contain or expose a design flaw are often marked as deprecated. For example, a deprecated method may allow incomplete objects to be used before all invariants are established. Such methods are usually removed from the library in the future. In this work, we assume that failed tests involving such code confirm previously known issues. Ignoring these issues allowed us to focus on unknown flaws.
- Stack overflows. Container classes, such as found in the Apache collection library or Guava, allow recursive nesting of data. For example, a list l_1 can be inserted into another list l_2 followed by an insertion of l_2 into l_1. Operations such as list iteration or toString will then never terminate on the resulting objects. It is possible to make iteration robust against infinite recursion, but a fix entails keeping track of previously visited object instances during iteration. This requires an amount of memory that is linear in the size of the collection; the cure would be worse than the disease in most cases.
- Internal packages. Such packages occur in project `javax.mail` and usually start with `com.sun`. They are specific to the given reference implementation and not meant to be used by others. Possibly unsafe API uses found by test cases are therefore irrelevant.

These filters remove over half of the failed test cases, leaving 90 failed tests. We further manually simplify the remaining tests. Often it becomes apparent that a given test is a variation of an earlier one. This can be confirmed by comparing their stack traces and the sequences of method calls. GRT uses Randoop's test minimization, which does not always achieve a theoretically optimal result. Manual test minimization can simplify tests further, up to the point where multiple minimized tests become identical and thus are shown to confirm the same fault. Sometimes, two tests are slightly different but trigger the same fault at the end; we consider such sequences to be equivalent.[1] Using this definition, we classify equivalent failed tests into distinct *issues*. When doing so, we count the same defect in two classes as two issues, as well as two different types of problems in the same class. This results 56 issues found by GRT. Each issue is then investigated more closely, and based on the API documentation and our own experience, we discount some as false positives.

Remaining open issues were reported to the issue tracker of each project. To limit the number of reports, we created only one issue ticket for similar defects across a given class or multiple classes. Based on feedback from developers, we classify the originally counted issues as false or true reports. Unconfirmed cases are counted as unknown. Finally, 23 issues were confirmed by developers, and most of our reported defects have already been fixed [9].

[1]For example, a class may have multiple constructors with optional extra parameters. If a fault does not depend on those extra parameters, any constructor can be used to trigger it.

Table I. SUMMARY OF BUG CLASSIFICATION [9]

Software	Failed tests	Filtered tests	Identified issues				Issue numbers
			issues	false	unkn.	true	
A. CLI	1	1	0	0	0	0	–
A. Codec	2	1	1	0	0	1	184
A. Collection	56	25	15	13	0	2	512–516
A. Compress	25	0	4	0	0	4	273–276
A. Math	76	45	13	6	0	7	1115–1118, 1224
A. Primitive	4	0	2	1	1	0	17
Guava	13	5	8	6	0	2	1722–1724
JavaMail	20	12	6	0	0	6	6365–6368
Mango	3	1	1	0	0	1	1
TinySQL	8	0	6	1	5	0	14–18
Total	208	90	56	27	6	23	

C. Defect classification

The defect detection results from previous work (see Table I [9]) show that GRT classifies about half of the issues as false positives. In our analysis we first consider the causes for the reported test failures (see Figure 1):

1) NullPointerException: The method under test throws such an exception because one of its arguments or one of the fields used is `null`. Such test failures may be true positives (in cases where a `null` value should be permitted, or where there is no documentation stating the contrary), or false positives (in cases where `null` values are prohibited in the documentation).
2) equals/hashCode: We consider two properties here:
    a) Equals must be reflexive, symmetric, and transitive. The first two properties are tested by randomly inserted checks.
    b) Two objects that are equal must have the same hashCode.
3) Incomplete initialization: These issues appeared in iterators of the Apache collections library. Many specialized iterators perform additional functions such as transformation or filtering. These iterators "will not function until the… [initialization] methods are invoked" [4].
4) Other: A ClassCastException and a missing dependency resulted in one failed test each.

