Oracle Problem in Software Testing

Gunel Jahangirova
FBK, Trento, Italy
UCL, London, UK
jahangirova@fbk.eu

ABSTRACT

The oracle problem remains one of the key challenges in software testing, for which little automated support has been developed so far. In my thesis work we introduce a technique for assessing and improving test oracles by reducing the incidence of both false positives and false negatives. The experimental results on five real-world subjects show that the fault detection rate of the oracles after improvement increases, on average, by 48.6% (86% over the implicit oracle). Three actual, exposed faults in the studied systems were subsequently confirmed and fixed by the developers. However, our technique contains a human in the loop, which was represented only by the author during the initial experiments. Our next goal is to conduct further experiments where the human in the loop will be represented by real developers. Our second future goal is to address the oracle placement problem. When testing software, developers can place oracles externally or internally to a method. Given a faulty execution state, i.e., one that differs from the expected one, an oracle might be unable to expose the fault if it is placed at a program point with no access to the incorrect program state or where the program state is no longer corrupted. In such a case, the oracle is subject to failed error propagation. Internal oracles are in principle less subject to failed error propagation than external oracles. However, they are also more difficult to define manually. Hence, a key research question is whether a more intrusive oracle placement is justified by its higher fault detection capability.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**;

KEYWORDS

Oracle Problem; test oracle; test case generation; mutation testing; oracle placement; failed error propagation;

ACM Reference format:
Gunel Jahangirova. 2017. Oracle Problem in Software Testing. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17-DOC),* 5 pages.
https://doi.org/10.1145/3092703.3098235

ISSTA'17-DOC, July 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3098235

1 RESEARCH PROBLEM AND MOTIVATION

Recent advances in test input generation have left the Oracle Problem as a key remaining bottleneck in improving the overall effectiveness and efficiency of the software testing process. The latter depends both on the quality of the test cases and that of the oracle. There are many techniques for assessing and improving the adequacy of test cases, e.g. their code coverage. In comparison, there is relatively little work to help the software tester with the *Oracle Problem*, i.e., the problem of defining accurate oracles, capable of detecting all but only faulty behaviours exercised during testing. Without a (good) oracle to determine whether or not the output they induce is correct, test inputs that satisfy the strictest adequacy criteria remain useless and testing is ineffective.

Good oracles combine generality and accuracy. They should be applicable to the widest possible range of test cases, in particular so that they can be used within automatically generated test suites. And crucially, they should be accurate in revealing all the faulty behaviours (*completeness*, no false negatives) and only the faulty ones (*soundness*, no false positives). Indeed, the precision and efficiency of oracles greatly affects testing cost/effectiveness: we don't want that test failures pass undetected, but on the other side we don't want either to be notified of many false-positives, which waste important resources.

Software faults are believed to be complicated and difficult to detect. One of the widely-attributed sources of such difficulty is driven by the possibility of *failed error propagation* (FEP): a fault may corrupt the program's internal state, yet this corruption fails to propagate to any point at which it is observed. Such non-propagating faults play the role of 'nasty unexploded mines'; lurking undetected in software systems, waiting for that slight change in execution environment that allows the corrupted error state to propagate, causing unexpected system failure. Thus, another critical decision about oracles is their placement in the code. Since oracle definition is a difficult and effort intensive task for developers, the number of oracles that can be placed in a program is limited. Hence, developers are faced with the problem of choosing the best oracle placement points in the program, so as to maximise their overall fault detection ability.

2 COMPLETED WORK

We introduce an approach [7] that is based on search based test case generation to identify false positives and mutation testing to identify false negatives in oracles represented in the form of program assertion. Our tool generates counterexamples as test cases that demonstrate incompleteness and unsoundness, which the tester uses to iteratively improve the assertion oracle. The process continues until the tool is unable to generate new counterexamples and finishes with an improved (more complete and sound) oracle. Our

approach necessarily places the human tester in the loop, because modifications made to the oracle to solve reported *oracle deficiencies* (i.e. false negatives or false positives) depend on the intended program behaviour (vs. the implemented behaviour), which we assume is known to developers through informal knowledge, requirement documents and other sources of documentation.

