Test Oracle Assessment and Improvement

Gunel Jahangirova
FBK, Trento, Italy &
UCL, London, UK

David Clark &
Mark Harman
UCL, London, UK

Paolo Tonella
FBK, Trento, Italy

ABSTRACT

We introduce a technique for assessing and improving test oracles by reducing the incidence of both false positives and false negatives. We prove that our approach can always result in an increase in the mutual information between the actual and perfect oracles. Our technique combines test case generation to reveal false positives and mutation testing to reveal false negatives. We applied the decision support tool that implements our oracle improvement technique to five real-world subjects. The experimental results show that the fault detection rate of the oracles after improvement increases, on average, by 48.6% (86% over the implicit oracle). Three actual, exposed faults in the studied systems were subsequently confirmed and fixed by the developers.

CCS Concepts

•Software and its engineering → Software verification and validation;

Keywords

Test oracle; test case generation; mutation testing

1. INTRODUCTION

Recent advances in test input generation have left the Oracle Problem as a key remaining bottleneck in improving the overall effectiveness and efficiency of the software testing process. The latter depends both on the quality of the test cases and that of the oracle [4, 29, 31]. 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 [18, 23] and symbolic execution [6] 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 but only faulty behaviours exercised during testing [13, 19, 22, 25, 26, 28]. Without a (good) oracle to determine whether or not the output they induce is correct, test inputs that satisfy the strictest adequacy criteria remain useless and testing is ineffective.

In the absence of an automated oracle, possibly derived from formal specifications, the task of determining output correctness typically falls to the human [1, 17]. Unfortunately, this is costly, slow and error prone, motivating the need for automated decision support to assist the human. We study the use of in-program logical assertions (assertion oracles) as decision support and the problem of ensuring that they faithfully reflect developer knowledge of the intended behaviour of the software (the perfect oracle).

Oracle performance depends on two properties: **Completeness**: All correct program states are accepted by the oracle, which raises an alarm only for faulty states, with no false alarms (no *false positives*). **Soundness**: All faulty program states are rejected by the oracle, so there are no missed faults (no *false negatives*).

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.

We introduce an approach that is based on search based test case generation [10, 16, 23] to identify false positives and mutation testing [20, 21] to identify false negatives. Our tool generates counterexamples as test cases that demonstrate incompleteness and unsoundness, which the tester uses to iteratively improve the assertion oracle. The process continues until the tool is unable to generate new counterexamples and finishes with an improved (more complete and sound) oracle. Our approach necessarily places the human tester in the loop, because modifications made to the oracle to solve reported 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.

Our primary contributions are:

1. A novel iterative oracle assessment and improvement approach, validated on five nontrivial real-world systems, demonstrating the improvements that can be achieved using our approach (on average, 48.6% increased fault detection compared to unimproved). Using our approach, we found three bugs in Apache Commons Math. Apache's developers have confirmed all of three to be genuine bugs and have already fixed them.

2. A formalisation of the oracle improvement step as a change in the mutual information between the actual and perfect oracles and a proof that a monotonic sequence of increases is always possible in practice.

2. 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. Define

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

i.e. the set of reachable states for P at pp on which the assertion is true.

Assume that a developer/tester knows exactly which states that occur at pp are correct (the perfect oracle). Call this set E_{pp}, and think of E_{pp} as the intersection between the set of correct states at pp for some "ghost program" [3], G, an error free version of the software under test, and R_{pp}, the reachable states of the SUT.

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

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

2.1 The oracle improvement process

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.

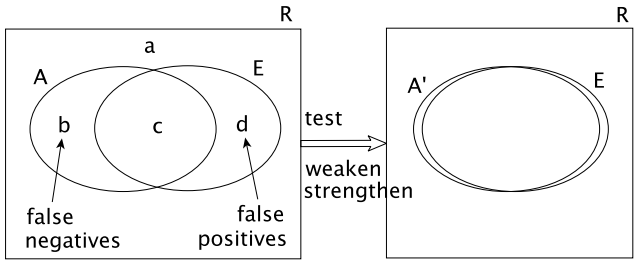

Figure 1: Assertion improvement process: the intersection between states where $\langle assert \rangle$ is true (A) and expected states (E), restricted to the subset of reachable states (R), is increased.

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 $(E - A)$ are 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.

3. APPROACH

In this section, we describe our technique for false positive and false negative detection, as well as its implementation based on the test case generator EvoSuite [10].

3.1 False positive detection

Given a program assertion, we detect its false positives by generating execution scenarios where the assertion fails when it should hold because the behaviour of the program is deemed correct. In such a case, failure of the assertion points to a bug in the assertion, not in the program.

To be able to generate the necessary execution scenarios (test cases), we perform a testability transformation [15] that transforms the criterion for false positive detection into the standard branch coverage criterion, for which automated test case generators are available.

Let us consider a program under test P containing n assertions $a_1 \ldots a_n : a_i = assert(c_i), i \in [1 \ldots n]$, where c_i is the boolean expression used in the assertion a_i. For each assertion $a_i, i \in [1 \ldots n]$ in P the proposed testability transformation takes c_i, negates it and replaces the assertion a_i with a new branch containing the negated condition: if $(!(c_i))$ {}.

Class `Subtract` in Figure 2 (top) has two assertions at lines 4 and 5. The transformation for false positive detection, takes the condition of the assert statement at Line 4 '`(result != x)`', negates it to '`(!(result != x))`' and replaces the assertion with the branch: '`if (!(result != x)) {}`'. By performing a similar transformation on the assert statement at Line 5 we get the transformed version of class `Subtract` shown in Figure 2 (bottom).

Test case generators are given two targets to cover: the '`then`' parts of the '`if`' statements at lines 4, 5. Test cases produced by the generator provide evidence that there are program executions that violate the assertions. In order to classify such execution scenarios as false positives of the

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

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

Figure 2: Example of class under test, including an incorrect assertion (at Line 4, top)

assertions, the behaviour of the program in such scenarios must be contrasted with the expected behaviour of the program, according to its requirements/specifications. If a test case violating an assertion has been generated and the program behaviour under such an execution scenario has been deemed correct, a false positive (i.e., a bug in the assertion) has been detected. This means that the assertion should be fixed in order not to reject a correct program behaviour.

