UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Fault Revealing Test Oracles, Are We There Yet? Evaluating The Effectiveness Of Automatically Generated Test Oracles On Manually-Written And Automatically Generated Unit Tests

Daniel Correia Bento

Mestrado em Engenharia Informática

Dissertação orientada por:
Prof. Doutor José Carlos Medeiros de Campos

2022

Acknowledgments

This thesis would not have been possible without the help and support from several people. To every one of them, my deepest gratitude.

Particularly, to my supervisor, Professor José Carlos Medeiros de Campos, without whom this work would not have been possible, for his support, advice and critical vision, which contributed to enrich every step of the work performed.

I will be forever indebted to my family, particularly to my parents and to my sister, for their unconditional love, help and patience throughout all this time.

Dedicated to my family and friends.

Resumo

Ferramentas de geração automática de testes têm sido utilizadas em cenários de desenvolvimento de *software* reais e já se provaram capazes de detetar falhas reais. No entanto, estas ferramentas geram testes de regressão, assumindo que o código para o qual estão a gerar testes está correto, com o objetivo de alcançar a maior quantidade de cobertura de código possível. Isto leva a que sejam gerados testes que, apesar de executarem código faltoso, não conseguem detetar falhas existentes no *software* devido à utilização de *oracles* fracos. Esta geraração de *oracles* incapazes de testar uma funcionalidade da forma correta, deve-se à dificuldade das ferramentas de distinguir funcionamento correto de funcionamento incorreto do *software* — *oracle problem* — pois não têm conhecimento dos requisitos do sistema ou das intenções do programador no momento da criação de determinada funcionalidade. De forma a resolver este problema, investigadores têm desenvolvido, não só ferramentas capazes de melhorar testes já existentes, como também várias abordagens para gerar, de forma automática, *oracles* que se assemelhem aos escritos manualmente.

Apesar de as ferramentas de geração automática de *oracles* serem capazes de gerar *oracles* semelhantes aos escritos por programadores, existem ainda algumas questões relativamente ao uso destas ferramentas em ambientes de desenvolvimento de *software* reais. Em particular, quão eficientes são os *oracles* gerados automaticamente a detetar falhas reais no *software*? Quanto tempo levam estas ferramentas a gerar um *oracle*? Estas duas perguntas são importantes pois, para que estas ferramentas sejam úteis em situações reais, a capacidade de gerar *oracles* que se assemelhem a *oracles* escritos por programadores não é tão importante quanto a capacidade destes oracles de detetar falhas existentes no *software*. Além disto, independentemente da capacidade destas ferramentas em detetar falhas no *software*, a sua utilização poderá ser irrelevante caso a geração de *oracles* seja demasiado demorada.

Para responder a estas questões, é necessário, em primeiro lugar, *software* real que contenha falhas já identificadas. Para isto, foi escolhido o DEFECTS4J, uma coleção de vários projetos *open-source*, com falhas documentadas e com testes que conseguem identificar essas falhas. Assim, e para que seja possível a utilização destes testes juntamente com *oracles* gerados automaticamente, foram removidos os *oracles* escritos manualmente de todos os testes capazes de detetar uma falha no DEFECTS4J. Para proceder à remoção

destes *oracles*, verificámos o *stacktrace*, manualmente e para todos os bugs detetatos nesta coleção, de forma a identificar todas as linhas que levam os testes a falhar. O procedimento seguido para a esta tarefa passa pela remoção da linha em que um teste falha e de todas as linhas seguintes, uma vez que a execução destas linhas poderia levar um teste a falhar por qualquer outra razão. Desta forma, foram identificadas 4 razões que levavam um teste a falhar: (1) *assert statements* – um teste falha devido a um resultado que difere do esperado. Aqui, é removida a linha onde é feito o *assert* e todas as seguintes; (2) *auxiliar test method* – é feita uma chamada a um método auxiliar, onde todas as verificações necessárias são feitas e o teste falha dentro desse método. Neste caso, apesar de ser uma simplificação, removemos apenas a chamada ao método auxiliar e todas as linhas seguintes do método de teste; (3) *expected exception* – é esperado o lançamento de uma exceção, ao executar determinado código, que não ocorre. Neste caso, o teste poderá falhar por não ser lançada uma exceção, ou por ser lançada a exceção errada. O *oracle* poderá ser um bloco *try/catch* e, neste caso, será removido todo o bloco e as linhas seguintes, ou uma anotação do JUnit e, nesse caso, será removida a anotação e todas as linhas do método de teste, uma vez que não é possível dizer, com certeza, qual o pedaço de código que deveria lançar a exceção; (4) *unexpected exception* – uma exceção que não era esperada é lançada ao executar um pedaço de código. Neste caso, é removida a linha em que a exceção é lançada e todas as linhas seguintes.

