Oracle Assessment, Improvement and Placement

Gunel Jahangirova

A report submitted in fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

April 16, 2019

Declaration

I, Gunel Jahangirova, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work. Some of the work presented in this thesis has been previously published in the following papers:

- Gunel Jahangirova, David Clark, Mark Harman and Paolo Tonella "Oracle Assessment and Improvement", in *Proceedings of the 25th International Symposium on Software Testing and Analysis* (ISSTA 2016), pages 247-258, 2016. This paper contains parts of Chapter 3. The author of the thesis has contributed to the main idea of the paper, to the theoretical foundations described in it, to the paper's writing and conducted all the experiments for empirical evaluation.

- Gunel Jahangirova "Oracle Problem in Software Testing", in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2017), pages 444-447, 2017. This is a doctoral symposium paper and it contains parts of Chapter 3, Chapter 4 and Chapter 5.

- Gunel Jahangirova, David Clark, Mark Harman and Paolo Tonella. OASIs: Oracle Assessment and Improvement Tool. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2018), pages 368-371, 2018. This paper contains parts of Chapter 3. The author of the thesis has contributed to the main idea of the paper and to its writing, and has implemented the tool that the paper describes.

Chapter 5 of the thesis has been submitted and is under review as the following paper:

- Gunel Jahangirova, David Clark, Mark Harman and Paolo Tonella "Oracle Improvement Process: Formal Model and Empirical Evaluation". The author of the thesis has contributed to the main idea of the paper, to the theoretical foundations described in it, to the paper's writing and conducted all the experiments with humans for empirical evaluation.

Abstract

The oracle problem remains one of the key challenges in software testing, for which little automated support has been developed so far. This thesis analyses the prevalence of failed error propagation in programs with real faults to address the oracle placement problem and introduces an approach for iterative assessment and improvement of the oracles.

To analyse failed error propagation in programs with real faults, we have conducted an empirical study, considering Defects4J, a benchmark of Java programs, of which we used all 6 projects available, 384 real bugs and 528 methods fixed to correct such bugs. The results indicate that the prevalence of failed error propagation is negligible. Moreover, the results on real faults differ from the results on mutants, indicating that if failed error propagation is taken into account, mutants are not a good surrogate of real faults. When measuring failed error propagation, for each method we use the strongest possible oracle as postcondition, which checks all externally observable program variables. The low prevalence of failed error propagation is caused by the presence of such a strong oracle, which usually is not available in practice. Therefore, there is a need for a technique to assess and improve existing weaker oracles.

We propose a technique for assessing and improving test oracles, which necessarily places the human tester in the loop and is based on reducing the incidence of both false positives and false negatives. A proof showing that this approach results in an increase in the mutual information between the actual and perfect oracles is provided. The application of the approach to five real-world subjects shows that the fault detection rate of the oracles after improvement increases, on average, by 48.6%. The further evaluation with 39 participants assessed the ability of humans to detect false positives and false negatives manually, without any tool support. The correct classification rate achieved by humans in this case is poor (29%) indicating how helpful our automated approach can be for developers. The comparison of humans' ability to improve oracles with and without the tool in a study with 29 other participants also empirically validates the effectiveness of the approach.

Impact Statement

The research work presented in this thesis is beneficial to the improvement of software quality and could have impacts on both software engineering research and industry.

Testing is the most common activity performed to validate software systems. In testing, the test oracle is the artifact that checks the validity of the obtained results, i.e. determines whether a software under test executes correctly. The effectiveness of the testing process is strongly dependent on the choice of test oracle.

The oracle problem is a well-known problem in both research and industry. However, while in most of the research literature there is an assumption that oracles are available, the applicable oracles are not described. In the current industrial practice of software testing, the oracle is often a human being. As the oracles manually generated by humans are costly and unreliable, there is a need for techniques to support developers in this task to ensure high testing quality while reducing the testing costs.

This thesis proposes an iterative approach for oracle assessment and improvement which places the developer in the loop of the process. We conducted experiments with real developers who had years of industrial experience to evaluate our approach. The results of our experiments show that developers using our tool achieve a higher quality oracles than the developers improving the oracles manually.

Testing is effective when it uncovers faults in the code. One of the reasons of hidden faults is the presence of failed error propagation, a case when the faulty statement is executed, the program transitions into an infectious state, but without propagating to the output. The majority of existing studies have analysed failed error propagation on programs with synthetic faults. In contrast, our empirical study used Java programs with real faults. Therefore, our results are indicative of real world scenarios and are more meaningful for the industrial community. The implications of our empirical study (presented in Section 3.5) have relevant suggestions for both practitioners and researchers.

A part of research in this thesis has already been published in academic papers at a high-level software engineering venue. 2 more papers composed of the later studies reported in the thesis have been recently submitted to academic journals. In addition to the results and observations presented in this thesis, the academic community will also benefit from the implemented tools and produced datasets, which are publicly available[1].

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

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisors Paolo Tonella, David Clark and Mark Harman for being the best supervisors one could hope for. Their academic advice, guidance, patience and understanding made this experience exciting and enjoyable throughout. In particular, I would like to thank Paolo Tonella for his exemplary strong work ethics, extremely nice personality and contagious passion for research. All of these made it a true pleasure to work with him and provided an endless amount of inspiration. I am very grateful to David Clark who took the challenge of supervising me in a distant mode (and did it excellently), visited me in Trento many times, and has been a true friend through all these years. I am also very grateful to my second supervisor Mark Harman for all the valuable comments, constructive criticism and continuous encouragement.

I would like to acknowledge Fondazione Bruno Kessler (FBK) for providing the scholarship which made this PhD possible. I want to thank all the members of Software Engineering group at FBK for being the nicest colleagues possible. Especially, I would like to express my warmest appreciation to dear Angelo Susi, discussions with whom were always insightful, but most importantly made me laugh a lot.

No words are enough to express my gratitude to Nargiz Humbatova and Olesya Razuvayevskaya who have been my 24/7 emotional support system for the whole duration of my PhD. They listened, and listened, and listened. Their patience, kindness and true friendship were invaluable and gave me a lot of strength.

Last but not least, I would like to thank my dear family members. My sister Leyla was always there for me and did all she could to reduce the bitterness of living far away from my family. My dear father and mother have always made me feel their continuing love and support. I should especially thank my mother who taught me what a privilege access to education is and did everything possible so that I always get the best opportunities in life.

Contents

List of Figures

List of Tables

1 Introduction

Testing is an essential activity in software engineering. Conceptually, it is a simple process: we run the software using some specified input, observe the software's execution, and decide if the execution appears to be correct. However, testing embraces a variety of activities, techniques and actors, and poses many complex challenges, especially with the pervasiveness and criticality of software growing ceaselessly. Several artifacts are involved in the testing process, including the set of *test inputs* to be run, the *test oracle*, or method for determining the correctness of the software, the *software* or program to be tested, and the *specification* the software is intended to implement [96].

Testing research, however, is predominately focused on determining what test data to use, e.g., creation and evaluation of test coverage criteria or automatic test generation tools. But no matter what coverage criterion is used, we need to know whether a given program executes correctly on a given input, as a test execution for which we are not able to discriminate between success or failure is useless. This corresponds to the so-called "*oracle*", ideally, a magical method that provides the expected outputs for all possible test cases; more often, a hardcoded assertion that can emit a pass/fail verdict over the observed test outputs [13].

In the absence of oracles, only "general" properties such as null pointer dereferencing, array bound errors, and program crashes can be checked. The criticality of the *oracle problem*, i.e. the problem of determining the correctness of a program's behaviour when tested [77], has been very early raised in the literature [105, 25, 86]. However, little attention has been paid to it in research and in practice few alternatives still exist to manually checking the program's output.

If the program under test has been developed following design-for-test principles, there will be a detailed, and possibly formal, specification of intended behaviour. In these situations, there is an automatable test oracle to which a testing tool can refer to check outputs. In case a full specification for the program under test does not exist, one may construct a partial test oracle that can check outputs for some inputs.

The so-called *sampling oracle* approach [45], which selects set of values based on some criteria, can be applied in this case. Boundary values, mid-points, minima and maxima are examples often chosen when testing. Once such values are selected, an oracle that provides the expected results for each of them should be created.

Another approach is the use of *metamorphic testing* [60, 19], a testing approach that uses metamorphic relations, i.e. the properties of the software under test represented in the form of relations among inputs and outputs of multiple executions. If exact results for a few inputs are available, these metamorphic relations can be used to do checks for the other inputs. The availability of simple relations is a key factor for the applicability of this approach.

When direct verification is not applicable, *redundant computations* [3] or *pseudo-oracle* [25, 106], i.e. testing one implementation against another, can be used instead. The second implementation could be performed by another development team or using another algorithm. For example, in case of searching algorithms, a binary search program could easily be tested by comparing the result with a linear search. In industry, this technique is often applied in regression testing, where the current version of the program is tested against its previous release to test the parts of the software which should not have been changed.

Despite the variety of test oracles, there is a common process that consists of few main steps that can be used to characterise and classify the different kinds of oracles [83]: (1) identifying the source of information for deriving the oracle, (2) recognising the program behaviour to be checked, (3) translating the source of information and the program behaviour into forms that can be checked against each other, and (4) executing the oracle. Strictly speaking, any test oracle needs some kind of human effort, since oracles rely on information about the expected behaviour of the system. Even if we assume the availability of full formal specifications, they should be created by the software designers. In case of sampling oracle, the expected output should be generated for each input in the sampled set. For the heuristic oracle, metamorphic relationships should be identified by the developers. To be able to perform redundant computations, the alternative implementation of the program by other developers should be available.

However, for many systems and most testing as currently practiced in industry, the tester does not have the luxury of formal specifications. Therefore, many organisations today depend on a *human oracle*. As a result, the tester faces the daunting task of manually creating oracles or manually improving any available partial ones. These are expensive and error-prone tasks. Unfortunately, methods for supporting humans in performing them are not common. Therefore, to achieve better quality of testing, we need a concerted effort to find ways to support developers in writing their oracles and in improving the already existing weak oracles.

1.1 Problem Statement

Oracle performance depends on two properties: completeness and soundness. *Completeness* means that all correct program states are accepted by the oracle and it raises an alarm only for faulty states, therefore it has no false alarms (i.e. no false positives). *Soundness* means that all faulty program states are rejected by the oracle, so there are no missed faults (i.e. no false negatives). Indeed, we don't want that test failures pass undetected, but on the other side we don't want either to be notified of many false positives, which waste important resources.

One of the widely-attributed sources of failures passing undetected (presence of false negatives) is driven by the possibility of *failed error propagation* (FEP): a fault may corrupt the program's internal state, yet this corruption fails to propagate to any point at which it is observed. A large amount of work [5, 112, 104, 59, 66, 73, 114, 67, 7] analyses the prevalence of failed error propagation in different subject programs by introducing faults into programs using mutants or seeded faults, to *simulate* real faults. While the reported rate of failed error propagation varies in each study, it is always significant.

Traditional test oracles are defined only on the *outputs* of test executions. The occurrence of failed error propagation in presence of such oracles can be caused by two different scenarios. In the first scenario, the fault actually affects the output of the test execution, but the existing output oracle is not strong enough and therefore can not observe a corrupted execution state, hence failing to report an error. To address these scenario there is a need in technique that will assess the existing weak oracle and provide support to the developer for its improvement.

In the second scenario, the fault does not affect the output of the execution, which implies that there is a need for *internal oracles* that will check the inner states of the program. A benefit of checking values internally would be knowing as soon as possible whether the program has entered into an erroneous state. This raises an *oracle placement problem*, i.e. the problem of finding the subset of program points that has the minimum size and that maximises the fault exposure probability of the oracles placed at the selected program points. The selection of optimal placement points for oracles has not been thoroughly investigated so far (with the exception of the preliminary idea described in a short ESEC/FSE-NIER paper [111]).

The recent research on oracle problem focuses on the automated generation of test case assertions [30, 78] and dynamic program invariants [27]. However, these synthesised assertions and program invariants are not oracles because they encode the observed behaviour observed of the program under test rather than the intended behaviour. To use them as test oracles, it is necessary to identify the incorrect ones (i.e. to perform *oracle assessment*) and then fix the them (i.e. to perform *oracle improvement*). This process requires human intervention, as to perform these actions it is necessary to understand what the system is supposed to do. While the quality of human input is crucial in these cases, only two works [81, 95] have studied the performance of the humans in the oracle assessment and improvement process. The first study

[81] uses CrowdSourcing to verify test case assertions, while in the second one [95] developers assess whether invariants generated by Daikon [27] are correct or incorrect. The results of the studies contradict each other: the second study indicates that human testers are not good at identifying correct test oracles, while the first one indicates that human testers can reliably identify correct test oracles and fix incorrect ones.

The work by Nguyen et al. [99] analyses mined specifications such as data invariants, temporal invariants and finite state automata, and demonstrates that they have a high false positive rate (around 47%). There are a few works that propose metrics to assess existing oracles, such as *checked coverage* [89] or the presence of *unused inputs* and *brittle assertions* [47]. While these metrics are indicative of the oracle's quality, they do not support the developer in the oracle improvement process.

Overall, developers face the daunting task of ensuring that the oracles they use in the testing process are complete and sound. To support them in this process, a further investigation of oracle placement and oracle strength is required, so as to ensure that failures do not pass unnoticed. An approach that would automatically assess oracles, i.e. detect false positives and false negatives in them, and will guide the developers in the oracle improvement process should be created.

1.2 Objectives

The objectives of this thesis are as follows:

- Analyse failed error propagation in methods with real faults and in methods where faults are introduced by mutations. What is the prevalence of failed error propagation? Is its occurrence dependent on the nature of the faults used? Is there a need of internal oracles to prevent failed error propagation? Assuming postconditions with optimal strength are available, can they prevent failed error propagation?

- Define a theoretical framework to assess oracle quality and formalise the oracle improvement process.
- Investigate methods to identify false positives and false negatives in oracles. How can we test an oracle to check if it can expose all the faults it is supposed to expose? How can we generate counterexamples showing that an oracle is violated in cases where it is supposed to hold?
- Analyse whether developers are good in detecting false positives and false negatives in the oracles manually.
- Analyse whether developers are good in improving oracles with false positives and false negatives manually.
- Empirically investigate whether the identified methods for false positive and false negative detection lead to the creation of better oracles and whether the generated counterexamples are helpful for developers in the oracle assessment and improvement process.

1.3 Structure of the Thesis

The remainder of the thesis is organised as follows:

Chapter 2 provides a comprehensive review of the literature that is most relevant to this thesis. The chapter starts with a summary of the early works that introduced and defined the term ”oracle”. It then reviews the previous research related to automated oracles taking the form of test case assertions and program invariants. Finally, it discusses the works related to oracle placement problem, such as the PIE framework and the existing studies on failed error propagation.

Chapter 3 presents our empirical study which analyses failed error propagation in Java programs with real faults. We describe our experimental procedure, where we measure different types of failed error propagation in methods with real faults. We compare this results against these results obtained from the methods where faults are injected by mutation operators. We also compare unit-level failed error propagation to the system-level one. We report the results of qualitative analysis on the nature of FEP in case of real faults and mutations and provide the implications of this work for testing research.

Chapter 4 introduces our proposed approach for oracle improvement and assessment. We present theoretical definitions of oracle quality, soundness, completeness, false positives and false negatives. Then we describe our iterative improvement process, the approach we use to identify false positives and false negatives and the implementation of the approach in form of a tool called OASIs. Finally, we present a formal model of oracle improvement using information theory.

Chapter 5 describes our extensive evaluation of the oracle assessment and improvement approach. First, we describe the evaluation on 5 different subjects and 3 different types of initial oracles, where the role of human in the loop was played by the author of the thesis. We report the number of iterations required to improve all three types of initial assertions and the increase in fault detection as a result of this improvement. We also compare the fault detection of oracles improved using our approach to the fault detection of test case assertions generated by automated test case generators. Then, we provide details on the second part of the evaluation which assessed the ability of humans to detect false positives and false negatives manually (without using OASIs). 39 participants including both students and professional developers were involved in this study. We report users' correct classification rate and analyse parameters affecting their performance. We also provide information on which type of oracle deficiencies is harder for them to detect and what are the most commonly occurring misclassification types in the assessment process. The last part of the evaluation, which involved 19 participants, compares the improvement of the oracles using our approach with the manual improvement. Here, the metric of comparison is the quality of the final improved assertions using each approach. Moreover, the characteristics of the iterative process such as number of iterations, detected oracle deficiencies, time spent on each iteration, are reported for each participant who played the role of the human in the loop.

Chapter 6 provides the conclusions derived from the work presented in this thesis and the plans for future work.

2 Literature Review

Studied since the late 1970s, the research literature on test oracles is a relatively small part of the research literature on software testing. However, in recent years test oracles and techniques to automatically generate test oracles have attracted a lot of attention and have witnessed an impressive growth.

This chapter reviews the work related to the oracle problem in software testing and most relevant to the thesis. First, the existing surveys on oracles are summarised. Then, an overview of the definitions of oracles and the attempts at formalisation of the oracle problem are provided. The review continues with the existing research on the two most widely used forms of oracles: *test case assertions* and *specifications*, considering in particular automated specification mining. Finally, it discusses the existing works on *failed error propagation* and the *oracle placement problem*.

2.1 Surveys on Oracle Problem

Five large surveys on topics related to oracles have been conducted till now. In 2001, Baresi and Young [8] presented a survey where they have grouped oracle systems based on implementation approaches (e.g., embedded assertions, execution log analyzers) and on the kinds of specifications they accept (e.g., interface specifications, design models, property- and model-based specifications of externally visible behavior). The main highlights of the paper are that (1) there is a need to bridge the gap between the concrete entities and specification entities when oracles are based on more abstract descriptions of program behavior; (2) oracle systems are usually ”partial”, i.e. they reject only some of incorrect behaviors; (3) while in an ideal oracle system, oracles would be orthogonal to test case selection, in reality it is more practical to determine acceptable behaviors for limited classes of test cases.

In 2009, Shahamiri et al. [90] performed a comparative analysis among six categories of test oracles: N-Version Diverse Systems and M-Model Program Testing; Decision Table; IFN (Info Fuzzy Network) Regression Tester; AI

(Artificial Intelligence) Planner Test Oracle; ANN (Artificial Neural Network) - Based Test Oracle; and Input/output Analysis Based Automatic Expected Output Generator. The authors compare these approaches in terms of their limitations and capabilities to automate oracle activities. The study concludes with two important messages. First, there are no existing techniques to completely automate the oracle process in non-regression testing with reasonable cost and reliability. Second, there is still no unique approach to automate all different kinds of oracle activities in any possible circumstances.

Oliveira et al. [76] used evidence from a pool of about 300 studies directly related to test oracles and presented a classification of test oracles based on a taxonomy that considers their source of information and notations. Based on this classification, they performed a quantitative analysis to highlight the shifts in the evolution of research on test oracles. Exploring geographical and quantitative information, they analysed the maturity of this field using co-authorship networks among published studies. Further, they determined the most prolific authors and their countries, main conferences and journals, supporting tools, academic efforts, and conducted a comparative analysis between academia and industry.

The survey by Pezze and Zhang[83] focuses on test oracles with particular attention to their automation. First, the survey presents the timeline showing main milestones in the evolution of the research on test oracles in chronological order. Then the authors identify main steps to characterise the different kinds of oracles: (1) identifying the source of information for deriving the oracle, (2) recognising the program behavior to be checked, (3) translating the source of information and the program behavior into forms that can be checked against each other, and (4) executing the oracle. Based on these steps, test oracles are classified according to the required information and different forms of checkable oracles (i.e. oracles expressed in a form directly checkable during the system execution). The conclusion of the survey highlights that the precision of automatically generated oracles depends on the information used for the generation. The role of the human is specifically underlined by stating that "the increasing availability of techniques to generate automatic or semi-automatic oracles may change the role of humans who may be required to provide different forms of information to increase the effectiveness of automatic generation techniques and the precision of automated oracles".

Barr et al. [9] have constructed a repository of 694 publications on test oracles and related areas. They analysed research trends on this topic by dividing oracles into four categories: when test oracles can be *specified*, when test oracles can be *derived*, when they can be built from *implicit information* and when there is *no automatable* oracle available. The results of the survey show that test oracles are difficult to construct, so oracle reuse is an important problem that merits attention, and while some work has begun on using test oracles as the measure of how well the program has been tested, more work is needed in this area.

Overall, these surveys provide wide range of structured information on different oracle taxonomies, multiple aspects of oracle automation and detailed summarisation of research trends. Moreover, they give an insight into the existing challenges in automated oracle generation, underlining that constructing oracles, defining/improving the precision of these oracles and investigating the role of humans in the automation process are important future research directions.

2.2 Formalisation of Oracles

The term "test oracle" was first introduced in William Howden's seminal work in 1978 [105] and is defined as the mechanism "that can be used to define the correctness of test output". To clarify the definition, the author notes that the most common test oracle is the comparision of *output variables* or *traces of selected program variables* for a given set of inputs. This process can be formally or informally defined. *Formally defined oracles* may consist of tables of values, algorithms for hand computation or formulae in the predicate calculus. *Informally defined oracles* are often simply the ability of the programmer to recognize correct output. In more theoretical terms, the author describes a test oracle for a program P as a source of information about a *hypothetical correct program* P^*.

In 1982 Weyuker and Davis [25][106] introduced the term "*oracle assumption*" - the belief that tester is routinely able to determine whether or not the test output is correct. They investigate the reasonableness of this assumption and conclude that in practice quite often it is impossible to define a complete and totally reliable oracle for all SUTs, thus introducing the notion of "non-testable program". Given the impossibility of defining an ideal oracle, the authors discuss two possible options: "*pseudo-oracles*" and "*partial oracles*". A pseudo-oracle is an independently produced program intended to fulfill the same specification as the original program. The two programs, which are to be produced in parallel by totally independent programming teams, are run on identical sets of input data, and the results are compared. The partial oracle is available in the cases when the tester is not in the possession of ideal oracle, but is not completely unaware of what the answer is. Frequently the tester is able to state with assurance that a result is incorrect without actually knowing the correct answer.

In 1992 Richardson et al. [86][85] defined test oracle as a mechanism that has two components: the *oracle information* specifies what constitutes correct behavior, while *oracle procedure* verifies test execution results with respect to the corresponding oracle information.

The work by Hoffman [45][46] is one of the first to recognise the complex nature of oracles. It introduces a list of the main characteristics of oracles that might be measured when relating an oracle to the Software Under Test (SUT): *completeness* of information, *accuracy* of information, usability, maintainability, complexity, temporal relations and cost. The accuracy of information of an oracle corresponds to the types of errors it might produce: miss actual wrong value and/or flag correct data as an error.

In 2011, Staats et al. [96] proposed a theoretical analysis that included test oracles in a revisitation of the fundamentals of testing. They extend Gourlay's

[37] approach and define a testing system as a collection $(P, S, T, O, corr, corr_t)$ where:

- S is a set of specifications
- P is a set of programs
- T is a set of tests
- O is a set of oracles
- $corr \subseteq P \times S$
- $corr_t \subseteq T \times P \times S$

Here, the predicate *corr* implies that p is correct with respect to s for $p \subseteq P$, $s \subseteq S$. Of course, the value of *corr(p, s)* is generally not known, so this predicate is just theoretical and used to explore how testing relates to correctness. The predicate $corr_t \subseteq T \times P \times S$ defines correctness with respect to a test $t \subseteq T$ and holds if and only if the specifications holds for program p when running test t. Using $corr_t$, authors introduce the definition of *complete*, *sound* and *perfect* oracle.