In the example shown in Figure 2, a test case can be produced that covers the first target: `TC=(0, 0)`. By contrast, the second target cannot be covered and a test case generator would probably fail or time out while trying to cover it. Since the expected result of the execution of `value` with input `(0, 0)` is indeed 0, we have detected a false positive of the assertion at line 4. The assertion is incorrect and the fix consists simply of removing it.

3.2 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 [14] to insert a (known) fault in program P that corrupts the program state so that the corrupted state reaches the given assertion and the assertion statement does not fail. We apply a testability transformation [15] that converts the false negative detection criterion to the standard branch coverage 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 $a_1, \ldots, a_n$: $a_i = assert(c_i), i \in [1 \ldots n]$. 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 is indicated as $(v_1^o, \ldots, v_{m_i}^o)$, while they are indicated as $(v_1^{M_j}, \ldots, v_{m_i}^{M_j})$ after running the same test case on mutant M_j. For each assertion a_i and mutant M_j we create a transformed version of P by going through the following steps:

- **Step 1**: In P, for each variable $v_1, \ldots, v_{m_i}$ we create a private field and a public setter method for this field.
- **Step 2**: In P, we replace each assertion a_i with the following branch:

  if $((c_i^o == c_i^{M_j})$ && $(v_1^{M_j} \neq v_1^o \mid\mid \ldots \mid\mid v_{m_i}^{M_j} \neq v_{m_i}^o))$ {}

Automated generation of test cases to cover the branch produced at Step 2 proceeds iteratively as follows:

1. The test case generator runs each newly generated test case on each mutant M_j.
2. If the mutant is strongly killed (i.e., P and M_j exhibit observably different behaviours), the test case generator stores the values $(v_1^{M_j}, \ldots, v_{m_i}^{M_j})$ into P by calling the public setter methods created at Step 1.
3. The test case generator runs the strongly killing test case on P.
4. If the test case executed on P covers the target branch created at Step 2, a false negative is reported. Otherwise, the test case generator modifies the test case so as to get closer to the target branch, hence producing a new test case to be run.

So, when a false negative is reported, the following conditions hold: (1) the program under test P contains a known fault (the mutation), associated with an observably different behaviour between P and M_j (strongly killing) condition; (2) the corrupted program state (*infection*) reaches the considered assertion a_i (at least one of the variables in scope has a different value in P vs. M_j); but, (3) the outcome of the assertion is the same for P and for M_j (presumably a pass; otherwise we are potentially in the presence of a false positive). This means that the assertion is not strong enough to capture the difference between P and M_j, although at least one variable accessible to the assertion has indeed a different value between the execution of P and that of M_j.

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

```
1  public class FastMath {
2      private int max_m;
3      private int a_m, b_m;
4      public void setMax_m(int max_m)
5                  { this.max_m = max_m; }
6      public void setA_m(int a_m) { this.a_m = a_m; }
7      public void setB_m(int b_m) { this.b_m = b_m; }
8      public int max (int a, int b) {
9          int max;
10         if (a >= b) {
11             max = a;
12         } else {
13             max = b;
14         }
15         if ((max_m >= a_m) == (max >= a) &&
16               (max_m != max || a_m != a || b_m != b))
17             {} // target
18         return max; } }
```

Figure 3: Class under test (top) and class transformed for false negative detection (bottom)

Figure 3 shows an example of the described transformation. Fields `max_m`, `a_m` and `b_m`, together with the respective setter methods, are added to class `FastMath`, to store the values of the variables visible at Line 9 in Figure 3 (top)

and observed during the execution of the mutant. The assertion at Line 9 in Figure 3 (top) becomes the `if` condition at Line 15 in the transformed program shown in Figure 3 (bottom). The `then` branch of this conditional statement is the target for test case generation. If the test generator succeeds in creating a mutation killing test case (in our example, one returning a different value of `max`) that covers this target, we obtain evidence of a false negative. In fact, although such a test case can strongly kill the mutant, the assertion `(max >= a)` does not fail (provided it did not fail in the original program), despite the presence of different values of either `max`, `a` or `b` in the original vs. mutated program.

Let us consider a mutant that changes the assignment `max = b;` at Line 7 in Figure 3 (top) into `max = a;`. The test case `TC=(0, 1)` can strongly kill this mutant, because the value returned by the original version of `max` is 1, while it is 0 when the mutant is executed. On the other hand, the assertion passes on both original and mutated programs, since on both we have that `max >= a` is true. This means that this test case satisfies the first part of the condition to reach the target (Line 15 in Figure 3 (bottom)). It satisfies also the second part (Line 16) because `max_m` and `max` are different (in this particular example, this happens to be the same condition as the strongly killing condition; in general, this might not be the case). So, this test case shows that it is possible to inject a fault in class `FastMath`, resulting in an observably different behaviour between original and mutated programs, which no present assertion can detect. This is an example of a false negative, requiring an intervention by the developers in order to make the assertion stronger. Specifically, it is possible to eliminate this false negative by replacing the assertion in Figure 3 (top) with `assert (max >= a && max >= b);`.

There are a few possible, though unlikely, corner cases. A bug might affect both the implementation *and* the assertions consistently, making the assertions pass on original and mutated program. In such a case, it would be prudent for the tester to check the output of mutant killing test cases, rather than assuming that only assertions can be wrong. Other cases are discussed in section 3.3 below.

3.3 Iterative improvement process

We propose a process for iterative oracle assessment and improvement based on the outcome of false positive/negative detection. The human is necessarily in the loop of the process, because we assume that knowledge about the intended program behaviour is available only informally or semi-formally to the developers, who are asked to manually refine the oracle whenever a false negative or a false positive is reported. Our approach might not be applicable in software processes that include complete formal specifications, from which oracles are derived automatically. In the authors' experience, industrial practice usually does not currently encompass complete formal specification, hence we think the proposed approach has wide applicability.