Além de remover as linhas que detetam os *bugs* existentes no código, é, também, adicionado um marcador nessa mesma linha que é necessário, por algumas ferramentas de geração automática de *oracles*, para identificar em que linha o novo *oracle* deve ser inserido. Todo este procedimento possibilita a integração de *oracles* gerados automaticamente em testes escritos manualmente.

Finalmente, utilizámos uma recente e promissora ferramenta de geração automática de *oracles* (T5) para gerar *oracles* para todos os testes resultantes do procedimento descrito anteriormente e registámos o quão eficazes são os *oracles* gerados por esta ferramenta na deteção de falhas reais, assim como o tempo necessário para a geração dos mesmos. Foram, também, identificadas outras ferramentas além do T5, que acabaram por ser descartadas pelas mais diversas razões (por exemplo, a ferramenta não estar disponível, ter pouca documentação, ou estar fora do âmbito do estudo realizado).

Para a execução da ferramenta selecionada (T5), é necessária a identificação de um método de teste que contenha um marcador para a posição do *oracle* a ser gerado e do método de foco desse teste, i.e., do método cujo funcionamento se pretente testar. Para identificar o método de foco de cada teste, foi criado um programa em Java. Este programa assume que o programador segue a prática comum de nomenclatura do JUnit, i.e., que o nome de uma classe de teste é composto pelo nome da classe a ser testada com o sufixo, ou prefixo, '*Test*' e que um método de teste tem, geralmente, no seu nome o nome do método a testar. Caso não seja identificado um método que corresponda a estas carac-

terísticas, então assume-se que o método de foco será o último método a ser invocado no método de teste (desde que este pertença ao projeto em questão).

Os nossos resultados demonstram que, em geral, o T5 está, ainda, longe de ser utilizável em cenários reais de desenvolvimento de *software*. Inicialmente, os *oracles* gerados pelo T5 não compilam. Isto deve-se ao facto de que o T5 gera *oracles* apenas em letra minúscula, podendo não corresponder aos nomes de classes, métodos e variáveis. Para combater isto, foi criado um script que, dados um método de teste, um método de foco e o *oracle* gerado, tenta corrigir a capitalização das letras através do nome de variáveis existentes no método de teste e dos seus tipos, e do nome do método de foco. Além disto, este script conta também com dados obtidos da API do Java 8 para corrigir o maior número de *tokens* possível.

Após a execução do processo descrito, dos 1696 *oracles*, apenas 466 compilam, dos quais, 58 conseguem detetar corretamente uma falha. É, também, importante de notar que o T5 foi treinado apenas para lidar com *asserts*, pelo que esta ferramenta não possui conhecimento suficiente para lidar com muitos dos casos existentes no DEFECTS4J. Quando consideramos todos os 835 *bugs* no DEFECTS4J, o T5 foi capaz de detetar 27, ou seja, 3.23% dos *bugs*. Além disto, o T5 requer, em média, 401.3 segundos para gerar um *oracle*.

As abordagens e datasets apresentados nesta tese aproximam a possibilidade de ferramentas de geração automática de *oracles* serem utilizadas para testar *software* real, pela informação que é fornecida sobre os problemas atuais de várias ferramentas assim como pela introdução de uma nova forma de testar ferramentas de geração de *oracles* em relação às suas capacidades de detetar falhas reais de *software*.

Palavras-chave: Teste de *software*, Testes unitários, *Test oracle*, Estudo empírico, Geração automática de *oracles*

Abstract

Automated test suite generation tools have been used in real development scenarios and proven to be able to detect real faults. These tools, however, do not know the expected behavior of the system and generate tests that execute the faulty behavior, but fail to identify the fault due to poor test oracles. To solve this problem, researchers have developed several approaches to automatically generate test oracles that resemble manually-written ones. However, there remain some questions regarding the use of these tools in real development scenarios. In particular, how effective are automatically generated test oracles at revealing real faults? How long do these tools require to generate an oracle?

To answer these questions, we applied a recent and promising test oracle generation approach (T5) to all fault-revealing test cases in the DEFECTS4J collection and investigated how effective are the generated test oracles at detecting real faults as well as the time required by the tool to generate them;

Our results show that: (1) out-of-the-box, oracles generated by T5 do not compile; (2) after a simple procedure, out of the 1696 test oracles, only 466 compile and 58 of them manage to correctly identify the fault; (3) when considering the 835 bugs in DEFECTS4J, T5 was able to detect 27, i.e., 3.23% of the bugs. Moreover, T5 required, on average, 401.3 seconds to generate a test oracle.