After checking against the documentation, we filtered out 20 issues as false positives and reported the remaining 36 issues (see Table II). 23 of these issues were confirmed as actual defects (true positives), six remain unconfirmed as to date, and seven additional false positives were reported. NullPointerExceptions are by far the most prevalent type of defect reported, causing over 70 % of all test failures. Problematic implementations of equals or hashCode are the second most important category (16 %), followed by incomplete initializations (9 %) and two tests that failed for other reasons (see Figure 1).

It is interesting to investigate what types of defects tend to be true or false positives, and why. Figure 2 shows that slightly less than half of all reported NullPointerExceptions are true positives while almost all issues relating to equals or hashCode are actual defects.

Among the false positives, three out of four cases of spurious issues were covered by the documentation of the given library. Five issues related to NullPointerExceptions

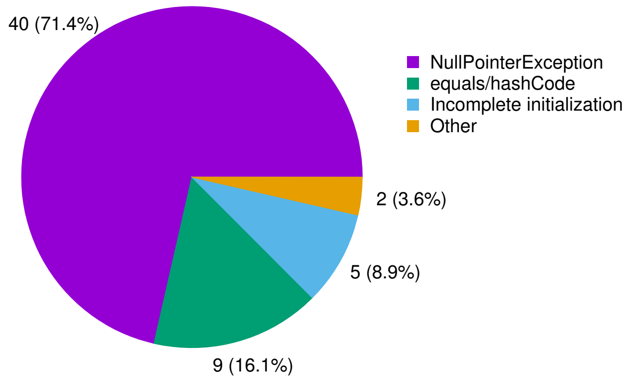

Figure 1. Causes for test failures.

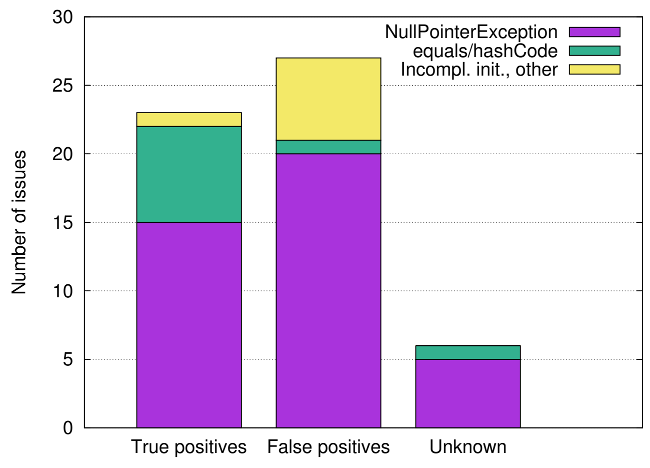

Figure 2. True and false positives for different types of issues.

were caused because the wrong generic type was used in a method call, and only two issues due to other reasons (wrong usage or lack of adherence to the documentation of the underlying Java library).

The true positives were eventually fixed; it is interesting to note that only about two thirds of all fixes were made in the implementation code while about one third of the issues were addressed by updating the documentation but not the program code (see Table III).

Furthermore, we can see that documentation is involved in confirming 30 out of 50 cases. Human-readable documentation often describes restrictions on API usage that were not obeyed by the generated test cases (see Table IV).[2] When looking at code (last row in Table IV), incorrectly generated generic types are responsible for five false positives. Such usage causes compiler warnings when compiling the failed test cases; these warnings were ignored by GRT. Conversely, 15 logical defects were fixed by code changes.

D. Discussion

In our case study, NullPointerExceptions constitute the largest class of defects found by random testing. Slightly less than half of these are true positives, which were fixed

[2]We also count "same as JDK" and "wrong usage" from Table II under "documentation" here.

