An Empirical Validation of Oracle Improvement

Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella

Abstract—We propose a human-in-the-loop approach for oracle improvement and analyse whether the proposed oracle improvement process is helping developers to create better oracles. For this, we conducted two human studies with 68 participants overall: an oracle assessment study and an oracle improvement study. Our results show that developers exhibit poor performance (29% accuracy) when manually assessing whether an assertion oracle contains a false positive, a false negative or none of the two. This shows that automated detection of these oracle deficiencies is beneficial for the users. Our tool OASIs (**O**racle **AS**sessment and **I**mprovement) helps developers produce assertions with higher quality. Participants who used OASIs in the improvement study were able to achieve 33% of full and 67% of partial correctness as opposed to participants without the tool who achieved only 21% of full and 43% of partial correctness.

Index Terms—Oracle Problem, Test Oracle, Oracle Assessment, Oracle Improvement, Human Study, Test Case Generation, Mutation Testing

1 Introduction

Recent advances in test input generation have left the Oracle Problem as a key remaining bottleneck in the improvement of the overall effectiveness and efficiency of the software testing process. Indeed, the effectiveness of testing depends both on the quality of the test cases and on the quality of the oracle [2], [40], [44]. There are many techniques for assessing and improving the adequacy of test cases, e.g., their code coverage, and many hundreds of studies about search-based [16], [28] and symbolic execution [5] techniques alone. 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 and only faulty behaviours exercised during testing [12], [18], [25], [36], [37], [38]. Without a (good) oracle to determine whether the test output is correct, test inputs that satisfy the strictest adequacy criteria remain useless and testing is ineffective.

The oracle's performance depends on two properties: **Completeness**: All correct program states should be accepted by the oracle, which should raise an alarm only on faulty states, with no false alarms (no *false positives*). **Soundness**: All faulty program states should be rejected by the oracle, so that there are no missed faults (no *false negatives*). Oracle assessment must thus identify and report false positives or false negatives (or both), so as to support the developer in improving the oracle soundness and completeness.

In our previous work [19] we introduced an approach that is based on search based test case generation [10], [15], [28] to identify false positives and mutation testing [21], [22] to identify false negatives. Our tool OASIs (**O**racle **AS**sessment and **I**mprovement) [20] generates counterexamples as test cases that demonstrate incompleteness and unsoundness. The tester uses them to iteratively improve the oracle. The process continues until OASIs 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 false positives and false negatives 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.

In the initial evaluation of our approach 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 OASIs.

In this paper we aim to analyse further whether our approach is helpful for testers to create better oracles. For this purpose we conducted two different human studies with 68 participants overall. 39 participants were involved in our *Oracle Assessment Study*, where we assessed the ability of humans to detect false positives and false negatives manually, without any tool support. The results of this study are indicative of how helpful the automated detection of oracle deficiencies could be for developers. Then, 29 different participants were involved in our *Oracle Improvement Study*, where they were assigned to two different groups (control and treatment). Participants from the first group were given initial assertion oracles (for which the oracle deficiency type was indicated) to be improved manually. Participants from the second group performed an iterative improvement process on the same initial oracles with the support of our tool, playing the role of the human in the loop. The comparison of the quality achieved in the final oracles validates empirically the effectiveness of the proposed approach.

This work extends our previous paper [19] with the following novel contributions:

- *G. Jahangirova is with Fondazione Bruno Kessler and University College London.*
- *D. Clark is with University College London.*
- *M. Harman is with Facebook and University College London.*
- *P. Tonella is with Università della Svizzera Italiana (USI).*

- A novel human study on oracle assessment;
- A novel human study on oracle improvement.

The paper is organised as follows. Section 2 summarises our previous work [19] and its preliminary evaluation. Section 3 reports new experimental results of the oracle assessment and oracle improvement studies, and enumerates the threats to validity. We discuss the related work in Section 4. Finally, Section 5 concludes the paper.

2 Background

2.1 Quality of Assertions

Let us consider a program point, pp, in a program under test, P. Let Σ be the set of all states that can occur in P and $I \subseteq \Sigma$ be the set of start states. We are interested in the set of states that reach pp via execution of P on I.

$$R_{pp} = \{s \mid \exists i \in I \wedge [\![P]\!]_{pp}\ i = s\}$$

where $[\![P]\!]_{pp}\ i$ indicates the state reached at pp by executing P on $i \in I$.

We place an assertion, $\langle assert \rangle$, at pp with the intention of using this assertion as an oracle. Let us define:

$$A_{pp} = \{s \in R_{pp} \mid \langle assert \rangle s = \mathrm{T}\}$$

to be the set of reachable states for P at pp on which the assertion is true.

Let us consider the states that occur at pp and are correct (the perfect oracle). We call this set E_{pp}, and we can think of E_{pp} as the intersection between the set of correct states at pp for a correct "ghost program" [1], G (an error free version of the program), and R_{pp}, the reachable states of the program under test.

$$E_{pp} = \{s \in R_{pp} \mid \exists i \in I \wedge [\![G]\!]_{pp}\ i = s\}$$

Subsequently we will drop the subscript pp from R, E and A where the program point is clear from context.

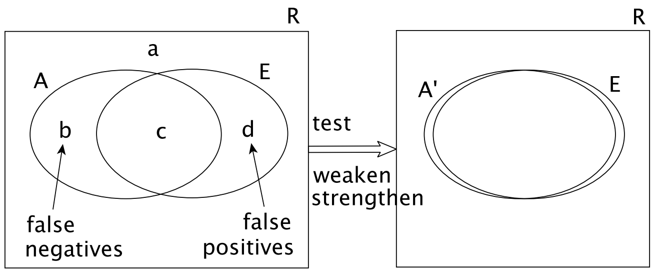

Fig. 1: Assertion improvement process: the intersection between states where $\langle assert \rangle$ is true (A) and expected states (E), restricted to the subset of reachable states (R), is increased.

The overall aim of the testing process is to expose and fix faults via a cycle of testing and revision of P so that E_{pp} is as large as possible at every program point in P, making P closer to G. Oracle improvement occurs within a given cycle, i.e., for a fixed P, during the testing phase. By *oracle improvement* we mean a process aimed at refining $\langle assert \rangle$ so as to obtain a new assertion, $\langle assert \rangle'$ for which A' has a larger overlap with the current E. Eventually, we would like to obtain a new assertion such that $A' \cap E = A' = E$ so that the states at pp on which the new assertion is true are exactly the "correct" states of the ghost program. The starting point of this process is represented in Figure 1, left.

Here, the region $(A - E)$ is the set of states of P which are not "correct" but on which $\langle assert \rangle$ is True, that is the set of reachable False Negatives, while $(E - A)$ is the set of "correct" states on which the assertion is False, that is the set of reachable False Positives.

Definition 1 (False Negatives) *A false negative is a reachable program state where the given assertion is True, although such state does not belong to the set of expected states according to the intended program behaviour.*

Definition 2 (False Positives) *A false positive is a reachable program state where the given assertion is False, although such state does belong to the set of expected states according to the intended program behaviour.*

The notions of False Positives and False Negatives are tightly connected with the notions of oracle soundness and completeness. An assertion $\langle assert \rangle$ is *Complete iff* the "correct" reachable states are a subset of the states accepted by the assertion, i.e. $E \subseteq A$. An assertion $\langle assert \rangle$ is *Sound iff* the accepted states are a subset of the "correct" reachable states, i.e. $A \subseteq E$. Completeness implies that the number of False Positives is zero; soundness implies that the number of False Negatives is zero.

After testing for False Positives and False Negatives we can strengthen $\langle assert \rangle$ to reduce the number of False Negatives and simultaneously weaken it to reduce the number of False Positives, producing a new assertion, $\langle assert \rangle'$ in the process illustrated in Figure 1, right. By reducing the number of False Positives and False Negatives, the proposed oracle assessment and improvement process will make the oracle more complete and sound.

2.2 Approach & Implementation

2.2.1 False Positive Detection

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

```
public int value(int x, int y) {
    int result = x − y;
    assert (result != x);
    return result; }
```

```
public int value(int x, int y) {
   int result = x − y;
   if (!(result != x)) {}; // target
   return result; }
```

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

Fig. 2: Example of False Positive Detection

First, we perform a testability transformation [14] that transforms the assertion in the code into a new branch. Let us consider a program under test P containing n assertions $a_1 \ldots a_n$:

$$a_i = \texttt{assert}(c_i), i \in [1 \ldots n]$$

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:

$$\texttt{if } (!(c_i))\{\}$$

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). After this transformation, the criterion for false positive detection turns into the standard branch coverage criterion. We developed a test case generator to cover the newly created branches as an extension of EvoSuite's branch coverage generator [9], [10]. 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*.

2.2.2 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.

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

Fig. 3: Example of False Negative Detection

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 and (2) the outcome of the assertion, i.e. whether it passes or fails. 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 $a_1, \ldots, a_n$:

$$a_i = \texttt{assert}(c_i), i \in [1 \ldots n]$$

Let us consider the variables $(v_1, \ldots, v_{m_i})$ in scope at the assertion point pp_i. Their values after running a test case on P are $(v_1^P, \ldots, v_{m_i}^P)$, while they are $(v_1^{M_j}, \ldots, v_{m_i}^{M_j})$ after running the same test case on mutant M_j.

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:

$$\forall i \in [1 \ldots n] : c_i^{M_j} = c_i^P$$

(2) at least, one of the variables visible at pp_i has different values in P and M_j:

$$\exists i \in [1 \ldots n] : v_1^{M_j} \neq v_1^P \vee \ldots \vee v_{m_i}^{M_j} \neq v_{m_i}^P$$

Condition (2) ensures that we exclude equivalent mutants from our analysis, as if the mutant is equivalent it could not lead to a change of value in any of the variables at the assertion point.

In Figure 3 (top) we provide an example of a method with a 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 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))`.

2.2.3 Iterative Improvement Process

We propose a process for iterative oracle assessment and improvement based on the outcomes of false positive/negative detection as reported by OASIs. OASIs is implemented as a command-line tool which takes five parameters as input: the 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 comprises the exact kind, in case oracle deficiency is detected, or just indicates that no deficiency was found. For each detected oracle deficiency, the evidence (in the form of a test suite) is provided.

However, the output of OASIs on its own is not sufficient for the improvement process. Therefore, the human is an integral part of this semi-automated process, as a source of knowledge about the intended behaviour of the program. The human in the loop is tasked with manually improving the oracle when a false negative or a false positive is reported.

The starting point for iterative oracle assessment and improvement is an initial oracle. This oracle can be the one defined manually by developers, or can be produced automatically by tools for invariant inference, like Daikon [7], or can even be the empty (implicit) oracle, which catches only program crashes or exceptions/errors. *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, 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. Incorrectly changing the oracle when, in fact, it is the program at fault will render the oracle unable to detect fault actually present in the code. The other two are expected to be extremely rare cases (no occurrence of the latter two cases was observed in our experiments).

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 no further counterexamples can be generated and finishes with an improved (more complete and sound) oracle.

2.3 Preliminary Validation

In this section we summarize our preliminary evaluation, during which 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 OASIs.

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 code in the form of JML specifications, which we transformed into standard Java assertions.