The starting point for iterative oracle assessment and improvement is an initial oracle, which can be defined manually, or can be produced automatically by tools for invariant inference, like Daikon [8], or can be even the empty (vacuous) oracle. *Oracle deficiencies* (i.e. false negatives or false positives) are detected and reported automatically by our tool. 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 cases: (1) A reported false positive might point to a bug in the program, not in the assertion; (2) A test case killing a mutant and triggering an assertion violation in the mutant might be associated with consistent bugs in both implementation and assertion; (3) A mutant might accidentally fix a fault in the program (this is expected to occur extremely rarely), causing a reported false negative to point to a bug in the program, not in the assertion. The first case is very important, since the improved oracle is immediately used for fault detection when this case occurs. Once assertions have been improved by the developer, the iterative process restarts and the new assertions are assessed for the presence of further oracle deficiencies. The overall outcome of the process is the improved oracle together with the bugs that such an improved oracle can find.

3.4 Implementation in EvoSuite

Our prototype tool for false positive and false negative detection is implemented as an extension of the EvoSuite [9, 10] test case generator[1].

For the detection of false positives, we use EvoSuite's branch coverage criterion. 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'.

Since 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 changed the fitness function of EvoSuite [11] into:

$$f_{B'} = \mid F \mid - \mid F_{\mathrm{T}} \mid + \sum_{b \in B' \setminus B} d(b, T)$$

where $\mid F \mid - \mid F_{\mathrm{T}} \mid$ is the set of unexecuted methods in the class under test; $d(b, T)$ is the minimal normalized branch distance.

For the detection of false negatives, we use EvoSuite's mutation killing criterion. 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. In fact, the test case assertions generated by EvoSuite capture the observable behaviour of the program, so a mutant is considered as strongly killed if the observable behaviour changes upon test case execution between the original and the mutated program. When the mutant is executed, but not strongly killed, the minimum normalized impact is measured in the fitness function [11]. Its inverse gives the level of propagation of the infected state in the program (propagation distance, d_p), with a wider impact (lower d_p) regarded as an indicator that the test case is getting closer to achieving the strong killing condition.

To detect false negatives, we have to further restrict the notion of mutation killing. For a given assertion a_i and mutation M_j the mutation is considered to be killed only if:

1. The original killing condition of EvoSuite is satisfied: a test case assertion fails.
2. The conditions in the program assertions do not change their values: $\forall i \in [1 \ldots n] : c_i^{M_j} = c_i^o$.

[1] http://www.evosuite.org

3. One of the variables visible at pp_i has different values in P and M_j: $\exists i \in [1 \dots n] : v_1^{M_j} \neq v_1^o \vee \dots \vee v_{m_i}^{M_j} \neq v_{m_i}^o$.

As a result, we changed the formula for the normalized propagation distance d_p, so that, when the mutant is killed, it returns the normalized distance for the following condition: $(\forall i \in [1 \dots n] : c_i^{M_j} = c_i^o) \wedge (\exists i \in [1 \dots n] : v_1^{M_j} \neq v_1^o \vee \dots \vee v_{m_i}^{M_j} \neq v_{m_i}^o)$, instead of returning zero.

4. EXPERIMENTAL RESULTS

We have conducted a set of experiments to answer the following research questions:

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

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

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

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

The goal is to assess the applicability of the proposed approach in different contexts, ranging from one where no oracle is present, hence fault detection relies entirely on the implicit oracle (program crashing or raising exceptions), to a context where the oracle is obtained automatically, by mining program specifications from the observed program behaviour, or is produced manually. The effectiveness of the improved oracle is assessed in terms of increased fault detection with respect to the initial and test case oracle.

To answer RQ1-2-3 we report the number of assertions added in each iteration to solve the false positives and negatives reported by our tool.

To answer RQ4 we analyse the mutation score reported by the popular and scalable mutation analysis tool PIT[2] for test case assertions and for program assertions before and after the improvement process.

There is empirical scientific evidence that mutants are an appropriate (and laboratory controllable) surrogate for real software faults [2, 21], making the mutation score a reasonable proxy for the actual fault detection rate. Since false negative detection relies also on mutation analysis, we used different tools (EvoSuite and PIT) for our technique and its evaluation, thereby avoiding any circularity in the evaluation. We further manually verified that these two techniques generate different mutants when the oracle is improved (using EvoSuite's internal mutant generator) compared to when the effectiveness of the generated oracles is assessed (using the PIT fault insertion tool).

During the experiments the human in the iterative assertion improvement process was the first author, who had no familiarity with the subjects and no previous experience in writing specifications. She of course knew how to interpret the tool's output very well.

4.1 Subjects

The subject systems used in our study are shown in Table 1. As each research question requires a different type of initial oracle, the subjects for each of them vary too. For the purpose of evaluation on programs with no initial oracles (RQ1, RQ2), we have selected Apache Commons Math, Apache Commons Collection and Apache Commons Lang, which are popular open source libraries that have been also used in previous testing research. To evaluate our approach on programs that include human written program assertions (RQ3), we have used the JavaFE front-end parser library and Logging framework, which contain JML contracts. All of the subjects are used to evaluate the increased fault detection capability (RQ4) of the improved oracles.

Table 1: Features of the subject systems

Id	Oracle	Name	NCLoC
CC	None	commons-collections	29,954
CM	None	commons-math4	83,929
CL	None	commons-lang	25,386
FE	JML	JavaFE	31,912
LG	JML	Logging	1,583

4.2 Experimental procedure

For RQ1 no initial oracle is needed, since the implicit one is used. To infer initial oracles for RQ2, first, the random test generation tool Randoop has been used to produce a large test suite T for the system under test (SUT), P. The training traces needed by the invariant inference tool Daikon [8] are obtained by running T on P. From such traces, Daikon infers properties of program P. These are used as initial oracles. For RQ3, the initial oracles are already provided with the SUTs. However, to make them compatible with our tool, the JML specifications have been manually transformed into standard Java assertions.

