OASIs: Oracle Assessment and Improvement Tool

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

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

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

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:

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].

OASIs1 (Oracle ASsessment and Improvement) 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 [5, 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

1https://github.com/guneljahan/OASIs
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 \( a_1 \ldots a_n : a_i = \text{assert}(c_i), i \in [1 \ldots n] \), where \( c_i \) is the boolean expression used in the assertion \( a_i \). For each assertion \( a_i, i \in [1 \ldots n] \) 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 \(! (c_i) \) [1]. Figure 2 shows an example of such a transformation. The condition of the assert statement at Line 4 `\((\text{result } \neq x)\)` in Figure 2 (top), is negated to `\! (\text{result } \neq x)` and then the assertion is replaced with the branch: `if (!(\text{result } \neq 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 \( \text{result} \) is actually equal to \( x \).

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 \( M_1, \ldots, M_k \). Program \( P \) and each of its mutants have \( n \) assertions \( a_1 \ldots a_n : a_i = \text{assert}(c_i), i \in [1 \ldots n] \). Let us consider the variables \( (v_1, \ldots, v_m) \) in scope at the assertion point \( p_i \). Their values after running a test case on \( P \) are \( (v_{10}^{}, \ldots, v_{m0}^{}) \), while they are \( (v_{1}^{M_1}, \ldots, v_{m}^{M_k}) \) after running the same test case on mutant \( M_j \).
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

<table>
<thead>
<tr>
<th>Initial Oracle</th>
<th>Improve</th>
<th>Assess</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual Refinement</td>
<td>FP Detection</td>
<td>FN Detection</td>
</tr>
<tr>
<td>Impeove</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OASIs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 ($p < 0.05$). The Vargha-Delaney effect size $A_{12}$ is always large (in our study, $A_{12} \geq 0.89$).

**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 Discovery, 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 successfullness 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**


