OASIs: Oracle Assessment and Improvement Tool

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

David Clark
UCL, London, UK
david.clark@ucl.ac.uk,

Mark Harman
Facebook, London, UK & UCL, London, UK
markharman@fb.com

Paolo Tonella
FBK, Trento, Italy
tonella@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. We introduce OASIs, a search-based tool for Java that assists testers in oracle assessment and improvement. It does so by combining test case generation to reveal false positives and mutation testing to reveal false negatives. In this work, we describe how OASIs works, provide details of its implementation, and explain how it can be used in an iterative oracle improvement process with a human in the loop. Finally, we present a summary of previous empirical evaluation showing that the fault detection rate of the oracles after improvement using OASIs increases, on average, by 48.6%.

CCS CONCEPTS

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

KEYWORDS

Test oracle; oracle assessment; oracle improvement; test case generation; mutation testing;

ACM Reference Format:
Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2018. OASIs: Oracle Assessment and Improvement Tool. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3213846.3229503

1 INTRODUCTION

The software testing process consists of two key components: generating test inputs and checking whether the outputs for these inputs are correct. The latter motivates the *oracle problem*, i.e., the problem of defining accurate oracles, capable of detecting all and only faulty behaviours exercised during testing. While there is a large body of research devoted to automated *test input generation*, methods for automatically generating *test oracles* are less common. Some initial steps in this direction are automated test case generation tools as EvoSuite [3, 4] and Randoop [12] which are able to automatically synthesize test assertions. However, these test case assertions encode the *implemented* behaviour of the program rather than the *intended* behaviour. Therefore, to turn them into oracles there is a need to identify and fix the incorrect ones, which requires human intelligence. Another form of automated oracles are *dynamically inferred invariants* [1]. They are extracted from a finite set of execution traces. As a result, they can have a high false positive rate [16] and, therefore, also require human intervention. However, developers struggle with this task. On average, they misclassify 9.1% to 31.7% of correct invariants as incorrect and 26.1%-58.6% of incorrect invariants as correct [15].

OASIs[1] (**O**racle **AS**sessment and **I**mprovement) is a search-based tool that aims to support the developer in assessing and improving oracles. It targets in-program logical assertions (program invariants) in Java and analyses oracles based on two properties: **Completeness**: All correct program states are accepted by the oracle, which raises an alarm only for faulty states, with no false alarms (no *false positives*). **Soundness**: All faulty program states are rejected by the oracle, so there are no missed faults (no *false negatives*). OASIs generates counterexamples as test cases that demonstrate incompleteness and unsoundness, which the tester uses to improve the assertion oracle.

In Section 2 we introduce our approach that is based on search based test case generation [3, 6, 11] to identify false positives and mutation testing [9, 10] to identify false negatives. In Section 3 we explain how OASIs can be used for an iterative oracle assessment and improvement process with the developer in the loop. Finally, in Section 4 we provide a summary of previous empirical evaluation of OASIs [8].

2 COMPONENTS & IMPLEMENTATION

OASIs is a command-line tool, see Figure 1, which takes five parameters as input: source code location of the Java class, the name of the class, the name of the method where the initial assertions are located, the search budget for FP detection and the search budget for FN detection. The last two parameters are optional and, if omitted, OASIs uses the default budgets of 60 seconds for FP and of 120 seconds for FN detection. OASIs starts the oracle assessment process by first looking for a False Positive. If no False Positive is detected, the search for False Negatives is initiated. The output of the tool consists of a message which, in case oracle deficiency is detected comprises the exact kind, or just indicates that no deficiency