Once the initial oracles are available, the iterative process of oracle assessment and improvement begins. Each iteration consists of two sub-processes: detecting and removing (1) false positives and (2) false negatives. The stopping condition for each sub-process is defined according to EvoSuite's default stopping criterion which is 60 seconds as search budget and 600 seconds as global budget. Note that the first sub-process is iterative itself, as we repeat it until no more false positives can be detected. It involves running our tool using its branch coverage criterion. The outputs of the tool in this case are the test cases which show the existence of false positives in the given assertions. Using these counterexamples, the assertions can be either improved to contain no false positives or can be removed completely, if deemed completely incorrect and useless. For the second sub-process, we run our tool enabling its strong mutation criterion. In this case, the outputs consist of the test cases and the list of mutations killed by each test case according to our redefined notion of mutation killing. These counterexamples show the cases for which there is no assertion that reacts to the fault injected by a mutation.

The nature of the mutation operations applied provides specific and detailed guidance during the improvement process of the assertions. Table 2 shows the list of mutation operators reported by EvoSuite grouped by the type of improvement actions required to remove false negatives.

Check variable value: the way to improve the assertions is first to identify whether the changed variable is indeed allowed to change its value during the execution of the

[2]http://pitest.org

Table 2: Procedure for assertion improvement

Improvement Action	Mutation Operator Reported
Check variable value	DeleteField InsertUnaryOperation ReplaceConstant ReplaceVariable
Check statement	DeleteStatement ReplaceArithmeticOperator ReplaceBitwiseOperator
Check condition	DeleteStatement NegateCondition ReplaceComparisonOperator

program. If not, we should add a check on the variable immutability. In case the change is allowed, the assertions should be revised so as to ensure that the variable is changed in accordance with the expected program behaviour.

Check statement: assertions fail to differentiate the original output of the statement from that of the mutated one. The typical improvement in this case consists of adding a check on the output value of the mutated statement.

Check condition: when assertions are not responsive to a mutated condition, the relationship between the changed condition and the output of the program is usually not captured in the assertions, so this relationship should be introduced into the assertions, in accordance with the intended conditional behaviour of the program.

To compare the fault detection capability of the initial, improved and test case oracles (RQ4), we used mutation analysis. First we generated the following test suites: (T_1) Randoop test suite without assertions; (T_2) Randoop test suite with test case assertions, minimised by line coverage as measured using Cobertura[3]; (T_3) EvoSuite test suite with test case assertions, generated according to the branch coverage criterion. Each class had three versions: (P_1) class with initial assertions; (P_2) class with improved assertions; (P_3) class without any assertions. Then, we used PIT to compute the mutation score using the following combinations of test suite and program version: (1) P_2, T_1 compared to P_1, T_1; (2) P_2, T_1 compared to P_3, T_2 and P_3, T_3.

4.3 Results

Table 3 shows a summary of the results obtained in our experiments. The interested reader can find detailed results for each class considered in our study in our companion Technical Report [4].

Column *C/M* in Table 3 reports the number of constructors and methods in each subject's classes. Column *Iteration1* shows the number of assertions available in the first iteration. For RQ1 (implicit oracle), it is the number of new assertions introduced to address the false negatives revealed initially by mutation analysis (subcolumn *New*). For RQ2 and RQ3 these are respectively the number of assertions produced by Daikon or those already available in the original programs (subcolumn *Init*). Columns *Iteration2* and *Iteration3* contain three subcolumns *New*, *FP*, *FN*, which report the number of newly added assertions, assertions in which false positives were detected and assertions in which false negatives were detected. The subcolumns *A*, *FP*, *FN* of column *Total* show the overall number of assertions generated, false positives and false negatives detected during all the iterations.

In terms of human effort we estimate that the average time spent to improve the assertion in the case of a detected false positive was 4 minutes and for a detected false negative it was 10 minutes.

4.3.1 RQ1 (Implicit Oracle)

To generate the experimental data necessary to answer RQ1 we ran our tool on 25 classes from *Apache Commons Math* and 25 classes from *Apache Commons Collections*.

For most classes (98%) the improvement process was completed in no more than three iterations. For 4% of the classes, all of which belong to *Collections*, the process was completed in just one iteration, which means that no oracle deficiencies were detected for the assertions generated in the first iteration. For 72% of the classes from *Math* and 80% from *Collections* two iterations were enough. Only 28% of classes from *Math* and 25% of classes from *Collections* required three iterations to find all the oracle deficiencies. In the third and last iteration, 90% of detected deficiencies were false negatives and only 10% false positives. There was only one class from *Collections* (`StringKeyAnalyzer`) that required 7 iterations to complete the process.

RQ1: *The proposed oracle improvement process effectively supported the creation of program assertions from scratch. The process typically involved two to three iterations of successive oracle refinement to converge to an oracle for which no deficiency is reported.*

4.3.2 RQ2 (Inferred Properties)

For RQ2 we considered *Apache Commons Lang* and *Apache Commons Math*. The size of the test suite generated by Randoop to create the training traces for Daikon ranges between 250 and 34,126, with an average of 4,141. The number of preconditions and postconditions generated by Daikon for each class was on average 10 and 30, respectively.

There were no classes for which Daikon was able to generate assertions without any oracle deficiencies. For 75% of the classes from *Lang* and 65% of the classes from *Math* one iteration was enough to complete the process. For the remaining classes, two iterations (after initial oracle creation) were needed. All of the detected false positives in Daikon-generated assertions were removed in the first iteration. The false positives in the second iteration (just 2 classes) are due to the new assertions added at the first iteration.

The preconditions generated by Daikon have been treated as filters for the postconditions. Hence, a false positive is found if a precondition holds and the postcondition fails. Failure of a precondition was regarded as a true positive (i.e. a needed check at the beginning of the method) if such a failure prevents an execution that results in some error. Otherwise the precondition was weakened or removed.