Moreover, they introduce two *oracle comparison metrics*: *power* and *PROBBETTER*. The first metric states that an oracle o_1 has a power greater than oracle o_2 with respect to a test set *TS* (written $o_1 \geq_{TS} o_2$) for program p and specification s if: $\forall t \in TS, o_1(t, p) \implies o_2(t, p)$. In other words, if o_1 fails to detect a fault for some test, then so does o_2. Oracle o_1 is stated to be more powerful than o_2 for test set *TS* ($o_1 >_{TS} o_2$) if: $\forall t \in TS, o_1(t, p) \implies o_2(t, p) \wedge \exists t' \in TS, \neg o_1(t', p) \wedge o_2(t', p)$. In other words, $o_1 \geq_{TS} o_2$ and for some test $t' \in TS$, o_1 detects a fault where o_2 fails to detect a fault. The *PROBBETTER* (PB) metric provides probabilistic comparison of two oracles. Thus, oracle o_1 is PB than oracle o_2 with respect to a test set TS, written as $o_1 PB_{TS} o_2$, for program p if for a randomly selected test $t \subseteq T$, o_1 is more likely to detect a fault than o_2. An oracle o_1 is universally PB than o_2 if $o_1 PB_T o_2$, where T is the entire set of tests that can be run against p.

Another formalisation of test oracle is in the previously mentioned survey by Barr et al. [9], which contains a section that presents definitions to establish a lingua franca in which to examine the literature on oracles. These definitions of test oracle and probabilistic test oracle are provided to avoid ambiguity throughout the survey. The authors also introduce the notion of *ground truth* and then define *soundness* and *completeness* of test oracle with respect to the notion of ground truth.

In general, only few attempts were made to formalise oracles and locate them in the overall theoretical framework for testing. However, no theoretical framework is defined for oracle quality and for the oracle improvement process.

2.3 Automatically Generating Oracles

The current research in automated oracles can be classified according to the two forms of oracle being considered: *test case assertions* and *mined specifications*. Synthesized test case assertions are generated by automated test case generation tools. However, as they encode observed behaviour, they need human input to be used as oracles. This human input is provided in different ways in different approaches: crowdsourcing [81] , manually written test cases [80], JavaDoc documentation [36]. The work that present tools to assess the quality of oracles analyse the quality of test case assertions using metrics, such as checked coverage [89] or the presence of brittle assertions and unused inputs [47]. Another group of tools support the construction of oracles by identifying the variables with the highest fault-detection capability.

Specification mining tools produce invariants which are used as oracles. These invariants might be incorrect, as they are generated from source code, so they capture the implemented, not the intended, behaviour. Thus, they also require human intervention.

Overall, without human intervention the synthesized test case assertions and mined specifications are not able to detect any faults related to the implemented functionality and are useful mostly for regression testing. In case human input is available, its quality is of crucial value for the proposed approaches to work. In fact, two existing studies respectively using CrowdSourcing [81] to verify test case assertions and using developers to determine user classification effectiveness for invariants [95] contradict each other. The second study indicates that human testers are not good at identifying correct test oracles, while the first one indicates that human testers can reliably identify correct test oracles and fix incorrect ones. This shows that there is a need of more experiments analysing the performance of human testers in the oracle improvement process.

2.3.1 Test Case Assertions

Generation of Synthesized Test Case Assertions

Automated test oracles in the form of test case assertions are implemented as part of modern test case generators such as EvoSuite [29, 30] and Randoop [78]. These tools have the capability to synthesise test cases that include assertions.

Randoop [78] allows annotation of source code to identify observer methods to be used for assertion generation. It classifies generated test suites as error-revealing or expected behaviour. The error-revealing tests show that the code violates its specification or contract. By default, Randoop checks some general contracts on Java object's `equals`, `hashcode`, `toString`, `clone` methods. The expected behaviour test suite contains the test cases with assertions reflecting the current behaviour of the program under test. While the error-revealing test suite is able to find simple errors in the current implementation, the expected behaviour one can be useful only for regression testing to find errors in future implementations.

In EvoSuite [29, 30] mutation-driven generation of oracles is used. This was originally developed as part of the tool μTest [33], which is now a component of EvoSuite. A test case detects a mutant only if it there is a test case assertion that can identify misbehaviour that distinguishes the mutant from the original program. To generate such assertions for a test case, the test case should be run against the original program and all mutants, using observers to record the necessary information. After the execution, the traces generated by the observers are analysed for differences between the runs on the original program and its mutants, and for each difference an assertion is added. Then the number of assertions is minimised by tracing for each assertion which mutation it kills, and finding a subset for each test case that is sufficient to detect all mutations that can be detected with this test case.

The test case oracles generated by Randoop and EvoSuite are specific for a single run, which makes them hard to understand because of the information that is specific to that run and is brittle with respect to future code changes. To overcome this problem, Fraser and Arcuri [32] present a novel approach which converts the method sequence in traditional test cases into *parameterised unit tests* (PUT) - unit tests containing symbolic pre- and postconditions characterising test input and test result. The process starts with an automatically generated concrete method sequence. Such a concrete method sequence has a very precise but implicit precondition; this precondition is encoded in the input objects and the setup performed on the unit under test. Similarly, the postcondition can be interpreted as the observable state after the test execution. These conditions are made explicit by determining all the conditions that hold for the given states. For example, all objects are compared with each other, all observer methods are observed, and so on. The resulting conditions overspecify the test case, therefore the approach tries to get rid of as many conditions as possible. For this, new tests are iteratively generated and executed on the original program and versions with seeded defects, thus effectively filtering irrelevant preconditions and postconditions. At the end of the process, we get a parameterised unit test that only contains the test statements, the relevant preconditions on the inputs, and an effective test oracle. The evaluation on 5 subjects shows that PUTs are more expressive, retain only 57% of the original statements and cover 72.6% more branches than the original concrete unit test. However, they are more expensive, requiring several minutes per test case generation, have 19.6% false negative rate and 8.3% false positive rate.

Human Input for Synthesized Test Case Assertions

Automatic synthesis of test assertions is an initial step towards automatic generation of oracles. However, the synthesised assertions are not oracles because they encode the behavior *observed* by executing the test case instead of the *intended* behavior. To turn synthesised assertions into oracles it is necessary to identify and fix the incorrect assertions, which can hardly be automated as it requires human intelligence. Oracles encode the intended behavior of the software system, so they must be provided by a human or generated from human-provided information such as a formal specification.

One approach to deal with this problem is to use the idea of CrowdSourcing. CrowdSourcing a problem consists of specifying it in the form of a Human Intelligence Task (HIT) and making the problem available on a CrowdSourcing platform, where registered workers can choose to complete HITs for a small remuneration. Pastore, Mariani and Fraser [81] 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 in order for the crowd to be able to determine right and wrong assertions.

The works by McMinn et al. [72] and Afshan et al. [1] argue that one source of human oracle cost is the inherent unreadability of machine-generated test inputs, which makes test cases hard to comprehend and time-consuming to check. The authors propose methods to extract knowledge from programmers, source code and documentation and to incorporate it into the automatic test data generation process to make produced test cases more realistic. The later work by McMinn et al. [2] focuses specifically on automatically generated string inputs. The authors present an approach in which they incorporate a natural language model into a search-based input data generation process with the aim of improving the readability of generated strings. They evaluate their approach by conducting a human study with participants recruited from CrowdFlower[2] crowdsourcing platform. The results show that 10 out of 17 test inputs generated using the proposed technique, the participants recorded significantly faster times when evaluating inputs produced using the language model, with medium to large effect sizes 60% of the time.

The approach proposed by Pastore and Mariani [80] to identifying the incorrectly synthesized assertions uses the manually written test cases as the source of human knowledge about the system. They presented a tool *ZoomIn* which pinpoints the wrong assertions by comparing the executions produced by the manual test cases to the executions produced by the automatically generated test cases at two abstraction levels simultaneously. The first level is code coverage, that is *ZoomIn* compares the statements covered by manual and automatic tests. The second level is program variables, where *ZoomIn* uses Daikon [27] to generate constraints about the values that can be legally assigned to program variables when the manual tests are executed. These two levels are combined according to the following intuition: 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. In practice, it is assumed that an automatic test case that follows a path similar to one covered by a manual test case while generating anomalous variable values is an automatic test case that reveals a failure by covering a special untested case of an already tested functionality. For the purpose of evaluation Apache Commons Math library and 7 real faults from it were selected and *ZoomIn* was applied to the test cases generated by EvoSuite. The results show that *ZoomIn* has been able to detect 50% of the analysed non-crashing faults requiring inspection of less than 1.5% of the automatically generated assertions. However, the process has its limitations: it requires the existence of manual tests and the output of the tool is directly dependent on the quality of the manual tests. Also, the empirical results are based only on one subject

[2]http://www.crowdflower.com

program and a small number of real bugs, which shows that results might be inapplicable to other types of subject programs.

The work by Goffi et al. [36] automatically generates oracles from human-written documentation, such as Javadoc comments. They implemented the tool Toradocu that consists of Javadoc extractor, condition translator and oracle generator. The *Javadoc extractor* identifies all the Javadoc comments that are related to exceptional behaviors. The *condition translator* translates each natural-language condition into Java boolean expressions. The *oracle generator* produces test oracles in the form of assertions and embeds them in the provided test cases. 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 reduces EvoSuite's false positives by 33%. To evaluate the study they have conducted a human study with

The work by Blasi et al. [17] introduces JDoctor, which extends Toradocu so that it produces specifications not only for exceptional behaviors, but also for preconditions and normal postconditions. Moreover, JDoctor adds a novel notion of semantic similarities. This handles comments that use terms that differ, despite being semantically related, from identifiers in code. In an empirical evaluation, JDoctor achieved precision of 92% and recall of 83% in translating Javadoc into procedure specifications. The JDoctor-derived specifications were also supplied to an automated test case generation tool, Randoop. The results show that the specifications enabled Randoop to generate test cases that produce fewer false alarms and reveal more defects.

Tools to Assess Quality of Test Case Assertions

In the work by Huo and Clause [47] the quality of the oracles is measured in terms of the presence of *brittle assertions* and *unused inputs*. The technique is based on dynamic tainting and works by tracking the flow of controlled and uncontrolled inputs along data- and control- dependencies at runtime. Intuitively, controlled inputs are inputs explicitly provided by the test itself (e.g., constants that appear in the test method) and all other inputs are considered uncontrolled. When a test finishes execution, the technique uses the tracked information 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 an assertion). These reports are then filtered to remove false positives and presented to testers. The experimental results on 4,000 real test cases showed that the proposed technique is able to detect 164 tests containing brittle assertions and 1,618 tests containing unused inputs.

The work by Schuler and Zeller [89] addresses the problem of traditional test coverage metrics not assessing the oracle quality and introduces the concept of *checked coverage* - the dynamic slice of covered statements that actually influence the oracle. The evaluation on 7 Java open-source projects showed that, for all the projects, checked coverage is lower than regular coverage, with an average difference of 24%. Furthermore, they measured how the proposed technique is sensitive to oracle decay - that is, how oracle quality is artificially reduced by removing checks. The results show that while all quality metrics decrease with oracle decay, checked coverage is more sensitive to missing assertions.

Tools to Support Construction of Test Case Assertions

Staats, Gay and Heimdahl [94] proposed a method supporting test oracle creation, which is based on the use of mutation analysis to rank variables in terms of fault-finding effectiveness. Evaluation on four industrial avionics systems was performed by comparing the proposed approach against two baseline rankings: (1) the *output-base* approach, which uses the outputs of the system under test as oracle data (2) simple *random* selection of the oracle data set. Results show that for oracle variable size (number of variables used in the oracle data set) smaller than the output-only oracle, the proposed approach tends to perform relatively well compared to the output-base approach, with improvements up to 145.8%. As the variable size grows closer in size to the output-only oracle, the improvement decreases, but in 50% of the cases their approach still outperforms output-only oracle, up to 26.4%. For cases when the test oracle grows in size beyond the output-only oracle, the relative improvement again grows, with improvements of 2.2 - 45%. In case of comparison with the random approach, every oracle generated by the proposed approach outperforms it. However, as the experiments were made only on avionics systems, it is questionable whether this approach is applicable to other domains such as object-oriented unit testing. In addition, the approach can have scalability problems due to the mutation analysis it uses.

Similarly, to support test oracle creation, Loyola et al. [61] propose a system called *Dodona* that ranks program variables based on the *interactions* and *dependencies* observed between them during program execution. Initially, a test input is executed, and their tool *Dodona* monitors the relationships that occur between variables during execution (via dataflow analysis). Following this, *Dodona* ranks the relevance of each program variable using techniques from network centrality analysis. A test engineer can then define an expected value oracle for the given test input, confident that their effort is directed towards aspects of the system behavior that are relevant under that input.

2.3.2 Specification Mining

Another form of automated oracles are mined specifications. The work by Nguyen, Marchetto and Tonella [99] evaluates three types of such automated oracles in terms of cost and effectiveness: data invariants, temporal invariants and Finite State Automata (FSA). The following tools are used as representatives of these mined specifications: *KLFA* [64] for FSA oracles, *Daikon* [27] for data invariants and *Synoptic* [15] for temporal invariants. The following procedure is adopted for the experiment design: 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 execution scenarios, 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). The experiments were conducted on 7 Java applications from different domains and different size, up to 94,550 NCLoCs, and 7 real faults from Apache Commons Collections. The results show that automated oracles have a moderate fault detection capability: *Daikon* truly revealed 1 fault, while *Synoptic* revealed 3 and *KLFA* revealed 2 faults. However, the false positive rate of these tools is very high: around 86% for *KLFA* and 30% for *Daikon* and *Synoptic*.

Unfortunately, existing approaches for inferring invariants necessarily require human intervention for two reasons. First, invariants are intended to act as specifications, but are generated from the source code we wish to verify. Extracting what the program should do from what the program actually does is impossible. Second, many existing approaches are dynamic, and use only a finite number of program traces to generate "likely" invariants, rather than correct invariants. Thus, if we assume that user classification effectiveness, defined as the percentage of invariants a user correctly classifies as correct or incorrect, is high in practice, then automatic invariant generation is a potentially effective method for generating automated test oracles, and existing results demonstrating the power of invariant generation may hold in practice. Staats et al. [95] conducted an empirical study with 30 participants to determine user classification effectiveness for invariants generated using dynamic invariant generation, 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. Invariants were generated using Daikon, a dynamic inference tool with a strong body of supporting research. Participants were asked to determine, for each generated invariant, if the invariant was correct or incorrect with respect to the Java class. On average, the study participants misclassified 9.1-39.8% of correct invariants and 26.1-58.6% of incorrect invariants. Second, the factors that lead to invariant misclassification appear surprisingly subtle. Despite examining a large number of factors, the authors were unable to clearly determine why users perform poorly at the classification task.

The work by Zhang et al. [113] aims to reduce the false negative rate for *Daikon* and presents *iDiscovery*, a technique that employs a feedback loop between symbolic execution and dynamic invariant discovery to infer more accurate and complete invariants until a fix-point is reached. In each iteration, *iDiscovery* transforms candidate invariants inferred by Daikon into assertions that are instrumented in the program. The instrumented program is analyzed with symbolic execution to generate additional tests to augment the initial test suite provided to Daikon. The key intuition behind *iDiscovery* is that the constraints generated on the synthesized assertions provide additional test inputs that can refute incorrect/imprecise invariants or expose new invariants. Therefore, when the new inputs are used to augment the previous test suite, dynamic invariant discovery will be based on a richer set of program executions enabling discovery of higher quality invariants. To mitigate the cost of symbolic execution, *iDiscovery* provides two optimisations: assertion separation and violation restriction. The experimental results on four Java artifacts show that *iDiscovery* is able to falsify from 24% to 72% of the invariants generated by the Daikon.

2.4 Oracle Placement

Only a few works have considered the problem of optimal oracle positioning or placement. They mainly focused on which variables to consider in the oracles [61, 34, 94]. The selection of optimal placement points for oracles has not been thoroughly investigated so far, with the exception of the preliminary idea described in a short ESEC/FSE-NIER paper [111]. On the other side, a large body of work has been devoted to the main motivating factor for oracle placement problem, i.e. to failed error propagation.

In this subsection we first introduce the PIE framework, which provides the basis for understanding the process of error propagation. Then, we review the existing studies on failed error propagation.

2.4.1 PIE framework

Voas and Jeffrey [102] introduced a dynamic failure-based **Propagation, Infection, Execution (PIE)** framework to estimate three probabilities: 1) the probability that a particular section of a program is executed, 2) the probability that the executed section affects the data state, and 3) the probability that the affected data state has an effect on the program output. The authors note that these three analyses can be made at different levels of abstraction - programs, modules, and statements. *Execution probability* can be calculated by simply running the program, and determining how often each location is executed. *Infection probability* can be estimated by simulating various faults using mutation analysis and checking whether they cause data state errors. *Propagation probability* can be approximated by introducing different data-state errors and seeing whether program's output has changed. The infection probability and propagation probability are calculated for each mutation and data state error respectively.

Based on the PIE analysis' estimates, the authors proposed a technique called *Sensitivity Analysis*, which is the process of determining the sensitivity of a location in a program. Here the word "sensitivity" means a prediction of the probability that a fault will cause a failure in the software at a particular location under a specified input distribution. The location's sensitivity is measured by multiplying the location's execution estimate, minimum infection estimate, and minimum propagation estimate.

The ideas presented in the paper were empirically evaluated using a single subject program with 2000 lines of code. The experimental procedure included 100 inputs (using uniform input distribution), 25 mutations and a single function to perturb data states. The results show a significant correlation coefficient between the estimate of the probability of failure measured by random software testing and the probability of failure predicted by the estimates of propagation analysis and execution analysis.

The work by Voas et al. [100] introduces the notion of *program testability* and defines it as the program's ability to hide faults when the program is black-box-tested with inputs selected randomly from a particular input distribution. According to this definition two programs that compute the same function may have different testabilities. A program with a high testability readily reveals faults, while a program with low testability is unlikely to reveal faults. The authors note that while sensitivity is related to testability, the terms are not equivalent. Testability encompasses the whole program and its sensitivities under a given input distribution. Sensitivity characterises only the sensitivity of a single location in a program. However, the program's testability can be defined from the collection of sensitivities over all locations. It is conservatively estimated to be the minimum sensitivity over all locations in the program.

The authors conducted an experiment to check the hypothesis that for an injected fault, the sensitivity for the location where the fault was injected is always less than or equal to the resulting failure probability estimate of any fault injected at that location. The subject program for the experiment was a single method with just 10 lines of code. Three different faults were injected into the program at different locations. The failure-probability estimates were based on 10,000 inputs for the two faults injected and 10,000 inputs for the one remaining fault. Results show that the hypothesis is supported.

The later work by Voas and Miller [101] views each location in the program as a point where an assertion checking the internal state can be placed. The authors advocate a middle ground between no program assertions at all (the most common practice) and the theoretical ideal of assertions at every location, introducing the problem of *optimal oracle placement*. Their compromise is to place assertions only at locations where traditional testing is unlikely to uncover software faults. The authors propose locations determined by sensitivity analysis for the assertion placement. They have evaluated this approach on one tiny example and the results show that adding assertion to the selected locations increases propagation probability. This idea was used in later works for determining the optimal data set for output-based oracles [94] and for determining locations to place input-specific internal oracles [111].

The PIE framework was also reiterated in the Reachability-Infection-Propagation (RIP) model described in Amman and Offutt [3]. The authors provide similar definitions of sensitivity and testability based on RIP model and consider applications of testability to common technologies. They note that object-oriented software and web applications present special challenges for testability. For object-oriented software the main reason is that objects encode state information in instance variables, and access to these variables is usually indirect because of *inheritance*. In web applications almost all of the infrastructure in web applications is intended to be invisible from the client's perspective, therefore accessing much of the state is impossible. On the other hand, the server side is likely to be distributed not only across multiple hardware platforms, but even across multiple corporate organisations. Bringing high testability to such systems is still a research topic.

The works by Li and Offutt [57, 58] extend the traditional RIP model to Reachability-Infection-Propagation-Revealability (RIPR) model. RIPR model underlines that if the fault propagates to the output, but the oracle does not check the particular portion of the state that contains erroneous value caused by this fault, the oracle will not see the failure. That is, the test oracle must also *reveal* the failure. To investigate the ability for test oracles to reveal failures, the authors define ten new test oracle strategies that vary in amount and frequency of program state checked for model-based systems. They compare these strategies to baseline test oracle strategies: *null test oracle strategy* (NOS), i.e. implicit oracle, and *state invariant oracle strategy* (SIOS), that checks the invariants of states reached after each transition. The results of the experiments show that using only null test oracle strategy is not enough to reveal all the faults. However, it is also not necessary to check the entire state, as checking partial states reveals nearly as many failures. When it comes to the frequency of the checks, checking less frequently is as effective as checking states more frequently.

Overall, PIE framework and sensitivity analysis include characteristics that are similar to mutation testing. However, the goals of the two techniques are different. Mutation testing seeks an improved set of test data, while infection analysis seeks to identify locations where faults are unlikely to change the data state. Propagation analysis mutates the data state, not the code, and then examines whether the output is affected.

Also, PIE analysis is distinct from fault-based testing because PIE analysis collects information concerning the semantics of the program; fault-based testing collects information concerning whether certain classes of faults exist in a program. PIE analysis does not reveal the existence of faults, since correctness is not the goal of this analysis. Indeed, this technique also does not directly evaluate the ability of inputs to reveal the existence of faults. Instead, it identifies locations in a program where faults, if they exist, are more likely to remain undetected during testing.

2.4.2 Studies on Failed Error Propagation

Failed Error Propagation (FEP) occurs when a test case executes the faulty statements but no failure is triggered. The PIE model emphasises that for a failure to be observed, the following three conditions must be satisfied: 1) the defect is executed, 2) the program has transitioned into an infectious state, and 3) the infection has propagated to the output.

A number of studies provide evidence of the occurrence of FEP. The motivations for these studies vary. Some of them analyse specific cases such as propagation of error codes in file systems [39, 87, 88]. However, the majority are motivated by FEP being undesirable for *Coverage-Based Fault Localization* (CBFL) techniques [108, 10, 49]. Therefore, there is a large body of work aiming to reduce the vulnerability of CBFL to FEP and this usually includes measuring the prevalence of FEP in the subjects of the experiments [66, 67, 68, 73, 59, 112, 114].

However, all of these studies use different terms (*error masking*, *fault masking*, *strong/weak coincidental correctness*) and definitions to express closely related notions, or use the same term with different meanings. To make existing studies comparable, we unify existing terms and definitions as follows, using the PIE framework as a well understood basis for unification:

- *Coincidental Correctness* (CC) occurs when a fault is *executed*, but is not *propagated* to the output.
- *Failed Error Propagation* (FEP) occurs when a fault is *executed*, it *infects* the data state, but it does not *propagate* to the output.