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3229503

[1] https://github.com/guneljahan/OASIs

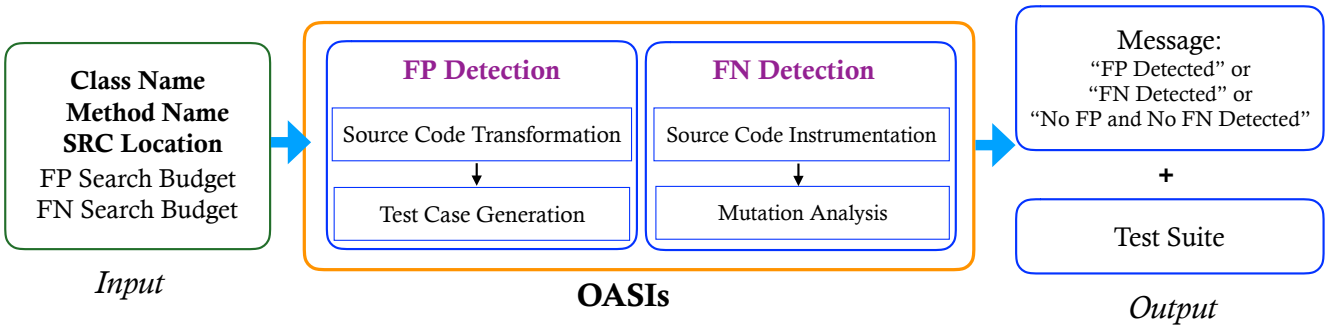

Figure 1: OASIs Components

was found. For each detected oracle deficiency, the evidence (in the form of test suite) is provided.

2.1 False Positive Detection

Given a program assertion, we detect its false positives by generating execution scenarios where the assertion fails yet 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.

Source code transformation. First, we perform a testability transformation [5] that transforms the assertion in the code into a new branch. 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)) {}. Figure 2 shows an example of such a transformation. The condition of the assert statement at Line 4 '`(result != x)`' in Figure 2 (top), is negated to '`(!(result != x))`' and then the assertion is replaced with the branch: '`if (!(result != x)) {}`' in Figure 2 (middle). The source code transformation also detects the lines of code where the newly-created branches are located and passes them to the test case generator, so that these branches can be differentiated from the already existing ones.

Test Case Generator. After this transformation, the criterion for false positive detection turns into the standard branch coverage criterion. The test case generator to cover the newly created branches is developed as an extension of EvoSuite's branch coverage criterion [2, 3]. Let P be the original program and B the set of branches in P. Let P' be the transformed version of P and B' the set of branches in P'. The standard version of EvoSuite will aim to cover all the branches in P'. However, we are interested in covering only branches $B_A = B' - B$, i.e., the set of branches that are created as a result of the transformation of assertions in P into branches. We altered the fitness function of EvoSuite so that it aims to cover only the '`then`' parts of the '`if`' statements at branches in B_A. In Figure 2, the bottom part shows an example of a test case generated as evidence of a False Positive for the assertion at line 4 in the top part. Indeed, if we execute the reported test case this assertion will fail, as *result* is actually equal to *x*.

```
1  public class Subtract {
2     public int value(int x, int y) {
3           int result = x - y;
4           assert (result != x);
5           return result; } }
```

```
1  public class Subtract {
2     public int value(int x, int y) {
3           int result = x - y;
4           if (!(result != x)) {}; // target
5           return result; } }
```

```
1  @Test(timeout = 4000)
2   public void test0() throws Throwable {
3     Subtract subtract0 = new Subtract();
4     int int0 = subtract0.value(1057, 0); }
```

Figure 2: Example of False Positive Detection, including an initial class with incorrect assertion (at Line 4, top), source code transformation for the class (middle), and generated test case to report False Positive (bottom)

2.2 False Negative Detection

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 into program P that corrupts the program state so that the corrupted state reaches the given assertion and the assertion statement does not fail.

Source-code instrumentation. First, we instrument the source code of the class so that we can monitor (1) the values of all variables visible at the program point where the assertion is located (2) the outcome of the assertion, i.e. whether it passes or fails.

Mutation Analysis. After the instrumentation, we use EvoSuite's strong mutation killing criterion. Let us consider the implementation under test P and its mutations $M_1, \dots, M_k$. Program P and each of its mutants have n assertions $a_1, \dots, a_n$: $a_i = assert(c_i)$, $i \in [1 \dots n]$. Let us consider the variables $(v_1, \dots, v_{m_i})$ in scope at the assertion point pp_i. Their values after running a test case on P are $(v_1^o, \dots, v_{m_i}^o)$, while they are $(v_1^{M_j}, \dots, v_{m_i}^{M_j})$ after running the same test case on mutant M_j.

```
1  public class FastMath {
2     public int getMax (int a, int b) {
3          int max;
4          if (a >= b) {
5               max = a; //max = -a;
6          } else {
7               max = b;
8          }
9          assert (max >= a && max >= b);
10         return max; } }
```

```
1    //1. getMax, Line 5 InsertUnaryOp Negation(max:-1,1)
2    @Test(timeout = 4000)
3    public void test0() throws Throwable {
4      FastMath fastMath0 = new FastMath();
5      int int0 = fastMath0.getMax((-1), (-110)); }
```