The postconditions generated by Daikon for the analysed classes can be classified as follows: (1) Daikon was able to generate the exact postcondition for all the methods in the class, so no false negatives were detected. This happened in 30% of the classes in *Lang* and 50% of the classes in *Math*. (2) Daikon was not able to generate the exact postcondition, but it was able to generate a very weak one, as for example, the check for non null-ness. This happened in 35% of the classes in *Lang* and 25% of the classes in *Math*. In this case,

[3]http://cobertura.sourceforge.net/

[4]TR-FBK-SE-2016-1: https://se.fbk.eu/technical-reports

Table 3: Oracle deficiencies (FP/FN) reported by our tool at each improvement iteration

RQ	Classes	Subj	C/M	Iteration1		Iteration2			Iteration3			Total		
				New	Init	New	FP	FN	New	FP	FN	A	FP	FN
RQ1	25	CM	66/203	285	0	24	13	98	0	0	12	309	13	110
RQ1	25	CC	69/302	307	0	43	23	53	2	0	6	352	23	59
RQ2	20	CL	49/191	0	605	55	114	44	6	0	2	660	114	46
RQ2	20	CM	26/107	0	1014	43	297	166	8	4	13	1065	301	179
RQ3	50	FE	55/155	0	465	21	0	106	0	2	17	486	2	123
RQ3	10	LG	13/55	0	134	26	0	33	3	0	5	153	0	38

the generated assertion was improved to contain no more false negatives. (3) Daikon was not able to generate any postcondition, so the new assertions were added to remove the false negatives. This was the case in 35% of the classes in *Lang* and 25% of the classes in *Math*.

RQ2: *The proposed oracle improvement process was extremely effective in improving weak assertions generated by Daikon or in adding assertions that were missed by Daikon. The process typically involved one iteration of Daikon oracle refinement.*

4.3.3 RQ3 (Manual Oracle)

While *JavaFE* and *Logging* do include JML specifications, the number of constructors and methods having contracts is indeed quite low. To apply our tool in a scenario different from that of RQ1, we have selected 50 classes from *JavaFE* and all the classes from *Logging*, which have at least two methods/constructors with at least one `requires` or `ensures` JML specification.

In 82% of the classes in *JavaFE* and in 60% of the classes in *Logging* no oracle deficiencies were detected after the first iteration. The remaining classes required just one more iteration. In 48% of the classes from *JavaFE* there was at least one method with no oracle deficiencies at all.

Overall, the oracle improvement process was not able to detect any false positives in these classes, but it was able to find at least one false negative in each class.

The improvements necessary to remove the identified oracle deficiencies are typically minor improvements. The most common case was the addition of some immutability check. Less frequent were cases where a very weak postcondition (such as `@ensures \result != null` or `@ensures \fresh (\result)`, had to be strengthened, or a postcondition had to be added to a method with only `@requires` and no `@ensures` clause.

RQ3: *The proposed oracle improvement process was able to detect deficiencies in manually defined JML contracts, but the associated improvements were typically minor ones, with the exception of a few cases of weak or missing postconditions.*

4.3.4 RQ4 (Fault Detection)

Table 4 shows the average mutation score computed by PIT for each subject with initial/test case (μ_s) and improved (μ'_s) oracle.

In case of comparison between the program assertions before and after iterative oracle improvement, the highest mutation score increase was observed for subjects with no initial oracle (other than the implicit one): the implicit oracle is unable to react to the injected faults in most cases. Remarkably, for 72% of the classes with no initial oracle, the mutation score increased from 0% to 100%.

Table 4: RQ4: 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%	81.6%	1.0	$1.4 \cdot 10^{-5}$
	CC	8.3%	98.4%	90.1%	0.98	$2.2 \cdot 10^{-5}$
Inferred	CL	60.5%	98.8%	38.3%	0.9	$9.0 \cdot 10^{-3}$
	CM	50.2%	95.8%	45.6%	0.91	$4.7 \cdot 10^{-4}$
Manual	FE	78.8%	100%	21.2%	0.9	$6.3 \cdot 10^{-7}$
	LG	81.5%	100%	18.5%	0.89	$1.7 \cdot 10^{-2}$
All	**All**	**50.1%**	**98.4%**	**48.3%**	0.92	$< 2.2^{-16}$
Randoop	All	45%	98.4%	53.4%	0.93	$5.3 \cdot 10^{-7}$
EvoSuite	All	46.9%	98.4%	51.5%	0.95	$3.8 \cdot 10^{-6}$

A substantial increase in the mutation score was observed for subjects equipped with Daikon assertions. A smaller, still quite relevant, mutation score increase occurred for subjects coming with manually written JML contracts. While for 20% of the classes with JML contracts the mutation score did not change at all, for the remaining 80% of the classes oracle improvement contributed to a higher mutation killing capability.

The improved program assertions achieve 51.8% and 53.4% higher mutation score than the test case assertions generated by EvoSuite and Randoop respectively. The average number of program assertions in the subject classes is 20 and the average number of test case assertions is 18 in EvoSuite and 55 in Randoop. This shows that program assertions require the manual validation of a lower(Randoop) or comparable(EvoSuite) number of assertions but have a higher fault detection capability.

In all cases, the observed mutation score increase is statistically significant ($p \leq 0.05$) according to the Wilcoxon non-parametric (paired, two-tailed) statistical test (p-values are presented in Table 4). The Vargha-Delanay effect size $\hat{A}_{12}$ is always *large* (in our study, $\hat{A}_{12} \geq 0.89$).

RQ4: *The improved oracle has significantly higher mutation score than the implicit, the inferred (Daikon), the manual (JML) and test case oracles.*

4.4 Qualitative Analysis

4.4.1 Improvement of implicit oracle

Figure 4 (top) shows the source code of method `add()` from class `MapBackedSet`, taken from Apache Commons Collections. This method does not contain any assertions. To create assertions for it, we first run our tool with mutation analysis enabled, getting the output shown in Figure 5.