The work by Daran & Thévenod-Fosse [24] reports the experimental comparison of error propagation mechanisms of software errors generated by real faults and by first-order mutations. The experiment was conducted on a single C program (approximately 1000 lines of code) from the civil nuclear field. It involves 12 known real faults and 24 mutations. The 12 real faults were uncovered during authors' previous experiments [97, 98]. The 24 mutations were selected so that a small and various sample can be obtained. Yet, in order to make the comparison feasible some (but not all) mutations were performed on instructions involved in the "fix" of the real faults. The results are reported not across all the executions, but only for the ones where the faults have infected the state. Among 88 of such executions, for real faults 19 (22%) propagate to output, while 69 (78%) fail to propagate. For mutations 41 (24%) out of 169 propagate and 128 (76%) do not.

The work by Xue et al. [112] analysed the prevalence of coincidental correctness on 4 Java programs from the Software Infrastructure Repository (SIR) [26]. 20 different faults were hand-seeded into these programs by other researchers. The executions to analyse for the presence of coincidental correctness were obtained by running the manually-written test suites (sizes between 54 and 214) for the subject programs. One of the hand-seeded faults did not expose any failure, so it was excluded from the experimentation. For the remaining 19 faulty programs, results show that the percentage of coincidental correctness is in the range from 1.2% to 22.2%, with an average of 7.4%.

The works by Masri et al. [66] and Masri & Assi [67, 68] analyse both the occurrence of coincidental correctness and failed error propagation. Three releases of NanoXML and seven programs from the Siemens Suite (136 - 7646 lines of code) were used in the studies. The experiments involved 148 seeded versions, among which 16 were derived from the NanoXML releases and 132 from the Siemens programs. As the analysis tools used in the study targeted only Java programs, the authors manually converted the Siemens programs from C into Java. The average rate of coincidental correctness was reported as 56.4%. 3.5% of subjects did not exhibit any coincidental correctness, while 28.5% exhibited a high level in the range [60%, 90%] and 30% exhibited an ultra-high level in the range [90%, 100%]. The rate of failed error propagation is 15.7% on average. 28% of the faulty programs did not exhibit any failed error propagation, while 13% exhibited a high level in the range of [60%, 100%].

Wang et al. [104] used three real world C programs with the sizes between 5,000 and 8,000 lines of code in their study. First, program mutations were created for the subjects. Then these mutants were executed using the whole test pool (between 5000 and 13585 test cases) and the ones that were not strongly killed were excluded. After this step, for each subject program 1000 mutants were randomly sampled in proportion to the occurrence frequency of their fault types. Results show an average of 36% coincidental correctness. The authors report that for 27% of the mutants the rate of coincidental correctness is over 80%. Having conducted the experiments with mutants, the authors further validated their results using 38 real faults in one of the subjects. In the case of real faults, the rate of coincidental correctness varies substantially between 0.15% and 99.77%, with an average of 54%.

The study by Miao et al. [73] measures the level of coincidental correctness in 6 C programs from the Siemens Suite. The experiments are conducted using the subject programs injected with 115 hand-seeded faults and their corresponding manually written test suites (sizes between 1052 and 5542). The rate of coincidental correctness is 56% on average. For around 18% the level of coincidental correctness is 100%, as the faulty versions not exhibiting any failure were not excluded from the study. On the contrary, in the study by Li & Liu [59] for each fault there is at least one failing test case. This study is conducted on 3 subject programs from Siemens Suite with 18 hand-seeded faults. Results show that the rate of coincidental correctness is between 0.5% and 44.4%, with an average of 20.4%.

Androutsopoulos et al. [5] introduce an information theoretic approach to FEP. They introduce five different metrics, based on measures of conditional entropy, and check whether these metrics are well-correlated with the probability of FEP. The subject programs were 17 very small programs and two real-world projects (810 - 286000 lines of code). To obtain faulty versions of the programs a mutation generator was used. In case too many mutants were generated for the subject program, 100 mutants were selected randomly. As all subject programs had numeric inputs, the Rng-Pack[3] library was used to generate the random numbers to be used as inputs. Each subject program and each of its mutations were executed with the same 5000 inputs. The results show an average rate of 14.74% for coincidental correctness and 9.85% for failed error propagation.

Xiong et al. [111] performed a quantitative study on how much inner oracles can improve the fault-detection capability of existing tests. For this, they generated mutations for subject programs and manually removed equivalent mutants. With each test and each mutant forming a test-fault pair, they got overall 97582 test-fault pairs. The results show that in 30.72%-69.65% of these pairs the fault is triggered but cannot be detected by traditional oracles on output, while these pairs can all be detected by inner oracles. This shows that inner oracles have a significant impact on both the fault-detection capability of tests.

Table 1 provides an overall summary of the studies on failed error propagation. Column *Language* shows the programming language of the subject programs. Column *Fault Type* shows what type of faults were analysed in the corresponding study: synthetic mutations, faults seeded into the source code by developers or real faults. Column *# of Faults* shows how many faults of the given type were generated. *FEP type* shows whether the study measured

[3]http://www.honeylocust.com/RngPack/

Coincidental Correctness or Failed Error Propagation, and *FEP ratio* shows the average rate of CC/FEP reported in the study.

Table 1: Studies on Failed Error Propagation

Study	**Language**	**Fault Type**	**# of Faults**	**FEP Type**	**FEP Ratio**
Daran et al. [24]	C	Real	12	FEP	78%
Masri et al. [66] Masri & Assi [67, 68]	Java, C	Seeded	148	CC, FEP	56.4%, 15.7%
Wang et al. [104]	C	Mutants Real	3000 38	CC CC	36% 54%
Miao et al. [73]	C	Seeded	115	CC	56%
Li & Liu [59]	C	Seeded	18	CC	20.4%
Xue et al. [112]	Java	Seeded	19	CC	7.4%
Androutsopoulos et al. [5]	C	Mutants	1408	CC, FEP	4.89%, 9.85%
Xiong et al. [111]	Java	Mutants	137	FEP	43.4%

As we can see from the table, previous work tends to suggest that there is a nontrivial proportion of faults that are subject to FEP. However, the ratio of FEP varies across different studies substantially: from 7.4% to 43.4%. The majority of these studies come from *fault localisation* and 4 out of 8 studies in the table use subjects from the same Siemens Suite. The majority of the studies use mutants or seeded faults, which are used to simulate real faults. Two studies [24, 104] analysing real faults use a single C subject and consider respectively 12 and 38 real faults for it. Only two previous papers considered FEP for Java programs. However, neither of them attempted to measure FEP on real faults.

3 Failed Error Propagation measured on Java Programs with Real Faults

Software faults are difficult to detect and fault removal consumes a significant proportion of software development and evolution [11, 14]. One of the widely-attributed sources of such difficulty is the possibility of *failed error propagation* (FEP): a fault may corrupt the program's internal state, yet this corruption fails to propagate to any point at which it is observed [38, 84, 107]. Such non-propagating faults play the role of 'nasty unexploded mines': lurking undetected in software systems, waiting for that slight change in execution environment that allows the corrupted error state to propagate, causing unexpected system failure.

Despite the importance of FEP, surprisingly few empirical studies in the literature assess the extent of the problem. Empirical evidence based on a few examples of real faults is available only for the C/C++ programming language [12, 18, 40, 24], while for Java results have been obtained only with mutants [68, 111], not with real faults. In the absence of robust empirical analysis, the research and practitioner community is left with suspicions of a silent menace of unknown proportions.

In order to bridge the gap between suspicions and empirical evidence we set out to perform a large empirical study of FEP on real faults from Defects4J [51], a large scale benchmark that has become the de-facto standard [52, 63, 6, 62, 65, 110, 82, 111, 91] for real faults in Java programs. Our study encompasses all six projects in Defects4J and the associated 386 real faults.

Since the occurrence of FEP is a statistical property of a method (in fact, it may occur in some executions and not in others), we faced the problem of obtaining a sample of empirical data that is large enough to draw statistically meaningful conclusions. This requires that the considered faults are executed multiple times, in program executions that differ from each other, and that the effects of the faults on the program state are observed along corresponding execution points of both faulty and fixed program. We have extended the

EvoSuite [30] test case generator to address the first problem and we have developed our own trace alignment algorithm to address the second problem.

Our study revealed a very surprising finding: in this significant corpus of real world bugs, the prevalence of unit-level FEP is negligible. We further experimented with seeded synthetic faults (mutants [48]), for which we observed that unit-level FEP *was* found to be much more prevalent. To further analyse the propagation of real faults we conducted experiments testing programs at the system level rather than on unit level. Our results show that the rate of system-level FEP with real faults is substantially higher than the rate of unit-level FEP both with real and synthetic faults.

The primary contributions of this chapter are:

1. A large empirical study of failed error propagation in 6 different subjects with 386 real bugs overall.
2. Comparison of FEP occurrence in programs with real faults to FEP occurrence in programs with synthetic faults.
3. Comparison of unit-level FEP occurrence to system-level FEP occurrence.

3.1 Failed Error Propagation

The effectiveness of testing depends on the use of oracles that are sensitive to any deviation from the intended program behavior and that report all such deviations as test failures. One of the key decisions about the use of oracles is their placement. Oracles can be placed in test cases in the form of a test case assertion, i.e., outside the method under test (unit level testing) or at the end of the entire system execution (system level testing); at the end of the execution of the method under test, before the return point (acting as a post-condition); or even internally, at any arbitrary execution point, predicating on the intermediate program states observed during method execution.

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

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

In the running example shown in Figure 1, consider the faulty statement `x = 3 * x`, whose corresponding fixed version is `x = 2 + x`, and the return point assertion at `pp6`. If the faulty program is executed with input `x==4` (see `test0` in Figure 1), it returns 0, while the expected value is 2, which indicates

```
int f(int x) {
   // pp0: assert(\old(x) == x))
   x = 3 * x; // fix: x = 2 + x;
   // pp1: assert(\old(x) + 2 == x))
   if (x > 0) {
      // pp2: assert(2+\old(x) > 0 && \old(x)+2 == x))
      x = x % 4;
      // pp3: assert(2+\old(x) > 0 && (\old(x)+2) % 4 == x))
   } else {
      // pp4: assert(2+\old(x) <= 0 && \old(x)+2 == x))
      x = x + 1;
      // pp5: assert(2+\old(x) <= 0 && \old(x)+3 == x))
   }
   // pp6: assert(2+\old(x) > 0 ? \result == (2+\old(x)) % 4 :
      \result == 3+\old(x));
   return x;
}
void test0() { assert(f(4) == 2); } // FAIL
void test1() { assert(f(5) == 3); } // PASS
```

Figure 1: Code example including 7 possible internal oracle placement points, `pp0` to `pp6`, as well as a test case (`test0`) exhibiting no FEP and one with external FEP (`test1`)

the fault can indeed affect an externally visible result, in some execution. If the program is executed with input `x==5`, we can observe a different execution state at program points `pp1` and `pp2`, where we have `x==15` in the faulty program, while we expect `x==7`. However, at program point `pp3` the same value of `x` is produced by both the faulty and the fixed program: `x==3`. When the assertion at `pp6` is executed, no difference is observed between faulty and fixed program. The external assertion inside `test1` also does not fail.

In the second program execution (with input `x==5`), the error fails to propagate to the assertion at `pp6` because the information about the different execution states in the faulty and fixed programs is destroyed by the execution of statement `x = x % 4`, which collapses the two different program states into the same one, `x==3`. This is a case of both internal and external FEP, which could be solved by introducing the internal assertion at `pp1` or `pp2`.

Consider a case where external/output FEP occurs, while internal FEP does not. Suppose we change the return type of `f` in the example shown in Figure 1 to `boolean` and change the return expression to `(x >= 0)`. With such a change, `test0` would pass, expecting and observing `true` as return value. However, the return point assertion at `pp6` would fail, since the observed value `x=0` differs from the expected value `x=2`.

Definition 1 (Coincidental Correctness) *Given a fault f at program point pp_f, a specific method execution e containing pp_f, represented as the sequence of program points $e = \langle pp_0, \ldots, pp_n \rangle$, is said to be subject to coincidental correctness (CC) if the faulty statement pp_f does not cause a state divergence between actual and expected execution states, $s[pp_f]$ and $s'[pp'_f]$.*

$$s[pp_f] = s'[pp'_f]$$

Definition 2 (Internal FEP) *Given a fault f at program point pp_f, a specific method execution e containing pp_f, represented as the sequence of program points $e = \langle pp_0, \ldots, pp_n \rangle$, is said to be subject to internal failed error propagation (intFEP) if execution of the faulty statement pp_f causes a state divergence between actual and expected execution states, $s[pp_f]$ and $s'[pp'_f]$, which is not observable at the return statement pp_n:*

$$s[pp_f] \neq s'[pp'_f] \wedge s[pp_n] = s'[pp'_n]$$

where program points pp'_f, pp'_n correspond to pp_f, pp_n in the fixed program; s and s' indicate the execution state of faulty and fixed program respectively.

Definition 3 (External FEP) *Given a fault f at program point pp_f, a specific method execution e containing pp_f, represented as the sequence of program* *points $e = \langle pp_0, \ldots, pp_n \rangle$, is said to be subject to external failed error propagation (extFEP) if execution of the faulty statement pp_f causes a state divergence between actual and expected execution states, $s[pp_f]$ and $s'[pp'_f]$, which is not observable outside the faulty method (extFEP):*

$$s[pp_f] \neq s'[pp'_f] \wedge ext = ext'$$

where program point pp'_f corresponds to pp_f in the fixed program; s and s' indicate the execution state of faulty and fixed program respectively; ext represents the values observable outside of the unit under test.

Definition 4 (System FEP) *Given a fault f at program point pp_f, a specific method execution e containing pp_f, represented as the sequence of program points $e = \langle pp_0, \ldots, pp_n \rangle$, is said to be subject to system failed error propagation (sysFEP) if execution of the faulty statement pp_f causes a state divergence between actual and expected execution states, $s[pp_f]$ and $s'[pp'_f]$, which is not observable in the output produced by the system (sysFEP):*

$$s[pp_f] \neq s'[pp'_f] \wedge out = out'$$

where program point pp'_f correspond to pp_f in the fixed program; s and s' indicate the execution state of faulty and fixed program respectively; out represents the values output by the system.

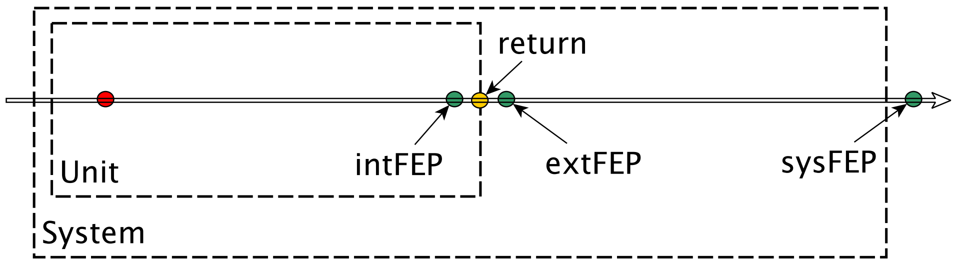

Figure 2: Execution points where state corruption disappears in the cases of internal, external or system FEP

It can be easily shown that internal FEP subsumes external FEP, which in turn subsumes system FEP ($intFEP \Rightarrow extFEP \Rightarrow sysFEP$). Figure 2 shows the three cases of FEP in graphical form. When a state corruption occurs in the execution (red leftmost dot), it might no longer be observable in the execution right before the return point (green dot labeled intFEP), right after the return point (extFEP dot) or when the entire system execution is over (sysFEP dot). Visually, the subsumption relation corresponds to the green dot (second from left) in the figure, which propagates from left to right (i.e., if a state corruption disappears, it remains unobservable until the end of the execution).

The definitions given above are tied to a particular execution of the faulty method. We can generalize such definitions and define the probability of failed error propagation of a method for a fault f as follows:

Definition 5 (Probability of FEP) *Given a fault f at program point pp_f, the probability of (internal/external/system) FEP is the proportion of method executions e containing pp_f that are subject to (internal/external/system) FEP across all method executions e containing pp_f:*

$$p(FEP_f) = \frac{\mid \{e \mid pp_f \in e \wedge e \textit{ is subject to FEP}\} \mid}{\mid \{e \mid pp_f \in e\} \mid}$$

Different types of oracles are required to prevent different types of FEP. We call the oracle placed at an internal program point an *inner oracle*. One variety of inner oracle is an oracle placed at the return points, which we call a *return point oracle*. An oracle that checks the externally visible state of the class is called a *unit-level oracle*. Similarly, an oracle that checks the output of the overall system is called a *system-level oracle*. Inner oracles (in case of high internal FEP), unit-level oracles (in case of low internal FEP and high external FEP) or system-level oracles (in case of low internal FEP, low external FEP and high system FEP) are needed to increase the fault detection capability of test cases.

Inner oracles are the most powerful form of oracles, since they can detect any deviation between actual and expected internal program states. However, defining inner oracles is quite difficult for developers, especially when they need to be placed within loops or within complex control structures. Manual oracle definition is supposed to be easier for return point oracles, and even simpler for unit-level oracles, which consider only the externally visible program state. Hence, understanding the relative strength of unit-level oracles, return point oracles and inner oracles has major practical implications for developers. It is also relevant for research, since generating, assessing and improving external vs. return point vs. inner oracles involves different approaches and techniques.

3.2 Experimental Procedure

In this section, we provide the details of the procedure we have followed to measure FEP occurrence on real faults. We first present the benchmark used in the empirical study. To obtain statistically significant measurements we needed large pools of inputs exercising the faulty statements. We describe the automated test case generation approach adopted for this purpose. Then, we describe how execution traces of faulty and fixed programs have been aligned so as to compute state differences at corresponding program points. Finally, we give detailed information on how the FEP measures were obtained from the aligned traces.

3.2.1 Benchmark

To analyse FEP occurrence in programs with real faults we used Defects4J [53][51] (version 1.1.0), a large scale database of existing faults, which contains 395 real bugs from 6 real-world Java open source projects (442 classes and 71455 SLOC per project on average).

For each bug we identify whether it is suitable for our study by checking if its fix is a change in a method/constructor. The results show that for 9 out of 395 bugs, the fix is a change in other class members as instance/static variables, static initialisation blocks or in the class declaration itself (as the interfaces it implements). As this kind of bugs can not lead to FEP, we exclude these bugs from our study. For system-level FEP rate analysis we can use only bugs from projects with system-level functionality. This condition is satisfied for 132 bugs from the Closure Compiler project, as the remaining 5 projects in Defects4J are libraries. For unit-level FEP rate analysis we exclude methods/constructors with only one statement, as there is no possibility for internal or external FEP in them.

Table 2 shows the projects contained in Defects4J, the number of bugs and changed methods/constructors for each of them. In total, we have 386 bugs and 459 methods/constructors available for unit-level FEP rate analysis and 132 bugs available for system-level FEP rate analysis.

Table 2: Defects4J Projects (M/C means Methods/Constructors)

Project Name	**Bugs**		**Number of M/C**	
	Fix in M/C	**All**	**> 1 LOC**	**All**
JFreeChart	25	26	36	39
Closure Compiler	132	133	153	172
Commons Lang	62	65	73	84
Commons Math	104	106	126	146
Mockito	37	38	31	68
Joda Time	26	27	40	51
Total	386	395	459	560

3.2.2 Input Generation

As FEP might occur only for specific inputs, to estimate its probability we need a large number of executions that cover the faulty statements.

To obtain these executions for unit-level FEP rate analysis, we extended the EvoSuite [30] test case generator (version 1.0.5). We identify the difference between the buggy and fixed versions of the method in terms of lines of code. The standard line coverage criteria of EvoSuite aims at generating a test suite that covers all lines of code. However, we need to cover only lines of code that contain faulty statements. Moreover, we need these lines of code to be covered multiple times, by different test cases. For this purpose, we made changes to EvoSuite's implementation, so as to handle the following new parameters:

1. *line_list*: list of lines of code to be covered by the generated test cases;
2. *goals_multiply*: number of times each line should be covered.

In our experiments we aimed to have 1,000 different executions covering each fault. This number of executions was judged a good balance between the total time spent on each experiment and the resulting size of the test pool per bug. Achieving a coverage goal results in a single execution that covers a single line. Each bug consists of a number of lines, the lines in *line_list*. Therefore, we calculate the value for the parameter *goals_multiply* by dividing 1,000 by the size of the list provided as the *line_list* parameter and round this number. We run our extension of EvoSuite giving it a cumulative, maximum search budget of 10,000 seconds (i.e., a maximum of 10 seconds per coverage goal). Since we generate 1,000 test cases per bug and these tests are generated on the faulty program version (as a developer would do to expose faults during development), each bug requires a separate test generation process, executed on a distinct program version, i.e., the one containing the considered bug.

For system-level FEP analysis, we needed inputs for *Closure Compiler*, which is a tool that accepts a JavaScript file as an input, analyzes it, removes dead code and rewrites and minimizes what's left. We downloaded the 15 most highly trending JavaScript projects from GitHub[4]. Trending projects are identified by looking at a variety of data points including stars, forks, commits, follows, and page views and weighting them appropriately. As a result, we got 3779 JavaScript files in total, and used these files as inputs to our system.

3.2.3 Trace Alignment

To identify the cases of internal and external FEP, we trace both faulty and fixed methods, and we compare the values of variables at corresponding program points in the faulty and fixed versions of the method. In simple scenarios,

[4]https://github.com/trending/javascript, downloaded on 18.09.2017

where the fault fix requires only a change in an existing statement, the correspondence between program points is trivially by position in the linearly ordered sequence of statements, i.e., corresponding program points are program points with the same line number. However, in more complex cases, in which the fix requires the addition of new statements and/or the deletion of existing statements, the statement sequences aligned by order must exclude program points that refer to added/deleted statements. Hence, the identification of the corresponding statements can be obtained by calculating the tree edit distance between the Abstract Syntax Trees (AST) of faulty and fixed methods. The *tree edit distance* [93] is the minimal-cost sequence of node edit operations that transform one tree into another, where the allowed edit operations are: CHANGE, INSERT, DELETE.

We represent the source code of faulty and fixed versions of a method as an AST using JavaParser[5] (version 2.3.1). We adapted the tree edit distance computation algorithm described in [93] so that it works with nodes which are objects of JavaParser's `Node` type. We assign the cost of 1 to the three edit operations (CHANGE, INSERT and DELETE) supported by the algorithm. As a result, we get an edit sequence which converts one tree into another and therefore a faulty method into the fixed one.

Figure 3 (a) shows an example of a simple method `test(int x)`, which, for the purpose of the explanation, we consider as a buggy method. Three hypothetical fixes are shown in Figure 3 (b), (c), (d), involving respectively the change of an existing statement, the addition of a new statement and the deletion of an existing statement. The edit scripts automatically produced by our implementation of the tree edit distance algorithm are shown in the right column of the figure.