TABLE 1: Average mutation score by subject for initial/test case (μ_s) and improved (μ'_s) oracle

Oracle	**Subj**	μ_s	μ'_s	Δ	$\hat{A}_{12}$	p**-value**
Implicit	CM	16%	97.6%	81.6pp	1.0	$1.4 \cdot 10^{-5}$
	CC.	8.3%	98.4%	90.1pp	0.98	$2.2 \cdot 10^{-5}$
Inferred	CL	60.5%	98.8%	38.3pp	0.9	$9.0 \cdot 10^{-3}$
	CM	50.2%	95.8%	45.6pp	0.91	$4.7 \cdot 10^{-4}$
Manual	FE	78.8%	100%	21.2pp	0.9	$6.3 \cdot 10^{-7}$
	LG	81.5%	100%	18.5pp	0.89	$1.7 \cdot 10^{-2}$
All	**All**	**50.1%**	**98.4%**	**48.3pp**	0.92	$< 2.2^{-16}$
Randoop	All	45%	98.4%	53.4pp	0.93	$5.3 \cdot 10^{-7}$
EvoSuite	All	46.9%	98.4%	51.5pp	0.95	$3.8 \cdot 10^{-6}$

The effectiveness of the improved oracle was assessed in terms of increased fault detection with respect to the initial and test case oracle. We analysed the mutation score for test case assertions and for program assertions before and after the improvement process. Table 1 shows results for each type of initial oracle and each subject (CM: Commons Math, CC: Commons Collections, CL: Commons Lang, FE: JavaFE, LG: Logging). The improvement in mutation score is 85.9pp (*pp* means percentage points) for implicit, 42.0pp for inferred and 19.9pp for manual assertions. The improved program assertions achieve 51.8pp and 53.4pp 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, $\hat{A}_{12} \geq 0.89$).

During our experiments we detected 4 real bugs in the Apache Commons Math project (MATH-1256, MATH-1258, MATH-1259, MATH-1414), which have been reported to (and then fixed by) the developers. Overall, results show that our approach is effective in improving all three types of initial oracles. The process typically involved from one to three iterations to converge to an oracle for which no deficiency is reported.

3 Human Study

We assessed (1) whether the proposed approach is beneficial for developers to detect oracle deficiencies by conducting an *Oracle Assessment Study*, and (2) whether the output of the tool helps developers remove oracle deficiencies and create better oracles by conducting an *Oracle Improvement Study*. Both of our studies were approved by UCL's Research Ethics Committee (REC 12857/001 and REC 12005/001). All the experimental data collected is available at the link: https://github.com/guneljahan/OASIs/humanstudy.

3.1 Oracle Assessment Study

To improve an oracle one should first be aware of its current deficiencies and then take actions to get rid of them. Our approach automatically detects false positives and false negatives in the assertions and reports them to the user. To check whether the first task, oracle assessment, is difficult for humans, which would indicate that the information provided by our tool is potentially useful, we conducted a study to analyse *how successful developers are at assessing oracles manually, with no tool support*. With this overall goal in mind, we explored the following research questions:

RQ1: How effective are developers in determining whether the oracle has a deficiency and, if it has one, what the deficiency type is?
RQ2: What are the common misclassifications developers make when assessing oracle deficiencies?

3.1.1 Object Selection

The starting point of our experimental design was the previous study by Staats et al. [39]. In this previous work Staats et al. analysed the user's ability to classify dynamically generated invariants by Daikon as correct or incorrect. An invariant is considered incorrect if there is a test input capable of violating the invariant, which is in line with our definition of a False Positive. Three Java classes were used as subject programs in this previous study: StackAr, Matrix and PolyFunction. *StackAr* is a stack class originally used in user studies about Daikon [29]. *Matrix* is a class representing a matrix, found in the JAMA linear algebra package, developed by The MathWorks and the National Institute of Standard and Technology (NIST) [17]. *PolyFunction* is a class representing a polynomial function, and is part of the Math4J package [27]. The users involved in the previous study analysed 336 invariants generated by Daikon for these classes during the experiments. Moreover, at the end of the task each participant was asked to manually write 5 invariants for each class.

We reused the subject programs of this study, and used the dynamically generated and manually written invariants as our initial oracles. However, the aim of our study was not limited to the analysis of the developers' ability to detect False Positives, but to also include the same analysis for False Negatives. Given this wider task, we decided to give subjects more time for oracle assessment than in the previous study. In our study, we provided participants with 10 assertions from two different classes to be evaluated in 30 minutes. By contrast, in the study by Staats et al. [39] subjects were asked to analyse 112 invariants on average in 60 minutes or 86 invariants on average in 35 minutes, depending on the session.

We selected 15 assertions (5 from each class) among 336 properties inferred by Daikon and 37 human-written assertions. The majority of assertions in this pool check general properties of the class or basic properties of the method (such as the immutability of some variables). Our selection process favoured assertions that were checking the functionality specific to the method under test rather than the general properties. For each assertion we run our tool to detect whether it has a false positive, a false negative or no oracle deficiencies. In case no oracle deficiency was found, we also analysed the assertion manually to ensure that the output of the tool is correct. We selected the final assertions so as to achieve a balance between the numbers of Daikon-generated and human-written assertions, as well as between the numbers of assertions with false positives, false negatives and no oracle deficiencies at all.

Before executing the empirical study, we conducted a pilot study with 2 volunteers (who were not included later in the experiment itself). The results of the questionnaire and the discussion after the pilot study showed that participants think that the time provided was insufficient to analyse 10 assertions in total. Therefore, we reduced the number of assertions to 6 for the main experiment (3 for each class). We also slightly reduced the source code of all three case examples to make the task more feasible.

Table 2 lists the classes from the work of Staats et al. [39] that we reused in our experiments and the number of lines of code, methods and assertions in them. Rows *Assertion 1*, *Assertion 2* and *Assertion 3* indicate whether each assertion has a false positive (FP), a false negative (FN) or no false positives and no false negatives (None) and whether it is human-written (H) or Daikon-generated (D).

TABLE 2: Assessment Study: Subject Programs

	StackAr	**Matrix**	**PolyFunction**
SLOC	94	142	152
# of Methods	11	17	12
# of Assertions	3	3	3
Assertion 1	FN, D	FP, D	FN, H
Assertion 2	None, H	FN, D	FP, H
Assertion 3	FP, D	None, H	FN, D

3.1.2 Participants

To answer our research questions we conducted three separate experimental sessions. The first and third session were conducted with master degree students of the *Security Testing* course at the *University of Trento*. The second session was conducted with professional developers who work at *Fondazione Bruno Kessler*. The analysis of user feedback for the first two sessions showed that participants thought that they did not have enough time to perform the task. Therefore, in the third session we changed the duration of the task from 30 minutes to 45 minutes.

TABLE 3: Assessment Study: Experimental Sessions

	Type of Part.	**# of Part.**	**Duration**
Session 1	MSc Students	20	75 min
Session 2	Prof. Developers	6	75 min
Session 3	MSc Students	13	90 min

Table 3 lists all the sessions conducted during the study, the type and number of participants in each of them along with the whole duration of the session. Overall, 33 master degree students and 6 professional developers participated in our experiments.

3.1.3 Experimental Procedure

At the beginning of each session we provided an identical 30 minute training to the participants: (1) explaining what the oracle problem is; (2) explaining what a false positive and a false negative is; (3) overviewing Java assertions; (4) showing multiple examples of false positives and false negatives in Java assertions; (5) introducing utility classes and constructs used to write assertions (e.g., the boolean implication operator and the way to refer to old values of variables). In the training, the motivation for the assertions in the program was explained to be regression testing, as in regression testing users can assume that the program behaves correctly as is. Correspondingly, the user's task is to determine whether assertions match the program's current behaviour. Indeed, asking participants to judge whether invariants match the *intended* program behaviour would have made the task overly difficult, since participants are not the developers of the classes under study. We also recommended participants to start their analysis of the assertions from the search for false positives. In fact, only after making sure that there is no false positive (the assertion is correct), it makes sense to check whether the assertion has false negatives (the assertion is strong enough to expose arbitrary faults).

After the training session, each subject received an experiment package, consisting of the randomly assigned group id, a statement of consent and instructions on how to proceed with the task. Participants were divided into groups in order to have a balanced number of responses for each subject class. Instructions directed the participants to the website where the source code of Java classes for their group could be downloaded and to the online questionnaire.

During the task, each participant was assigned two Java classes with three assertions each. The objective was to indicate for each assertion whether (1) it has a false positive (2) it has a false negative (3) it has no false positives and no false negatives. In case the subject did not know the answer the option "I don't know" was provided as well. Once the 30 minute (45 minute for the third session) period assigned to the task was complete, the participants proceeded to the questionnaire to answer questions about their background and to provide feedback about the session.

3.1.4 Results

RQ1: User Effectiveness

To answer RQ1 we calculated the correct/incorrect classification ratios for each participant group, investigated the parameters that affect users' performance and measured the agreement rate between participants.

Classification Results. Table 4 presents the results for the two sessions (Session 1 - SS1, Session 3 - SS3) conducted with students. Column *All* shows the overall number of classifications obtained for each assertion. Columns *Correct* and *Incorrect* show the number of correct and incorrect classifications respectively. Column *Don't Know* reports the number of cases when the option "I don't know" was picked for the assertion. While the duration of these sessions was different (30 min vs. 45 min), the results for them are quite similar with the 25% and 26% correct classification rates.

Table 5 shows the results for the 6 professional developers. With 48% correct classification rate they exhibited almost twice better performance than students. For 4 out of 9 assertions, professional developers had no incorrect classifications at all, either always correctly classifying an assertion (*M3*) or selecting the answer "I don't know" rather than giving an incorrect answer (*P2*, *S2*, *S3*).

Figure 4 provides more insight into the participants' performance by showing the number of participants giving the same number of correct answers (which range from 0 to 6). As it can be seen, 10 out of 33 students were not able to correctly classify a single assertion. This was not the case for professional developers, as each of them was able to correctly assess from at least 1 to up to 4 assertions. The highest performance of 5 and 6 correct answers was exhibited by just one student.

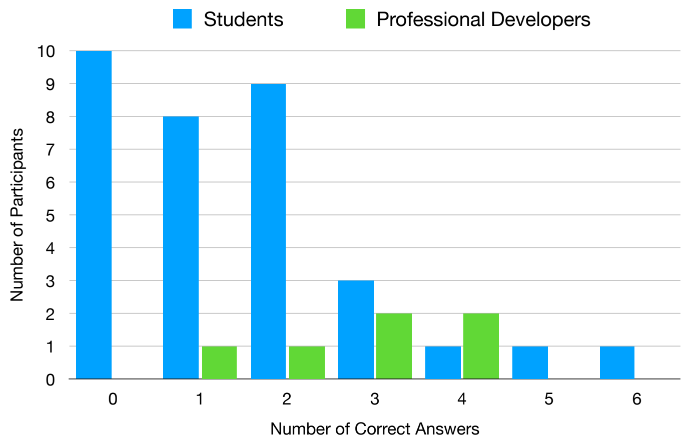

Fig. 4: Number of Participants grouped by Number of Correct Answers

Overall, for 39 participants the average correct classification ratio is only 29%, see Table 6. There are no assertions that were incorrectly or correctly classified by all participants. In 22% of cases the option "I don't know" was picked and in 49% the provided classification was wrong.

We tested the statistical significance of our results. According to the Pearson-Klopper method for calculating binomial confidence intervals (at 95% confidence level), for students the correct classification rate is in the range [0.1936:0.2190] with mean 0.2525; for professional developers in the range [0.3040:0.6451] with mean 0.472; and for all participants in the range [0.2293:0.3488] with mean 0.2863. The difference between students' and professional developers' performance is statistically significant according to Fisher's exact test (two-sided) with $p = 0.01488$ at 95% confidence level. We conclude that there is inferential statistical evidence that the professional developers were significantly better at oracle assessment than students.