Figure 3: Example of False Negative Detection, including an initial class with weak assertion (at Line 9, top), and generated test case and mutations to report False Negative (bottom)

In EvoSuite, a mutant is strongly killed if EvoSuite can create a *test case assertion* (not to be confused with the *program assertions* that are assessed for false negatives) that evaluates to false if the test is executed on the mutant and to true if it is executed on the original class. To detect false negatives, we further restricted the notion of mutation killing by adding two additional conditions to be satisfied: (1) The conditions in the program assertions do not change their values: $\forall i \in [1 \dots n] : c_i^{M_j} = c_i^o$. (2) At least, one of the variables visible at pp_i has different values in P and M_j: $\exists i \in [1 \dots n] : v_1^{M_j} \neq v_1^o \vee \dots \vee v_{m_i}^{M_j} \neq v_{m_i}^o$.

Output Improvement. The original output of EvoSuite's strong mutation killing criterion produces a test suite in which, for each test case, it lists mutations that are strongly killed by the test case. We change the output, so that for each mutation we also list variables that have changed their values as a result of mutation. If a variable has a primitive type we also provide the values in the original and mutated version. This provides additional support for the developer in the improvement process by indicating which variables the program assertion ignores or does not check strongly enough.

In Figure 3 (top) we provide an example of a method with weak assertion (at line 9). OASIs reports a False Negative for this assertion, as in Figure 3 (bottom). The report contains a test case and a description of the mutation in the comments above the test case. As it follows from the description, the mutation applies a unary negation operator to variable *a* at line 5, changing the value of variable *max* from -1 to 1. However, the assertion in the method does not react to this change, as in the mutated version *max* is equal to 1, which is still greater than the value of *a* == -1 and *b* == -110. This False Negative can be eliminated by replacing the assertion in Figure 3 (top) with `assert (max >= a && max >= b && (max == a || max == b));`.

3 ITERATIVE IMPROVEMENT PROCESS

We propose a process for iterative oracle assessment and improvement based on the outcomes of false positive/negative detection by OASIs. As illustrated in Figure 4, the human is an integral part of the process, as a source of knowledge about the intended behaviour of the program. Moreover, the human in the loop is asked to manually improve the oracle when a false negative or a false positive is reported.

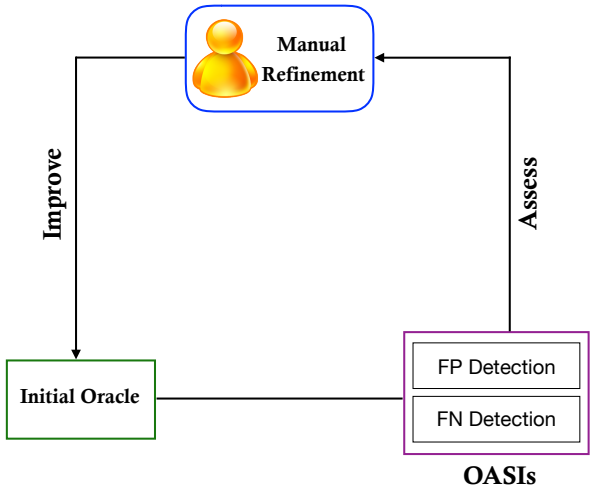

Figure 4: Iterative Improvement Process

The starting point for iterative oracle assessment and improvement is an initial oracle. This oracle can be defined manually by developers, or can be produced automatically by tools for invariant inference, like Daikon [1], or can even be the empty (implicit) oracle. *Oracle deficiencies* (i.e. false negatives or false positives) are detected and reported automatically by OASIs. The developer fixes the assertions in the program, based on the reported oracle deficiencies. Some care must be taken in this step, in order to recognise the following corner cases: (1) A reported false positive might point to a bug in the program, not in the assertion; (2) A test case killing a mutant and triggering an assertion violation in the mutant might be associated with consistent bugs in both implementation and assertion; (3) A mutant might accidentally fix a fault in the program (this is expected to occur extremely rarely), causing a reported false negative to point to a bug in the program, not in the assertion. The first case is important, since the improved oracle is immediately used for fault detection when this case occurs.