After the edit script is generated, we start the instrumentation process. For both the buggy and fixed versions of the method we instrument the starting program point *pp0*. Then we visit the nodes in the ASTs of the two methods, according to the pseudocode of Algorithm 1 If a node is associated with the

[5]http://www.javaparser.org

```
public int test(int x) {
   //pp0 (b, c, d)
   int y = x + 1;
   //pp1 (b, c)
   y = y % 4;
   //pp2 (b, c) //pp1 (d)
   return y; }
```

(a)

```
public int test(int x) {
   //pp0
   int y = x + 1;
   //pp1
   y = y % 3;
   //pp2
   return y; }
```

```
KEEP int y = x + 1;
CHANGE y = y % 4; to
       y = y % 3;
KEEP return y;
```

(b)

```
public int test(int x) {
   //pp0
   int y = x + 1;
   y = y * 3;
   //pp1
   y = y % 4;
   //pp2
   return y; }
```

```
KEEP int y = x + 1;
INSERT y = y * 3;
KEEP y = y % 4;
KEEP return y;
```

(c)

```
public int test(int x) {
    //pp0
    int y = x + 1;
    //pp1
    return y; }
```

```
KEEP int y = x + 1;
DELETE y = y % 4;
KEEP return y;
```

(d)

Figure 3: Buggy method (a) and hypothetical fixed versions, obtained by changing an existing statement (b), by adding a new statement (c), or by removing an existing statement (d)

Algorithm 1: Program point instrumentation

```
1  Procedure visit(n, i)
       Input:
       n: AST node to be visited
       i: instrumentation index
2      begin
3          if n is labeled as KEEP or CHANGE $\wedge$ type(n) is not (RETURN
             or THROW) then
4              while next(n) is labeled as DELETE or INSERT do
5                  $n :=$ next($n$)
6              $i := i + 1$
7              instrumentAfter($n$, $pp_i$)
8          else
9              visit(next($n$), $i$)
10         if type(n) is not (FOR or WHILE) then
11             for $m \in$ children(n) do
12                 visit($m$, $i$)
```

KEEP or CHANGE operators and if it is not of the RETURN or THROW statement types, we instrument the program point after this node (line 7), skipping any sequence of INSERT and DELETE nodes (lines 4-5). Otherwise, if it is associated with the INSERT or DELETE operators, we skip the program point and proceed with the next node (line 9). Then, if the node is not a *while* or *for* loop, the visit proceeds recursively on the subtrees (lines 10-12). We exclude program points within loops because of the practical difficulty of defining oracles for the program state inside a loop.

By following this procedure, we obtain the program point correspondence indicated within comments in Figure 3, associating program points in version (a) with those in (b), (c), (d). The placement of program points in ⟨(a), (b)⟩ is straightforward. In case of ⟨(a), (c)⟩, the program point before added statement `y = y * 3` is skipped during the visit of (c) due to the while loop at lines 4-5 in Algorithm 1. Similarly, the deleted statement `y = y % 4` is jumped over during the visit of (a). As a consequence, the program point after `y = x + 1` in (a) has no corresponding program point in (d).

3.2.4 Measuring FEP Rate

After running the generated inputs on the instrumented methods, we obtain the values of variables at each program point, for each execution. Algorithm 2 shows how we identify whether a given execution is subject to FEP.

As shown at lines 2-3, if unit-level FEP analysis is performed and the externally observable state is affected by the faulty execution as compared to the fixed execution, we report no FEP. Similarly, if system-level FEP analysis is performed and the output of the system is affected by the fault, we report no FEP (lines 5-6). However, if the output of the system remains the same for the faulty and fixed execution, but the externally visible state is different, we report system-level FEP (lines 7-8). Otherwise, we check whether the state at the program point before return is different (lines 9-10) and if it is so, we report external FEP. If there is no external FEP, and the program points traversed by the executions in the buggy and fixed methods are different (lines 11-12), then internal FEP is detected – here, the executions in the faulty and the fixed methods took different paths, so if we place an internal oracle in the buggy method checking for the predicates in the path of the fixed execution, it would detect the fault. In the case that all program points in the two executions are the same, we iterate through them and report internal FEP if the state in at least one aligned pair of them is different (lines 13-15). Finally, it is also possible that for some inputs, the bug in the method does not lead to any changes at all in the pair of executions being compared. This is also a case of no FEP (line 16). The FEP value returned by the algorithm is expanded with the addition of the subsumed values (invocation of *closure* in Algorithm 2). This means for instance that if intFEP is reported for a system level analysis, extFEP and sysFEP are also reported as true.

Algorithm 2: Measuring FEP

```
Input:
$type = \langle sys \mid unit \rangle$: type of analysis, system-level or unit-level
$out$, $out'$: output of the system, used only for system-level analysis
$ext$, $ext'$: externally observable state after buggy/fixed methods' executions
$pp = \langle pp_0, \ldots, pp_n \rangle$: program points executed in fixed method
$pp' = \langle pp'_0, \ldots, pp'_k \rangle$: program points executed in buggy method
$s$, $s'$: state by program point in buggy/fixed methods
Result:
$fepType$: ⟨ sysFEP | intFEP | extFEP | noFEP ⟩
1  begin
2      if $type = unit$ && $ext \neq ext'$ then
           // $s \neq s'$, $ext \neq ext'$
3          return noFEP
4      if $type = sys$ then
5          if $out \neq out'$ then
               // $s \neq s'$, $ext \neq ext'$, $out \neq out'$
6              return noFEP
7          if $ext \neq ext'$ then
               // $s \neq s'$, $ext \neq ext'$, $out = out'$
8              return closure(sysFEP, $type$)
9      if $s[pp_n] \neq s'[pp'_k]$ then
           // $s \neq s'$, $ext = ext'$
10         return closure(extFEP, $type$)
11     if $pp \neq pp'$ then
           // $pp \neq pp'$, $ext = ext'$
12         return closure(intFEP, $type$)
13     for $i \in [1 : n - 1]$ do
14         if $s[pp_i] \neq s'[pp'_i]$ then
               // $s \neq s'$, $ext = ext'$
15             return closure(intFEP, $type$)
       // $s = s'$
16     return noFEP
```

where *closure(FEP, type)* applies the implication $intFEP \Rightarrow extFEP \Rightarrow sysFEP$ when $type = sys$ and $intFEP \Rightarrow extFEP$ when $type = unit$.

We run this algorithm for each system or method/constructor execution, as appropriate, and then we calculate the proportion of either system-level FEP or unit-level FEP (both internal and external) across all of the executions that cover the considered fault, to estimate the probability of FEP for such a fault. In this algorithm, the value of variables at each program point may represent Java objects that need to be stored and compared with each other. For this we use the XStream framework[6] (version 1.4.9), which can serialize any Java object without requiring their classes to implement the *java.io.Serializable* interface (including private and final fields). We serialize these objects to JSON format and consider two objects equal when their JSON representations are the same.

3.3 Results

3.3.1 Research Questions

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

- **RQ1**: *What is the prevalence of unit-level failed error propagation with real faults?*
- **RQ2**: *Does the prevalence of unit-level failed error propagation change if real faults are replaced by mutants?*
- **RQ3**: *Does the prevalence of failed error propagation with real faults change if it is measured at the system level instead of unit level?*

RQ1 is the key research question that motivates this study. The answer to this question has implications for oracle placement. It is potentially relevant for both practitioners and researchers, since it estimates the probability of missing / detecting a fault depending on where oracles are placed (i.e., internally, at return points, or externally).

[6]http://x-stream.github.io/

While, to the best of our knowledge, no previous study investigated the occurrence of FEP on real Java faults, there are experimental results [5, 68, 111] on FEP rate computed when mutations are used as surrogates for real faults in both Java and C. Such results provide evidence for the occurrence of FEP on mutants. With RQ2 we want to investigate whether results on mutants correspond to the results obtained on real faults.

Since a system level execution typically involves a long chain of concatenated unit level executions, there is potentially more opportunity for a corrupted state to disappear during such a system-level execution, becoming undetectable at the output. In RQ3 we want to check whether the prevalence of FEP changes (and in particular, whether it increases) when we consider test executions at the system level instead of unit level as expected.

We also report some observations obtained from a qualitative analysis performed to better understand the patterns of prevalence behind FEP or no FEP, either with real faults or with mutations, considering the root cause of each occurrence.

3.3.2 Experimental Data

RQ1 (FEP Rate in Programs with Real Faults)

Table 3 shows a summary of the results obtained in our experiments. Column *Changed Methods* indicates the overall number of methods changed as a result of a bug fix, while column *Methods with TS* reports the number of methods for which our extended version of EvoSuite was able to generate a large test suite, consisting of test cases that exercise the faulty statements. While the target size of these test suites was 1,000 test cases, sometimes EvoSuite generated slightly smaller test suites in the allowed generation time (the average test suite size is 863).

Column *Number of Executions* shows the overall number of executions obtained as a result of running the test cases. Column *Externally Detectable* shows the number of executions where the fault resulted in a program state deviation that is observable outside of the methods. Columns *Internal FEP*

Table 3: Internal and external FEP on real faults (RQ1)

Project Name	**Changed Methods**	**Methods with TS**	**Number of Executions**	**Externally Detectable**	**Int FEP**	**Ext FEP**
JFreeChart	36	28	18,785	10,678	0	0
Closure Compiler	153	102	89,078	42,078	0	0
Commons Lang	73	54	35,153	24,854	0	0
Commons Math	126	92	78,489	45,065	0	0
Mockito	31	25	20,967	8,348	0	0
Joda Time	40	28	15,900	7,987	0	0
Total	459	329	258,372	139,010	0	0

and *External FEP* show that, among the 258,372 executions, the fault was externally observable in 139,010 cases (53.8%). In the remaining cases (119,362 test case executions), in order for FEP to happen an internal program state deviation, not propagated to the output, should be observed. However, this was never the case. There was no single case where an internal state deviation occurred, i.e. no state infection.

We have tested the statistical significance of our results, which depends on sample size and observed values. According to the Pearson-Klopper method for calculating binomial confidence intervals, internal/external FEP is in the range [0:1.43^{-5}] with mean = 0 at confidence level 95%. This means that even if intFEP and extFEP could occur in other subjects (we might have not observed it just by chance), their likelihood can be assumed to be very low with high confidence.

RQ1: *Our experiments show that the probability of unit-level FEP in Java methods with real faults is extremely low.*

RQ2 (FEP in Mutated Programs)

For RQ2, instead of real faults we consider faulty versions of methods obtained by means of mutation analysis (i.e., we generate mutants of the fixed

Table 4: Methods from benchmark grouped by LOC

Project Name	**2-25**	**26-50**	**51-100**	**101-200**	**>200**
JFreeChart	22	6	5	3	0
Closure Compiler	60	45	35	8	5
Commons Lang	34	16	11	12	0
Commons Math	59	22	24	16	5
Mockito	25	6	0	0	0
Joda Time	26	12	2	0	0

Defects4J methods). As we need a large test suite for each mutant and the number of mutations generated per method can be high, we did not conduct this analysis on all the methods available in the benchmark. Instead, we sampled the methods based on their lines of code. We divided methods into 5 groups: 2-25 LOC, 26-50 LOC, 51-100 LOC, 101-200LOC, $>$ 200LOC. Table 4 shows the number of methods in each group for each project. We randomly selected one method from each group for each project and we generated mutants for the selected representative using Major [50] (version 1.1.6) and applying all the mutation operators available in this tool. Then, among the generated mutations, we selected only strongly killable mutants, to avoid the inclusion of equivalent mutants. In fact, an internal state deviation in an equivalent mutant is always associated with external FEP, but this is by definition a false positive, because the internally observed difference is not an indicator of a fault: since the mutant is equivalent to the original program, it does not introduce any fault into the program, so there is no fault to be detected internally at all. Hence, we conservatively measure FEP only on mutants proved to be strongly killable by test generation. In cases when EvoSuite was unable to generate a large test suite for any of the mutations of a method, or when none of them is strongly killable, we randomly select another method from the group.

Table 5: Mutants generated.

Project Name	**Mutants**	**Strongly Killed**
JFreeChart	37	25
Closure Compiler	468	350
Commons Lang	360	215
Commons Math	765	502
Mockito	28	15
Joda Time	212	18
Total	1870	1125

Table 5 shows the overall number of mutants and the number of mutants that are strongly killable by the generated test suites. We can see from Table 6 that when we replace real faults with mutations, for 3 subjects there are cases of both internal and external FEP. Among all 831,789 executions in these 3 subjects, 51% of faults were externally detectable. In 1.6% of executions there was an occurrence of internal and in 3.7% of external FEP. In the remaining cases (46.9%) the internal state was always identical to the expected one, i.e., the fault did not infect the execution.

According to the Pearson-Klopper method, internal FEP is in the range [0.0159:0.0164], with mean = 0.0161, at confidence level 95%; external FEP is in the range [0.0210:0.0216], with mean = 0.0213, at confidence level 95%.

RQ2: *Mutants behave in a substantially different way than real faults when the FEP rate is considered for Java methods: there is higher probability of both internal and external FEP when the fault is introduced by mutation.*

RQ3 (System-level FEP)

For RQ3 we have run *Closure Compiler* on 5,070 different JavaScript input files, on 132 bugs of this project. For each bug, we made a run on both faulty and fixed versions of the system and saved the pairs of outputs and method

Table 6: Internal/external FEP on mutants (RQ2)

Project Name	**Num of Execs**	**Externally Detectable**	**Int FEP**	**Ext FEP**
JFreeChart	25,842	6,217	0	0
Closure Compiler	320,678	180,562	2,587	4,783
Commons Lang	89,043	45,800	1,567	2,623
Commons Math	422,068	200,865	10,222	25,956
Mockito	16,284	7,321	0	0
Joda Time	15,460	8,970	0	0
Total	889,375	449,735	14,376	33,362

executions obtained. The output of *Closure Compiler* is also a JavaScript file and if the output files generated are different we consider the error to be *Externally Detectable.* Table 7 lists the ID of the bugs which we were able to execute with our inputs. 22 bugs out of 132 were executed leading to an overall number of 528 executions. For each of these 22 bugs there was at least one execution which was externally detectable, i.e. that caused a change in the output file generated by *Closure Compiler.* Overall, 424 out of 528 (80.3%) executions were externally detectable. 60 executions (11.4%) provide evidence of FEP occurring for 4 different bugs. For 8 executions (1.5% of all executions) of **Bug 1** we observed unit-level internal and external FEP. This bug affects neither the externally observable state of the class nor the final output of the system. However, it causes the program states in faulty and fixed versions to differ, which is evidence of both internal and external FEP. During the unit-level analysis our test case generator was not able to generate any test cases for this bug, therefore no unit-level FEP was reported in RQ1.

According to the Pearson-Klopper method for calculating binomial confidence intervals, system-level FEP is in the range [0.0878:0.1438], with mean = 0.1136, at confidence level 95%.

Table 7: System-Level FEP on real faults (RQ3)

Closure Bug ID	**Num of Execs**	**Externally Detectable**	**Sys FEP**	**Int FEP**	**Ext FEP**
1	20	12	8	8	8
4	15	5	8	0	0
8	200	159	41	0	0
13	36	24	0	0	0
16	15	10	0	0	0
20	22	22	0	0	0
21	4	4	0	0	0
22	4	4	0	0	0
29	1	1	0	0	0
34	13	10	0	0	0
50	1	1	0	0	0
52	13	13	0	0	0
56	2	2	0	0	0
60	5	5	0	0	0
62	57	50	0	0	0
63	57	50	0	0	0
87	23	12	3	0	0
115	3	3	0	0	0
116	4	4	0	0	0
127	14	14	0	0	0
131	9	9	0	0	0
133	10	10	0	0	0
Total	528	424	60	8	8

RQ3: *The prevalence of FEP changes when we test programs at the system level instead of unit level: 11.4% of the overall executions provide evidence of system-level FEP, which is substantially higher than the probability of unit-level FEP, both with real faults and with mutants.*

3.4 Qualitative Analysis: Factors Affecting FEP

To understand the reasons behind the absence of FEP in programs with real faults and their existence in the mutations, we performed a qualitative analysis on all the 384 bugs from the Defects4J benchmark and the mutations generated by Major. For methods from Defects4J, we manually compared the buggy version of the methods with the fixed version and analysed the bug fixes. As a result of this analysis we identified two main classes of explanations for the absence of FEP: (1) the fix of the bug affects the output directly; (2) the state change resulting from the fix is such that it always propagates to the output. In case (1), clearly both internal and external FEP are impossible, since all state deviations are immediately returned to the unit-level oracle. In case (2), the state change propagates to the output because the computation performed between the fault and the return statement does not "squeeze" the state (i.e., it never collapses correct and incorrect values into the same value, as happens e.g. with statement `x = x % 4` in Figure 1).

During manual analysis, one commonly occurring fix pattern was a change in the *return* statement of a method. For example, in Figure 4 the bug is at line 6 and the fix is as indicated within a comment at line 7. As this fix changes the return statement directly, it is not possible to observe any difference between the fixed and buggy versions at some internal point in the method, so no FEP can be observed in such cases. Another typical pattern for a bug fix is the addition of an *if* statement containing *return* or *throw* statements inside. In Figure 5 the bug is fixed by adding the *if* statement at lines 7-12. So whenever this *if* statement is executed, the method will return the object produced by the invocation at line 11. If this differs from the object generated by the faulty

```
1  public Complex divide(double divisor) {
2    if (isNaN || Double.isNaN(divisor)) {
3         return NaN;
4     }
5     if (divisor == 0d) {
6         return NaN;
7         //return isZero ? NaN : INF;
8     }
9     if (Double.isInfinite(divisor)) {
10         return !isInfinite() ? ZERO : NaN;
11     }
12    return createComplex(real / divisor,
13                            imaginary / divisor);
14 }
```

Figure 4: Commons Math Bug 46

version at line 14, the difference will be definitely observable by unit-level oracle. If it does not differ, we have coincidental correctness, but no FEP.

To quantify this class of FEP, we considered the edit scripts generated for trace alignment and used JavaParser to identify the following cases: (1) when the edit script contains a CHANGE operator which changes one return statement into another; or, (2) when the edit script contains an INSERT operator which adds an *if* statement containing a *return* or *throw* statement inside. Table 8 reports the number of occurrences of both cases. As we can see, in 32% of the methods the bug fix includes these type of changes.

Another typical pattern preventing the occurrence of FEP is when a state change resulting from a bug fix always propagates to output. In Figure 6 the bug is at lines 5-6 and the fix is as indicated at lines 7-8. If the buggy statement is executed, it might cause a difference in the value of the *chiSquare* variable. However, whenever this happens, this difference of value is ensured to always propagate to the return statement of the method, hence being externally

```java
public static LocalDate fromDateFields(Date date) {
  if (date == null) {
      throw new IllegalArgumentException
                ("The date must not be null");
   }

   //if (date.getTime() < 0) {
   //    GregorianCal cal = new GregorianCal();
   //    cal.setTime(date);
   //    return fromCalendarFields(cal);
   // }

   return new LocalDate(
      date.getYear() + 1900,
      date.getMonth() + 1,
      date.getDate());
  }
```

Figure 5: Joda Time Bug 12

observable. When there is no difference, we have coincidental correctness, but no FEP.

Table 9 shows the number of bugs for each project where this scenario holds. These cases were identified performing manual analysis on the bug fixed. Overall, it happens in 13% of the bugs.

As the results for RQ2 show, when real faults are replaced with mutants, there is evidence of FEP. To analyse the reasons behind that, we investigated mutants which lead to the occurrence of internal and external FEP. In Figure 7 we have method `getInitialDomain(double p)` and two mutations for it, `mut0` at line 3 and `mut1` at line 6, generated by Major. In case of `mut0`, whenever the if condition at line 7 is true, variable *ret* is reassigned a new value. So, while the values of *ret* at method's return and program point *pp_ret* in the buggy and fixed method are the same, they are different at program point

Table 8: Fixes affecting the output directly

Project	**Fixed Methods**	**Return Change**	**If Addition**
JFreeChart	39	11	3
Closure Compiler	172	21	13
Commons Lang	88	17	10
Commons Math	146	28	28
Mockito	76	28	7
Joda Time	51	9	6
Total	572	114	67

Table 9: Fixes directly propagating to output

Project Name	**Bugs**	**Fix visible at output**
JFreeChart	25	8
Closure Compiler	131	14
Apache Commons Lang	61	4
Apache Commons Math	104	17
Mockito	37	1
Joda Time	26	4
Total	384	48

pp1, which indicates the presence of internal FEP. Actually, the assignment at line 8 "squeezes" the information associated with variable *ret*, which is no longer available at the return point and externally.

For `mut1`, when the *if* statement at line 7 is false in the original, fixed program, variable *ret* keeps its initial value equal to 0.0. However, the value of variable *d* at program point *pp_ret* might be different from 0.0, since any value lower than or equal to 2.0 makes the *if* condition false. So we may observe two different values for variable *d* at program point *pp_ret* in original vs. mutated

```java
public double getChiSquare() {
      double chiSquare = 0;
      for (int i = 0; i < rows; ++i) {
          final double residual = residuals[i];
          chiSquare += residual * residual *
                             residualsWeights[i];
          //chiSquare += residual * residual /
          //             residualsWeights[i];
      }
    return chiSquare;
  }
```

Figure 6: Commons Math Bug 65

program, while in both versions the value of *ret* is the same, i.e., 0.0. This is a clear case of external FEP.

The conclusion from our qualitative analysis of FEP in mutants is that the effect of mutation operators on the program state and on the propagation of incorrect program states is substantially different from the effect of real faults.

3.5 Implications

The empirical results presented in this chapter have relevant implications for practitioners and researchers:

Inner oracles The absence of internal FEP when real faults are considered for Java units (classes) indicates that internal oracles do not have higher fault detection capabilities than return point or unit-level oracles when performing unit testing of classes. Rather than attempting to include assertions about the internal execution state, Java developers might better invest their time to strengthen the assertions that check the program state at return points or within test cases. In fact, if such assertions are sufficiently strong to capture any deviation from the expected execution state, they will miss no fault that manifests itself internally, because the internal state deviation tends to reach

```java
protected double getInitialDomain(double p) {
    double ret = 0.0;
    //mut0: double ret = 1.0;
    //pp1
    double d = getDenominatorDegreesOfFreedom();
    //mut1: d = 0.0;
    if (d > 2.0) {
        ret = d / (d - 2.0);
    }
   //pp_ret
    return ret;
 }
```

Figure 7: Commons Math Bug 95

them. Researchers interested in Java faults should focus on techniques to improve the oracles that can be defined at return points or within test cases, because these can be made equally effective as internal oracles.

The non-negligible occurrence of FEP at the system level indicates that checking the overall output of a system might be not enough and that probes for the intermediate computations should be inserted into the test case execution to avoid that the effects of faults disappear when proceeding to the computation of the overall system output. While such intermediate oracles can still be based on post conditions or test case assertions, and do not require the observation of internal execution states, they might represent a challenge for system level testing. In fact, at this testing level the system is usually considered as a black box, whose intermediate steps are not visible. According to our results, monitoring and checking such intermediate steps is quite important for avoiding system FEP.

Post-conditions The *programming by contract* method prescribes that every method be equipped with pre-conditions, post-conditions and invariants. This approach to programming offers several benefits, among which are the following possibilities: to formally express the specifications that each method must satisfy, in a way that is machine interpretable; to reuse the oracle across test cases; to document a method in an unambiguous way. One may question what part of the execution state should be checked in a post-condition. In fact, at return points the whole internal state of the method under test is accessible. According to our results, the absence of internal FEP indicates that checking the externally visible effects of a method execution is enough to expose faults as soon as they corrupt the execution state. It is unlikely that the effort to create internal oracles will be beneficial for early fault exposure. Rather, post-conditions at return points can be focused on the externally visible effects of the execution, disregarding the inner details. This is consistent with the programming by contract paradigm, where only the externally visible contract is typically specified.