Let us consider the mutations in `test0()` and the assertions that should be added to detect them: (1) mutations 1, 4 and 7 lead to the change of the method's return value, so

the check for this value is necessary; (2) mutations 2, 5 and 8 show that we should check whether the given parameter was inserted into the map; (3) mutations 3 and 6 show that the relationships between the values of `size` and `map.size()` should be checked. Based on this analysis, we add the new assertion shown in Figure 4 (middle).

However, when we check the newly added assertion for false positives, we get a test case violating the assertion. By analysing the test case we can see that it adds elements with key equal to `null` into the map twice. As the map does not keep two values with the same key, the second inserted element replaces the first one, so the size of the map does not change and the assertion fails. Taking this situation into account, we improve our assertion as shown in Figure 4 (bottom) and the check for false positives confirms this improvement.

```
1   public boolean add(final E obj) {
2      final int size = map.size();
3       map.put(obj, dummyValue);
4       return map.size() != size; }
```

```
1   public boolean add(final E obj) {
2      final int size = map.size();
3       map.put(obj, dummyValue);
4       boolean result = map.size() != size;
5
6       assert (map.get(obj) == dummyValue) &&
7                  map.size() == size + 1 &&
8              (result == (map.size() != size)));
9
10     return result; }
```

```
6       assert (
7           map.get(obj) == dummyValue &&
8           result == (map.size() != size) &&
9           implication (result == true,
10                      map.size() == size + 1)));
```

Figure 4: Method add() with no assertions (top), with the assertion added at iteration 1 (middle) and with the final assertion (bottom)

4.4.2 *Improvement of inferred oracle*

Figure 6 (top) shows the source code of the `getSize()` method of class `Interval` from the Apache Commons Math library with the postconditions generated for it by Daikon. Following the described process, we first checked the given assertions for the existence of false positives. The output of the tool for this step is a test case calling the constructor of `Interval` with input parameters `(-1, -1)` and then calling `getSize`. Indeed, following the test case execution we can see that `result` = -1.0 - (-1.0) = 0, so it is greater than `old_upper` which has the value of -1.0. Hence, the 4th assertion (line 19) contains a false positive. Moreover, `result` = 0 also shows the existence of a false positive in the 3rd assertion (line 18), declaring that `result` cannot be zero.

After removing the two assertions with false positives, we ran the tool to check the remaining assertions for the existence of false negatives. The output of the tool for this step is in Figure 7. As we can see, it shows that if we replace the '-' sign in the code with either '+' or '∗', there is no assertion that reacts to this injected fault. To prevent this situation we add two new assertions that check the value of the result as follows: `assert (result == upper - lower)`, `assert (result >= 0)`. The new version of class `Interval` with improved oracle is shown in Figure 6 (with improved assertions at the bottom).

```
//Test case number: 0
/* 7 covered goals:
 * 1 add(Ljava/lang/Object;)Z:4 - ReplaceConstant -
     true -> false
 * 2 add(Ljava/lang/Object;)Z:3 - DeleteField:
     mapLjava/util/Map;
 * 3 add(Ljava/lang/Object;)Z:2 - DeleteField:
     mapLjava/util/Map;
 * 4 add(Ljava/lang/Object;)Z:4 -
     ReplaceComparisonOperator != -> ==
 * 5 add(Ljava/lang/Object;)Z:3 - DeleteStatement:
     put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/
     lang/Object;
 * 6 add(Ljava/lang/Object;)Z:2 - DeleteStatement:
     size()I
 * 7 add(Ljava/lang/Object;)Z:4 - DeleteStatement:
     size()I
 * 8 addAll(Ljava/util/Collection;)Z:3 - DeleteField:
     dummyValueLjava/lang/Object; */
@Test
public void test0() throws Throwable {
    HashMap<String, Object> hashMap0 = new
        HashMap<String, Object>();
    MapBackedSet<String, Integer> mapBackedSet0 =
    MapBackedSet.mapBackedSet((Map<String, ? super
        Integer>) hashMap0, (Integer) (-144));
    boolean boolean0 = mapBackedSet0.add("");
    assertEquals(true, boolean0);
}
```

Figure 5: FN detection for method add()

After this improvement, we start the next iteration, and the tool detects a false positive, which happens to be a true positive, i.e. a real bug of class `Interval`. The new assertion #6 (at line 19) is violated when the constructor of class `Interval` is called with input parameters 0.0, -1.0. In such a case the returned size of the interval is negative, while an interval size is supposed to be always non-negative. The bug has been reported to the Apache Commons Math developer community (bug report # MATH-1256) and was immediately fixed by the developers, by raising an exception inside the constructor of `Interval` when `upper < lower` (see Figure 6, middle).

In a similar way, we have detected two more bugs in Apache Commons Math. One involves five classes: `CanberraDistance`, `ChebyshevDistance`, `EarthMoversDistance`, `EuclideanDistance` and `ManhattanDistance`. Each of them contains a method to compute a distance between two arrays. If the length of the first array is greater than the length of the second, method `compute()` in all five classes gives an error (`ArrayIndexOutOfBoundsException`). Quite strangely, if the length of the second array is greater than the first, the method terminates silently. The bug was reported to developers (bug report # MATH-1258) and fixed.

The third bug is in class `Incrementor`. If an instance of this class is initialized with a negative number, its method `canIncrement` returns false, although the upper bound set in the class has not yet been reached (bug report # MATH-1259). The reported bug led to the discussion that the overall functionality of the class does not serve its purpose, so the solution was to replace the class `Incrementor` with a new class with the correct functionality, to deprecate `Incrementor`

```
1   public class Interval {
2
3      private final double lower;
4      private final double upper;
5
6      public Interval(double lower, double upper) {
7         this.lower = lower;
8         this.upper = upper;
9      }
10
11     public double getSize() {
12       double old_upper = upper;
13       double old_lower = lower;
14       double result = upper - lower;
15
16       assert (this.lower == old_lower); //1
17       assert (this.upper == old_upper); //2
18       assert (result != 0); //3: removed (FP)
19       assert (old_upper >= result); //4: removed (FP)
20
21       return result; } }
```

```
7     if (upper < lower) { // Fix for bug #MATH-1256
8        throw new NumberIsTooSmallException(
9           LocalizedFormats.ENDPOINTS_NOT_AN_INTERVAL,
10          upper, lower, true);
```

```
16       assert (this.lower == old_lower); //1
17       assert (this.upper == old_upper); //2
18       assert (result == upper-lower); //5: new (FN)
19       assert (result >= 0); //6: new (FN)
```

Figure 6: Method getSize() with Daikon assertions before (top) and after (bottom) oracle improvement; a real bug was reported and fixed (middle)

in Math 3.6 (so as to ensure backward compatibility for some time) and to remove it in Math 4.0.