The approaches and datasets presented in this thesis bring automated test oracle generation one step closer to being used in real software, by providing insight into current problems of several tools as well as introducing a way to test automated test oracle generation tools that are being developed regarding their effectiveness on detecting real software faults.

Keywords: Software testing, Unit testing, Test oracle, Empirical study, Automated oracle generation

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software can be found in various types of things, from the simplest, like fridges and washing machines to the more complex, such as banks or planes, systems upon which human lives currently depend on.

As the complexity of the system being developed increases, so does the difficulty to write reliable software. Due to this, software faults are bound to be found in any software system at some point in time. Software faults cause a program to behave in an incorrect or unexpected way and their impact on our lives can vary, from being relatively low, like forcing a program to close, to being extreme, like causing a plane to crash, having a major repercussions on human lives.

1.1 Motivation

When faulty software is deployed, the undetected faults, not only get more expensive to fix as time passes [52], but can also lead to major disasters as we have seen in the last years.

For example, NASA's Mars Climate Orbiter, which had a building cost of 125$ million, was lost in space due to a software fault. After the source code was inspected, it was discovered that the failure was originated from a navigational error, due to a miss conversion of values. A portion of the source code was computing the force in 'pounds of force' and, another portion of the code, read this data in the metric unit ('newtons per square meter')[1].

Other faults, however simple, can cause the loss of millions of dollars to companies. For example, the case of Compound, a popular decentralized-finance staking protocol, where a one-character boundary condition ('>' arithmetic comparison should have been '>=') costed 90$ million[2]. Or worse, the loss of human lives, e.g., the two Boeing 737

[1] `https://www.wired.com/2010/11/1110mars-climate-observer-report/`

[2] `https://www.cnbc.com/2021/10/01/defi-protocol-compound-mistakenly-gives-away-millions-to-users.html`

Max crashes, which killed 346 people[3]. Security faults might also, for example, enable a malicious user to bypass access controls to obtain unauthorized privileges, as recently happened with Twitch, where a server configuration error allowed its source code, detailed users' payouts and encrypted passwords to be leaked[4].

To reduce the number of faults present in software, and therefore, lower the probability of some kind of an error occurring, software must be tested [44]. Software testing checks whether a program's execution is according to its expected outcome [25] and is one of the most crucial, challenging, and expensive parts of a software's lifecycle [21].

1.2 Problem

Thoroughly testing a software system is a laborious and error-prone task which often requires knowledge of the system and its source code [21, 4]. Thus, automation is often advocated to reduce the manual effort. In particular, efficient techniques to automatically generate unit tests for object-oriented software have been developed and resulted in popular tools such as AGITARONE [32], RANDOOP [45], and EVOSUITE [22, 2]. These tools generate unit tests by analysing and executing the system to be tested, reducing the amount of work that developers need to do in order to create a usable test suite.

Although automatic test generation tools have been able to achieve good results (e.g., high code coverage [23, 24, 50, 10] or being able to detect real faults [1, 56]), they still have several limitations [18, 56, 53]. For example, a recent study conducted by Shamshiri et al. [56] reported that 44.3% of 357 real faults in the DEFECTS4J database [33] are not detected by automatically generated tests and that 63.3% of these cases were due to weak *test oracles*. In line with Shamshiri et al. [56]'s work, Schwartz et al. [53] reported that stronger *test oracles* could also increase the effectiveness of manually-written tests.

In unit testing of object-oriented software, test oracles are represented as test assertions that check properties of objects created as part of the test. Take the following piece of code as an example.

Listing 1.1: Example of a method with a faulty implementation.

```java
public class Foo {
   public int sum(int a, int b) {
      // return a + b; // correct implementation
      return 0;        // faulty implementation
   }
}
```

Listing 1.1 shows a badly implemented `sum` function. This function takes two numbers and should return the sum of those two numbers, however, as it is, it always returns 0.

[3]https://www.fierceelectronics.com/electronics/killer-software-4-lessons-from-deadly-737-max-crashes

[4]https://blog.twitch.tv/en/2021/10/15/updates-on-the-twitch-security-incident/

Without knowing the intended behavior of the function, current automated test generation tools would always generate a test that expects the function to return the value 0 (see line 5 in Listing 1.2), allowing this fault to pass unnoticed.