Subsystem testing The higher prevalence of failed error propagation at system level over unit level might indicate that testing subsystems of the software in isolation could make it easier to expose bugs. While the effect of a bug is externally visible in the class to which it belongs, it is not always visible at the level of the whole system. This supports the idea of bottom-up integration testing, in which we build on unit-level results by testing higher-level combination of units in successively more complex scenarios.

Mutants vs. real faults The software engineering community has witnessed a long debate on the use of mutants as surrogate for real faults [4, 52]. Such a replacement may be valid for the purpose of evaluating the adequacy of a test suite, owing to the high correlation between mutation score and fault detection rate. We, rather, are interested in investigating the propagation of an error to the oracle that can detect it. Our results show that such propagation is less prevalent with mutants, while it is always successful with real faults. Our qualitative analysis indicates that mutants corrupt the internal state differently from real faults. In fact, the latter state corruption tends to always generate an externally visible misbehaviour, while the former might remain invisible if only the external state is inspected. Hence, practitioners should not decide where to place their oracles based on the propagation of errors as simulated with mutants. Researchers could instead investigate mutation operators that behave similarly to real faults with respect to the propagation of the corrupted internal state to the externally visible state.

Previous work Previous work on failed error propagation tended to suggest that there is a nontrivial proportion of faults that manifest the FEP property. Our results differ markedly from these previous findings. One possible explanation could be differences in the subjects and the types of faults. Daran et al. [24] analysed 12 real faults in a C program with 1000 lines of code. Wang et al. [104] analysed 38 real faults in a C program with 6000 lines of code. By comparison the Defects4J contains 395 real faults which come from six large Java projects. An intriguing possibility lies in the potential differences between the two language (C vs. Java) styles; perhaps some programming languages have inherently higher or lower failed error propagation propensity than others. Hence, one of the implications of our findings is the pressing need for further work on FEP in different programming languages and corpuses. Taken together, our findings and those in the previous literature do tend to suggest that there may be differences between different programming paradigms with respect to error propagation behaviour, and that there are certainly differences between unit and system level FEP. These differences clearly have implications for software testability [16, 103], because FEP tends to inhibit testability. Such findings may also suggest testability transformations [42, 71] that could reduce the likelihood of failed error propagation, leading to reformulations of software systems (e.g., by inserting probes for intermediate steps) that are inherently more testable.

3.6 Threats to Validity

In this section we discuss potential threats to the validity of our empirical findings. These are mostly in the external and internal validity categories.

Threats to *external validity* affect the generalisation of our results. We carried out our experiments on a well established benchmark for Java, Defects4J, which includes 395 real bugs from 6 different projects. While Defects4J is becoming de-facto a standard benchmark for Java testing, replication of our study on further subjects beyond Defects4J would be quite important. We do not claim generalisability to programming languages other than Java. On the contrary, we suspect that the programming style of Java, which encourages the decomposition of the software into small computations assigned to methods, favours the creation of code units where information is not squeezed when propagating from inner states to the output. Other programming styles might favour the creation of longer and more complex computational units, where information squeezing might be more likely to occur, due e.g. to variable reassignments, which erase and replace the information hold by the reassigned variables..

Threats to *internal validity* come from factors that could influence our results. Among them, the most important factor that influences our conclusions on the differences between real faults and mutants on FEP, is the set of mutations that have been considered. To limit such a threat, we used a well-established mutation analysis tool, Major. However, different tools and different mutation operators might lead to different sets of synthetic faults. Moreover, we have not been able to perform mutation analysis of all the buggy methods available in Defects4J, because of the enormous computation time involved, since we generate test cases for all mutants that Major produces for each method. We have defined a sampling strategy that takes method size into account, in order to consider representatives of the various possible method size categories. However, this does not ensure that the results obtained on the selected sample would remain exactly the same if extended to the entire dataset of the buggy methods.

Another factor that might have influenced the results is the way we filtered equivalent mutants from the full set of mutants generated by Major. We *conservatively* kept only killable mutants. This means that among the excluded mutants, some may be non equivalent and may be subject to FEP. As a consequence, when mutants are considered instead of real faults, our measures of internal/external FEP are conservatively underestimating the true values. Even with such a conservative underestimation, we observed a non negligible number of occurrences. Our conservative underestimation may also explain the lower incidence of FEP on mutants in comparison with the values reported in the literature [68, 111].

Finally, our results are potentially affected by the limitations of the test generator used to exercise the faults. EvoSuite was indeed unable to generate large test suites for some faults and EvoSuite might have produced larger test suites if given additional test generation budget. To avoid that small test suites could affect our results, we have excluded all test suite with less than 150 test cases. The test generation budget allocated to EvoSuite (10,000 seconds per test suite) was the maximum compatible with the overall duration of the empirical study.

3.7 Conclusions and Future Work

In this chapter we have presented empirical evidence from a large corpus of real-world faults in Java systems that reveals a surprisingly low unit-level FEP amongst the 386 faults studied. These empirical findings contradict earlier work on failed error propagation and, if replicated in other fault corpuses and/or for other languages, would have profound implications for software testing. On the other hand, with system-level inputs we get a substantially higher rate of FEP. This shows that when oracles are defined for an individual Java class, postconditions that predicate on the externally observable state or test case oracles are sufficient to detect faults as soon as they corrupt the internal state. On the contrary, when we analyse a complete software system, the output alone does not provide enough information to expose faults as soon as they manifest themselves, necessitating the observation of intermediate computation steps.

When we turn our attention to studying the synthetic faults introduced by program mutants (a widespread practice believed to be good at simulating real faults), we find noticeably different behaviour at the unit level: the artificial faults denoted by mutants *do* exhibit substantial FEP, unlike the real faults we studied. While such synthetic faults may be good proxies for estimating whether test cases that reveal them will also reveal real faults, there do appear to be non-trivial differences in the behaviour of synthetic faults and real faults, with respect to their error propagation in Java classes.

These findings suggest further work to investigate the prevalence of FEP in other programming languages and bug data sets, and the need to further investigate the relationship between mutation testing and real faults. We studied only single faults, but future work could also extend our findings to multiple faults, which may have additional implications for higher order mutation testing, one of the main motivations of which is the ability to model fault masking.

4 Oracle Assessment and Improvement

In this section, we introduce our approach to oracle assessment and improvement that is based on search based test case generation [30, 43, 70] to identify false positives and mutation testing [48, 53] to identify false negatives. Our technique generates counterexamples as test cases that demonstrate incompleteness and unsoundness, which the developer then 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.

The main contributions of this chapter are:

1. 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. A novel iterative oracle assessment and improvement approach and its implementation.

4.1 Formal Model

4.1.1 Quality of Assertions

Let us consider a program point, pp, in some software under test (SUT), 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 denote R_{pp} as a 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.

Although this knowledge is generally unavailable to developers, for the sake of the formalisation we indicate by E_{pp} the set of states that occur at pp and are correct (the perfect oracle). One may think of E_{pp} as the intersection between the set of correct states at pp for some "ghost program"[5], G, an error free version of the software under test, and R_{pp}, the reachable states of the SUT:

To make a state comparison possible between the two program versions, we assume that the differences between G and P are sufficiently small and that pp occurs in both programs. We can drop the subscript pp and use R,E and A when pp is clear from the context. The relationship between R, E and A at pp can be represented in a Venn diagram as shown in Figure 8.

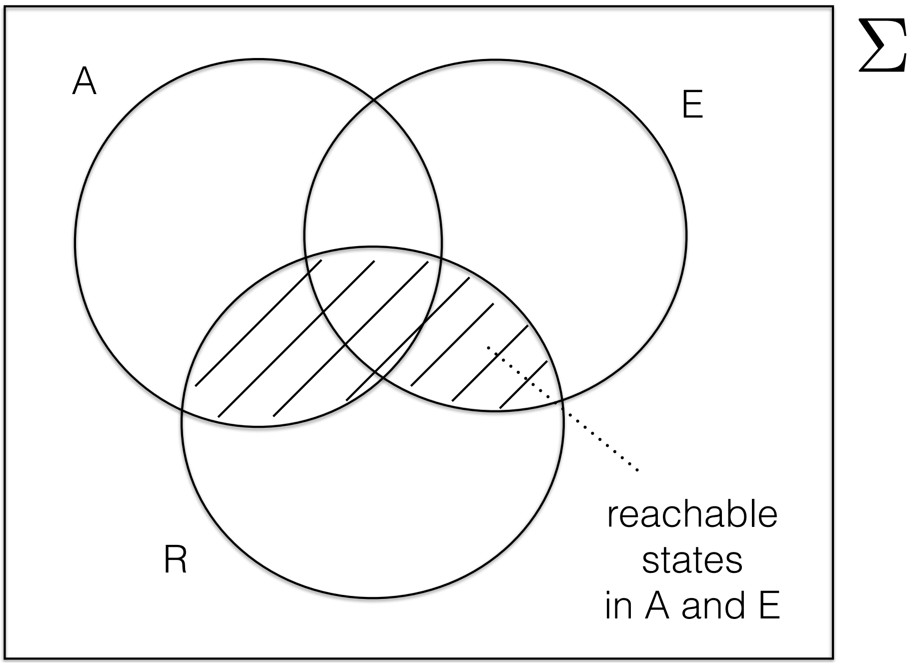

Figure 8: Relationship between R, E and A

The overall aim of the testing process is to make the software behaviour as close as possible to the expected/intended behaviour. At the end of this process we will have adjusted the states of the SUT at pp so that $E \cap R$ is as large as possible (and similarly for all the other program points in the SUT). However, our focus is on the oracle improvement: improving $\langle assert \rangle$ so that we obtain a new assertion, $\langle assert \rangle'$ for which the domain of True has as large an overlap with E as possible, i.e. improve the size of $A \cap E$. Ideally we would like 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. However, the set of states that we *actually* have access to, and can test, are the states of the SUT, i.e. the states in R. In terms of the relationship between A and E these are the partitions $A \cap E \cap R$, $(A - E) \cap R$ and $(E - A) \cap R$ in Figure 8.

The situation can be represented more simply, as in Figure 9, by taking R as the set universe. Here, the region $(A - E) \cap R$ is the set of states of the SUT which are not "correct" but on which $\langle assert \rangle$ is True, that is the set of false negatives, while $(E - A) \cap R$ are the set of correct states on which the assertion is false, that is the set of reachable false positives.

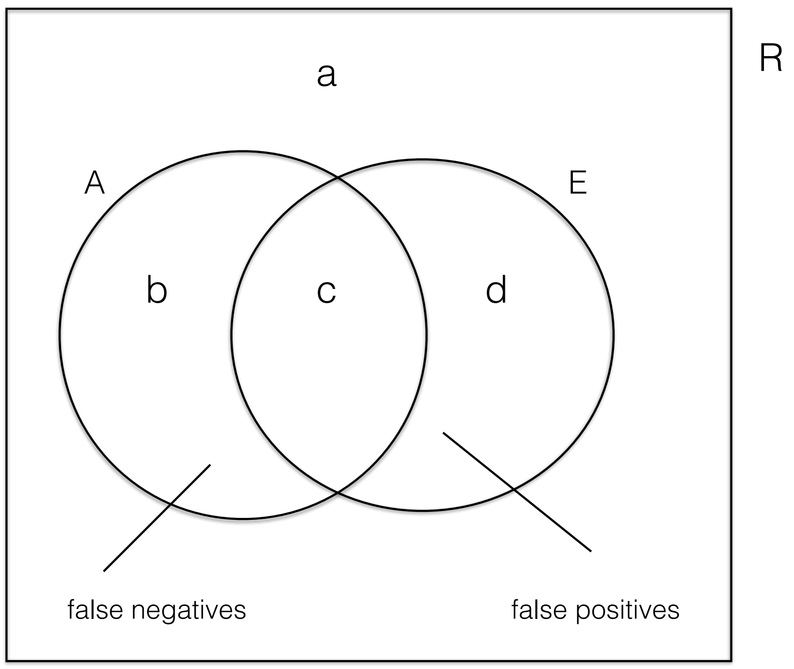

Figure 9: E and A limited by R

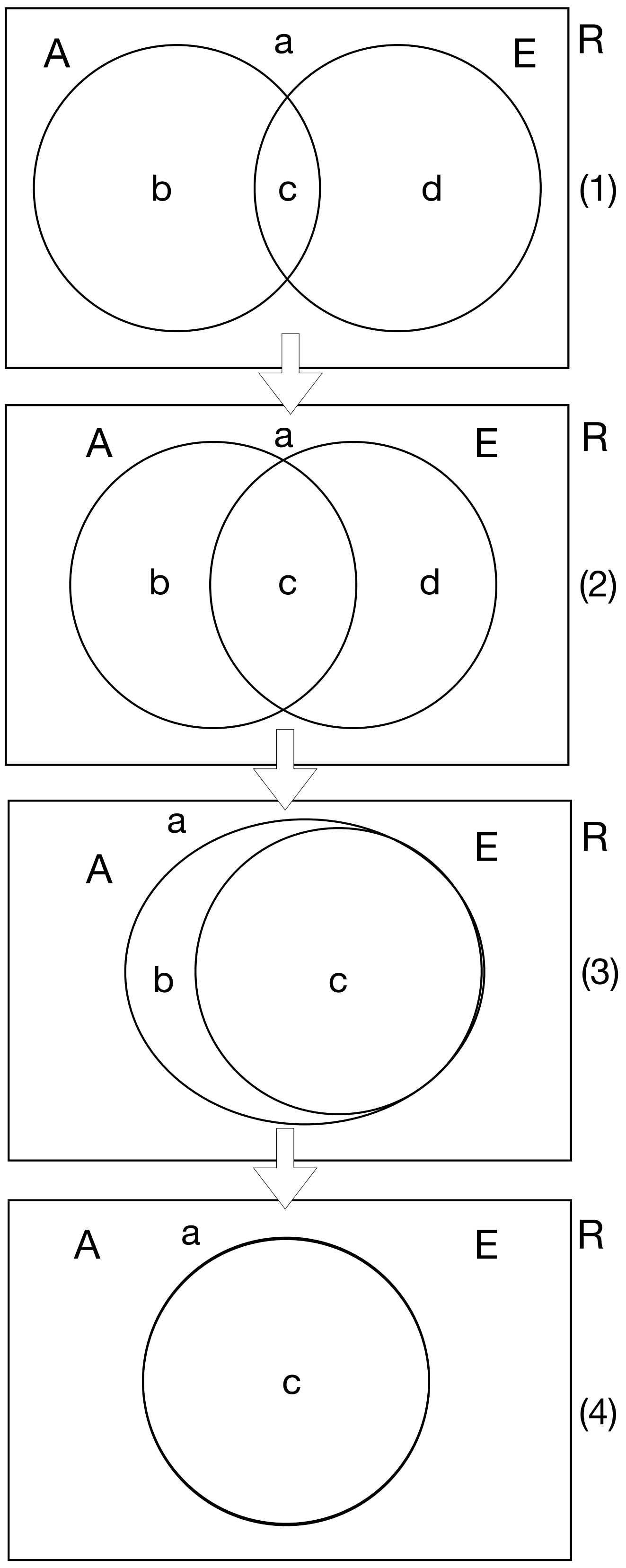

Figure 10: The Assertion Improvement Process

Definition 6 (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 7 (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.*

Our proposed *assessment process* tests oracles in terms of presence of false positives and false negatives, which we call *oracle deficiencies.* 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.

Our proposed *improvement process* strengthens $\langle assert \rangle$ to reduce the number of oracle deficiencies, producing a new assertion $\langle assert \rangle'$. This process is illustrated in Figure 10: the initial assertion in diagram (1) is improved into new assertion in diagram (2) with fewer false positives and false negatives. The number of false negatives and false positives (in Figure 10 the sizes of b and d respectively) are the indicators of *oracle quality.* In the ideal situation, after the improvement process, we should have a *Fully Correct* final oracle. However, generating fully correct oracles might be an expensive and difficult process, as an oracle that detects all faults could be as complex as the system under test itself. Therefore, a *Partially Correct* oracle might be regarded as sufficiently adequate in practice.

Definition 8 (Full Correctness) *An oracle is fully correct if it has no false positives and no false negatives, i.e. it is both complete and sound.*

Definition 9 (Partial Correctness) *An oracle is partially correct if it has no false positives, but has false negatives, i.e. it is complete, but not sound.*

Diagram (3) in Figure 10 shows a step in the improvement process where partial correctness has been achieved. Here, the size of d is zero (no false positives), while the size of b is not (there are false negatives). The size of b can be quantified to indicate the level of partial correctness. Diagram (4) in Figure 10 demonstrates the case of full correctness: both b and d has size zero (no false positives and no false negatives), i.e. A = E. The improvement process terminates with a more complete and sound oracle once the desired level of either partial or full correctness has been achieved.

4.1.2 Information Theory Based Model

In what follows we consider a probability distribution on the set of states that occur at a program point in a program. Such a probability distribution can be formally constructed by considering the semantics of a program (e.g., Cousot and Cousot's reachability semantics, which is an abstract interpretation of their partial trace semantics [21]) and then applying Kozen's principles for building probabilistic semantics for programming languages on the basis of input distributions and non-probabilistic semantics [54].

Consider R_{pp} as above and let σ be the normalised probability distribution on the members of R (i.e., we consider R as a random variable on the program states that reach *pp*).

Let $o : R \to Bool$ be an oracle on R. Since o induces a probability distribution on *Bool* from the one on R, o is a random variable on *Bool*. In fact any random variable corresponds to a partition over some event space equipped with a probability distribution [20], with a Boolean valued random variable being simply a binary partition on the domain event space. Let $\mathcal{O}_R$ be the set of all possible binary partitions on R. This is also the set of all possible oracles on R interpreted as random variables.

Suppose that we have two oracles, $\alpha, \gamma \in \mathcal{O}_R$, that can observe the states at *pp* and make decisions as to whether they are correct. Here, α is an oracle created from an assertion by our program transformation techniques and γ is an oracle that is ideal in the sense that it perfectly encapsulates ground truth knowledge about the intended, correct behaviour of the program, P (we introduce γ to support theoretical analysis, but we do not assume that γ is explicitly available in practice). As described before, the aim of our improvement process is to make practical oracle α more similar to ideal oracle γ by reclassifying α's false positives and false negatives so that they align with the decisions of γ.

An oracle improvement step either reclassifies some states that were false positives as true negatives or reclassifies some states that were false negatives as true positives. These two kinds of steps are quite independent of each other and may change the labelling on different numbers of states. Each step creates a new, improved oracle from the old one. We model these two kinds of steps as self maps on the domain of oracles on R.

$$N, \Pi : \mathcal{O}_R \to \mathcal{O}_R$$

Since we are using testing, i.e., a dynamic, incomplete method, the approach is necessarily existential, that is, in each step we discover either *some* false positives or *some* false negatives, so we interpret a self map in an existential way. An N step converts some states incorrectly labelled by α as positive (i.e. failures) to negative in better alignment with γ

$$\exists s \in R \;.\; \alpha(s) = \mathrm{F} \wedge \gamma(s) = \mathrm{T} \wedge N(\alpha)(s) = \mathrm{T}$$

while a Π step converts false negatives to true positives.

$$\exists s \in R \;.\; \alpha(s) = \mathrm{T} \wedge \gamma(s) = \mathrm{F} \wedge \Pi(\alpha)(s) = \mathrm{F}$$

We model our oracle improvement process using Shannon's information theory [92]. With reference to the diagram in Figure 9, we interpret the regions labelled a, b, c, d as probability masses:

$$a = p(\alpha = \mathrm{F}, \gamma = \mathrm{F}) = \textstyle\sum_{s \in R \wedge \alpha = F \wedge \gamma = F} \sigma(s)$$

$$b = p(\alpha = \mathrm{T}, \gamma = \mathrm{F}) = \textstyle\sum_{s \in R \wedge \alpha = T \wedge \gamma = F} \sigma(s)$$

$$c = p(\alpha = \mathrm{T}, \gamma = \mathrm{T}) = \textstyle\sum_{s \in R \wedge \alpha = T \wedge \gamma = T} \sigma(s)$$

$$d = p(\alpha = \mathrm{F}, \gamma = \mathrm{T}) = \sum_{s \in R \wedge \alpha = F \wedge \gamma = T} \sigma(s)$$

Removal of false negatives can be seen as a repartition of R so that $b' \leq b$ and $a' \geq a$ and $a + b = a' + b'$, i.e. the probability of false negatives is reduced and the probability of true positives increased by the same amount. Similarly, removal of false positives can be seen as a repartition of R so that $c' \geq c$ and $d' \leq d$ and $c + d = c' + d'$. The oracle α is complete when $d = 0$ and is sound when $b = 0$.

The probability of oracle α detecting true faults is:

$$p(\gamma = \mathrm{F} \mid \alpha = \mathrm{F}) = \frac{a}{a + d} \tag{1}$$

Similarly, the probability of oracle α accepting correct executions is:

$$p(\gamma = \mathrm{T} \mid \alpha = \mathrm{T}) = \frac{c}{b + c} \tag{2}$$

A reduction, Π, of false negative probability repartitions the probability weights to create a new oracle, α' where $a' = a + \Pi$, $b' = b - \Pi$. Similarly, reducing false positive probability by N creates a new oracle, α', where $c' = c + N$, $d' = d - N$[7]. Note that the two operations are independent and that while A changes to A', E does not change (see Figure 10). The intuition is that α' is a better approximation to γ than α.

Proposition 1 *Oracle improvement increases conditional probabilities of detecting true faults and of accepting correct executions.*

$$p(\gamma = T \mid \alpha = T) = p(\gamma = T, \alpha' = T)$$
$$p(\gamma = F \mid \alpha = F) = p(\gamma = F, \alpha' = F)$$

Proof: As it was noted before, false negative reduction causes the following repartitions: $a' = a + \Pi$, $b' = b - \Pi$.

The Equation 1 and the fact that $b' > b$ lead to the following inequation:

[7]With some notation overload, we indicate with the same letters N, Π the self maps modelling oracle improvement as well as the amount of oracle improvements, measured as the removed false negatives/positives, since the context allows for an easy disambiguation.