Once assertions have been improved by the developer, the iterative process restarts and the new assertions are assessed for the presence of further oracle deficiencies. The process continues until the OASIs is unable to generate new counterexamples and finishes with an improved (more complete and sound) oracle.

4 EVALUATION SUMMARY

In this section we summarize our previously published empirical evaluation [8].

Different types of initial oracles. We have assessed the applicability of our approach for three types of initial oracles: (1) implicit oracle where no assertion is present, hence fault detection relies entirely on program crashing or raising exceptions (2) inferred

properties, where we use invariants generated by Daikon as initial assertions (3) manual oracle where initial oracles are already provided with the SUTs in form of JML specification, which we transformed into standard Java assertions. Overall, results show that OASIs is effective in improving all three types of initial oracles. The process typically involves from one to three iterations to converge to an oracle for which no deficiency is reported.

Effectiveness. The effectiveness of the improved oracle is assessed in terms of increased fault detection with respect to the initial and test case oracle. We analyse the mutation score for test case assertions and for program assertions before and after the improvement process. The results show an 85.9% improvement for implicit, 42.0% for inferred and 19.9% for manual assertions. The improved program assertions achieve 51.8% and 53.4% higher mutation score than the test case assertions generated by EvoSuite and Randoop respectively. In all cases, the observed mutation score increase is statistically significant ($p \leq 0.05$). The Vargha-Delaney effect size $\hat{A}_{12}$ is always *large* (in our study, $\hat{A}_{12} \geq 0.89$).

Real bug detection. During our experiments we detected 4 real bugs in Apache Commons Math project (MATH-1256, MATH-1258, MATH-1259, MATH-1414), which have been reported to (and then fixed by) the developers.

Human in the loop. During our experiments the human in the iterative oracle improvement process was represented by the first author. She had no familiarity with the subjects, no previous experience in writing specifications but, of course, knew very well how to interpret the output of the tool. We defined precise rules and procedures for oracle improvement to be followed by the human experimenter, to mitigate internal threat to validity. As a result, the human in the loop in our experiments has behaved largely deterministically and unimaginatively.

5 RELATED WORK

Different metrics have been proposed to assess test oracle quality. In their work Huo and Clause [7] measure it 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 4,000 real test cases, it has a high false positive rate. The work by Schuler and Zeller [14] 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. [17] introduces *iDiscovery*, which aims to improve the quality of the oracles iteratively using symbolic execution. However, it is applicable only to automatically inferred oracles.

There are only two existing studies that evaluate the successfulness of humans in improving automated oracles. They respectively use CrowdSourcing [13] to verify test case assertions and use developers to determine user classification effectiveness for invariants [15]. 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 qualified human testers can reliably identify correct test oracles and fix incorrect ones. This shows a need in more experiments analysing the performance of human testers in the oracle improvement process.

6 CONCLUSION AND FUTURE WORK

We present OASIs, a tool for assessing and improving test oracles by reducing the incidence of both false positives and false negatives. Experimental results show that OASIs is able to identify both false positives and false negatives in three important types of initial oracles (implicit, inferred and manual), leading to an average 48.6% improvement of mutation score over all the analysed classes and exposing real faults that have been reported to and fixed by the developers. Our next goal is to validate our results by conducting experiments where OASIs is used by real developers.

REFERENCES

[1] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69:35–45, December 2007.

[2] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In M. Núñez, R. M. Hierons, and M. G. Merayo, editors, 11^{th} *International Conference on Quality Software (QSIC)*, pages 31–40, Madrid, Spain, July 2011. IEEE Computer Society.

[3] 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.

[4] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Trans. Software Eng.*, 38(2):278–292, 2012.

[5] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Trans. Software Eng.*, 30(1):3–16, 2004.

[6] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing (keynote). In 8^{th} *IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*, Graz, Austria, April 2015.

[7] 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.

[8] 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.

[9] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649 – 678, September–October 2011.

[10] 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.

[11] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[12] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

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

[14] 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.

[15] 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.

[16] P. Tonella, C. D. Nguyen, A. Marchetto, K. Lakhotia, and M. Harman. Automated generation of state abstraction functions using data invariant inference. In *Proceedings of the 8th International Workshop on Automation of Software Test (AST)*, 2013.

[17] 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.