Parameters affecting user effectiveness. In the background questionnaire we asked participants questions about their programming experience, their assessment on the understandability of training material and their satisfaction with the time provided for the task. To analyse whether any of these factors affected subjects' effectiveness, we calculated the ratios of correct, incorrect and "I don't know" answers within each group corresponding to different parameter values. The first/second columns in Table 7 show the parameter values and the number of overall responses within each group, while the next columns list the oracle assessment answers. Column *Conf. Int.* shows confidence intervals (at 95% confidence level) for each response.

As the table shows, the rate of correct answers increases when we switch from the group with "< 1 year" to the group with "$1-3$ years" of programming experience. However, this increase does not continue for the group with "> 3 years" of programming experience. A similar pattern holds for Java and Industry Experience. Regarding the time provided for the task, the number of responses are equal for both groups, but the ratio of correct answers is higher when the answer was "yes". The user effectiveness also increases as the understandability of the provided training material

TABLE 4: Results: Students (SS1 - 1st session, 20 students; SS3 - 3rd session, 13 students)

Assertion	Deficiency	All		Correct		Incorrect		Don't Know	
		SS1	SS3	SS1	SS3	SS1	SS3	SS1	SS3
M1	FP	13	8	2 (15%)	1 (13%)	10 (77%)	7 (88%)	1 (8%)	0 (0%)
M2	FN	13	8	7 (54%)	5 (63%)	3 (23%)	1 (13%)	3 (23%)	2 (25%)
M3	None	13	8	6 (46%)	3 (38%)	5 (38%)	1 (13%)	2 (15%)	4 (50%)
P1	FN	13	9	1 (8%)	2 (22%)	9 (69%)	5 (56%)	3 (23%)	2 (22%)
P2	FP	13	9	2 (15%)	1 (11%)	5 (38%)	4 (44%)	6 (46%)	4 (44%)
P3	FN	13	9	1 (8%)	1 (11%)	8 (61%)	5 (56%)	4 (31%)	3 (33%)
S1	FN	14	9	3 (21%)	1 (11%)	8 (57%)	8 (89%)	3 (21%)	0 (0%)
S2	None	14	9	5 (36%)	3 (33%)	9 (64%)	5 (56%)	0 (0%)	1 (11%)
S3	FP	14	9	3(21%)	3 (33%)	8 (57%)	5 (56%)	3 (21%)	1 (11%)
		120	78	30 (25%)	20 (26%)	65 (54%)	41 (52%)	25 (21%)	17 (22%)
		198		**50 (25%)**		**106 (53%)**		**42 (21%)**	

TABLE 5: Results: Professional Developers (2nd session, 6 developers)

Assertion	All	Correct	Incorrect	Don't Know
M1	4	1 (25%)	3 (75%)	0 (0%)
M2	4	2 (50%)	1 (25%)	1 (25%)
M3	4	4 (100%)	0 (0%)	0 (0%)
P1	5	3 (60%)	1 (20%)	1 (20%)
P2	5	1 (20%)	0 (0%)	4 (80%)
P3	5	2 (40%)	2 (40%)	1 (20%)
S1	3	0 (0%)	2 (67%)	1 (33%)
S2	3	2 (67%)	0 (0%)	1 (33%)
S3	3	2 (67%)	0 (0%)	1 (33%)
	36	**17 (48%)**	**9 (25%)**	**10 (27%)**

TABLE 6: Results: Overall (39 participants)

Assertion	All	Correct	Incorrect	Don't Know
M1	25	4 (16%)	20 (80%)	1 (4%)
M2	25	14 (56%)	5 (20%)	6 (24%)
M3	25	13 (52%)	6 (24%)	6 (24%)
P1	27	6 (22%)	15 (56%)	6 (22%)
P2	27	4 (15%)	9 (33%)	14 (52%)
P3	27	4 (15%)	15 (56%)	8 (30%)
S1	26	4 (15%)	18 (69%)	4 (15%)
S2	26	10 (38%)	14 (54%)	2 (8%)
S3	26	8 (31%)	13 (50%)	5 (19%)
	234	**67 (29%)**	**115 (49%)**	**52 (22%)**

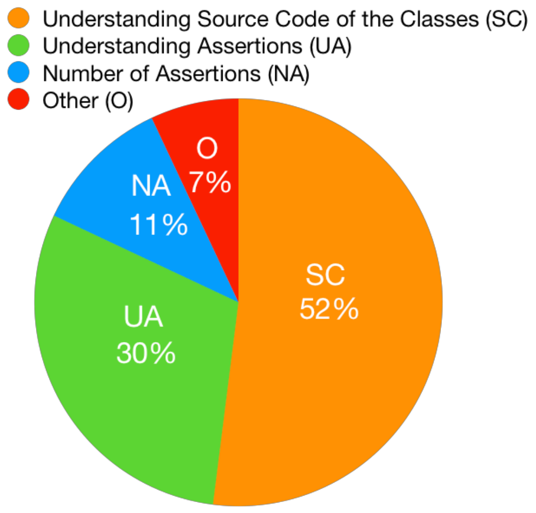

Fig. 5: What was the main challenge while performing the task?

increases (according to participants). However, even when participants think that the time allocated for the task was enough, their average effectiveness is only 32%. Similarly, when they rate the provided training material with the highest possible mark, the average effectiveness is still only 35%.

We calculated the Pearson correlation coefficient between the ratio of correct answers and each of the factors in Table 7. The correlation coefficients are positive for all factors except Java Experience. Industry Experience is the factor with the highest correlation rate and the only one where correlation is statistically significant ($p \leq 0.05$). Even for this factor, the correlation is *moderate*, not *strong*. The permutation test for the analysis of co-factors gives similar results.

To get participants' opinion on the difficulties associated with the task, we asked them a multiple-choice question "What was the main challenge while performing the task?". We got responses from all 39 participants with 54 answers selected. As Figure 5 shows, the main challenge for participants was to understand the source code of the classes, followed by understanding the assertions.

Agreement rate between participants. To analyse how much homogeneity there is between the classifications provided by users, we measured the degree of inter-rater agreement. Fleiss' kappa [8] is the most common statistical measure for assessing the reliability of agreement between a fixed number of raters when classifying items. It calculates the degree of agreement in classification over the one that would be obtained by chance. However, as we have overall 9 assertions and each participant classified only a subset (6) of them, Fleiss' kappa is not applicable to our data. Hence, we instead used Krippendorff's alpha [23] coefficient, which generalizes Fleiss' kappa to incomplete (missing) data. Krippendorff's alpha takes value between 0 and 1, where 0 is perfect disagreement and 1 is perfect agreement. When it is less than 0 disagreements are systematic and exceed what can be expected by chance.

Table 8 shows the number of raters and Krippendorff's alpha value for each subject group and for all subjects (i.e., students, professionals and all participants). The highest agreement rate is for the assertions in class *Matrix*, for professionals. According to Landis and Koch's [24] interpretation of agreement rate values, professionals have reached a *fair* agreement. This is related to the fact that all professionals have classified one of the assertions (*M3*) in

TABLE 7: Results for different parameter values

	All	Correct	Incorrect	Don't Know	Conf. Int.	Pearson Correlation		Co-Factor Analysis	
						Coeff.	p-value	Coeff.	p-value
Progr. Exp. (37)									
<1 year (7)	42	8(19%)	30(71%)	4(10%)	[0.09:0.34]	0.0728	0.6642	2.2370	0.9220
1 - 3 years (13)	78	27(35%)	31(40%)	20(26%)	[0.24:0.46]				
>3 years (17)	102	26(25%)	49(48%)	27(26%)	[0.17:0.35]				
Java Exp. (37)									
None (3)	18	5(28%)	9(50%)	4(22%)	[0.10:0.53]	-0.0310	0.8649	-3.830	0.6860
<1 year (12)	72	23(32%)	35(49%)	14(19%)	[0.21:0.44]				
1-3 years (16)	96	22(23%)	48(50%)	26(27%)	[0.15:0.33]				
>3 years (6)	36	11(31%)	18(50%)	7(19%)	[0.16:0.48]				
Industry Exp.(36)									
None (19)	114	18(16%)	62(54%)	34(30%))	[0.10:0.24]	0.4126	0.0112	10.4270	0.0190
<1 year (9)	54	20(37%)	26(48%)	8(15%)	[0.24:0.51]				
1-3 years (5)	30	15(50%)	11(37%)	4(13%)	[0.31:0.69]				
>3 years (3)	18	6(33%)	8(44%)	4(22%)	[0.13:0.59]				
Enough Time (36)									
No (18)	108	26(24%)	57(53%)	25(23%)	[0.16:0.33]	0.2001	0.2334	12.770	0.4900
Yes (18)	108	35(32%)	49(45%)	24(22%)	[0.24:0.42]				
Training (38)									
1 (1)	6	0(0%)	5(83%)	1(17%)	[0.00:0.46]	0.2681	0.0989	4.2590	0.6670
2 (3)	18	3(17%)	4(22%)	11(61%)	[0.04:0.31]				
3 (12)	72	20(28%)	39(54%)	13(18%)	[0.18:0.40]				
4 (12)	72	19(26%)	36(50%)	17(24%)	[0.17:0.38]				
5 (10)	60	21(35%)	29(48%)	10(17%)	[0.23:0.48]				

this class correctly, therefore fully agreeing. The agreement rate for class *StackAr* between professionals and also for all participants is negative (*poor*). In all the other cases, there is a *slight* agreement between students, professionals and all participants.

TABLE 8: Agreement rate between participants

	Students		Professionals		All	
	#	Alpha	#	Alpha	#	Alpha
Matrix	21	0.124	4	0.324	25	0.091
PolyFunction	22	0.006	5	0.042	27	0.011
Stack	23	0.005	3	-0.102	26	-0.006
	33	0.010	6	0.015	39	0.049

Overall, these low agreement rate values show that although all subjects, even those with industry experience, find oracle classification hard, there is no evidence for systematic bias nor consistent misunderstanding among subjects regarding their incorrect oracle inferences. For example, it is never the case that participants consistently agree on classifying an assertion which actually has a false positive as an assertion with a false negative.

RQ1 (effectiveness): *Our experiments show that subjects can only achieve a poor correct classification rate (29%) when assessing whether an assertion contains a false positive, a false negative or none of the two. Professional developers achieve a significantly higher correctness rate (48%) than students (25%), but still such correctness rate is largely below the desirable value (100%). The inter-rater agreement was also quite poor, confirming that the oracle assessment is indeed a difficult task for humans. We observed a moderately strong evidence that industrial experience is correlated with correct classification rate, but found no such evidence of any other correlations.*

RQ2: Misclassifications

Harder to Detect Oracle Deficiencies. To investigate which type of oracle deficiencies is harder to detect for developers, we summarized the results of the oracle assessment task for each type of oracle deficiency and participant group (see Figure 6). As the figure reveals, both students and professional developers are more successful in detecting false negatives than false positives (27% vs. 21% overall). However, the best result is achieved for assertions with no oracle deficiencies at all. For these assertions, professionals were able to provide correct classifications in 87% of the cases.

We asked the question "Which oracle deficiency is harder to detect?" to the participants in the exit questionnaire. As Figure 7 shows, the number of people finding false positives harder to detect than false negatives is slightly higher, which is inline with our results. However, to check whether the response of participants considering false negatives harder than false positives is consistent with the actual results we observed for their performance, we calculated correct classification rates for false positives and false negatives by both the ”FP is harder” and ”FN is harder” groups. The results show that the ”FN is harder” group is more successful in detecting false positives (29%) than false negatives (19%). Similarly, the ”FP is harder” group shows better results for assertions with false negatives (31%) than for the ones with false positives (18%). Therefore, the participants' intuition about the difficulty of each oracle deficiency type is confirmed by the results observed for each group of deficiency.