$p(\gamma = \mathrm{T} \mid \alpha = \mathrm{T}) = \dfrac{c}{b+c} < \dfrac{c}{b'+c} = p(\gamma = \mathrm{T}, \alpha' = \mathrm{T})$

For the repartition of a,

$$p(\gamma = \mathrm{F} \mid \alpha = \mathrm{F}) = \frac{a}{a+d} = \frac{1}{1+\frac{d}{a}} < \frac{1}{1+\frac{d}{a'}} = p(\gamma = \mathrm{F}, \alpha' = \mathrm{F})$$

Similar proof can be performed also for false positives. □

We can measure how closely connected two random variables (oracles) are by measuring their mutual information, a measure of their lack of independence [22]:

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

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

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

$$\mathcal{I}(\alpha;\ \gamma) = \begin{cases} -(b+c)log_2(b+c) - (a+d)log_2(a+d) \\ -(a+b)log_2(a+b) - (c+d)log_2(c+d) \\ +a\, log_2\, a + b\, log_2\, b + c\, log_2\, c + d\, log_2\, d \end{cases} \tag{3}$$

One might conjecture that mutual information always increases as the oracle is being improved. However this is not necessarily true, as neither $f(x) = x\, log(x)$ nor $\mathcal{I}(A;\ B)$ are monotonic functions. In fact f is concave on x in the interval $[0, 1]$ and $\mathcal{I}$ is concave on (A, B) in the interval $[0, \infty)$. For example, suppose we have an oracle α with the following probability masses:

$$a = \frac{1}{8},\ b = \frac{3}{8},\ c = \frac{1}{4},\ d = \frac{1}{4}$$

Therefore, α disagrees with γ on fail $\dfrac{3}{8}$ of times and the mutual information between them $\mathcal{I}(\alpha;\ \gamma) = 0.0487$. An improvement step on α may lead to a new oracle α' and the following probability masses change:

$$a = \frac{1}{4},\ b = \frac{1}{4},\ c = \frac{1}{4},\ d = \frac{1}{4}$$

However, mutual information $\mathcal{I}(\alpha';\ \gamma) = 0$, i.e., it has decreased.

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

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

Proof: Given a, b, c, d, the mutual information $\mathcal{I}(\alpha';\ \gamma)$ can be written as a function of Π as follows:

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

To find the values of Π which make it increase, we need to find the points of minimum of $\mathcal{I}(\alpha';\ \gamma)$. For this, we calculate the derivative of $\mathcal{I}(\alpha';\ \gamma)$ in terms of Π.

$$\frac{d\mathcal{I}}{d\Pi} = log_2(b+c-\Pi) + \frac{1}{ln2} - log_2(a+d+\Pi) - \frac{1}{ln2} + log_2(a+\Pi) + \frac{1}{ln2} - log_2(b-\Pi) - \frac{1}{ln2} = log_2\frac{(b+c-\Pi)(a+\Pi)}{(a+d+\Pi)(b-\Pi)}$$

$$log_2\frac{(b+c-\Pi)(a+\Pi)}{(a+d+\Pi)(b-\Pi)} = 0$$

$$\frac{(b+c-\Pi)(a+\Pi)}{(a+d+\Pi)(b-\Pi)} = 1\ ,\ \Pi = \frac{bd - ac}{c + d}$$

So, when $\Pi > \frac{bd - ac}{c + d}$, the mutual information increases. $\square$

If we consider a step that improves false positives by N and we compute the derivative of $\mathcal{I}(\alpha';\ \gamma)$ in terms of N, we obtain a very similar result:

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

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

Proof: Immediate from Theorem 1, via substitutions. □

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. We denote an oracle for which $ac < bd$, i.e. an oracle for which the product of the probabilities of the inaccuracies is bigger than the product of the probabilities of its accuracies, as a *bad oracle*. For such an oracle there exists a *symmetric* corresponding oracle, called *good oracle*, for which $ac > bd$ and the mutual information with γ does not change. An example of a bad oracle and its corresponding good oracle is shown in Figure 11.

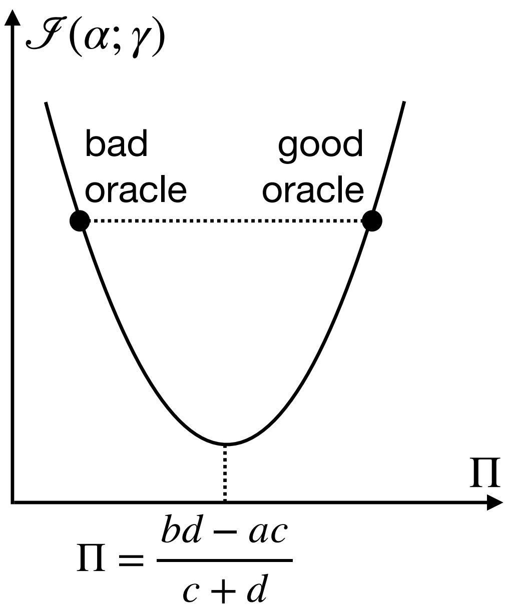

Figure 11: Bad oracles

Proposition 2 *Given a bad oracle $\alpha[a, b, c, d]$, the symmetric oracle $\alpha'[a', b', c', d'] = \alpha[b, a, d, c]$ is a good oracle with the same mutual information as $\alpha[a, b, c, d]$.*

Proof: α is a bad oracle, therefore $bd < ac$. The proposed transformation suggests that $a'c' = bd$ and $b'd' = ac$, as a result, $b'd' > a'c'$, which proves that α' is a good oracle. Mutual information remains the same, as replacing a with b and c with d in Equation 3, does not change its value. □

Corollary 2 *Oracle improvement increases mutual information between actual and perfect oracle assuming bad oracles are first transformed into good oracles by negation.*

Proof: If $ac > bd$ the mutual information increases because of Theorem 1. If $ac < bd$ the negated (symmetric) oracle $\alpha' = \alpha[b, a, d, c]$ satisfies the condition $a'c' > b'd'$, hence mutual information is ensured to increase because of Proposition 2. $\square$

The case of an initially decreasing function $\mathcal{I}(\alpha;\ \gamma)$ corresponds to an extremely poor initial oracle whose behaviour is opposite to the expected one. Theorem 2 shows that such a bad oracle can be made into a good one by simply swapping a with b and c with d. This swap can be interpreted as negating the oracle predicate, since the swap just accounts for giving opposite results for false negatives/true positives (resp. false positives/true negatives). Therefore, for assertion oracles it means just negating the assertion's verdict. This negation will cause the assertion oracle to jump over the minimum of $\mathcal{I}(\alpha;\ \gamma)$ and to reach a region where $\mathcal{I}(\alpha;\ \gamma)$ is monotonically increasing, as indicated in Theorem 2 and Figure 11.

Proposition 3 *For good oracles, the probabilities of the perfect oracle are lower bounds for the conditional probabilities.*

$$p(\gamma = T, \alpha = T) > p(\gamma = T)$$
$$p(\gamma = F, \alpha = F) > p(\gamma = F)$$

Proof: $ac > bd \implies a > \dfrac{bd}{c} \implies a + b + c + d > b + c + d + \dfrac{bd}{c}$.

As $a + b + c + d = 1$, $1 > b + c + d + \dfrac{bd}{c}$.

Therefore, $c > bc + c^2 + dc + bd = (b + c)(c + d) \implies \dfrac{c}{b + c} > c + d.$

The last expression is equal to:

$$p(\gamma = \mathrm{T}, \alpha = \mathrm{T}) > p(\gamma = \mathrm{T}).$$

A similar derivation holds also for fault detection, therefore:

$$p(\gamma = \mathrm{F}, \alpha = \mathrm{F}) > p(\gamma = \mathrm{F}).$$

□

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.

4.2 Approach

In this section, we describe our technique for oracle improvement via false positive and false negative detection.

4.2.1 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 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 such execution scenarios (test cases), we perform a testability transformation [41] that transforms the criterion for false positive detection into the standard branch coverage criterion.

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

Class `Subtract` in Figure 12 (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

```java
public class Subtract {
    public double value(int x, int y) {
         int result = x-y;
         assert (result != x);
         assert (result == x-y);
        return result;
    }
}
```

```java
public class Subtract {
    public double value(int x, int y) {
         int result = x-y;
         if (!(result != x)) {}; // target
         if (!(result == x-y)) {}; // target
         return result;
    }
}
```

```java
//Subtract.value(II)I:Branch Line 4
@Test(timeout = 4000)
public void test0() throws Throwable {
  Subtract subtract0 = new Subtract();
  int int0 = subtract0.value(0, 0);
}
```

Figure 12: Example of False Positive Detection

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 12 (middle).

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 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 12, 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.

In our approach assertions are part of the source code, but they should not cause any side effects. Therefore it is unacceptable that they lead to an exception during program execution, i.e. cause a *Crash*. 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 crashing assertion.

4.2.2 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 [35] 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 [41] 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 mutant M_j we create a transformed version of P, P'_j, 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 $(((c_i == c_i^{M_j})$ && $(v_1^{M_j} \neq v_1 \mid\mid \ldots \mid\mid v_{m_i}^{M_j} \neq v_{m_i})) \mid\mid (v_1^{M_j} == v_1$ && $\ldots$ && $v_{m_i}^{M_j} == v_{m_i}))$ {}

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 and P.
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'_j by calling the public setter methods created at Step 1.
3. The test case generator runs the strongly killing test case on P'_j.
4. If the test case executed on P'_j covers all the target branches 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 branches, 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 at least one of the considered assertions (and at least one of the variables in its scope has a different value in P vs. M_j); but, (3) the outcome of all the assertions 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 assertions are not strong enough to capture the difference between P and M_j, although at least one variable accessible to the assertions has indeed a different value between the execution of P and that of M_j.

Figure 14 shows an example of the described transformation for the class in Figure 13 (top). Fields `max_m1`, `a_m1`, `b_m1`, `max_m2`, `a_m2` and `b_m2` together with the respective setter methods, are added to class `FastMath`, to store the values of the variables visible at Lines 9 and 10 in Figure 13 (top) and observed during the execution of the mutant. The assertions at Lines 9 and 10 in Figure 13 (top) become the `if` conditions at Lines 18 and 22 in the transformed program shown in Figure 14. The `then` branches of these conditional statements are the targets 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 both of these targets, we obtain evidence of a false negative. In fact, although such a test case can strongly kill the mutant, the assertions `(max >= a)` and `(max >= b)` do not fail (provided they 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 M_1 that changes the assignment `max = b;` at Line 7 in Figure 13 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. However, the assertion at line 9 passes on

```java
public class FastMath {
    public int findMax (int a, int b) {
       int max;
       if (a >= b) {
        max = a;
       } else {
        max = b; // M1: max = a; M2: max = b + 1;
       }
       assert (max >= a);
       assert (max >= b);
       return max;
  }
}
```

```java
  //1. findMax, Line 7 IINC +1(max:0,2)
 @Test(timeout = 4000)
 public void test0() throws Throwable {
   FastMath fastMath0 = new FastMath();
   int int0 = fastMath0.findMax(0, 1);
 }
```

Figure 13: Class FastMath: An Example of a False Negative

```java
public class FastMath {
    private int max_m1, a_m1, b_m1;
    private int max_m2, a_m2, b_m2;
    public void setMax_m1(int max_m1) { this.max_m1 = max_m1; }
    public void setA_m1(int a_m1) { this.a_m1 = a_m1; }
    public void setB_m1(int b_m1) { this.b_m1 = b_m1; }
    public void setMax_m2(int max_m2) { this.max_m2 = max_m2; }
    public void setA_m2(int a_m2) { this.a_m2 = a_m2; }
    public void setB_m2(int b_m2) { this.b_m2 = b_m2; }
    public int findMax (int a, int b) {
        int max;
        if (a >= b) {
          max = a;
        } else {
          max = b;
        }
        if (((max_m1 >= a_m1) == (max >= a) &&
             (max_m1 != max || a_m1 != a || b_m1 != b))
                   || (max_m1 != max && a_m1 != a && b_m1 != b))
         {} // target1
        if (((max_m2 >= b_m2) == (max >= b) &&
             (max_m2 != max || a_m2 != a || b_m2 != b))
                   || (max_m2 == max && a_m2 == a && b_m2 == b))
         {} // target2
        return max;
   }
}
```

Figure 14: Class Transformation for False Negative Detection

both original and mutated programs, since on both we have that `max >= a` is true. On the other hand, the assertion at line 10 does not pass on the mutated program, as `max >= b` is false. Therefore, there is no false negative, as at least one of the assertions reacts to the injected fault.

Another mutant M_2 changes the assignment `max = b;` at Line 7 in Figure 13 into `max = b + 1;`. The test case `TC=(0,1)` again strongly kills the mutant. However, for this mutation both the assertions at Line 9 and 10 pass, as the value of `max` is greater than the value of both `a` and `b`. So, this test case and mutation show 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 13 with `assert (max >= a && max >= b && (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 4.2.3 below.

4.2.3 Iterative Improvement Process

We propose a process for iterative oracle assessment and improvement based on the outcome of false positive/negative detection, see Figure 15. 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 needed in software processes that include complete formal specifications, from which oracles are derived automatically. In our experience, industrial practice usually does not currently encompass complete formal specification, hence we think the proposed approach has wide applicability.

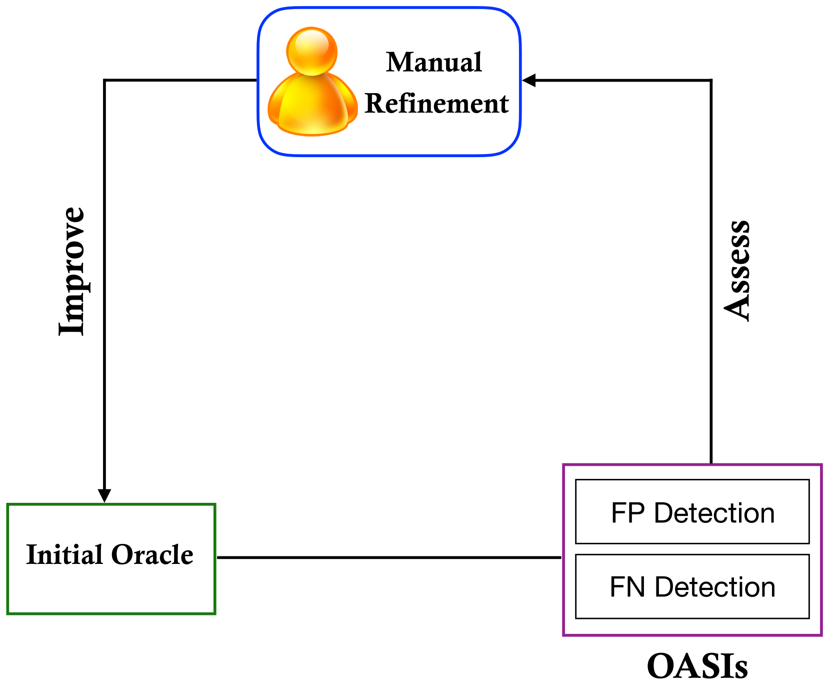

Figure 15: Iterative Improvement Process

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 [27], or can be even the empty (vacuous) oracle. Oracle deficiencies 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, 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. The last two cases are expected to occur very rarely in practice and actually have never occurred during our experiments.

Depending on the improvement step the developer has taken to fix the assertion, the new assertion can, in the best case, be fully correct, can have an oracle deficiency (of the same or new type) or can lead to a *Crash* (e.g., due to an exception) in the program. Figure 16 shows all the possible state changes for the assertion during the improvement process.

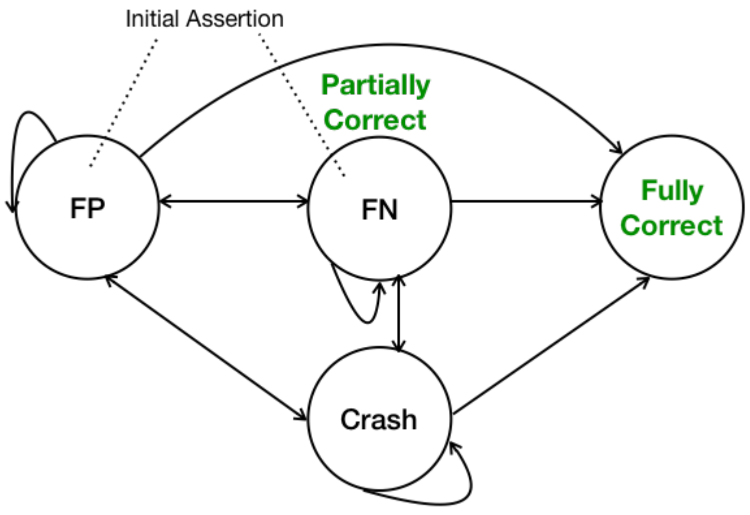

Figure 16: Oracle Improvement Process: Possible Outcomes

To demonstrate examples of each state change, let's see the shortened version of class *StackAr* in Figure 17. In method `pop` there is an initial assertion which has a False Negative. Figure 18 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 18 causes a *Crash*. It can lead to *ArrayIndexOutOfBoundsException*, if the value of variable `topOfStack` is equal to -1 when the assertion gets executed. 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);
  }
}
```

Figure 17: 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 18. 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 full 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).

```
(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))
```

Figure 18: Class StackAr, Method pop: Examples of Improved Assertions

This shows the case when instead of fully correct assertions, partially correct ones as the initial one in method pop or the third one in Figure 18 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.

After each improvement step, the iterative process restarts and the new assertions are assessed for the presence of further oracle deficiencies. After some iterations of oracle refinement, no oracle deficiencies will be reported to the user. This means the oracle has been strengthened to solve the reported false negatives and false positives, eventually getting closer and closer to the "ghost" program oracle E (see Figure 10). The overall outcome of the process is the improved oracle together with the bugs that such an improved oracle can find.

4.3 Implementation

We have implemented our approach for false positive and false negative detection as a command-line tool OASIs (**O**racle **AS**sessment and **I**mprovement), see Figure 19. OASIs takes five parameters as input: source code location of the Java class, the name of the class, the name of the method where the initial assertions are located, the search budget for false positive detection and the search budget for false negative detection. The last two parameters are optional and, if omitted, OASIs uses the default budgets of 60 seconds for false positive and of 120 seconds for false negative detection. OASIs starts the oracle assessment process by first looking for a false positive. If no false positive is detected, the search for false negatives is initiated. The output of the tool consists of a message which, in case an oracle deficiency is detected comprises the exact kind, or just indicates that no deficiency was found. For each detected oracle deficiency, the evidence (in the form of a test suite) is provided.

4.3.1 False Positives

For false positive detection we first perform a testability transformation that transforms the assertion in the code into a new branch. For this we use Java-

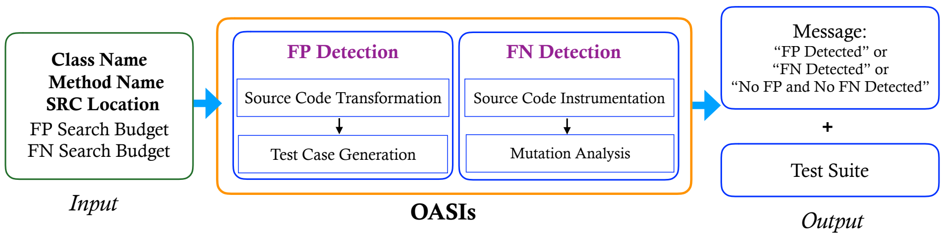

Figure 19: OASIs Components

Parser[8] (version 2.3.1) which provides a set of tools to parse, analyze, transform and generate Java code. The source code transformation also detects the lines of code where the newly created branches are located and passes them to the test case generator, so that these branches can be differentiated from the already existing ones. Our test case generator to cover these newly created branches is implemented as an extension of the EvoSuite[9] [29, 30] test case generator (version 1.0.5).

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'. The original fitness function [31] (to be minimized) for branch coverage, denoted $f_B(T)$, measures the number of methods not executed by keeping track of the set of executed methods F_T out of the set of all methods F and adds to it the sum of the minimal normalized branch distances $d(b, T)$ for each branch $b \in B$:

$$f_B(T) = \mid F \mid - \mid F_{\mathrm{T}} \mid + \sum_{b \in B} d(b, T)$$

where:

$$d(b,T) = \begin{cases} 0 & \text{if the branch has been covered} \\ \nu(d_{\min}(b,T)) & \text{if the predicate has been} \\ & \text{executed at least twice} \\ 1 & \text{otherwise} \end{cases}$$

[8]http://www.javaparser.org
[9]http://www.evosuite.org

with ν a normalisation function, such as $x/(x+1)$.

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 [31] into:

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

Once the test suite is generated, it is reported to the developer as evidence of a false positive. The test case in Figure 12 (bottom) shows an example of such a report for the method in Figure 12 (top). Along with providing a test case, that will make the program assertion in the method fail, the output of the tool also specifies in the comments the line number (Line 4 in this case) where the failing assertion is located.

4.3.2 False Negatives

As described in the previous section, the approach for the detection of false negatives is also based on the branch coverage test case generation criteria. However, creating a transformed version of program P for each mutation is quite inefficient. For that reason, in the prototype implementation instead of the branch coverage we have adapted the strong mutation coverage criteria of EvoSuite.

First, we instrument the source code of the class using JavaParser so that we can monitor (1) the values of all variables visible at the program points where the assertions are located and (2) the outcome of the assertions, i.e. whether they pass or fail.

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 [31]. 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. These considerations result in the following definition of propagation distance d_p that is used in EvoSuite's fitness function for a test suite T and a mutant M_j:

$$d_p(M_j, T) = \begin{cases} 0 & \text{if a TC assertion fails} \\ 1 & \text{if } d_i(M,T) > 0 \\ \frac{1}{1+impact_{min}(M_j,T)} & \text{if } d_i(M,T) = 0 \end{cases}$$

The infection distance $d_i(M,T)$ is calculated using the following formula:

$$d_i(M,T) = \begin{cases} 1 & \text{if } M \text{ was not reached} \\ \nu(d_{min}(M,T)) & \text{if } M \text{ was reached} \end{cases}$$

Here $d(M,T)$ is the branch distance and $\nu(x)$ is a normalizing function, as defined in Section 4.3.1.

To detect false negatives, we have to further restrict the notion of mutation killing. For a given 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. None of the conditions in the program assertions change their values: $\forall i \in [1 \ldots n] : c_i^{M_j} = c_i^o$.

3. One of the variables visible at one of the program points where assertions are located has different values in P and M_j: $\exists i \in [1 \ldots n] : v_1^{M_j} \neq v_1^o \vee \ldots \vee v_{m_i}^{M_j} \neq v_{m_i}^o$.

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: $(\forall i \in [1 \ldots n] : c_i^{M_j} = c_i^o) \wedge (\exists i \in [1 \ldots n] : v_1^{M_j} \neq v_1^o \vee \ldots \vee v_{m_i}^{M_j} \neq v_{m_i}^o)$, instead of returning zero.

The test suite generated by OASIs as an evidence of a false negative consists of the test cases each of which comes with a list of mutations. If each mutation in the list is injected into the method under test and the provided test case is executed, none of the program assertions will react to the mutation. For each mutation we also report variables that have changed their values as a result of the mutation. If a variable has a primitive type we provide the values of that variable in the original and mutated versions. This provides additional support for the developer in the improvement process by indicating which variables the program assertion ignores or does not check strongly enough.

Figure 13 (bottom) shows an example of OASIs' report for a false negative in program assertions in Figure 13 (top). The report contains one test case `test0` and a description of the mutation in the comments above the test case. As it follows from the description, the mutation applies an increment by 1 operator at line 7, changing the value of variable `max` from 1 to 2. However, none of the assertions in the method reacts to this change.

5 Oracle Assessment and Improvement: Empirical Evaluation

We conducted a large empirical evaluation of our oracle assessment and improvement approach. The goal was to assess its applicability to different types of initial oracles, different subject programs and with different developers representing the human in the loop. A common problem with controlled empirical studies involving human subjects is that, due to their cost and complexity, their size is limited. In order for this limitation not to affect the size/variety of subject programs and initial oracles being considered in the empirical evaluation, we have conducted three separate empirical studies.

In our large scale study we used our approach to assess and improve oracles in 5 large real-world systems. The initial oracles in this study were ranging from the case 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. During these experiments the human in the iterative assertion improvement process was the author of the thesis, 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.

Then we conducted an *Oracle Assessment Study* with 39 participants to *assess* 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.

Another 29 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.

Both of our human studies were approved by UCL's Research Ethics Committee. All the experimental data collected is available at the link: https://github.com/guneljahan/OASIs/humanstudy.

The primary contributions of this chapter are:

1. The validation of our oracle assessment and improvement approach on five nontrivial real-world systems and three types of initial oracles.
2. A novel human study on oracle assessment.
3. A novel human study on oracle improvement.

5.1 Large Scale Study

In this empirical evaluation we conducted a set of experiments to answer the following research questions:

RQ1 (Implicit oracle): How effective is the computation of oracle deficiencies in introducing and iteratively improving new program assertions in classes without assertions?

RQ2 (Inferred properties): What is the effectiveness of oracle deficiencies computation for the improvement of automatically inferred program properties?

RQ3 (Manual oracle): How effective is the proposed approach in revealing oracle deficiencies in classes that include human written program assertions?

RQ4 (Comparison with Initial Program Assertions): Can the improved oracle reveal more faults than the initial (implicit, automatically inferred, manual) oracle?

RQ5 (Comparison with Test Case Assertions): Can the improved oracle reveal more faults than the test case oracle?

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[10] with program assertions before and after the improvement process.

For RQ5 we compare the mutation score of program assertions after the improvement process with the mutation score of the test case assertions generated by automated test case generation tools as EvoSuite and Randoop.

There is empirical scientific evidence that mutants are an appropriate (and laboratory controllable) surrogate for real software faults [4, 53], 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.

5.1.1 Subjects

Table 10: 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

The subject systems used in our study are shown in Table 10. 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 (version 3.5), Apache Commons Collection (version 3.2) and Apache Commons Lang (ver-

[10]http://pitest.org

sion 3.4), 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 contracts written using Java Modeling Language (JML), which is a specification language for Java programs. All of the subjects are used to evaluate the increased fault detection capability (RQ4, RQ5) of the improved oracles.

5.1.2 Experimental Procedure

The experimental procedure for RQ1, RQ2, RQ3 involves two major activities: *obtaining initial oracles* and *running the assessment and improvement loop.* While the first step is different for each type of initial oracle and accordingly for each research question, the second step remains the same across all the three research questions.

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 (1000 test cases) for each class P. The training traces needed by the invariant inference tool Daikon [27] 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 subject programs. 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. The human in the loop was instructed to run OASIs with its default parameters during this process. If false negatives are detected, the nature of the mutation operations reported by the tool provides guidance during the improvement process of the assertions. To ensure that the human experimenter behaves deterministically we defined precise rules and procedures for oracle improvement to be followed, prescribing what to do for each reported deficiency (e.g., for each EvoSuite mutation operator triggering a false negative).

Table 11 shows the list of mutation operators reported by EvoSuite grouped by the type of improvement actions required to remove false negatives.

Table 11: 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

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 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, RQ5), we used mutation analysis. First we generated the following test suites: (T_1) Randoop test suite without test case assertions; (T_2) The same Randoop test suite as in T_1, but with test case assertions; (T_3) EvoSuite test suite without test case assertions, generated according to the branch coverage criterion; (T_4) The same EvoSuite test suite as in T_3, but with test case assertions. 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) For RQ4: P_2, T_1 compared to P_1, T_1 for RQ4 (2) For RQ5: P_2, T_1 compared to P_3, T_2 and P_2, T_3 compared to P_3, T_4.

The comparison of these mutation scores provides an insight into the improvement in fault detection.

5.1.3 Results

Table 12 shows a summary of the results obtained in our experiments. Column *C/M* in Table 12 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.

Table 12: 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	62/186	283	0	15	15	106	0	0	13	298	15	119
RQ1	25	CC	40/234	296	0	31	31	44	0	1	5	327	32	49
RQ2	20	CL	54/170	0	605	55	114	44	6	0	2	660	114	46
RQ2	20	CM	30/112	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

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.

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.*

RQ2 (Inferred Properties)

For RQ2 we considered *Apache Commons Lang* and *Apache Commons Math*. The size of the test suite generated by Randoop (version 3.0.3) 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, 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.*

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.*

RQ4 (Comparison to Initial Program Assertions)

Table 13 shows the average mutation score computed by PIT (version 1.1.7) for each subject 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 13: RQ4: Average mutation score by subject for initial (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	$< 2.2^{-16}$

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.

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 13). 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) and the manual (JML) initial oracles.*

RQ5 (Comparison to Test Case Assertions)

Table 14 reports the average mutation score computed by PIT for all subjects (1) with test case assertions generated by Randoop and Evosuite (μ_s) (2) with program assertions after iterative oracle improvement (μ'_s).

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.

Table 14: RQ5: Average mutation score by subject for test case (μ_s) and improved (μ'_s) oracle

Oracle	**Subj**	μ_s	μ'_s	Δ	$\hat{A}_{12}$	p-**value**
Randoop	All	45%	98.4%	53.4%	0.93	$5.3 \cdot 10^{-7}$
EvoSuite	All	46.9%	98.4%	51.5%	0.95	$3.8 \cdot 10^{-6}$

As in RQ4, the observed mutation score increase is statistically significant and the Vargha-Delaney effect size $\hat{A}_{12}$ is always *large*.

RQ5: *The improved oracle has significantly higher mutation score than the test case assertions generated by EvoSuite and Randoop.*

5.1.4 Qualitative Analysis

To provide a better understanding of the iterative process and the nature of the improvements it provides let's have a look at some examples in detail.

Improvement of Implicit Oracle

Figure 20 (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 false negative detection enabled, getting the output shown in Figure 21.

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 20 (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 20 (bottom) and the check for false positives confirms this improvement.

Improvement of Inferred Oracle

Figure 22 (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.

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