Given a program assertion, we detect its false positives by generating execution scenarios where the assertion fails when it should hold because the behaviour of the program is deemed correct. In such a case, failure of the assertion points to a bug in the assertion, not in the program. To be able to generate the necessary execution scenarios (test cases), we perform a testability transformation that transforms the criterion for false positive detection into the standard branch coverage criterion, for which automated test case generators are available. Let us consider a program under test P containing n assertions $a_1 \dots a_n : a_i = assert(c_i), i \in [1 \dots n]$, where c_i is the boolean expression used in the assertion a_i. For each assertion $a_i, i \in [1 \dots n]$ in P the proposed testability transformation takes c_i, negates it and replaces the assertion a_i with a new branch containing the negated condition: if $(!(c_i))$ {}.

An assertion has no false negatives if it exposes all faults. Therefore, if we deliberately insert a fault into the source code of program P, a sound oracle ought to always report the presence of this fault. Hence, to find evidence of false negatives we use mutation testing to insert a (known) fault in program P that corrupts the program state so that the corrupted state reaches the given assertion and the assertion statement does not fail.

Our prototype tool for false positive and false negative detection is implemented as an extension of the EvoSuite [5]. We have conducted a set of experiments to answer the following research questions:

RQ1 (Implicit oracle): Can new program assertions be introduced and iteratively improved in classes without assertions thanks to the computation of oracle deficiencies?

RQ2 (Inferred properties): Can automatically inferred program properties be improved thanks to the computation of oracle deficiencies?

RQ3 (Manual oracle): Can the proposed approach reveal oracle deficiencies in classes that include human written program assertions?

RQ4 (Fault detection): Can the improved oracle reveal more faults than the initial (implicit, automatically inferred, manual) oracle and the test case oracle?

The goal is to assess the applicability of the proposed approach in different contexts, ranging from one where no oracle is present, hence fault detection relies entirely on the implicit oracle (program crashing or raising exceptions), to a context where the oracle is obtained automatically, by mining program specifications from the observed program behaviour, or is produced manually. The effectiveness of the improved oracle is assessed in terms of increased fault detection with respect to the initial and test case oracle. To answer RQ1-2-3 we report the number of assertions added in each iteration to solve the false positives and negatives reported by our tool. To answer RQ4 we analyse the mutation score reported for test case assertions and for program assertions before and after the improvement process. The human in the loop during these experiments was represented by the author.

Overall, results show that the proposed oracle improvement process is effective in improving three important types of initial oracles (implicit, inferred and manual). The process typically involved from one to three iterations of successive oracle refinement to converge to an oracle for which no deficiency is reported. In case of comparison between the program assertions before and after iterative oracle improvement, the highest mutation score increase was observed for subjects with no initial oracle (other than the implicit one): the implicit oracle is unable to react to the injected faults in most cases. Remarkably, for 72% of the classes with no initial oracle, the mutation score increased from 0% to 100%. A substantial increase in the mutation score was observed for subjects equipped with Daikon assertions. A smaller, still quite relevant, mutation score increase occurred for subjects coming with manually written JML contracts. While for 20% of the classes with JML contracts the mutation score did not change at all, for the remaining 80% of the classes oracle improvement contributed to a higher mutation killing capability. The improved program assertions achieve 51.8% and 53.4% higher mutation score than the test case assertions generated by EvoSuite and Randoop respectively. The average number of program assertions in the subject classes is 20 and the average number of test case assertions is 18 in EvoSuite and 55 in Randoop. This shows that program assertions require the manual validation of a lower(Randoop) or comparable(EvoSuite) number of assertions but have a higher fault detection capability.