Misclassification types. To analyse the type of mistakes participants made when assessing oracles, we calculated how often each of the 6 possible misclassifications has occurred. Column *Class-Misclass* in Table 9 lists these mis-

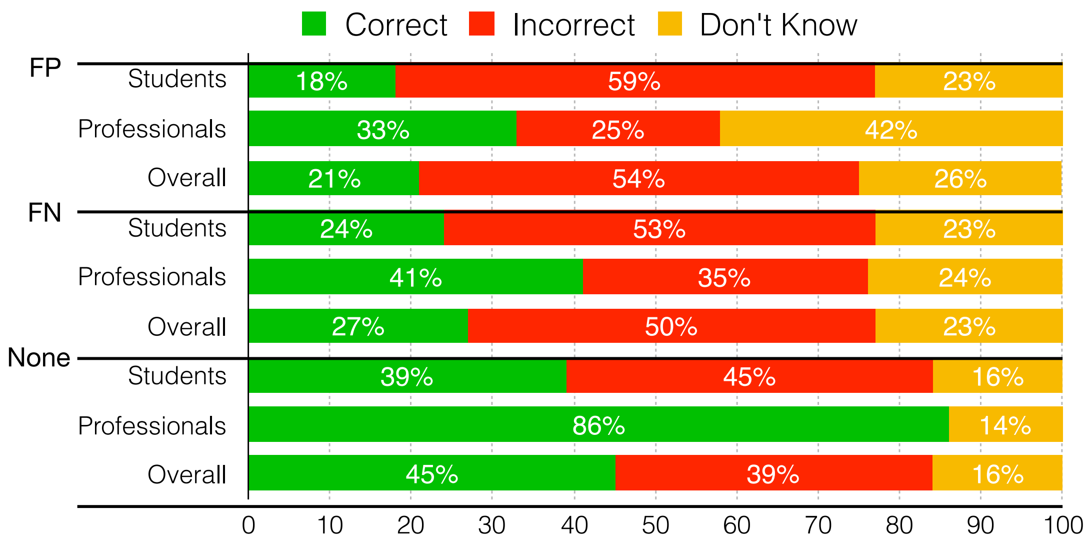

Fig. 6: Correctness rate divided by FP, FN and None

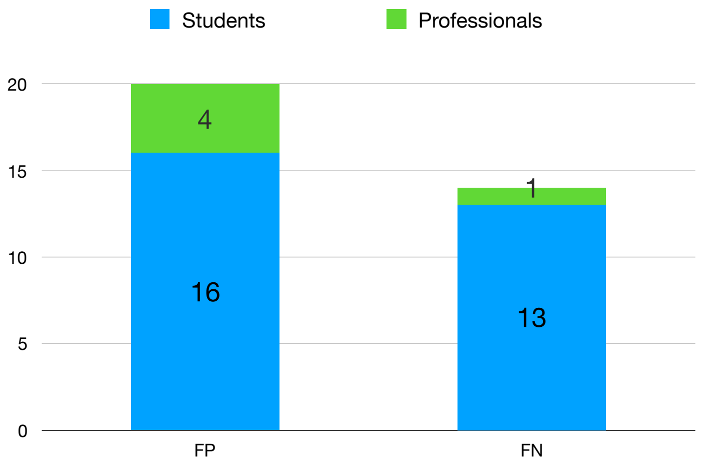

Fig. 7: Which Oracle Deficiency is harder to detect?

classifications, where the notation *OD1-OD2* means that the assertion has an oracle deficiency of type *OD1*, but was classified as having *OD2*. Columns *Students*, *Professionals* and *All* show the rate of each misclassification for the corresponding participant group. These rates were calculated by dividing the number of times the misclassification *OD1-OD2* took place by the overall number of assertions with *OD1*.

TABLE 9: Misclassifications

Class-Misclass	Students	Professionals	All
FP-FN	29%	25%	28%
FP-None	30%	0%	26%
FN-FP	17%	18%	17%
FN-None	36%	18%	33%
None-FP	25%	0%	22%
None-FN	20%	0%	18%

As the Table 9 shows, students have made each possible misclassification. In contrast, for professional developers three out of six possible erroneous classifications never took place. The ratio of each misclassification is higher for students than for developers, except FN-FP, for which the difference is negligible. Students misclassify false positives as false negatives or "None" at very close ratios (29% vs. 30%), while for professionals such difference is more perceptible (25% vs. 0%). Despite the fact that false positives are being misclassified more often, the most common error for all participants is FN-None. This shows that users often fail to recognise the bugs that the assertion can miss, and therefore tend to classify weak assertions as strong. One of the least prevalent misclassifications is None-FN, showing that strong assertions are classified as weak more rarely.

RQ2 (misclassifications): *False positives were perceived (and were actually found) to be the hardest category to identify for all subjects. The most common misclassification consists of weak assertions regarded as free of deficiencies, showing that identifying faults potentially missed by an assertion is a quite difficult task for humans.*

3.2 Oracle Improvement Study

Once developers are aware of assertion deficiencies, they must improve the assertion so as to remove deficiencies. To support developers in this process, our tool automatically generates counterexamples that demonstrate the reason for each type of oracle deficiency. To check whether this leads to a more effective oracle improvement process, we conducted a study to compare the improvement process when using our tool against manual improvement unaided by our tool. We addressed the following research questions:
RQ3: What is the quality of assertions improved using our tool compared to assertions improved manually?
RQ4: When using our tool, how many iterations and how much human effort does the iterative improvement process require to remove all oracle deficiencies in the assertion?

3.2.1 Participants

In our approach, the developer is an integral part of the oracle improvement process. To analyse how beneficial is

TABLE 10: Improvement Study: Participants

Participant	Group	Experience	# of Jobs	Amount	Job Title
P1	Without Tool	8 years	7	1000+ USD	C, C++, Java Developer
P2	Without Tool	3 years	0	0 USD	Software Quality Assurance Analyst
P3	Without Tool	3 years	18	1000+ USD	Software Tester
P4	Without Tool	3 years	4	3000+ USD	Full Stack Software Engineer
P5	Without Tool	5 years	0	0 USD	Expert in Automation QA
P6	With Tool	3 years	0	0 USD	Software Quality Assurance Engineer
P7	With Tool	1 year	2	15 USD	Test Automation Engineer
P8	With Tool	3 years	1	40 USD	Test Manager (Manual,Automation,Security)
P9	With Tool	6 years	0	0 USD	QA Automation Engineer
P10	With Tool	5 years	0	0 USD	Full Stack Java Enterprise Developer

the use of our tool for developers with various backgrounds, two different groups of participants were involved in our experiments. We recruited the participants for the first group by sending personal email invitations to 28 PhD students from *Fondazione Bruno Kessler* and to 19 PhD students and 2 postdoctoral researchers from *University College London*. No financial incentive was offered in this invitation. Overall, 17 PhD students and 2 postdoctoral researchers agreed to participate.

Our second group of participants were developers from *Upwork*. Upwork is a global freelancing platform where businesses and independent professionals collaborate remotely. To hire developers on this platform, we registered there as a *client*, by filling in necessary details and then adding and verifying the payment method. After registration, we posted two different fixed-price jobs: 1) without using the tool, with a payment of 20 USD; 2) using the tool, with a payment of 30 USD. The difference in the price is due to the training on how to use our tool job, an extra activity that is carried out only for the second job. For both jobs we required candidates to pass a qualification test. Overall, we received 20 job proposals for the first and 12 job proposals for the second job. We aimed to have five freelancers completing each job. To reach this quota we had to hire 15 freelancers overall: four of them did not pass the qualification test and one did not submit the last part of the task.

Participants for each job were selected so that there is a balance in terms of experience between control and treatment groups on average. Table 10 lists our final list of participants from the Upwork platform. Column *Group* shows whether each participant worked on a task with or without the tool. Column *# of Jobs* shows the number of jobs each freelancer did on the Upwork platform and Column *Amount* shows how much money each freelancer has earned overall.

We had limited control on the group composition (we could just approximately balance the level of *Experience*). In fact, it turned out that the group *Without Tool* includes participants with slightly higher *# of Jobs* and *Amount*, possibly giving a slight unfair advantage to this group of subjects. We deemed this possible bias acceptable since it reduces the chance of Type I errors (incorrectly inferring that our tool provides benefits to its users).

3.2.2 Experimental Procedure

The main structure of our experimental procedure is shown in Figure 8. The PhD student/Postdoc sessions were organised individually for each participant as a single 1.5 - 2 hour session. In Upwork we divided our experimental session into *milestones*, i.e., subtasks with separate budgets and deliverables. Each participant had to pass each milestone to be able to proceed with the next one. The green bars in Figure 8 show the content and the payment offered for each milestone.

Each experimental session started with a 30-minute *Oracle Improvement Training*, which contained all the information from the *Oracle Assessment Study* training material, with the addition of multiple examples on how to improve the assertions to remove oracle deficiencies. For the participants from Upwork, this material was provided in written form, while for the PhD student/Postdoc sessions it was delivered in the form of a presentation.

The training was followed by an *Oracle Improvement Practice Task*, where participants were provided with 4 simple Java methods with an initial assertion each. The objective of the task for the participants was to improve the assertions so that they have no false positives and no false negatives. The aim of the task was to ensure that participants understand the oracle improvement process. In the Upwork setting, participants submitted their improved assertions online. In case any of the four assertions still had oracle deficiencies left, the written feedback explaining the reason for the oracle deficiency was sent to them. Participants could resubmit based on the feedback provided. In case the participant was not able to finish the improvement process after two iterations of feedback, her/his participation in the experiment was terminated. This was the case for 4 participants out of 15. In the PhD student/Postdoc sessions, this part was conducted in a more interactive way, where participants could write the improved assertion and receive immediate feedback, possibly followed by a discussion, and could subsequently improve the assertion until all deficiencies were removed. All participants from PhD student/Postdoc sessions have passed the practice task.

The *Tool Training* was conducted only with participants from the treatment (*With Tool*) group. The training material was provided in written form to participants from Upwork and in the form of a presentation to the others. The training included information on: (1) how to run the tool; (2) the output of the tool for False Positives; (3) the output of the

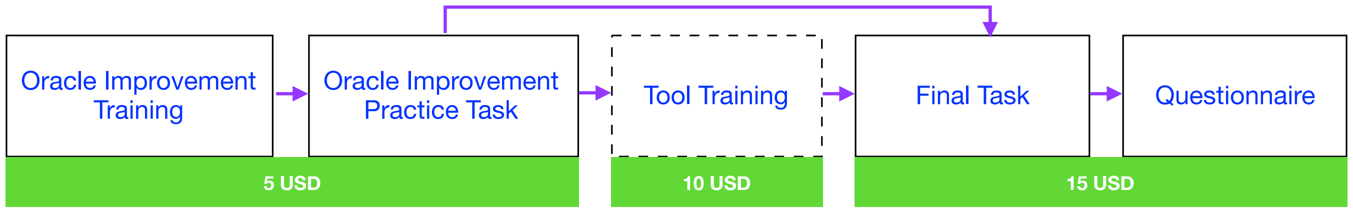

Fig. 8: Oracle Improvement Study: Experimental Procedure

tool for False Negatives, including the explanation of each mutation operator that could be applied to the source code.

To give a hands-on experience on the use of the tool, participants ran the tool and analysed its output for the methods from the *Oracle Improvement Task*. We reused these methods to ensure that participants performing the task with the tool did not get more examples and experience of oracle improvement than participants not using the tool. We provided a machine with pre-installed tool to the participants in the PhD student/Postdoc sessions.

We provided instructions on where to download the tool and how to run it on their machine to participants from Upwork. Participants from Upwork were also required to submit a written description of the output produced by the tool for each method, to check that they could understand it properly. For False Positives they had to explain why the generated test case makes the assertion fail. For False Negatives they had to describe the applied mutations and why the assertion does not react to them. Examples of such descriptions were provided in the training material.