```
public boolean add(final E obj) {
   final int size = map.size();
    map.put(obj, dummyValue);
    boolean result = map.size() != size;

    assert (map.get(obj) == dummyValue) &&
              map.size() == size + 1 &&
           (result == (map.size() != size)));

   return result; }
```

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

Figure 20: Method add(): No Assertions (top), Assertion Added at Iteration 1 (middle), Final Assertion (bottom)

```
/* 1 add, Line 4 - ReplaceConstant - true -> false
 * 2 add, Line 3 - DeleteField: mapLjava/util/Map;
 * 3 add, Line 2 - DeleteField: mapLjava/util/Map;
 * 4 add, Line 4 - ReplaceComparisonOperator != -> ==
 * 5 add, Line 3 - DeleteStatement:
    put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/ lang/Object;
 * 6 add, Line 2 - DeleteStatement: size()I
 * 7 add, Line 4 - DeleteStatement: size()I
 * 8 addAll, Line 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 21: FN Detection for Method add()

```java
public class Interval {

  private final double lower;
  private final double upper;

  public Interval(double lower, double upper) {
     this.lower = lower;
     this.upper = upper;
  }

  public double getSize() {
    double old_upper = upper;
    double old_lower = lower;
    double result = upper - lower;

    assert (this.lower == old_lower); //1
    assert (this.upper == old_upper); //2
    assert (result != 0); //3: removed (FP)
    assert (old_upper >= result); //4: removed (FP)

    return result; } }
```