Listing 1.2: Test case generated with EvoSuite for the pseudo-method `sum`.

```
@Test
public void test0() {
   Foo foo0 = new Foo();
   int int0 = foo.sum(1, 2);
   assertEquals(0, int0);
}
```

Providing accurate — fault revealing — test oracles for a test generated automatically can be a difficult task [4]. Generated tests may not represent realistic scenarios and may not be as nicely readable as human-written tests. Test generation tools also tend to produce large numbers of tests. Thus, it may not be feasible for a human developer to annotate all generated tests for a program under test with a test oracle. But, is that even necessary? Recently, several techniques have been proposed to improve the results obtained by automatically generated test suites [73, 72] and also to automatically generate test oracles that resemble their manually-written counterparts [8, 65, 61, 40]. However, to the best of our knowledge, there is no evidence that such techniques are able to generate fault revealing test oracles.

1.3 Approach

In this work, we conducted an empirical study to assess the effectiveness of automatically generated test oracles at revealing faults on manually-written unit tests. For this, we used the Defects4J database [33], which contains 835 real faults from 17 open-source projects.

Specifically, our study answers the following research questions:

RQ1: How effective are manually-written tests augmented with automatically generated test oracles at revealing real faults? In this question, we investigate whether T5 [49, 40] is able to generate fault revealing test oracles for manually-written tests.

RQ2: How long does it take to automatically generate fault revealing test oracles? In this question, we investigate the time required by test oracle generation approaches such as T5 [49, 40] to generate fault revealing oracles.

1.4 Contributions

The main contributions of this thesis are as follows:

1. A survey of previously proposed tools for automated test oracle generation and/or test case augmentation, and a review on the usability of five of these tools.

2. A patch dataset that removes oracles from fault-revealing tests in DEFECTS4J [33] and labels the line where an oracle must be injected to detect each bug. This allows others, not only to precisely identify and study the type of oracles present in DEFECTS4J, but also to evaluate their own tools.

3. An empirical evaluation of the effectiveness and efficiency of T5 [49, 40] at generating fault revealing test oracles for 835 real faults in the DEFECTS4J database [33].

4. A repository[5] with all scripts and data generated and used in our study to allow its reproduction and replication.

5. An adaptation of RANDOOP [45][6] and EVOSUITE [22, 2][7] that allows the generation of test cases containing an oracle placeholder. These adaptations can be used in a future work, to evaluate the automated generation of test oracles in automatically generated test cases.

1.5 Structure of the document

This document is organized as follows:

- Chapter 2 describes the required background to understand the contents of this work.
- Chapter 3 describes the techniques and tools proposed in the literature to address the oracle problem, as well as other empirical studies similar to the one proposed with this work.
- Chapter 4 presents the methodology and experimental setup of our analysis.
- Chapter 5 presents the questions we mean to answer with this study.
- Chapter 6 presents the final remarks and what should be done in continuation of this work.

[5]`https://github.com/jose/meaningful-assert-statements-data`
[6]`https://github.com/jose/meaningful-assert-statements-randoop`
[7]`https://github.com/jose/meaningful-assert-statements-evosuite`

Chapter 2

Background

2.1 Structure of a Unit Test

A unit test is composed of the setup, which leads the software to a determined state, the test action, where the focal method is called upon to do something, and the oracle, which determines the correctness of the focal method's execution.

Listing 2.1: Example test method depicting the structure of a unit test.

```java
@Test
public void testSum() {
   int a = 1;
   int b = 1;
   int expected = 2;
   int result = sum(a, b);
   assertEquals(expected, result);
}
```

The piece of code in Listing 2.1 shows an example of a simple test case `testSum()`. A test method is usually composed of three parts:

- **Test Setup**. This part is composed of the steps that need to be taken before the execution of the test method's core body. In the previous example, this includes the initialization of the three variables `a`, `b` and `expected`.

- **Test Action**. In this part, the unit under test (UUT) executes some determined action, which can, or not, be the focal method [48], i.e, the unit we want to test. In the previous example, this is the call to `sum(a, b)`, which is the focal method. Note that, in some cases, a test's action can be performed inside the oracle, e.g., `assertEquals(expected, sum(a, b))`.

- **Test Oracle**. A test oracle determines whether the outcome of the focal method's execution is as expected. In this example, an assertion oracle is used, however, there are other kinds of oracles, which will be explored further in this work.

2.2 Oracle

Assertion oracles, as used in Listing 2.1, are not the only types of oracles. Barr et al. [4] identify four types of test oracles, however, those can fit in two big groups: `automated oracles` and `human oracles`.