After participants received all the necessary training, they proceeded with the *Final Task*. In this task they were provided with a single Java class *StackAr* which had an assertion with a false positive in the `top` method and an assertion with a false negative in the `pop` method. The objective of the task was to improve both assertions so that they have no oracle deficiencies. The aim of the task was to compare the outcome of the oracle improvement process when participants use the tool and when they do not. Participants from both groups knew the type of oracle deficiency each assertion has.

The control group was instructed to improve the assertions manually. The treatment group had the tool to guide them: for each improvement step they could run the tool and if an oracle deficiency was detected, based on the test cases reported as an evidence they could decide on the next improvement step. The stopping point for the participants from the treatment group was when the tool reported no oracle deficiencies, while for the control group it was only the participant's own confidence in the final assertions. In the Upwork experiments we offered a bonus of 5 USD to participants from the control group who were able to submit assertions with no oracle deficiencies.

Once the task was over, participants were asked to submit their final assertions along with the information about their background, as well as their assessment of the experimental session through the exit questionnaire.

3.2.3 Oracle Improvement Process: Possible Cases

In our experiments, the initial assertions have either a false positive or a false negative. After getting the report of the tool for the initial oracle deficiency, the developer has to decide on the improvement step to take. Depending on this improvement step, the new assertion can, in the best case, be *Fully Correct* (no oracle deficiencies), can have an oracle deficiency (of the same or new type) or can lead to a *Crash* in the program. Figure 9 shows all the possible state changes for the assertion during the improvement process.

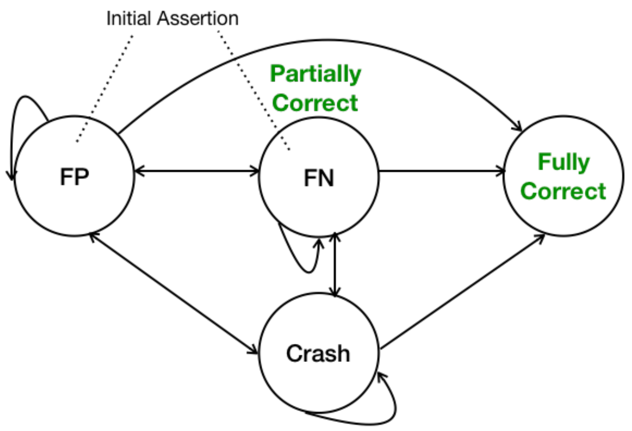

Fig. 9: Oracle Improvement Process: Possible Outcomes

The example in Figure 10 (top) shows a shortened version of the class *StackAr* that we have used in our study. In method `pop` there is an initial assertion which has a False Negative. The bottom part of Figure 10 shows assertions that were produced by different developers as an improvement to the initial one after one iteration of improvement process.

The first assertion in Figure 10 (bottom) causes a *Crash*. It can lead to *ArrayIndexOutOfBoundsException*, if the value of variable `topOfStack` is equal to -1 when the assertion gets executed. Assertions are part of source code, but they should not cause any side effects. Therefore it is unacceptable that they lead to an exception during program execution. When our tool performs False Positive detection, if during the search process any test case causes an exception, such that the error stack trace for this exception contains the line number of the assertion in the code, the test case gets reported to the developer as an evidence of *Crash*.

The second assertion shows how an attempt to fix a False Negative can lead to the introduction of a False Positive. This assertion claims that the value of `topOfStack` has been incremented, while in fact it was decremented. This makes the assertion fail any time it gets executed.

```
public class StackAr {
 private Object[] theArray;
 private int topOfStack;

 public StackAr(int capacity) {
    theArray = new Object[capacity];
    topOfStack = -1;
 }

 public void pop() throws UnderflowException {
   //instrumentation
   int old_topOfStack = topOfStack;
   //instrumentation
   Object[] old_theArray =
         Arrays.copyOf(theArray, theArray.length);

   if (topOfStack == -1)
     throw new UnderflowException();

   theArray[topOfStack] = null;
   topOfStack = topOfStack - 1;

   assert (theArray[topOfStack + 1] == null);
  }
}
```

```
(1) assert (theArray[topOfStack] == null);
(2) assert (theArray[topOfStack + 1] == null &&
          topOfStack - 1 == old_topOfStack);
(3) assert (theArray[topOfStack + 1] == null  &&
          old_topOfStack - 1 == topOfStack);
(4) assert  (theArray[topOfStack + 1] == null  &&
          old_topOfStack - 1 == topOfStack  &&
          validateArray(theArray, old_theArray,
              old_topOfStack))
```

Fig. 10: Class StackAr: method pop

The third assertion shows an example of a correct improvement step. The initial assertion checked the property stating that the value of `theArray` at index `topOfStack` is equal to null. The improved assertion adds an additional check stating that the value of `topOfStack` was changed correctly, i.e., it was decremented by one. This assertion is stronger than the initial one, but it still has a False Negative. The *Fully Correct* assertion for method `pop` would be assertion number 4 in Figure 10 (bottom). Along with the previous checks, it also ensures that method `validateArray` returns true. In turn, method `validateArray` loops through the array and checks whether all the elements in `theArray` and `old_theArray`, except the one at index `topOfStack + 1`, are equal. Therefore, to ensure *Fully Correctness* in this case one should check that the method has changed correctly the part of the stack state it was supposed to change (i.e., the values of `topOfStack` and `theArray[ind]`, with `ind = old_topOfStack`) and that it has not affected the rest of the state (i.e., the values of elements in `theArray[ind]`, except for index `ind = old_topOfStack`).

Generating fully correct oracles might be an expensive and difficult process. Therefore, *Partially Correct* assertions as the initial one in method `pop` or the third one in Figure 10 (bottom) might be regarded as sufficiently adequate in practice. In fact, a complete specification of the state changes that a method should perform might provide, in practice, a powerful enough method to catch most incorrect implementations, even if such assertion is only partially correct, by not ruling out method implementations that operate state changes on the part of the state that is supposed to be untouched by the operation implemented by the method.

In our approach the level of partial correctness can be quantified as the mutation score of the assertion: a higher mutation score indicates that the assertion is capable of ruling out a higher number of incorrect state changes performed by buggy implementations (mutants), possibly including state changes that affect the supposedly unchanged substate.

3.2.4 Results

Quality of Final Assertions

Table 11 shows the results for the participants who improved the assertions manually. Column *Overall T.* shows the overall time spent on improving each assertion, as reported by each participant. Column *Outcome* shows the oracle deficiency or the level of correctness the final assertion has reached, where the distinction among FN, Partially Correct and Fully Correct is that an assertion labelled FN has some initial mutation score m; an assertion labelled Partially Correct has mutation score $> m$ and < 1; an assertion labelled Fully Correct has mutation score = 1 (assuming in all three cases that there is no residual false positive, which would cause otherwise the labelling FP).

The results presented in Table 11 show that only five out of nine participants in the PhD student/Postdoc sessions achieved full correctness for Assertion 1. The assertions submitted by the remaining four participants either still have a false positive or cause a crash in the program. None of the participants was able to improve Assertion 2 to the point of full correctness, but five out of nine participants have achieved partial correctness. The participants from Upwork (UP1-UP5) performed worse for Assertion 1 and better for Assertion 2 in comparison to the participants from PhD student/Postdoc sessions. For the first assertion, only one participant achieved full correctness. For the second assertion four participants submitted partially correct assertions and no one submitted a fully correct one.

Table 12 shows the results for the participants who used our tool to improve the assertions. Here, column *Overall T.* comprises the running time of the tool, reported in column *Tool T.*, and the time the developer spent on analysing the output of the tool and improving assertions, i.e., the human cost, reported in column *Human T.* Every time the participant ran our tool, we recorded the time of the day and the assertions in the code. Based on this information, we calculated the human cost as the sum of time intervals between tool runs and the running time of the tool as the sum of tool run durations for all iterations.

When using the tool, all the participants from both PhD student/Postdoc sessions and Upwork sessions have achieved full correctness for Assertion 1. As our PhD student/Postdoc experimental sessions were limited in time, initially we configured the tool so that it reports false negatives for Assertion 2 only until partial correctness was achieved (as in the third assertion in Figure 10). Five participants (marked with an asterisk in Table 12) have run the tool with this configuration. As they achieved the desired partial correctness in a relatively short time, we used the standard

TABLE 11: Improvement Study: Results Without Tool

Participant	**Assertion1**		**Assertion2**	
	Outcome	Overall T.	Outcome	Overall T.
P1	Crash	45:00	Partially Correct	30:00
P2	Fully Correct	5:00	FP	10:00
P3	FP	5:00	FP	10:00
P4	Crash	25:00	Partially Correct	10:00
P5	Fully Correct	2:00	Partially Correct	4:00
P6	FP	6:00	Partially Correct	6:00
P7	Fully Correct	10:00	FN	7:00
P8	Fully Correct	16:00	FP	10:00
P9	Fully Correct	7:00	Partially Correct	2:00
UP1	Partially Correct	20:00	Partially Correct	20:00
UP2	Fully Correct	45:00	FN	40:00
UP3	FN	45:00	Partially Correct + FP	30:00
UP4	Partially Correct	17:00	Partially Correct	18:00
UP5	Partially Correct	25:00	Partially Correct + FP	35:00
	21% Partially Correct 43% Fully Correct	18:12	64% Partially Correct	15:28

TABLE 12: Improvement Study: Results With Tool

Participant	**Assertion1**				**Assertion2**			
	Outcome	Overall T.	Tool T.	Human T.	Outcome	Overall T.	Tool T.	Human T.
P10*	Fully Correct	19:02	03:53	15:09	Partially Correct	14:07	07:48	06:19
P11*	Fully Correct	18:01	05:22	12:39	Partially Correct	24:06	14:16	09:50
P12*	Fully Correct	21:11	03:52	17:19	Partially Correct	13:59	07:47	06:12
P13*	Fully Correct	16:37	10:26	06:11	Partially Correct	10:56	07:52	03:04
P14*	Fully Correct	10:27	06:03	04:24	Partially Correct	20:03	11:27	08:36
P15	Fully Correct	12:59	06:41	06:18	Partially Correct + FP	52:18	15:04	37:14
P16	Fully Correct	19:51	10:49	09:02	Partially Correct + FP	44:06	19:16	24:50
P17	Fully Correct	12:20	07:10	05:10	Partially Correct + FP	47:44	28:08	19:36
P18	Fully Correct	12:44	06:03	06:41	Fully Correct	40:40	15:55	24:45
P19	Fully Correct	47:38	14:15	33:23	Partially Correct	34:43	16:17	18:26
UP6	Fully Correct	13:46	07:48	05:58	Fully Correct	22:14	10:48	11:26
UP7	Fully Correct	15:17	09:15	06:02	Partially Correct	31:38	10:47	20:51
UP8	Fully Correct	08:24	04:57	03:27	Fully Correct	22:16	10:05	12:11
UP9	Fully Correct	09:25	05:34	03:51	Fully Correct	06:28	03:57	02:31
UP10	Fully Correct	16:36	08:53	07:43	Fully Correct	28:20	11:16	17:04
	100% Fully Correct	16:57	07:24	09:33	33% Fully Correct 67% Partially Correct	27:33	12:42	14:51

configuration of the tool reporting all false negatives for the rest of the participants. As a result, these participants received a false negative report after achieving partial correctness. However, only one of them was able to improve the assertion to the point of full correctness. Three participants (P15, P16, P17) understood the reason of the reported false negative and made steps towards improvement, but the added checks contained a false positive which they were not able to remove by the end of experimental session. Participant P19 was not able to understand the reason for the reported false negative, and, therefore did not improve the assertion beyond the point of partial correctness. The same scenario occurred also for Upwork Participant UP9. The rest of Upwork participants (four out of five) were successful in achieving full correctness.