4.5 Threats to Validity

Internal validity: The first author has been involved in a number of tasks carried out during the experiments. Specifically, she has developed the tool being evaluated and she has manually refined the oracles during the experiments, playing the role of the human in the loop. We carefully mitigated this validity threat by defining precise rules and procedures for oracle improvement to be followed by the human experimenter, prescribing what to do for each reported deficiency (e.g., for each EvoSuite mutation operator triggering a false negative). As a result, the human in the loop in our experiments has behaved largely deterministically and unimaginatively, as determined by these procedures. Moreover, to mitigate the single-annotator bias risk we followed a cross-checked-annotator approach, in which the first author's implementation of the protocol was cross-checked by another author. Developers properly trained on the usage of our tool and on the changes to apply for each oracle deficiency can be as efficient as the first author, but possibly even more effective, given their higher domain knowledge and freedom to improve the oracle.

External validity: We have validated our approach on a set of classes from five different subjects and with three different types of initial oracles. While we expect similar results to hold for other subjects, generalizability of our findings requires further replications on additional subjects.

```
//Test case number: 1
/* 2 covered goals:
 * 1 Strong Mutation 9:
     org.apache.commons.math4.geometry.euclidean.oned.
    Interval.getSize()D:14 -
        ReplaceArithmeticOperator - -> +
 * 2 Strong Mutation 11:
     org.apache.commons.math4.geometry.euclidean.oned.
    Interval.getSize()D:14 -
        ReplaceArithmeticOperator - -> * */
@Test
public void test1() throws Throwable {
    Interval interval0 = new Interval((-1.0), (-1.0));
    double double0 = interval0.getSize();
    assertEquals (double0, 0.0); }
```

Figure 7: FN detection for method getSize()

5. A FORMAL MODEL OF ORACLE IMPROVEMENT

We model our oracle improvement process using Shannon's information theory [27] and prove that every improvement step *can* make the information in the actual oracle closer to the information in the perfect oracle. These oracles are modelled as a pair of Boolean-valued random variables, $\alpha, G : R \rightarrow Bool$ respectively, assuming the set of reachable states at pp, R, is equipped with some probability distribution. With reference to the left diagram in Figure 1, we interpret the regions labelled a, b, c, d as probability masses: $a = p(\alpha = \mathrm{F}, G = \mathrm{F})$, $b = p(\alpha = \mathrm{T}, G = \mathrm{F})$, $c = p(\alpha = \mathrm{T}, G = \mathrm{T})$, and $d = p(\alpha = \mathrm{F}, G = \mathrm{T})$. A reduction, Δ, of false negative probability repartitions the probability weights to create a new oracle, α' where $a' = a + \Delta$, $b' = b - \Delta$. Similarly, reducing false positive probability by Γ creates a new oracle, α', where $c' = c + \Gamma$, $d' = d - \Gamma$. We can measure how closely connected two random variables (oracles) are by measuring their mutual information, a measure of their lack of independence [7]:

$$\mathcal{I}(X;\ Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y)\, log_2 \frac{p(x,y)}{p(x)\, p(y)}$$

When they are completely independent $\mathcal{I}(X;\ Y) = 0$ and when they are completely dependent they contain the same information.

We can define $\mathcal{I}(\alpha;\ G)$ in terms of a, b, c, d, getting:

$$\mathcal{I}(\alpha;\ G) = \begin{cases} -(b+c)log_2(b+c) - (a+d)log_2(a+d) \\ -(a+b)log_2(a+b) - (c+d)log_2(c+d) \\ +a\, log_2\, a + b\, log_2\, b + c\, log_2\, c + d\, log_2\, d \end{cases}$$

As before, we can always consider improvement steps Δ and Γ separately as they don't affect each other. We would like that each improvement step for α to α' always increases the mutual information between the actual and perfect oracles. Unfortunately this is not always true, but we can prove that it is always true whenever conditions on Δ and Γ are met. First, consider a step that improves false negatives.

THEOREM 1. *Let α, α' and G be Boolean-valued random variables modelling oracles (as above) and let α' be obtained from α via an improvement step Δ (as above). Then*

$$\Delta > \frac{bd - ac}{c + d} \ \Rightarrow\ \mathcal{I}(\alpha';\ G) \geq \mathcal{I}(\alpha;\ G)$$

Proof: *(sketch)* Rewrite $\mathcal{I}(\alpha';\ G)$ as a function of Δ with constants a, b, c, d.

$\mathcal{I}(\alpha';\ G) = -(b+c-\Delta)log_2(b+c-\Delta)-(a+d+\Delta)log_2(a+d+\Delta)-(a+b)log_2(a+b)-(c+d)log_2(c+d)+(a+\Delta)\,log_2\,(a+\Delta)+(b-\Delta)\,log_2\,(b-\Delta)+c\,log_2\,c+d\,log_2\,d$

Differentiate $\mathcal{I}(\alpha';\ G)$ with respect to Δ to find the single turning point and the region of monotonic increase. ■

We can immediately consider a step that improves false positives and obtain a very similar result.