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

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

Figure 22: Method getSize() with Daikon assertions before (top) and after (bottom) oracle improvement; a real bug was reported and fixed (middle)

```
//Test case number: 1
/* 1. org.apache.commons.math4.geometry.euclidean.oned.
     Interval.getSize()D: Line 14 ReplaceArithmeticOperator - -> +
 * 2. org.apache.commons.math4.geometry.euclidean.oned.
     Interval.getSize()D: Line 14 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 23: FN detection for method getSize()

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 23. 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 22 (with improved assertions at the bottom).

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

The last detected bug is in the class `Complex`. The method `reciprocal` returns INF only if the real and imaginary parts are exactly equal to 0.0. In the cases when real and imaginary parts are double numbers very close to 0.0, it does not return INF. The bug was reported to the developers (bug reports # MATH-1259, # NUMBERS-22) and subsequently fixed by adjusting the output of the method to IEEE and C99 standards.

5.1.5 Threats to Validity

The main threats to validity are the authors' bias and the external validity threat.

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. Therefore, the way the oracles have been refined might have influenced the results. 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 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, generalisability of our findings requires further replications on additional subjects.

5.2 Human Study: Oracle Assessment

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:

RQ6: How effective are developers in determining whether the oracle has a deficiency and, if it has one, what the deficiency type is?

RQ7: What are the common misclassifications developers make when assessing oracle deficiencies?

5.2.1 Object Selection

The starting point of our experimental design was the previous study by Staats et al. [95]. In this previous work Staats et al. analysed the user's ability to classify invariants dynamically generated 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 [74]. *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) [44]. *PolyFunction* is a class representing a polynomial function, and is part of the Math4J package [69]. 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.

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. [95]. Table 15 shows that while the approach described in the paper [95] found

73 assertions with a false positive among 324 assertions, our approach found false positives in 60 more assertions.

Table 15: OASIs applied to the invariants from [95]

Class	**# of Assertions**	**Incorrect**	
		In Paper [95]	OASIs
Matrix	122	18	42
Poly	121	26	50
StackAr	81	29	41
Overall	324	73	133

We reused the subject programs of the study by Staats et al. [95], and used the dynamically generated and manually written invariants as our initial oracles. However, we used the output of OASIs for classifying these invariants. 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. [95] 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. Our selection process favoured assertions that were checking the functionality specific to the method under test rather than general properties of the class (as most Daikon-generated invariants do). 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.

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 16 lists the classes from the work of Staats et al. [95] 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 16: 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

5.2.2 Participants

To answer our research questions we conducted three separate experimental sessions. The first and third sessions 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 17: 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 17 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.

5.2.3 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 that participants 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 partially 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 for the task was completed, participants proceeded to the questionnaire to answer questions about their background and to provide feedback about the session.

5.2.4 Results

RQ1: User Effectiveness

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

Classification Results. Table 18 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 similar, respectively with 25% and 26% correct classification rates.

Table 18: Results: Students (SS1 - 1st session, 20 students; SS3 - 3rd session, 13 students)

As.-n	**OD**	**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 19: 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 19 shows the results for the 6 professional developers. With a 48% correct classification rate they exhibited almost twice as good a 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 24 provides more insight into the participants' performances by showing the number of participants giving the same number of correct answers (which ranges from 0 to 6). 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 up to 4 assertions. The best performance of 5 and 6 correct answers was exhibited by one student.

Overall, for 39 participants the average correct classification ratio is only 29%, see Table 20. 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.

Table 20: 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%)**

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 [0.193:0.219] with mean 0.253; for professional developers it is in the range [0.304:0.645] with mean 0.472; and for all participants it is in the range [0.229:0.349] with mean 0.286. The difference between students' and professional developers' performances 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 of the understandability of the 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 21 show the

Table 21: 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]				

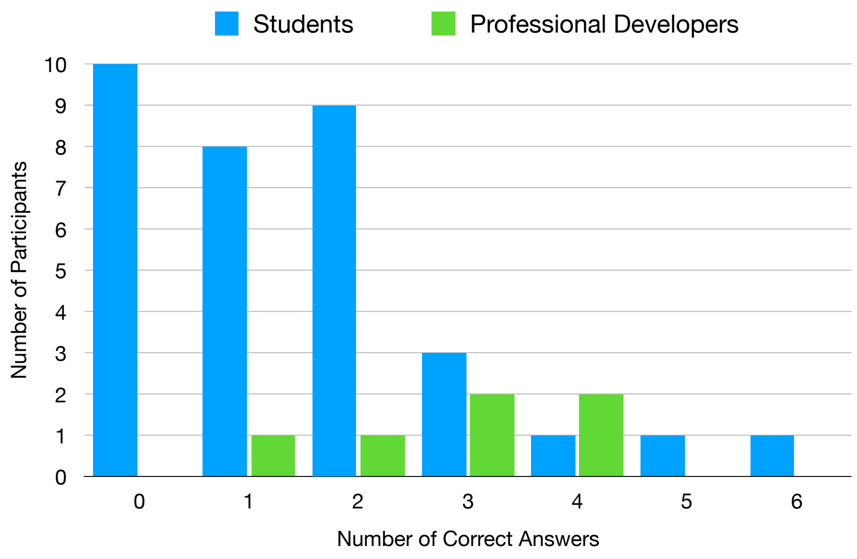

Figure 24: Number of Participants Grouped by Number of Correct Answers

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 subjective comprehensibility of the provided training material 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 21. The correlation coefficients are positive for all factors except Java Experience. Industry Experience is the

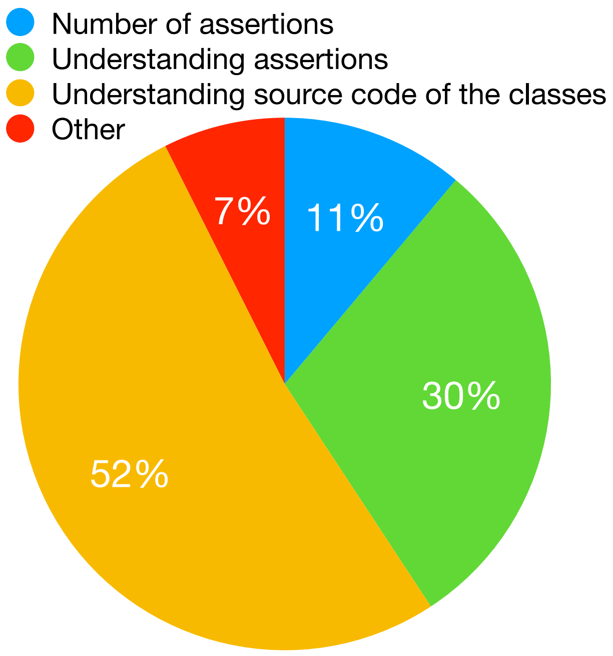

Figure 25: What Was the Main Challenge while Performing the Task?

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' opinions 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 25 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 [28] 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 [55] coefficient, which generalizes Fleiss' kappa to incomplete (missing) data. Krippendorff's alpha takes a 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 22: 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

Table 22 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*, among professionals. According to Landis and Koch's [56] 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 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.

Overall, these low agreement rate values show that although all subjects, even those with industry experience, find oracle classification hard, there is no evidence of 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.

RQ6 (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 a correctness rate is largely below the desirable value (100%). The inter-rater agreement was also quite poor with no consistent misclassifications. We observed moderately strong evidence that industrial experience is correlated with the correct classification rate, but found no similar evidence of any other correlations.*

RQ2: Misclassifications

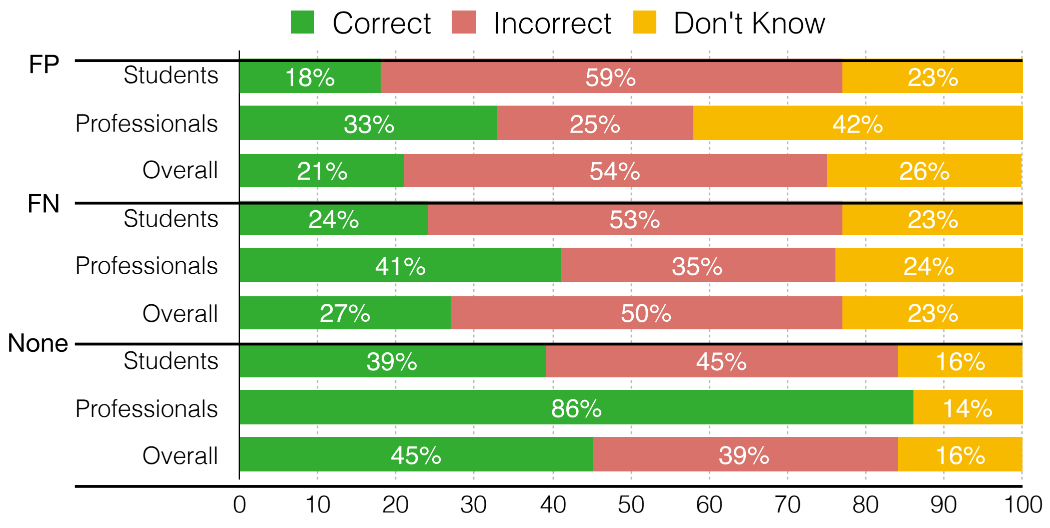

Figure 26: Correctness Rate for FP, FN and None

Harder to Detect Oracle Deficiencies. To investigate which type of oracle deficiency is harder to detect for developers, we summarised the results of the oracle assessment task for each type of oracle deficiency and participant group (see Figure 26). 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 86% of the cases.

We asked the question “Which oracle deficiency is harder to detect?” to the participants in the exit questionnaire. As Figure 27 shows, the number of people finding false positives harder to detect than false negatives is slightly higher, which is in line 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 in the experimental results, we calculated the 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.

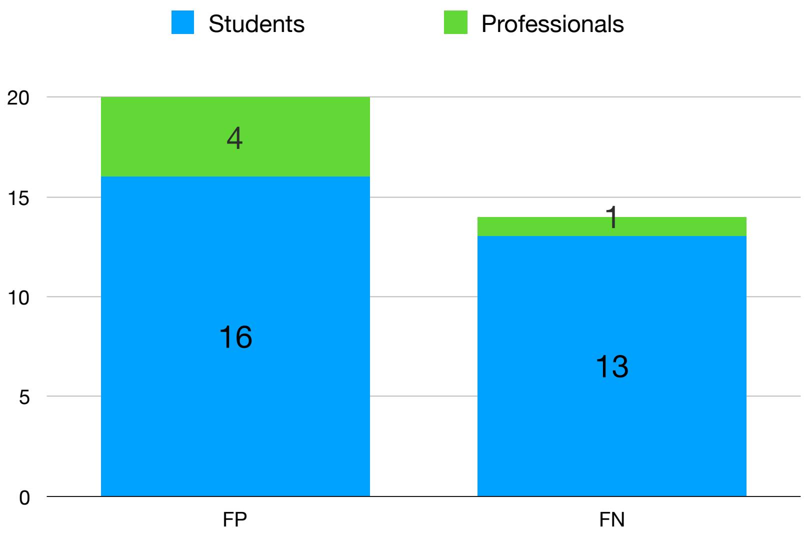

Figure 27: Which Oracle Deficiency is Harder to Detect?

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 23 lists these misclassifications, 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 23: 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 23 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.

RQ7 (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.*

5.2.5 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.

Our measurements of user effectiveness are obtained by comparing participants' results against the outcome of OASIs. While it provides evidence for any oracle deficiency it detects, it may report no oracle deficiencies even if some (undetected) deficiency is actually there. To deal with this issue, the authors thoroughly examined each assertion with no oracle deficiencies according to OASIs, to ensure that the tool's judgement was 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 code not written by them. We have selected three relatively simple Java classes for our studies due to the limited time of the experimental sessions. We acknowledge that our results cannot be generalised to other Java classes. However, we had a large number of participants in the study, and therefore we believe that our results provide insight in the expected behaviour of developers with different experience and backgrounds in the oracle assessment process.

5.3 Human Study: Oracle Improvement

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:

RQ8: What is the quality of assertions improved using our tool compared to assertions improved manually?

RQ9: 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?

5.3.1 Participants

Table 24: Improvement Study: Participants

Part.-t	Group	Exp.	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
P9	With Tool	6 years	0	0 USD	QA Automation Engineer
P10	With Tool	5 years	0	0 USD	Full Stack Java Developer

In our approach, the developer is an integral part of the oracle improvement process. To analyse how beneficial is 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, 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 24 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 *Exp.* shows the experience of each developer in years. Column *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).

5.3.2 Experimental Procedure

The main structure of our experimental procedure is shown in Figure 28. 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 28 show the content and the payment offered for each milestone.

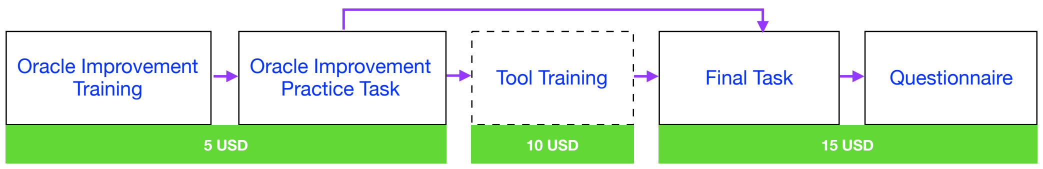

Figure 28: Oracle Improvement Study: Experimental Procedure

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. 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.

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 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.

5.3.3 Results

Quality of Final Assertions

Table 25 shows the results for the participants who improved the assertions manually. Column *OT* (Overall Time) 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 mutation score = 0; an assertion labelled Partially Correct has mutation score > 0 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 otherwise cause the labelling FP).

The results presented in Table 25 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

Table 25: Improvement Study: Results Without Tool

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

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 26 shows the results for the participants who used our tool to improve the assertions. Here, column OT (Overall Time) comprises the running time of the tool, reported in column TT (Tool Time), and the time the developer spent on analysing the output of the tool and improving assertions, i.e., the human cost, reported in column HT (Human Time). 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

Table 26: Improvement Study: Results With Tool

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

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 18). Five participants (marked with an asterisk in Table 26) have run the tool with this configuration. As they achieved the desired partial correctness in a relatively short time, we used the standard configuration of the tool reporting all false negatives for the rest of the participants. As a result, the latter 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 the Upwork participants (four out of five) were successful in achieving full correctness.

In Tables 25 and 26 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, 43% of participants achieved full correctness and 21% achieved partial correctness when improving assertions manually versus 100% of developers achieving full correctness when improving assertions using our tool. For Assertion 2, in case of manual improvement 64% of developers achieved partial correctness, while when using the tool 33% of them got to a point of full correctness and 67% of them to a point of partial correctness.

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 $p < 2 * 10^{-16}$.

RQ8 (Quality of Final Assertions): *The tool helped developers produce assertions with higher quality. On average, when using the tool participants achieved full correctness in 67% and partial correctness in 33% of cases, while participants without tool achieved full correctness in 21% and partial correctness in 43% of cases. The difference is statistically significant.*

Iterative Improvement Process

Analysis of the time required to complete the iterative improvement process (see Tables 25, 26) 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

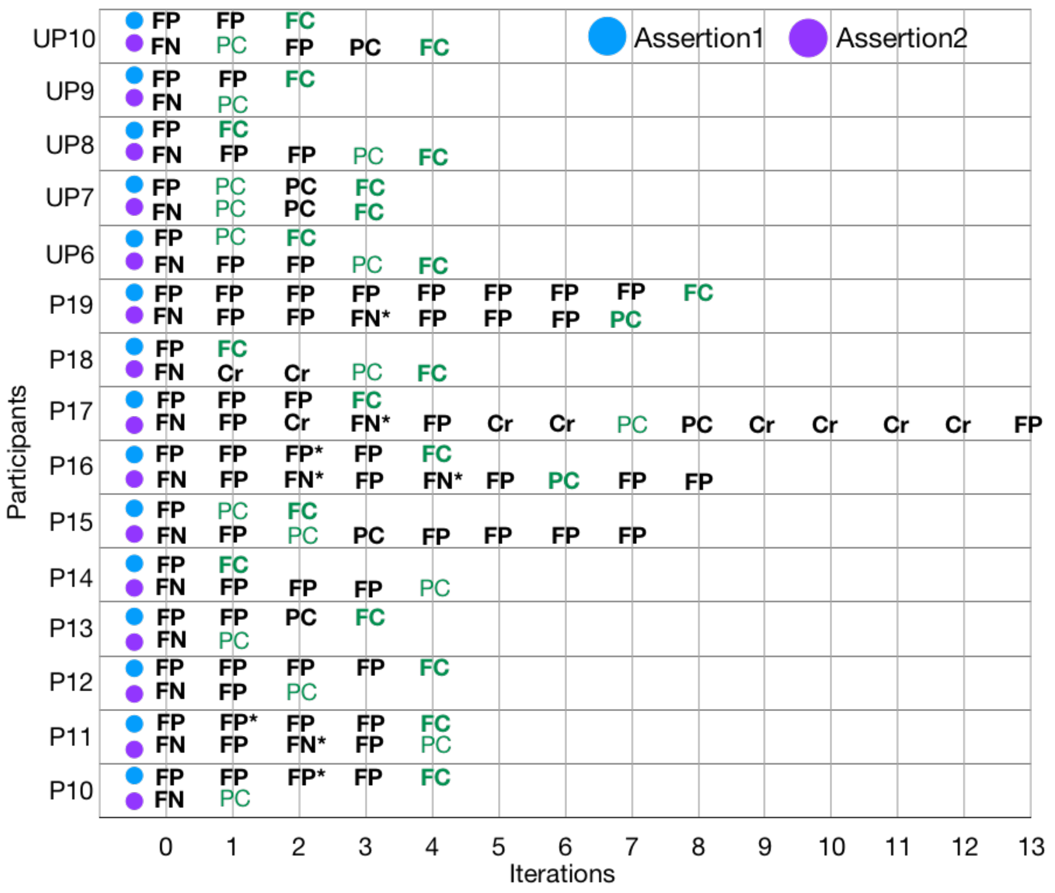

Figure 29: Improvement Study Results: Iterative Process Details

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 26), tends to be lower than the human overall time when no tool is available (see Table 25). It is only when the tool time (Column *Tool T.*) is added that we get comparable times to the setting without the 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 29 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' 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 29 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 have 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.

RQ9 (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 during oracle improvement process varied between 1 and 13, with an average of 3.9 iterations. In each iteration, developers spent, on average, 193 seconds of manual effort between tool runs to fix oracle deficiencies.*

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 a false negative is initiated. Therefore, the detection time for false negatives includes the whole search budget of a false positive search (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 30 lists the questions and 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 31 shows the percentages of chosen answers. For false positives, under-

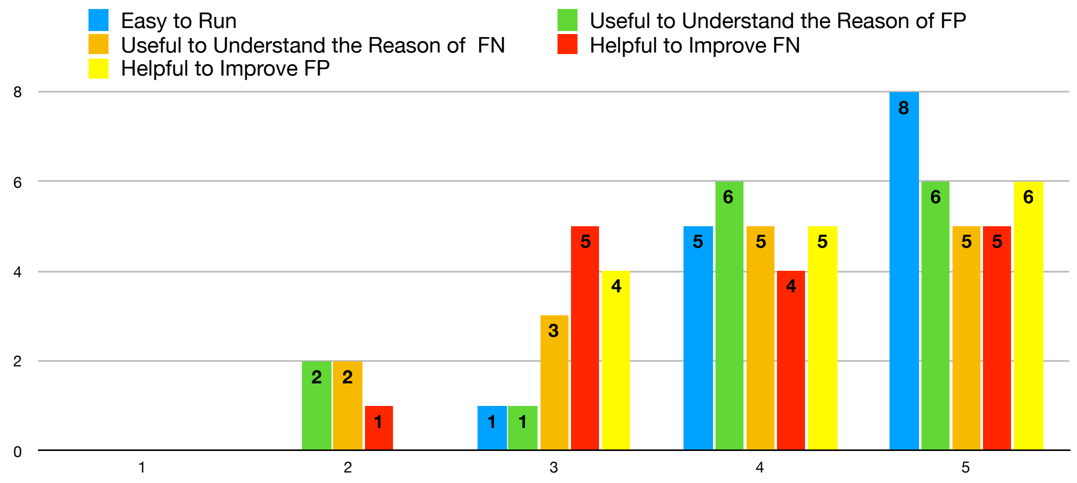

Figure 30: User Feedback on Tool

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

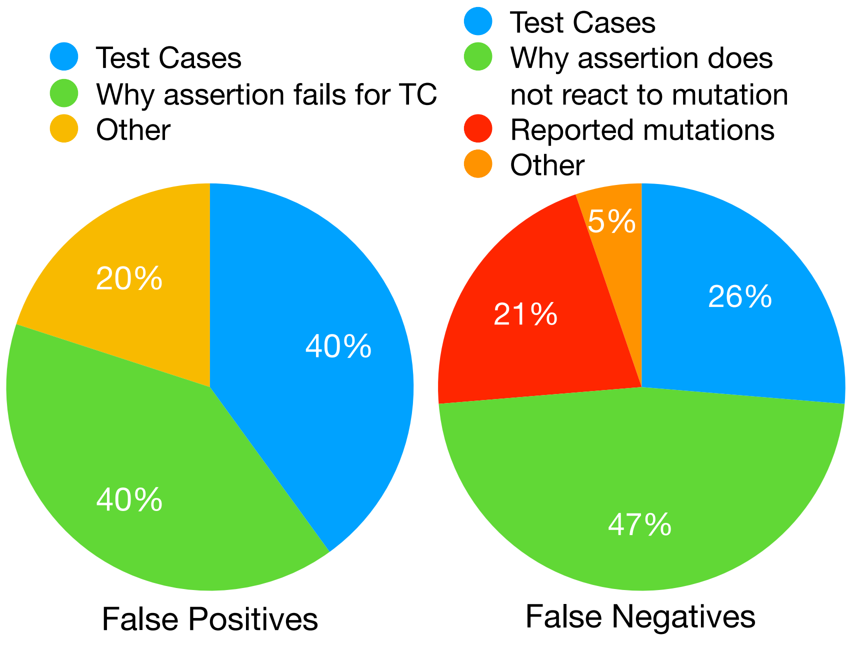

Figure 31: Difficulties in Understanding Tool's output

5.3.4 Threats to Validity

Internal. To mitigate the threat to internal validity regarding the understandability of the training material and experimental objectives for the participants, we included a *practice task* in our study 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 the test.

A part of the study was performed in a remote setting using the Upwork freelancing platform. The training provided to these participants was in 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 time information in our results, but acknowledge that it is not reliable. Overall, cofactor analysis shows that results of Upwork participants are not significantly different from the results of other participants.

External. As in the *Oracle Assessment* study, the classes used in this study were not developed by our participants and may have been unfamiliar to them. We also acknowledge that the results of our *Oracle Improvement* study cannot be generalised to other Java classes. However, due to the large number of participants with varying experiences and background, we believe that our study provides meaningful insights about the behaviour of developers in the oracle improvement process.

A further threat to external validity is that our results might be biased by the population of developers who are 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 that we would be able to recruit a subset of workers with similar skills in any platform.

5.4 Conclusions and Future Work

We have proposed an iterative technique for the assessment and improvement of oracles, which is based on test case generation for the identification of false positives and mutation testing for the identification of false negatives. Our 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. In this experimental setup the human in the loop was represented by the author of the thesis.

In our further evaluation 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. This study indicates that humans find it very difficult to assess the deficiencies of program oracles. Hence, there is a strong need for automated support in such task. Our tool OASIs aims at addressing this need.

When provided with information on the type of oracle deficiency for the assertion and asked to improve it manually, developers, on average, achieved full and partial correctness in 21% and 43% of cases respectively. These numbers increased significantly, with developers achieving 67% of full and 33% of partial correctness when 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 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 that are more sound and complete oracles. Our future work will be to optimise the performance of OASIs, so that OASIs takes less time to run and leads to a smoother incremental improvement process. The analysis of the iterative oracle improvement process using OASIs shows that around 45% of time in 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 work on mutant selection [75, 109] can become a part of our implementation in future.

The user feedback collected about the understandability and helpfulness of the tool's output, including the difficulty in understanding the automatically generated test cases, will also be addressed in our future work. The existing work in the area of test code understandability such as techniques to improve the readability of automatically generated test cases [23] or to provide test case summaries in natural language [79] can be incorporated into OASIs.

6 Conclusions and Future Work

This chapter summarises the overall conclusions of this thesis and how the presented work addresses the objectives it aimed to investigate. It also discusses how the approaches presented can be extended and enhanced in future work.

6.1 Summary of Achievements

The main contributions of the thesis are:

- **Empirical Study on Failed Error Propagation in Programs with Real Faults**

  We have presented empirical evidence from a large corpus of real-world faults in Java systems that reveals a surprisingly low level of failed error propagation amongst the 384 faults studied; all state corruptions caused by these faults can be observed, and none failed to propagate. These empirical findings contradict earlier work on failed error propagation and, if replicated in other fault corpuses and/or for other languages, would have profound implications for software testing. Furthermore, when we turn our attention to studying the synthetic faults introduced by program mutants, a widespread practice believed to be good at simulating real faults, we find very different behaviour: the artificial faults denoted by mutants do exhibit failed error propagation, unlike the real faults we studied. These findings concerning mutants provide additional nuances on earlier work on the suitability of mutation testing for simulating real faults. Such synthetic faults may be closely coupled to real faults in the sense that test cases that reveal them also tend to reveal real faults. Nevertheless, there do appear to be non-trivial differences in the behaviour of synthetic faults and real faults, with respect to their error propagation. Lack of failed error propagation is due to the use of the strongest possible oracle as postcondition, which checks all externally observable program variables. This requires techniques to assess and improve existing oracles, that might not be no as strong as the optimal postcondition oracle.

- **Formal Model of Oracle Improvement**

  We have proposed a formal model of oracle improvement using Shannon's information theory. We first proved that oracle improvement increases conditional probability of accurate acceptance/rejection given the probability of the ideal oracle doing so. By modelling the actual and perfect oracles as a pair of boolean-valued random variables, we measured how closely connected they are using mutual information. We then proved that every improvement step can make the information in the actual oracle closer to the information in the perfect oracle.

- **Approach for Oracle Assessment and Improvement**

  We have proposed an iterative technique for the assessment and improvement of the oracles. We use search-based test case generation to detect false positives and mutation analysis to detect false negatives. Our approach necessarily places a human in the loop of the iterative process as the source of information about program's intended behaviour. We have implemented this approach as a tool, named OASIs, for Java programs. Experimental results show that OASIs is able to identify both false positives and false negatives in three important types of initial oracles (implicit, inferred and manual), leading to an average 48.6% improvement of mutation score over all the analysed classes and exposing real faults that have been reported to and fixed by the developers. Moreover, the program assertions improved using our approach detect, on average, 52.6% more faults than the test case assertions generated by automated test case generators.

- **Human Study on Oracle Assessment**

  We conducted a large empirical study with 39 participants (33 students, 6 professionals) overall to assess developers' ability to detect oracle deficiencies manually, with no tool support. The results show that subjects achieve a low correct classification rate (29%) when performing the oracle assessment task. The performance of professional developers (48%) is significantly higher than the performance of students (25%), but it is still below the desirable value (100%). The analysis of parameters that might affect users' performance shows a moderate evidence that industrial experience is correlated with correct classification rate and no evidence of any other correlations. These results confirm that the oracle assessment is a difficult task for humans and that automatic detection of false positives and false negatives by OASIs is indeed useful.

- **Human Study on Oracle Improvement**

  We conducted an Oracle Improvement Study, where participants (19 overall) were assigned to the control or treatment group. The participants from the control group had to improve the provided initial oracles manually, while the participants from the treatment group had the support of OASIs to perform the same task. The results show that OASIs helped developers produce higher quality assertions. Participants who used the tool were able to achieve full correctness in 67% of cases and partial correctness in 33% of cases, while participants without tool achieved full correctness in only 21% and partial correctness in only 34% of cases. The number of iterations during oracle improvement process varied between 1 and 13, with an average of 3.9 iterations. The introduction of OASIs in the process did not impact the overall time spent on oracle improvement to any major extent. If the tool execution time is excluded, it actually reduced the time required from humans. Therefore, using OASIs in oracle improvement process leads to a higher quality final oracles and requires less human effort.

6.2 Future Work

Further Experiments on Failed Error Propagation

As shown in Section 2.4.2, the existing work on failed error propagation is focused mainly on synthetic or hand-seeded faults. Only two works [24, 104] consider 12 and 38 real faults in C programs respectively. While our study on failed error propagation addresses real faults in Java programs, there are no existing studies analysing it on a large corpus of real faults in C/C++ programs. To analyse the generalisability of our results to programming languages other than Java, our future work will focus on the empirical evaluation of failed error propagation in C/C++ programs with real faults.

There are existing metrics that serve as predictors of failed error propagation in the programs under test. The works by Voas and his collaborators [100, 102] introduced a metric called "testability" for this purpose (described in detail in Section 2.4.2). The work by Androutsopoulos et al. [5] proposed 4 new information theory-based metrics and demonstrated they are well-correlated with failed error propagation. We plan to calculate these metrics for the subject programs with real faults we have used in our empirical study and analyse whether their values are also in line with the low level of FEP we have observed.

When measuring the level of FEP in our empirical study, we considered the maximum oracle, i.e. an oracle that was checking all the externally visible members of a Java class. However, such an oracle is rarely available. To measure the level of FEP in cases when weaker oracles are provided, we plan to gradually weaken the maximum oracle by excluding the externally visible variables that we consider and measure the level of FEP with such an oracle. Moreover, we plan to evaluate whether the change in the level of FEP as a result of change of the oracle is correlated with the metrics mentioned in the previous paragraph.

Improvements on OASIs

We plan to add a plugin to OASIs which will convert the final improved program assertions into the format required by automated test case generators such as Randoop. Therefore, these program assertions will serve as specifications which will be respected during automated test case and assertion generation.

The automated test case generator Randoop has a parameter using which developers can provide a specification of the expected behaviour of the code under test. This specification indicates the circumstances when the method can be called, and how it should behave when called. Randoop uses such a specification to better classify method calls as error-revealing, expected behaviour, or invalid. The corresponding parameter in Randoop should be set to file containing the method specifications. The file format is a JSON list with elements indicating pre-conditions, post-conditions and throw-conditions.

References

[1] Sheeva Afshan and Phil McMinn. An investigation into qualitative human oracle costs. In *Psychology of Programming Interest Group Annual Workshop (PPIG 2011)*, 2011.

[2] Sheeva Afshan, Phil McMinn, and Mark Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, pages 352–361, Washington, DC, USA, 2013. IEEE Computer Society.

[3] Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, New York, NY, USA, 1 edition, 2008.

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

[5] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M. Hierons, and Mark 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.

[6] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, 2016.

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

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

[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[10] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving Test Suites for Efficient Fault Localization. In *28th International Conference on Software Engineering (ICSE 06)*, Shanghai, China, 2006. ACM. selection : 9%.

[11] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[12] Henrik Bengtsson. Linux settimelimit bug description.

[13] Antonia Bertolino. Software testing research and practice. In *Proceedings of the Abstract State Machines 10th International Conference on Advances in Theory and Practice*, ASM'03, pages 1–21, Berlin, Heidelberg, 2003. Springer-Verlag.

[14] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In Lionel Briand and Alexander Wolf, editors, *Future of Software Engineering 2007*, Los Alamitos, California, USA, 2007. IEEE Computer Society Press.

[15] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael 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.

[16] R. V. Binder. Design for testability in object–oriented systems. *Communications of the ACM*, 37(9):87–101, 1994.

[17] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 242–253, 2018.

[18] S. Chandra and P. M. Chen. How fail-stop are faulty programs? In *28th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-28)*, pages 240–249, June 1998.

[19] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[20] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation*, 15(2):181–199, 2005.

[21] Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. In *Building the Information Society*, pages 359–366, Boston, MA, 2004. Springer US.

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

[23] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley 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.

[24] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 158–171, New York, NY, USA, 1996. ACM.

[25] Martin D. Davis and Elaine J. Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM '81 Conference*, ACM '81, pages 254–257, New York, NY, USA, 1981. ACM.

[26] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.

[27] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69:35–45, December 2007.

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

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

[30] Gordon Fraser and Andrea 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.

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

[32] Gordon Fraser and Andreas 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.

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

[34] Gregory Gay, Matt Staats, Michael W. Whalen, and Mats Per Erik Heimdahl. Automated oracle data selection support. *IEEE Trans. Software Eng.*, 41(11):1119–1137, 2015.

[35] Robert Geist, A. Jefferson Offutt, and Frederick C. Harris Jr. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, 41(5):550–558, 1992.

[36] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro 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.

[37] J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Trans. Softw. Eng.*, 9(6):686–709, November 1983.

[38] Michael Grottke and Kishor Trivedi. A classification of software faults. In *Supplemental Proc. Sixteenth International IEEE Symposium on Software Reliability Engineering*, pages 4.19 – 4.20, 2005.

[39] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. Eio: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and*

Storage Technologies, FAST'08, pages 14:1–14:16, Berkeley, CA, USA, 2008. USENIX Association.

[40] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error handling is occasionally correct. In Mary Baker and Erik Riedel, editors, *6th USENIX Conference on File and Storage Technologies (FAST 2008)*, pages 207–222, 2008.

[41] Mark Harman, Lin Hu, Robert M. Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Trans. Software Eng.*, 30(1):3–16, 2004.

[42] Mark Harman, Lin Hu, Robert Mark Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, January 2004.

[43] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing (keynote). In 8th *IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*, Graz, Austria, April 2015.

[44] Joe Hicklin, Cleve Moler, Peter Webb, Ronald F Boisvert, Bruce Miller, Roldan Pozo, and Karin Remington. Jama: A java matrix package. *URL: http://math. nist. gov/javanumerics/jama*, 2000.

[45] Douglas Hoffman. A taxonomy for test oracles. In *Software Quality Methods, LLC*, 1998.

[46] Douglas Hoffman. Using oracles in test automation. In *Software Quality Methods, LLC*, 2001.

[47] Chen Huo and James 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.

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

[49] J. A. Jones. Fault localization using visualization of test information. In *Proceedings. 26th International Conference on Software Engineering*, pages 54–56, May 2004.

[50] René Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 433–436, San Jose, CA, USA, July 23–25 2014.

[51] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[52] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 2014.

[53] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon 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.

[54] Dexter Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.

[55] Klaus 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.

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

[57] Nan Li and Jeff Offutt. An empirical analysis of test oracle strategies for model-based testing. pages 363–372, 03 2014.

[58] Nan Li and Jeff Offutt. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43:1–1, 01 2016.

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

[60] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2014.

[61] Pablo Loyola, Matt Staats, In-Young Ko, and Gregg 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.

[62] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. How does regression test prioritization perform in real-world software evolution? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, 2016.

[63] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. Grt: Program-analysis-guided random testing (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[64] Leonardo Mariani and Fabrizio 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.

[65] Matias Martinez and Martin Monperrus. Astor: A program repair library for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, 2016.

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

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

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

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

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

[71] Phil McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In Franz Rothlauf, editor, *Genetic and Evolutionary Computation Conference (GECCO 2009)*, pages 1689–1696, Montreal, Québec, Canada, 2009. ACM.

[72] Phil McMinn, Mark Stevenson, and Mark Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *Proceedings of the First International Workshop on Software Test Output Validation*, STOV '10, pages 1–4, New York, NY, USA, 2010. ACM.

[73] Yi Miao, Zhenyu Chen, Sihan Li, Zhihong Zhao, and Yuming Zhou. Identifying coincidental correctness for fault localization by clustering test cases. In *SEKE*, pages 267–272, 2012.

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

[75] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, 1996.

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

[77] Alessandro Orso and Gregg Rothermel. Software testing: A research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 117–132, New York, NY, USA, 2014. ACM.

[78] Carlos Pacheco and Michael 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.

[79] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald 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.

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

[81] Fabrizio Pastore, Leonardo Mariani, and Gordon 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.

[82] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, 2017.

[83] Mauro Pezzè and Cheng Zhang. Automated test oracles: A survey. *Advances in Computers*, 95:1–48, 2015.

[84] Brian Randell. On failures and faults. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *International Symposium of Formal Methods Europe (FME)*, volume 2805 of *Lecture Notes in Computer Science*, pages 18–39. Springer, 2003.

[85] Debra J. Richardson. Taos: Testing with analysis and oracle support. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '94, pages 138–153, New York, NY, USA, 1994. ACM.

[86] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 105–118, New York, NY, USA, 1992. ACM.

[87] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 270–280, New York, NY, USA, 2009. ACM.

[88] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. *SIGPLAN Not.*, 44(6):270–280, June 2009.

[89] David Schuler and Andreas 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.

[90] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, and Siti Zaiton Mohd-Hashim. A comparative study on automated software test oracle methods. In *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, ICSEA '09, pages 140–145, Washington, DC, USA, 2009. IEEE Computer Society.

[91] Sina Shamshiri. Automated unit test generation for evolving software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, 2015.

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

[93] Dennis Shasha and Kaizhong Zhang. *Approximate tree pattern matching*, pages 341–371. Oxford University Press, 1997.

[94] Matt Staats, Gregory Gay, and Mats Per Erik 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.

[95] Matt Staats, Shin Hong, Moonzoo Kim, and Gregg 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.

[96] Matt Staats, Michael W. Whalen, and Mats Per Erik 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.

[97] P. Thévenod-Fosse and H. Waeselynck. Statemate applied to statistical software testing. In *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '93, pages 99–109, New York, NY, USA, 1993. ACM.

[98] P. Thévenod-Fosse and H. Waeselynck. Statemate applied to statistical software testing. *SIGSOFT Softw. Eng. Notes*, 18(3):99–109, July 1993.

[99] Paolo Tonella, Cu D. Nguyen, Alessandro Marchetto, Kiran Lakhotia, and Mark 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.

[100] J. Voas, L. Morell, and K. Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2):41–48, March 1991.

[101] J. M. Voas and K. W. Miller. Putting assertions in their place. In *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*, pages 152–157, Nov 1994.

[102] Jeffrey M. Voas. Pie: A dynamic failure-based technique. *IEEE Trans. Softw. Eng.*, 18(8):717–727, August 1992.

[103] Jeffrey M. Voas and Keith W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.

[104] Xinming Wang, Shing-Chi Cheung, Wing Kwong Chan, and Zhenyu Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 45–55, 2009.

[105] W.E.Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, 4(4):293–298, 1978.

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

[107] Eric Wong and Bojan Cukic. *Adaptive Control Approach for Software Quality Improvement.* World scientific Series on Software Engineering and Knowledge Engineering, 2011. Volume 20.

[108] W. E. Wong, Y. Qi, L. Zhao, and K. Y. Cai. Effective fault localization using code coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 449–456, July 2007.

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

[110] Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, 2017.

[111] Yingfei Xiong, Dan Hao, Lu Zhang, Tao Zhu, Muyao Zhu, and Tian Lan. Inner oracles: input-specific assertions on internal states. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 902–905, 2015.

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

[113] Lingming Zhang, Guowei Yang, Neha Rungta, Suzette Person, and Sarfraz 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.

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