In Tables 11 and 12 we do not indicate explicitly the level of partial correctness (i.e., the mutation score), because it is the same across all participants: Partial Correctness for Assertion 1 has mutation score = 75%, while for Assertion 2 it is 92%.

Overall, for Assertion 1, participants achieved 43% of full and 21% of partial correctness when improving assertions manually versus 100% of full correctness when improving assertions using our tool. For Assertion 2 manual improvement led to 64% of partial correctness as opposed to 33% of full and 67% of partial correctness when using the tool. We checked the statistical significance of the difference between the manual and tool-supported improvement by applying the Fisher's exact test (two-sided) in two different configurations. In the first configuration we compared the outcomes of assertions in terms of achieving partial correctness and in the second in terms of achieving full correctness. In both cases the difference is statistically significant at 95% confidence level, with $p = 0.00025$ in the first configuration and $p = 0.00067$ in the second.

We conducted a co-factor analysis to check if the type of participants (whether they are from Upwork or PhD student/Postdoc sessions) is significantly affecting their performance. Another co-factor here is whether the tool was used or not, while mutation score is the dependent variable. The permutation test shows that the effect of participant type is not statistically significant with $p = 0.33333$, but the effect of tool usage is statistically significant with $p < 2 * 10^{-16}$.

RQ3 (Quality of Final Assertions): *The tool helped developers produce assertions with higher quality. Participants who used the tool were able to achieve 67% of full and 33% of partial correctness, while participants without tool achieved only 21% of full and 43% of partial correctness. The difference is statistically significant.*

Manual Improvement Process

To understand the approach of developers when improving the assertions manually we asked them what was their strategy for the detection and improvement of false positives and false negatives. Figure 11 shows the participants' answers to the multiple-choice questions.

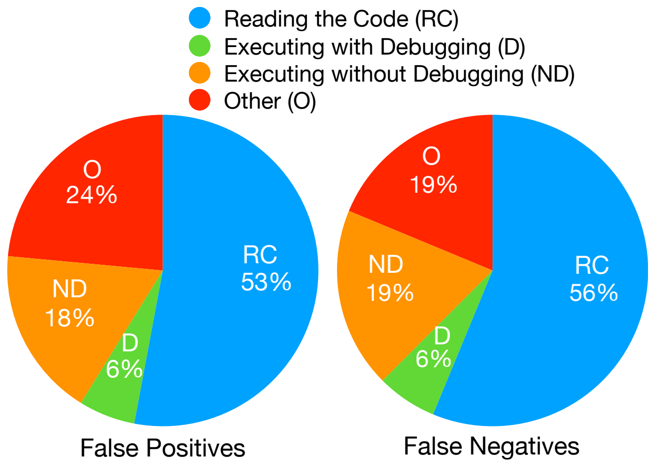

Fig. 11: Strategies to Manually Detect FP and FN

Around 55% of participants just read the code when performing manual improvement, while around 40% also executed the code with or without debugging enabled. In case the option "Other" was picked, the participant could describe his/her strategy using the textbox provided in the questionnaire. Only three participants have provided meaningful descriptions of their strategies. The first participant noted that he ran the method with the improved assertion distinguishing "empty case, one element, more than one element". The second participant described his strategy as "reading the commented description of what the function should do" and then encoding his interpretation of the expected behaviour in a boolean formula using the "old" and updated variables. The description of the third participant was "to try to study instances that will make the assertion fail", define what the assertion for this method should look like and then compare it with the current one and improve it.

Overall, the participants' approach to oracle improvement seems strongly based on static inspection of code and documentation. Dynamic analysis, tracing and debugging are not widely used. We conjecture the following reasons for such strategies: (1) the definition of assertions might be perceived as a coding/documentation activity; (2) debugging and fixing bugs in assertions is not as common as debugging and bug fixing in the code. There seems to be no standard practice for handling issues that affect assertions – hence, the need for a well-established approach and for supporting tools.

Iterative Improvement Process

Analysis of the time required to complete the iterative improvement process (see Tables 11, 12) is quite problematic, because we had to measure time differently in the different settings of the experiments. Specifically, the PhD student/Postdoc group without tool marked time in a paper sheet in a strictly controlled classroom setting, so their reported time is quite reliable.

On the contrary, Upwork participants self reported the time spent to improve the assertions without tool in an uncontrolled environment. They might have inflated times a bit to justify their remuneration and they might have been quite approximate in their time measurement. Time values measured for both groups when using the tool were obtained in a completely different way, since these values have been extracted from the tool execution logs. This means that they are very accurate, but also quite different from the times that humans self-report. Because of such differences, we can make only limited claims on time.

Overall, we observe that the order of magnitude is the same. In fact, the overall average time ranges between 15:28 and 27:33, considering both groups and treatments, with two intermediate values at 16:57 and 18:12. This indicates that the introduction of the tool can be extremely beneficial to the assertion quality (as shown in previous section) without having any remarkable impact on the time developers take to complete the improvement process. We can also notice that the human time (Column *Human T.*) when the tool is used (see Table 12), tends to be lower than the human overall time when no tool is available (see Table 11). It is only when the tool time (Column *Tool T.*) is added that we get comparable times to the setting without tool. These findings indicate that the tool execution time has a significant impact on the improvement process and that any performance improvement that could be achieved on the tool speed (the tool is a research prototype and was not optimized for performance) could directly benefit the overall iterative improvement time experienced by the tool users.

Figure 12 shows the overall number of iterations and the outcome of each iteration for both assertions and for all 15 participants who used the tool in the oracle improvement process. For the first assertion the number of iterations varied from 1 to 8 and the average number of iterations required to achieve full correctness was 2.93. For the second assertion the number of iterations varied from 1 to 13 and the average number of iterations was equal to 4.66. The average number of iterations participants went through to achieve full correctness was 3.8, while for partial correctness it was 3.66. Since these two numbers are approximately the same, we can conjecture that participants who were able to achieve full correctness performed bigger improvement steps, since they achieved higher quality in approximately the same number of iterations. At each iteration developers spent on average 195 seconds for the analysis of tools'

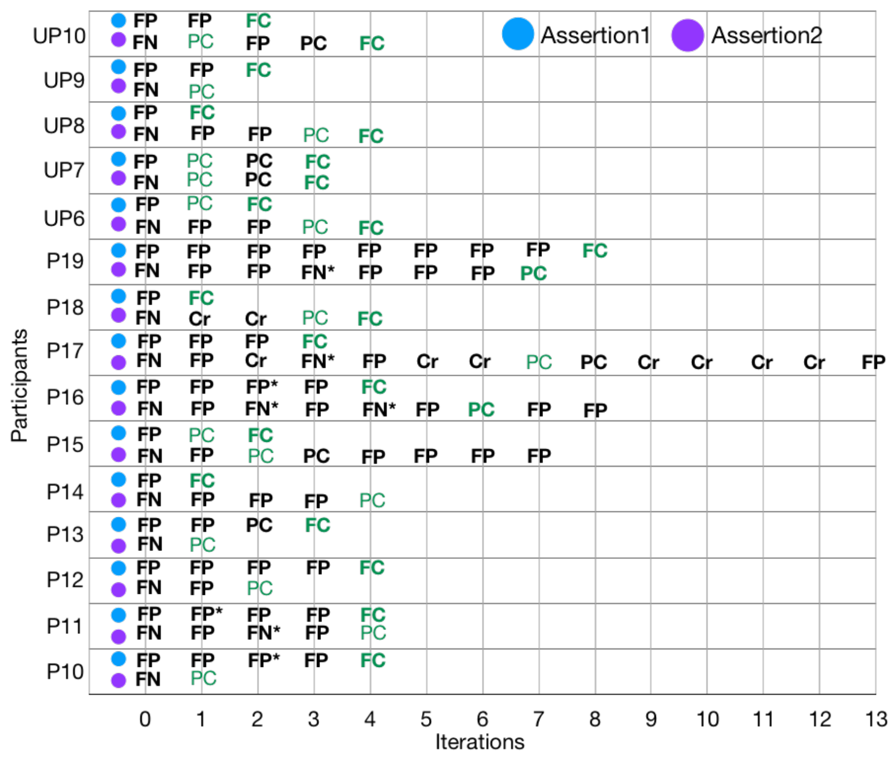

Fig. 12: Improvement Study Results: Iterative Process Details

output and fixing the oracle deficiency in case of Assertion 1 and 191 seconds in case of Assertion 2.

Only three participants (P14, P18, UP8) were able to improve the first assertion to the point of full correctness immediately after getting the report for the initial false positive, i.e. in one iteration. The more common scenario is to have a sequence of iterations (from 2 to 8) in which the tool still reports false positives. When trying to fix the false negatives in Assertion 2, 9 participants have introduced a false positive and 2 participants have introduced a crash into the assertion. A very peculiar case is the improvement process followed by Participant P17, since in 7 out of 13 iterations the tool reported a Crash.

The oracle deficiencies with an asterisk in Figure 12 denote the cases where the tool was run on an assertion identical to the initial one. This means the participant has decided to restart the process from the initial assertion. Five participants has acted so in eight different cases after on average 2.3 iterations of improvement. While it is understandable that after making a series of unsuccessful changes to the assertion, developers roll them back and restart from scratch, the initial iterations serve apparently no purpose, as the same deficiency that was already reported initially is analysed later in the process.

RQ4 (Human Effort for Iterative Process): *The introduction of the tool in the process does not impact the overall iterative improvement time to any major extent. If we exclude the tool execution time, it actually reduces the time required from humans. The number of iterations varied between 1 and 13, with an average of 3.9 iterations. In each iteration, developers spent, on average, 193 seconds of manual effort.*

Tool Performance and User Feedback

We measured the performance of our tool during the experiments as the amount of time it took to report the presence or absence of oracle deficiencies. The tool starts each iteration from a search for a false positive. In case no false positive is detected, the search for false negative is initiated. Therefore, the detection time for false negative includes the whole search budget of a false positive (60 seconds by default). Similarly, the tool uses its search budget for both false positives and false negatives before reporting that no evidence of oracle deficiencies was found. On average, during our experiments false positives were reported in 60, crashes in 62 and false negatives in 162 seconds, while the report for no oracle deficiencies took 271 seconds.

To get insight into the perceived quality of the tool, we asked participants to rate their experience with it in the exit questionnaire. We asked five Likert scale format questions, with a range of options from 1 (strongly disagree) to 5 (strongly agree). Figure 13 lists the questions an shows the answers of participants to each of them. As results show, the tool was assessed to be easy to run (4.5 on average). The usefulness of its output to understand the reason of a false positive was rated as 4.07, while its helpfulness to fix a false positive was evaluated as 4.13. For false negatives both of these numbers were a bit lower: 3.87 on average.