COROLLARY 1. *Let α, α' and G be Boolean-valued random variables modelling oracles (as above) and let α' be obtained from α via an improvement step Γ (as above). Then*

$$\Gamma > \frac{bd - ac}{a + b} \ \Rightarrow\ \mathcal{I}(\alpha';\ G) \geq \mathcal{I}(\alpha;\ G)$$

Surprisingly, in spite of these limiting conditions on which improvement steps increase mutual information, we can guarantee that for every given oracle we can construct another oracle for which any improvement increases the mutual information.

COROLLARY 2. *For all actual oracles, α there is an oracle, α', constructed from α, such that for* all *improvement steps, Δ or Γ on α' that produce an oracle α'', $\mathcal{I}(\alpha'';\ G) \geq \mathcal{I}(\alpha';\ G)$.*

Proof: Consider an arbitrary oracle α. Associated with α is a partition of probability weights $\{a, b, c, d\}$. If $bd \leq ac$ then $bd - ac \leq 0$. As $\Delta > 0$ and $\Gamma > 0$ by assumption, application of either to produce a new oracle satisfies the appropriate condition of either Theorem 1 or Corollary 1 respectively. So $\alpha' = \alpha$. Otherwise, $ac < bd$. Since the oracle, α, corresponds to a logical assertion, $\langle assert \rangle$, let α' be the oracle corresponding to the negation of $\langle assert \rangle$, $!\langle assert \rangle$, in which case $a' = b, b' = a, c' = d, d' = c$ and we have $b'd' < a'c'$ and any improvement on this oracle satisfies the appropriate condition. ■

The proof of Corollary 2 makes clear some intuitions about assertion oracles. For a poor oracle the product of the probabilities of the inaccuracies is bigger than the product of the probabilities of its accuracies. However a poor oracle can be made into a good one by simply negating it. For a good oracle any improvement in its inaccuracy at all brings its information closer to that of the perfect oracle.

An oracle having high mutual information with the perfect oracle is one that agrees with the perfect oracle most of the time. This means it tends to accept/reject correct/-faulty program executions whenever the perfect oracle does so. Since the proposed oracle improvement process increases mutual information between actual and perfect oracles, it leads to an oracle which, in agreement with the perfect oracle, reveals all the faults it can reveal, while at the same time accepting all correct executions it should accept.

6. 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 [25, 29, 31]. 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. [4].

The work by Fraser and Zeller [12] presents an approach to generate parameterised unit tests for which oracles are represented in the form of pre- and postconditions characterising test input and test result. They use the definition of false positives and false negatives close to ours, but only to assess the accuracy of generated postconditions, without any guidance on further improvement. The later work of these authors [13] uses mutation testing to generate both test inputs and test oracles for standard JUnit test cases. However, test cases generated in this way express the observed behaviour of the program under test rather than the intended behaviour. Moreover, the focus is on test case oracles, not program oracles.

In their work, Staats, Gay and Heimdahl [28] propose a method supporting test oracle creation, which is based on the use of mutation analysis to rank variables in terms of fault-finding effectiveness. Similarly, Loyola et al. [22] propose a system that ranks program variables based on the interactions and dependencies observed between them during program execution. While these approaches provide a good basis for oracle variable selection, they do not provide any support for oracle assessment and improvement.

The work by Nguyen, Marchetto and Tonella [30] studies the training cost, false positive rate and fault finding capability of three types of automated oracles: Data Invariants, Temporal Invariants and Finite State Automata. 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) and the next step for the successful adoption of automated oracles should be the decrease in this false positive rate.

In their work Huo and Clause [19] measure the quality of the oracles in terms of the presence of brittle test case assertions and unused inputs. During their evaluation of 4,000 real test cases they were able to detect 164 tests containing brittle assertions and 1,618 tests containing unused inputs, with a quite high false positive rate. They considered test case, not program, assertions. The work by Schuler and Zeller [26] introduces the concept of checked coverage – the dynamic slice of covered statements that actually influence the oracle. The results of their study show that checked coverage is a better indicator of the quality of testing than coverage alone. However, no guidance is provided on how to improve the oracle quality.

The work by Zhang et al. [32] introduces *iDiscovery* which, similar to our approach, aims to improve the quality of the oracles iteratively using symbolic execution. However, it is applicable only to automatically inferred oracles and the level of improvement is limited by the initial set of candidate invariant templates. Hence, our approach can be applied to further improve the invariants produced by *iDiscovery*.

Other previous work has sought to reduce the manual oracle effort by reducing the number of test cases generated and by increasing their realism [1, 5, 17, 24].

7. CONCLUSION

We have proposed an iterative technique for the assessment and improvement of the oracles. Experimental results show that our tool is able to identify both false positives and false negatives in three important types of initial oracles (implicit, inferred and manual), leading to an average 48.6% improvement of mutation score over all the analysed classes and exposing real faults that have been reported to and fixed by the developers.

8. REFERENCES

[1] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *International Conference on Software Testing, Verification and Validation (ICST 2013)*, pages 352–361, March 2013.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *27th International Conference on Software Engineering (ICSE)*, pages 402–411, 2005.

[3] 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.

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

[5] M. Bozkurt and M. Harman. Automatically generating realistic test input from web services. In J. Z. Gao, X. Lu, M. Younas, and H. Zhu, editors, *IEEE 6th International Symposium on Service Oriented System Engineering (SOSE 2011)*, pages 13–24, Irvine, CA, USA, December 2011. IEEE.

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

[7] T. M. Cover and J. A. Thomas. *Elements of information theory, 2nd ed.* John Wiley & Sons, 2012.

[8] 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.

[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. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2015.

[12] 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.

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

[14] R. Geist, A. J. Offutt, and F. C. H. Jr. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, 41(5):550–558, 1992.

[15] 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.

[16] 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.

[17] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In 3^{rd} *International Workshop on Search-Based Software Testing (SBST 2010)*, Paris, France, April 2010.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] P. McMinn, M. Stevenson, and M. Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In 1^{st} *International Workshop on Software Test Output Validation (STOV 2010)*, pages 1–4, Trento, Italy, July 2010.

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

[26] 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.

[27] C. E. Shannon. A mathematical theory of information. *Bell System Technical Journal*, 27(3):379–423, 1948.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.