In all cases, the observed mutation score increase is statistically significant ($p \leq 0.05$). The Vargha-Delanay effect size $\hat{A}_{12}$ is always *large* (in our study, $\hat{A}_{12} \geq 0.89$).

In addition, we detected three real bugs in Apache Commons Math project which have been reported to and then fixed by the developers.

3 FUTURE PLAN

3.1 Empirical Studies with Humans

The goal of our study is to investigate to what extent the oracle deficiency reports provided by our tool improve the ability of developers to fix assertions. We measure such an impact in the context of a testing scenario in which a Java method is provided and it has oracles with a false positive or a false negative. The *quality* focus concerns the easier improvement process of the oracles when the reports are provided compared to the improvement process without reports. We therefore plan to design our study to answer the following research questions (RQs):

RQ1: *Do oracle deficiency reports lead to a better detection of false positives and false negatives?* Our first objective is to check whether in case of the presence of an oracle deficiency humans are able to identify it without the use of the tool.

RQ2: *Do oracle deficiency reports impact the quality of final assertions after the improvement process?* Our second objective is to verify whether our approach leads to the generation of a higher quality oracles (with less false positives and false negatives).

RQ3: *Do oracle deficiency reports lead to a faster oracle improvement process?* The aim is to assess whether it takes less time to

improve oracles when our approach for oracle improvement is applied.

Before starting the experiment, we plan to conduct a training session, in which we will provide information about oracles, program assertions, false positives, false negatives and mutation testing. Then two sessions will be conducted. In the first session, each participant will be provided with oracles and will be asked to indicate whether in their opinion this oracle contains a false positive or a false negative. The aim of this session is to check whether our tool is able to provide oracle deficiency reports in cases when humans are not successful with the detection of these deficiencies. In the second session participants will be provided with 4 tasks: in two of them they will be asked to improve oracles without the output of our tool and in the other two with the output of the tool. The participants will record the time they spent on each task.

3.2 Analysis of FEP in Programs with Real Faults

The effectiveness of testing depends on the use of oracles that are sensitive to any deviation from the intended program behavior and that report all such deviations as test failures. One key decision about the use of oracles is their placement. Oracles can be placed in test cases, i.e., outside the method under test; at the end of the execution of the method under test, acting as a post-condition for it; or even internally, predicating on the intermediate program states observed during method execution.

An external oracle (aka test case oracle) has a limited capability to discriminate between incorrect and correct method executions, since it can only check the value returned by the method under test and the externally observable state affected by the method under test (e.g., global variables, externally observable object states, persistent changes in the environment, etc.). In a specific program execution, an error may escape detection by an external oracle if it generates an internal state that differs from the expected one without producing any externally visible effect. This means it returns the expected value and it changes the externally visible state in the expected way. Of course, in order for this to be an error, there must be at least one execution where the error produces an externally visible incorrect effect. Hence, external oracles can eventually detect all faults, but they may require a lot of test cases if there is only a low probability that the internal state differences propagate to externally visible differences. When this happens, we say the method is subject to *external failed error propagation* (FEP), aka *unobserved error propagation*. External oracles are weak in comparison with return point or inner oracles when external FEP happens.

A return point oracle (i.e., an internal oracle placed at return points) is more powerful than an external oracle because it can predicate on the entire execution state at the return points, not just on the externally visible state. However, return point oracles may also be subject to FEP – in this case, called *internal FEP* or *squeezed error propagation*. In fact, in a specific program execution, the error, which we assume as detectable externally in other executions, might generate an internal state which differs from the expected one, but such a difference might disappear when the execution proceeds from the faulty statement to the return statement, where no state difference with respect to the expected state is observed.

To analyse internal and external FEP in programs with real faults we plan to use Defects4J [9][8], a database of existing faults, which contains 395 real bugs from 6 real-world Java open source projects. For each bug, we will perform an analysis to identify methods/constructors that have been changed as a result of a bug fix.