We also asked a multiple-choice question about the main difficulties users face when trying to interpret the output of the tool for each oracle deficiency. Figure 14 shows the percentages of chosen answers. For false positives, understanding the reported test cases (40%) and understanding why the test case makes the assertion fail (40%) were equally challenging for participants. For false negatives the main difficulty was figuring out why the assertion does not react

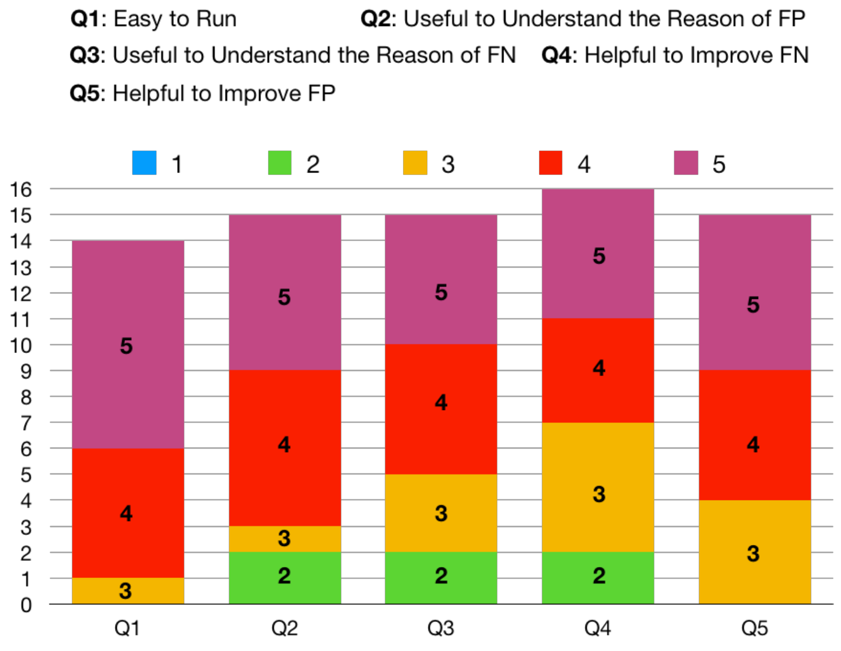

Fig. 13: User Feedback on Tool

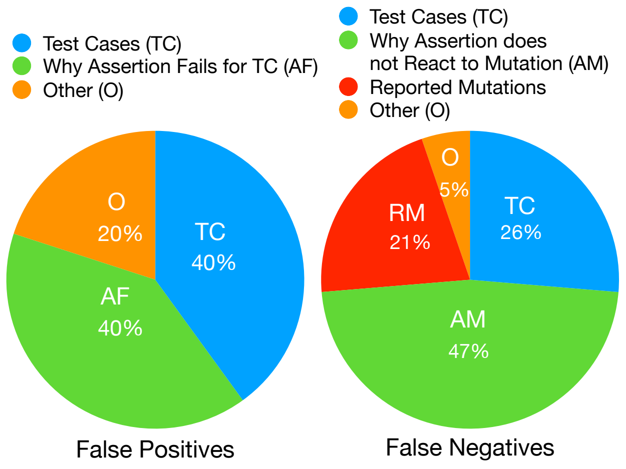

Fig. 14: Difficulties in Understanding Tool's output

to the mutation (47%), followed by understandability of the reported test cases (26%) and reported mutations (21%).

3.3 Threats to Validity

Internal. A threat to internal validity may result if the training material or experiment objectives were unclear to participants. To mitigate this threat we thoroughly revised all our training materials and tested them on a pilot study. Moreover, in the *Improvement Study* we included a *practice task* and ensured that participants had successfully completed it before proceeding to the real task. For Upwork participants, who received the training material and performed the tasks in remote mode, after each type of training (oracle improvement and tool) we required a test to be completed. They could proceed with the final task only after passing these tests.

In the *Assessment Study* participants were assigned tasks randomly to avoid bias and to have the same number of data points for all classes. In the *Improvement Study* we assigned participants from PhD student/Postdoc sessions' to control/treatment groups randomly, while Upwork participants were assigned based on their programming experience. During the *Improvement Study* each subject was either using the tool or not using it for both assertions, so as to eliminate learning effects that could influence our results.

A part of our *Improvement Study* was performed in a remote setting using the Upwork freelancing platform. The training provided to these participants was in a written form. Moreover, participants could work on the tasks at their own discretion and we could not oversee their behaviour. In the exit questionnaire, Upwork participants rated the training material as 4.8 out of 5, on average, which indicates that they were satisfied with its quality. For the participants who used the tool, we collected metadata on each tool run, therefore we could check the timeframe and iterative process for each assertion. Participants who did not use the tool self-reported time spent on each assertion. We include this information in our results, but acknowledge that it is not reliable. Overall, co-factor analysis shows that results of Upwork participants are not significantly different from the results of other participants.

In both studies, our measurements of user effectiveness are obtained by comparing participants' results against the outcome of OASIs. While OASIs provides evidence for any oracle deficiency it detects, it may report no oracle deficiencies even if some (undetected) is actually there. To deal with this issue, the authors thoroughly studied each assertion with no oracle deficiencies according to OASIs, to ensure that the tool's judgement is indeed correct.

External. The classes used in our study were not developed by our participants and may have been unfamiliar to them. However, it is a common practice that developers test the code not written by them. We selected three Java classes for our Assessment Study due to limited time for the sessions. Similarly, our Improvement Study analyses only one Java class. Furthermore, our case examples were chosen to be simple enough to be quickly understood. We acknowledge that our results cannot be generalised to other Java classes. However, we had a large number of participants in each study, and therefore we believe that our results provide insight into the behaviour of developers with different experience and background in the oracle assessment and improvement process.

A further threat to external validity is that our results might be biased by the population of developers who were registered at Upwork. Results could have been different if we had involved a different population of professional developers (e.g., using another freelancing platform). We mitigated this threat by introducing a qualification test. By adopting such a filter, we expect to be able to recruit a subset of workers with similar skills in any platform.

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 [36], [40], [44]. For a recent survey on the oracle problem and techniques for defining software oracles the reader is referred to the comprehensive review by Barr et al. [2]. In this review of related work we focus on previous work on oracle generation, improvement and studies of software engineers' behaviour with regard to oracles.

4.1 Automated Oracle Generation

4.1.1 Test Case Assertions

Modern test case generators as EvoSuite [9], [10] and Randoop [31] have the capability to automatically synthesize test case assertions. Randoop annotates the source code of the class to identify observer methods and uses them in assertion generation. EvoSuite applies the mutation-driven approach, where for each test case it selects assertions with the highest mutation killing score [12].

The work by Staats, Gay and Heimdahl [38] uses an approach similar to Evosuite's, where for each test case input it uses mutation analysis to rank variables (not assertions) in terms of fault-finding effectiveness. It then reports a set of top-ranked variables to the developers, so that they can manually write assertions that check the expected values for each variable. The work by Loyola et al. [25] explores a similar scenario, but program variables are ranked based on the dependencies observed between them during program execution. The analysis begins by using data flow analysis to construct a network of program variables for each test input. Then network centrality metrics are used to rank variables in terms of relevance or centrality in the resulting network.

The test case oracles generated by all these approaches are specific to a single run. To capture general, rather than concrete behaviour, the work by Fraser and Zeller [11] generates parameterised unit tests for which oracles are represented in the form of pre- and postconditions characterising test input and test result. The evaluation shows that parameterised unit tests are more expressive and cover 72.6% more branches than concrete unit tests. However, they are more expensive to produce and may require several minutes per test case generation.

Different approaches have been proposed to assess the quality of already generated test case assertions. The work by Schuler and Zeller [37] addresses the problem of traditional coverage metrics not reflecting the actual oracle quality and 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.

Huo and Clause [18] introduce a technique that is based on dynamic tainting and works by tracking the flow of controlled (explicitly provided by the test itself) and uncontrolled inputs along data- and control-dependencies. When a test finishes execution, the tracked information is used to generate reports that identify *brittle assertions*, assertions that check values that are derived from uncontrolled inputs, and *unused inputs*, inputs that are controlled by the test but are not checked by any assertion. In the evaluation of 4,000 real test cases, 164 tests were found to contain brittle assertions and 1,618 tests to contain unused inputs. While the technique contains a separate step to filter false positives, the final false positive rate is still very high: 63% on average for brittle assertions and 40% for unused inputs.

4.1.2 Specification Mining

Another form of automated oracles are mined specifications. The work by Nguyen, Marchetto and Tonella [42] evaluates three types of such automated oracles in terms of cost and effectiveness: data invariants, temporal invariants and finite state automata. The following tools are used as representatives of these mined specifications: *KLFA* [26] for FSA oracles, *Daikon* [7] for data invariants and *Synoptic* [3] for temporal invariants. The procedure adopted for the experimental design is as follows: while a subject system P is running, its execution is monitored to obtain traces, and different automated oracles are inferred from those traces. Then, due to the new usages, the automated oracle may report alarms when the execution violates them. Alarms might be due to a fault that has been triggered, or they may be wrong (false positives). Results show that automated oracles can detect several real faults, but such fault detection capability comes at the price of a quite high false positive rate (30% on average).

The work by Zhang et al. [46] presents *iDiscovery*, a technique that employs a feedback loop between symbolic execution and dynamic invariant discovery to infer accurate and complete invariants. In each iteration, *iDiscovery* transforms Daikon invariants into assertions and adds them to the program. The instrumented program is analysed with symbolic execution to generate additional tests to augment the initial test suite provided to Daikon. With the newly added inputs, dynamic invariant discovery will be based on a richer set of program executions enabling discovery of higher quality invariants. Experimental results show that *iDiscovery* is able to falsify from 24% to 72% of the invariants generated by Daikon.

Overall, the existing work in generation and assessment of oracles focuses mainly on the oracles for single test inputs. Our preliminary evaluation showed that assertion oracles generated using our approach have higher fault detection capability than the ones generated by Evosuite and Randoop. A few approaches as the generation of parameterised unit tests [11] or *iDiscovery* [46] address oracles for more general behaviour. The evaluation of oracles in automatically generated parameterised unit tests shows that they still have 19.6% false negative and 8.3% false positive rate. Therefore, our approach can be applied also to these oracles to further improve them. Similarly, it can be applied to the final Daikon invariants generated by *iDiscovery*.

4.2 Human Studies

Automatically generated oracles capture the implemented behaviour of the program rather than intended behaviour. Therefore, to turn them into oracles useful for fault detection developers have to identify and fix the incorrect ones, which requires human intelligence. Rather than using the human input directly, some approaches reuse the artefacts produced by humans for the program under test.

The work by Pastore and Mariani [34] aims to identify the incorrectly synthesized assertions using the manually written test cases as the source of human knowledge about the system. They present their tool *ZoomIn*, which pinpoints wrong assertions by comparing the executions produced by the manual test cases to the executions produced by the automatically generated test cases. Their intuition is that the execution of an automatic test case is likely to constitute a failure if it produces anomalous variable values while covering a case already tested by the developers. For the purpose of evaluation 7 real faults from Apache Commons Math library were selected and *ZoomIn* was applied to the test cases generated by EvoSuite. Results show that *ZoomIn* has been able to detect 50% of the analyzed non-crashing faults requiring inspection of less than 1.5% of the automatically generated assertions.

The work by Goffi et al. [13] introduced *Toradocu*, which uses developer-written Javadoc comments to create automated oracles for exceptional behaviours. The experimental evaluation of Toradocu shows that it improves the fault-finding effectiveness of EvoSuite and Randoop test suites by 8% and 16% respectively, and it reduces EvoSuite's false positives by 33%. The later work by Blasi et al. [4] extends *Toradocu* so that it produces specifications not only for exceptional behaviours, but also specifications capturing for normal pre- and postconditions. Such extended *Toradocu* achieves a precision of 91% and a recall of 83% in translating Javadoc comments into method specifications. These specifications enable Randoop to generate test cases that reveal more defects and produce fewer false alarms.