Table II. DETAILED CLASSIFICATION OF REPORTED AND UNREPORTED ISSUES

Software	Issue	Verdict	Fix/reason	Issue #
A. Codec	NullPointerException	true	code	184
A. Collection	hashCode vs. equals	true	documentation	512
	NullPointerException	false	documentation	513
	NullPointerException	false	same as JDK	514
	ClassCastException	false	wrong usage	515
	NullPointerException	true	documentation	516
	NullPointerException	false	documentation	—
	Incomplete initialization	false	documentation	—
	Incomplete initialization	false	documentation	—
	Incomplete initialization	false	documentation	—
	Incomplete initialization	false	documentation	—
	Incomplete initialization	false	documentation	—
	NullPointerException	false	documentation	—
	NullPointerException	false	documentation	—
	NullPointerException	false	documentation	—
	NullPointerException	false	documentation	—
A. Compress	NullPointerException	true	code	273
	NullPointerException	true	code	274
	Dependency	true	documentation	275
	NullPointerException	true	code	276
A. Math	NullPointerException	true	code	1115
	NullPointerException	true	documentation	1116
	NullPointerException	true	documentation	1116
	NullPointerException	false	documentation	1116
	NullPointerException	true	code	1117
	NullPointerException	false	documentation	1117
	hashCode vs. equals	true	code	1118
	hashCode vs. equals	true	code	1118
	NullPointerException	true	documentation	1224
	NullPointerException	false	documentation	—
	NullPointerException	false	documentation	—
	NullPointerException	false	documentation	—
	NullPointerException	false	documentation	—
A. Primitive	hashCode vs. equals	no response		17
	equals is not reflexive	false	documentation	—
Guava	hashCode not consistent	false	documentation	1722
	NullPointerException	true	documentation	1723
	NullPointerException	true	documentation	1723
	NullPointerException	false	wrong generic type	1724
	NullPointerException	false	wrong generic type	—
	NullPointerException	false	wrong generic type	—
	NullPointerException	false	wrong generic type	—
	NullPointerException	false	wrong generic type	—
JavaMail	equals not reflexive	true	code	6365
	NullPointerException	true	code	6365
	NullPointerException	true	code	6366
	hashCode vs. equals	true	code	6367
	hashCode vs. equals	true	code	6367
	NullPointerException	true	code	6368
Mango	NullPointerException	true	code	1
TinySQL	NullPointerException	no response		14
	NullPointerException	no response		15
	NullPointerException	no response		16
	NullPointerException	no response		17
	NullPointerException	no response		18
	NullPointerException	false	documentation	—

Table III. TYPES OF FIXES APPLIED PER DEFECT TYPE.

Bug type	Fixed in code	Fixed in documentation
NullPointerException	10	5
`equals/hashCode`	5	2
Other	0	1

Table IV. TRUE OR FALSE POSITIVES SPLIT BY DOCUMENTATION OR CODE.

Artifact	Related to true positive	Related to false positive
Documentation	8	22
Code	15	5

either in the code or in the documentation. Issues related to equals/hashCode are rarer but tend to be true positives. Other issues relate to correct API usage that is documented in a human-readable format but not readily accessible to program analysis tools.

III. Related Work

Although there exists a large body of work on developing automated test generation techniques and tools [2], work analyzing real faults in software is limited.

Xiao et al. [15] inspect the issues that limit automated tools in obtaining high structural coverage on practical software, and found the main problems to be 1) dependencies on external methods, 2) finding a suitable method sequence to derive desired input object states. However, their study is not based on real faults. Ramler et al. investigated the bug-finding ability of humans vs. automated tools on a database of examples with seeded faults and found that the two approaches tend to complement each other [12], [13].

To facilitate research on real faults, Just et al. developed Defects4J [7], a collection that contains 357 known real faults that can be used as a platform for a controlled bug detection study. In their follow-up work, they analyze three automated test generators using Defects4j: Randoop [11], Evosuite [6], and Agitar [1], to study whether these automated tools are helpful to uncover real faults and the challenges of existing tools that limit that bug detection ability [14].

Unlike previous work, this paper focuses on unknown defects. We generate failed tests by GRT [9], and perform in-depth analysis and classification on these tests to show how they enable unknown real faults detection. Similar to work that led to the creation of Defects4J [7], a bug database of previously found faults in Java programs, our work required much manual effort. Using the results of our work to enhance collections like Defects4J is promising future work.