As FEP might occur only for specific inputs, to measure its probability we need a large number of executions that cover the faulty statements. To obtain these executions, we are going to extend the EvoSuite [5] test case generator. The standard line coverage criteria of EvoSuite aims at generating a test suite that covers all lines of code. However, we need to cover only lines of code that contain faulty statements. Moreover, we need these lines of code to be covered multiple times, by different test cases. To achieve this, we will change EvoSuite's implementation so that it covers the given list of lines of code the given number of times.

To identify the cases of internal and external FEP, we should trace both faulty and fixed methods, and compare the values of variables at corresponding program points in the faulty and fixed versions of the method. In simple scenarios where the fault fix requires only a change in an existing statement, the correspondence between program points is trivially by position in the linearly ordered sequence of statements, i.e., corresponding program points are program points with the same line number. However, in more complex cases, which require the addition of new statements and/or the deletion of existing statements, the statement sequences aligned by order must exclude program points that refer to added/deleted statements. We plan to identify corresponding statements by calculating the tree edit distance between the Abstract Syntax Trees (AST) of faulty and fixed methods. We will represent the source code of faulty and fixed versions of a method as an AST using a tool to parse Java source code and then adapt tree edit distance computation algorithm.

We will also apply this analysis to the faulty versions of the methods created using a mutation generation tool. With this we want to investigate whether results on mutants correspond to the results obtained on real faults. Moreover, we will conduct a manual analysis of all the available real faults and generated mutations to better understand the prevalence patterns behind FEP or no FEP.

By conducting this experimental procedure we aim to answer the following research questions:

RQ1: *What is the prevalence of external and internal failed error propagation with real faults?*

RQ2: *Does the prevalence of failed error propagation change if real faults are replaced by mutants?*

RQ3: *What are the factors affecting the existence or absence of FEP?*

4 RELATED WORK

The importance of oracles as an integral part of the testing process has been a key topic of research for over three decades [17, 22, 24]. Five large surveys on the topics related to oracles have been conducted till now [3] [20] [15] [18] [4]. The work by Staats et al. [22] proposed a theoretical analysis that included test oracles in a revisitation of the fundamentals of testing.

In their work Huo and Clause [6] measure the quality of the oracles in terms of the presence of brittle test case assertions and

unused inputs. While their approach was able to detect 164 tests containing brittle assertions and 1,618 tests containing unused inputs among 4000 real test cases, it has a high false positive rate. The work by Schuler and Zeller [19] introduces the concept of checked coverage - the dynamic slice of covered statements that actually influence the oracle. The results of their study show that checked coverage is a better indicator of the quality of testing than coverage alone. However, no guidance is provided on how to improve the oracle quality. The work by Zhang et al. [27] introduces *iDiscovery* which aims to improve the quality of the oracles iteratively using symbolic execution. However, it is applicable only to automatically inferred oracles and the level of improvement is limited by the initial set of candidate invariant templates.

There are only two existing studies that evaluate the quality of human input for their proposed approaches to work. They respectively use CrowdSourcing [16] to verify test case assertions and use developers to determine user classification effectiveness for invariants [21] and their results contradict each other. The second study indicates that human testers are not good at identifying correct test oracles, while the first one indicates that human testers can reliably identify correct test oracles and fix incorrect ones. This shows that there is a need of more experiments analysing the performance of human testers in the oracle improvement process.

The most related work to FEP [1, 2, 10–12, 14, 23, 25, 26, 28] reports evidence of FEP based on mutants (or seeded faults), used to *simulate* real faults. Almost without exception, programs are written in C. Only a few previous papers [13, 25] considered FEP for Java programs. However, none of them attempted to measure FEP on real faults.