Only two human studies have been conducted to evaluate the capability of humans to improve automated oracles. They respectively used CrowdSourcing [35] to verify test case assertions and real developers to determine the quality of invariants [39]. In general, CrowdSourcing a problem consists of specifying it in the form of a Human Intelligence Task (HIT) and making the problem available in a CrowdSourcing platform, where registered workers can choose to complete HITs for a small remuneration. Pastore, Mariani and Fraser [35] proposed the idea of CrowdOracles, where test cases with synthesized assertions are verified with respect to the documentation and fixed by the crowd. The results show that *CrowdOracles* are a viable solution to address the oracle problem. However, to be successful, this approach requires a qualified crowd, which is not easy to find; monetary investment, which can be high in case of a big number of test cases and assertions; and also the existence of a good documentation for the programs under test, for the crowd to be able to determine right and wrong assertions.

Staats et al. [39] conducted an empirical study with 30 participants to determine the user classification effectiveness for invariants generated using Daikon, and to understand what factors lead to successful or unsuccessful classification. In each study, participants were given one of three Java classes with automatically generated invariants. Participants were asked to determine, for each generated invariant, if the invariant was correct or incorrect with respect to the Java class. On average, study participants misclassified 9.1-39.8% of correct invariants and 26.1-58.6% of incorrect invariants.

To evaluate classifications made by each participant, authors needed to determine whether each invariant was correct or incorrect. For this they employed two automated approaches to try to falsify invariants. First, they applied Randoop using 100,000 test inputs (far more than the 1,000 used to generate the invariants). Second, a different, manually written random test generation harness was produced for each case example, and then applied for a long period of time (24 hours). For any remaining invariants, three of the authors manually examined each one, attempting to develop a test input capable of violating the invariant. When failing, they tried to understand whether the invariant was indeed correct. Invariants that they could not falsify were accepted as correct. As we have noted before, the definition of invariant correctness in this study is in line with our definition of false positives. So, to recheck the classification of the authors, we applied OASIs, considering only false positive detection, to the invariants used in the study by Staats et al. [39]. Table 13 shows that while the approach described in the paper [39] found 73 assertions with a false positive among 324 assertions, our approach found false positives in 60 more assertions.

TABLE 13: Improvement Study: Results Without Tool

Class	# of Assertions	Incorrect	
		In Paper [39]	OASIs
Matrix	122	18	42
Poly	121	26	50
StackAr	81	29	41
Overall	324	73	133

Our human study was designed to assess the usefulness of the information that our tool provides to developers. Evidence for the necessity and importance of such kind of studies can be found in empirical research on program debugging tools. Indeed, our tool in some ways resembles these in that it assists the developer to "repair" the oracle. The work by Parnin and Orso [33] analyses whether automated debugging tools are actually helping programmers. Their results show that debugging tools are helpful in completing a task significantly faster, but only for experienced developers and simpler code. The study by Tao et al. [41] shows that automatically generated high-quality patches significantly improve debugging correctness of developers, but this effect is limited to difficult bugs. Moreover, debugging time is significantly affected by participant type and the specific bug to fix. The user study by Wang et al. [43] demonstrates that the support provided by information retrieval based techniques is helpful to developers only in getting to the faulty file quickly, but not in understanding and fixing the bug within that file. In contrast, our human study confirmed our hypothesis that OASiS is useful to the developer in all tasks associated with oracle improvement.

Overall, the results of the two existing human studies [35], [39] addressing the oracle problem are not consistent with 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 the incorrect ones. While this might be partially explained by the different nature of the oracles considered in the two studies (test case assertions vs. Daikon invariants), it also shows that there is a strong need for more experiments analysing the performance of human testers in the oracle improvement process.

4.3 Tool Output Improvement

Our tool OASIs can provide the evidence of existing oracle deficiencies and can guide developers in fixing them. Such evidence/guidance takes the form of automatically generated test cases. Any improvement in the understandability of these test cases is beneficial for the adoption of our approach. Therefore, the techniques to improve the readability of automatically generated test cases [6] or to provide test case summaries in natural language [32] are all related to our work. The analysis of the iterative oracle improvement process using OASIs shows that around 45% of time for each iteration is spent on actually running the tool. The main cost associated with the execution of OASIs is the mutation analysis step, performed to identify false negatives. One performance optimisation could be to avoid analysing all possible mutations for a method, considering only a meaningful/representative subset of such mutations. Therefore, the works on mutant selection [30], [45] can become a part of our implementation in future work.

5 Conclusion

We analysed whether our approach for oracle assessment and improvement with the human in the loop supports the creation of better oracles. The role of the human in the loop was played by developers with different backgrounds and experience: master degree students, PhD students, postdoctoral researchers, professional developers and freelancers from the Upwork platform. Our results show that humans perform poorly when assessing oracles manually. Their correct classification rate is 29%, on average. Professional developers (48%) show almost twice better performance than students (25%), but still misclassify more than half of oracle deficiencies. Overall, false positives are harder to detect than false negatives. However, the most common misclassification type is when an assertion with a false negative is classified as an assertion having no oracle deficiencies. We analysed the effect of multiple factors (experience, level of satisfaction with time allocated for the task and with the training material provided) on users' performance, but found no strong correlation with any of them.

When provided with the information on the type of oracle deficiency the assertion has and asked to improve it manually, developers, on average, achieved 21% of full and 43% of partial correctness. These numbers increased significantly, with developers achieving 67% of full and 33% of partial correctness when the they used our tool OASIs for the improvement process. The overall number of iterations varied from 1 to 13, with an average of 3.8 for full and of 3.66 for partial correctness. Results show that developers struggle with achieving full correctness. None of the participants doing manual improvement was able to improve any of the assertions in our study to a fully correct state. 3 participants from the group with the tool ran it for 2.6 extra iterations on average after achieving partial correctness to produce a fully correct assertion, but they did not succeed. While the reports of OASIs, informing users that their assertions are only partially correct, were judged definitely useful (they prevent developers from believing the their oracles will not miss any faults), in practice users might prefer to stop the improvement process at a partially correct state, due to the substantial effort incurred to achieve full correctness.

Overall, our results show that the proposed approach supports the developer in both the oracle assessment and oracle improvement processes, and leads to the creation of more sound and complete oracles. Our future work will be to optimise the performance of OASIs, so that it takes less time to run and leads to a smoother incremental improvement process. The user feedback collected during these studies about the understandability and helpfulness of the tool's output, including the difficulty in understanding the automatically generated test cases, will be addressed by incorporating existing works in the area of test code understandability into the implementation of OASIs.

Acknowledgments

This work was partially supported by the ERC Advanced project Precrime (ERC Grant Agreement n. 787703) and the UKRI EPSRC project InfoTestSS (EP/P005888/1).

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] 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.

[3] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 267–277, New York, NY, USA, 2011. ACM.

[4] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos. Translating code comments to procedure specifications. In *ISSTA 2018, Proceedings of the 2018 International Symposium on Software Testing and Analysis*, Amsterdam, Netherlands, July 2018.

[5] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, Feb. 2013.

[6] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 107–118, New York, NY, USA, 2015. ACM.

[7] 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.

[8] J. Fleiss et al. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

[9] 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.

[10] 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.

[11] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 364–374, New York, NY, USA, 2011. ACM.

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

[13] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 213–224, New York, NY, USA, 2016. ACM.

[14] 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.

[15] 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.

[16] M. Harman, A. Mansouri, and Y. Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, Department of Computer Science, King's College London, April 2009.

[17] J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. Miller, R. Pozo, and K. Remington. Jama: A java matrix package. *URL: http://math. nist. gov/javanumerics/jama*, 2000.

[18] 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.

[19] 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 2016, pages 247–258, New York, NY, USA, 2016. ACM.

[20] G. Jahangirova, D. Clark, M. Harman, and P. Tonella. OASIs: Oracle assessment and improvement tool. In *Proceedings of the 27th International Symposium on Software Testing and Analysis (Tool Demonstrations)*, ISSTA 2018, New York, NY, USA, 2018. ACM.

[21] 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.

[22] 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.

[23] k. krippendorff. Content analysis: An introduction to its methodology. sage, thousand oaks krippendorff k (2011) principles of design and a trajectory of artific iality. 28, 01 2004.

[24] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1), 1977.

[25] P. Loyola, M. Staats, I. Ko, and G. Rothermel. Dodona: automated oracle data set selection. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 193–203, 2014.

[26] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, ISSRE '08, pages 117–126, Washington, DC, USA, 2008. IEEE Computer Society.

[27] Math4J. A java numerics package. 2005.

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

[29] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. *SIGSOFT Softw. Eng. Notes*, 27(4):229–239, July 2002.

[30] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, 1996.

[31] 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.

[32] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 547–558, New York, NY, USA, 2016. ACM.

[33] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[34] F. Pastore and L. Mariani. Zoomin: Discovering failures by detecting wrong assertions. In *Proceedings of the International Conference on Software Engineering*, 2015.

[35] 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.

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

[37] 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.

[38] M. Staats, G. Gay, and M. P. E. Heimdahl. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 870–880, 2012.

[39] 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.

[40] 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.

[41] Y. Tao, J. Kim, S. Kim, and C. Xu. Automatically generated patches as debugging aids: A human study. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 64–74, New York, NY, USA, 2014. ACM.

[42] 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.

[43] Q. Wang, C. Parnin, and A. Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 1–11, New York, NY, USA, 2015. ACM.

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

[45] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.

[46] 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.

Jahangirova Gunel is a PostDoctoral Researcher at the Software Institute of Universita' della Svizzera Italiana (USI) in Lugano, Switzerland. She had her PhD in a joint program between Fondazione Bruno Kessler, Trento, Italy and University College London, London, UK. Her PhD work was about oracle problem in software testing, in particular, assessment, improvement and placement of test oracles. Her current research interests include automatic generation of program assertions, mutation testing, failed error propagation and testing of deep learning systems.

Mark Harman works full time at Facebook London and also holds a part-time professorship at UCL. At Facebook he worked on the deployment of Sapienz to test mobile apps, leading to thousands of bugs being automatically found and in multimillion line communications and social media apps in daily use by over 1.4Bn people worldwide. He co-founded the field SBSE, a research area with authors spread over more than 40 countries, and is also known for work on source code analysis, software testing, app store analysis and empirical software engineering. He received the IEEE Harlan Mills Award and the ACM Outstanding Research Award in 2019 for this work. In addition to Facebook itself, Mark's scientific work is supported by the European Research Council (ERC), with an advanced fellowship grant, and has also been supported by the UK Engineering and Physical Sciences Research Council (EPSRC), with platform and programme grants.

Paolo Tonella is Full Professor at the Faculty of Informatics and at the Software Institute of Universita' della Svizzera Italiana (USI) in Lugano, Switzerland. He is also Honorary Professor at University College London, UK. Until mid 2018 he has been Head of Software Engineering at Fondazione Bruno Kessler, Trento, Italy. Paolo Tonella holds an ERC Advanced grant as Principal Investigator of the project PRECRIME. In 2011 he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for his paper: "Analysis and Testing of Web Applications". He is the author of "Reverse Engineering of Object Oriented Code", Springer, 2005, and of "Evolutionary Testing of Classes", ISSTA 2004. Paolo Tonella was Program Chair of ICSM 2011 and ICPC 2007; General Chair of ISSTA 2010 and ICSM 2012. He is/was associate editor of TOSEM/TSE and he is in the editorial board of EMSE and JSEP. His current research interests include deep learning testing, web testing, search based test case generation and the test oracle problem.

David Clark is a Reader in Program Analysis at the Department of Computer Science of University College London. He gained his PhD at Imperial College while his first permanent appointment was at Kings College London before moving to UCL in 2010. He is most widely known for his seminal work in applying information theory as a program analysis to measure flow insecurity. His current research interests are dominated by software testing and, in particular, an attempt to construct an information theory based set of principles for it. Current topics of interest to him in this area include test suite diversity, using Deep Neural Nets for property based testing, and problems associated with test oracles.