IV. Conclusion and Future Work

Random testing can generate diverse test cases. However, due to the limited test oracle, it is restricted to finding crashes (such as NullPointerExceptions) and relatively few generic problems. Due to the inability to distinguish between expected and unexpected exceptions, false positives remain a problem with random testing. In our case study, many cases of NullPointerExceptions were covered by documentation, which is not designed to be machine-readable. Annotations such as `@NonNull` are possible in the grammar of Java 8 but currently not standardized in the main library [5]. Our results show that machine-readable annotations could reduce the false-positive rate significantly; in particular, 20 out of 27 false positives relate to NullPointerExceptions that could be documented via annotations. We think that the additional effort of using annotations instead of unstructured documentation is small.

We think that agreed-on type annotations should be standardized such that they can be included in upcoming Java releases, covering issues like null pointers and parameter ranges. This would benefit a wide range of program analysis tools, from static checkers to test case generators. Furthermore, it may be interesting to consider a light-weight documentation analysis to reduce the number of false positives in code that does not use annotations.

Acknowledgements

This work was supported by the SEUT project from the University of Tokyo and *kaken-hi* grants 23240003 and 26280019 from the Japanese Society for the Promotion of Science (JSPS). We also thank Michael Ernst for his comments.

References

[1] AgitarOne. http://www.agitar.com/.

[2] S. Anand, E. Burke, T. Chen, J. Clark, M. Cohen, W. Grieskamp, M. Harman, M. Harrold, and P. Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, August 2013.

[3] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[4] M. Benson, J. Carman, S. Colebourne, R. Donkin, M. Delagrange, G. Gregory, M. Hawthorne, D. Laha, G. Magnusson, L. Maisonobe, C. McClanahan, T. Neidhart, A. Nistor, P. Steitz, A. Thomas, R. Waldhoff, and H. Yandell. Apache Commons Collections API, 2015. https://commons.apache.org/proper/commons-collections/apidocs/.

[5] E. Costlow. Java 8's new type annotations, 2015. https://blogs.oracle.com/java-platform-group/entry/java_8_s_new_type.

[6] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE'11, pages 416–419, Szeged, Hungary, 2011.

[7] R. Just, D. Jalali, and M. Ernst. 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*, ISSTA'14, pages 437–440, San Jose, CA, USA, 2014.

[8] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. GRT: An automated test generator using orchestrated program analysis. In *IEEE/ACM Int. Conference on Automated Software Engineering*, ASE'15, pages 842–847, Lincoln, Nebraska, USA, 2015.

[9] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R Ramler. GRT: Program-analysis-guided random testing. In *IEEE/ACM Int. Conference on Automated Software Engineering*, ASE'15, pages 212–223, Lincoln, Nebraska, USA, 2015.

[10] L. Ma, C. Artho, C. Zhang, H. Sato, M. Hagiya, Y. Tanabe, and M. Yamamoto. GRT at the SBST 2015 tool competition. In *Proceedings of the Eighth International Workshop on Search-Based Software Testing*, SBST'15, pages 48–51, Florence, Italy, 2015. IEEE Press.

[11] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE'07, pages 75–84, Washington, DC, USA, 2007.

[12] R. Ramler, D. Winkler, and M. Schmidt. Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code? In *36th Conf. on Software Engineering and Advanced Applications*, pages 286–293. IEEE Computer Society, 2012.

[13] R. Ramler, K. Wolfmaier, and T. Kopetzky. A replicated study on random test case generation and manual unit testing: How many bugs do professional developers find? In *Proc. 37th Annual Int. Comp. Software & Applications Conf.*, COMPSAC'13, pages 484–491, Washington, DC, USA, 2013. IEEE Computer Society.

[14] S. Shamshiri, R. Just, J. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 201–211, Lincoln, NE, USA, November 11–13, 2015.

[15] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 611–620, New York, NY, USA, 2011. ACM.