5 CONCLUSION

This paper presented our approach for oracle improvement and assessment and the current results which demonstrate an improvement in the quality of initial assertions in terms of mutation score and the possibility to detect real faults during improvement process. Our next goal is to validate our results conducting experiments with real developers. Another problem we plan to address is oracle placement problem. First we plan to analyse failed error propagation in Java programs with real faults. In case the results show that it is common for the faults not to propagate to the externally visible state, we will investigate the methods to identify the program points for oracle placement which will provide the maximum fault-detection capability.

REFERENCES

[1] K. Androutsopoulos, D. Clark, H. Dan, R. M. Hierons, and M. Harman. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 573–583, 2014.

[2] A. Bandyopadhyay. *Mitigating the effect of coincidental correctness in spectrum based fault localization.* PhD thesis, Colorado State University. Libraries, 2007.

[3] L. Baresi and M. Young. Test oracles. technical report. Technical report, Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, 2011.

[4] E. T. 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, May 2015.

[5] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, pages 416–419. ACM, September 5th - 9th 2011.

[6] C. Huo and J. Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 621–631, 2014.

[7] G. Jahangirova, D. Clark, M. Harman, and P. Tonella. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA*, pages 247–258, 2016.

[8] R. Just, D. Jalali, and M. D. 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 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[9] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *International Symposium on Foundations of Software Engineering (FSE)*, pages 654–665, 2014.

[10] Y. Li and C. Liu. Using cluster analysis to identify coincidental correctness in fault localization. In *Computational and Information Sciences (ICCIS), 2012 Fourth International Conference on*, pages 357–360. IEEE, 2012.

[11] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 1–5. ACM, 2009.

[12] W. Masri and R. A. Assi. Cleansing test suites from coincidental correctness to enhance fault-localization. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 165–174. IEEE, 2010.

[13] W. Masri and R. A. Assi. Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM Trans. Softw. Eng. Methodol.*, 23(1):8:1–8:28, 2014.

[14] Y. Miao, Z. Chen, S. Li, Z. Zhao, and Y. Zhou. Identifying coincidental correctness for fault localization by clustering test cases. In *SEKE*, pages 267–272, 2012.

[15] R. A. P. Oliveira, U. Kanewala, and P. A. Nardi. Automated test oracles: State of the art, taxonomies, and trends. *Advances in Computers*, 95:113–199, 2015.

[16] F. Pastore, L. Mariani, and G. Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *ICST'13: Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, pages 342–351. IEEE Computer Society, 2013.

[17] D. K. Peters and D. L. Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

[18] M. Pezzè and C. Zhang. Automated test oracles: A survey. *Advances in Computers*, 95:1–48, 2015.

[19] D. Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pages 90–99, 2011.

[20] S. R. Shahamiri, W. M. N. W. Kadir, and S. Z. Mohd-Hashim. A comparative study on automated software test oracle methods. In *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, ICSEA '09, pages 140–145, Washington, DC, USA, 2009. IEEE Computer Society.

[21] M. Staats, S. Hong, M. Kim, and G. Rothermel. Understanding user understanding: Determining correctness of generated program invariants. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 188–198, New York, NY, USA, 2012. ACM.

[22] M. Staats, M. W. Whalen, and M. P. E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 391–400, 2011.

[23] X. Wang, S. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 45–55, 2009.

[24] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, Nov. 1982.

[25] Y. Xiong, D. Hao, L. Zhang, T. Zhu, M. Zhu, and T. Lan. Inner oracles: input-specific assertions on internal states. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 902–905, 2015.

[26] X. Xue, Y. Pang, and A. S. Namin. Trimming test suites with coincidentally correct test cases for enhancing fault localizations. In *IEEE 38th Annual Computer Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21-25, 2014*, pages 239–244, 2014.

[27] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 362–372, New York, NY, USA, 2014. ACM.

[28] Z. Zheng, Y. Gao, P. Hao, and Z. Zhang. Coincidental correctness: An interference or interface to successful fault localization? In *Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on*, pages 114–119. IEEE, 2013.