2.2.1 Automated Oracles

Automated oracles can be performed by the system without any human input. Two groups of automated oracles were previously identified by Dinella et al. [17]:

1. **Expected Exception Oracles**. This kind of oracle verifies if the execution of some code with an invalid argument raises an exception.

   Listing 2.2: Example test method using an expected exception oracle.

   ```
   @Test
   public void test() {
      String s = "Hello World";
      int index = -1;
      try {
         s.charAt(index);
         Assert.fail();
      } catch (Exception e) {
         verifyException(e, IndexOutOfBoundsException);
      }
   }
   ```

2. **Assertion Oracles**. This kind of oracle is used in the form 'assert*' and verifies if a returned value matches the expected result of an execution.

   (a) **Boolean Assertions**. Used to check whether a value is true or false.

   Listing 2.3: Example test method using a boolean assertion oracle.

   ```
   @Test
   public void test() {
      String s = "Hello World";
      assertTrue(s.startsWith("H"));
      assertFalse(s.isEmpty());
   }
   ```

   (b) **Nullness Assertions**. Usually used to check the returned value from a function.

   Listing 2.4: Example test method using a nullness assertion oracle.

   ```
   @Test
   public void test() {
      LinkedList ll = new LinkedList();
      assertNull(ll.peekFirst());
      ll.add("A");
      assertNotNull(ll.peekFirst());
   }
   ```

(c) **Equality Assertions**. Used to compare the value returned from a function against an expected value.

Listing 2.5: Example test method using an equality assertion oracle.

```java
@Test
public void test() {
   String initial_s = "HELLO WORLD";
   String expected_s = "hello world";
   assertEquals(expected_s, initial_s.toLowerCase());
}
```

2.2.2 Human Oracles

If no automated oracle can be used, the only source for a test oracle remaining is the human, who knows informal specifications, the domain of the problem and what the expectation for the system's output is.

2.3 Summary

In this chapter, we briefly described and demonstrated both, the structure of a unit test and the parts that it can be split into, as well as what types of oracles there are and how they are used. With this, our aim is to familiarize the reader with the notions of unit testing and test oracles used in this document.

Chapter 3

Related Work

3.1 Automated Generation of Test Oracles

Writing unit tests is a complicated and tedious task for humans to perform. The automatic generation of unit tests brings along the problem of knowing what is the correct outcome of a program or software. This problem is known as the *oracle problem* [4]. Some solutions to solve this problem have been proposed, for example, by Braga et al. [8], who presented an approach based on Machine Learning, capturing usage data from an application and using that data to train the generation of an oracle suitable for the application under test. Pastore et al. [46] used a different approach and tried to solve the oracle problem by exploiting CrowdSourcing, dividing the task of generating an oracle among several users.

3.1.1 Test Suite Automation

Some researchers have developed tools to automate the generation of test suites. Among these tools, the three most mentioned in the literature are: (1) AGITARONE [32], that automatically creates high code-coverage (about 80%) tests by analysing the project's source code and taking advantage of mocking technology to solve many of the dependency issues; (2) RANDOOP [45], which uses feedback-directed random test generation; and (3) EVOSUITE [22], which integrates μTEST [26], making use of mutation analysis in object oriented languages to generate the most effective subset of test cases possible. Both EVOSUITE and RANDOOP are able to automatically generate *regression oracles*, i.e., JUnit assertions or exceptions that exercise the current behavior of the program under test. EVOSUITE first generates all possible oracles for each test case and then filters out these oracles based on their mutation score, i.e., on their ability to *kill* mutants [26]. RANDOOP runs each test case, collects the output of any public API of the class under test, and generates oracles based on the collected data [45].

TESTNMT [67] and the solution proposed by Fontes et al. [21] can generate "partial tests". These tests may have syntactical errors or not generate the oracles, which are needed to infer the correctness of the software, and need to be adapted by the developer to produce a final desired test.

Despite the increasing number of tools that have been developed over time, integration into everyday's software development cycle remains a problem that needs to be solved. Arcuri et al. [2] solve this problem by developing three plugins — one for Apache Maven, another for the IntelliJ Integrated Development Environment (IDE) and a last one for Jenkins Continuous Integration (CI) system — for EvoSuite. This improves the integration of EvoSuite into the development process of software projects.

Fontes and Gay [20] do a literature review over the use of machine learning to generate test oracles and list the main challenges of this approach. Test oracles generated through the use of machine learning are limited by the quality and quantity of training that they are exposed to. These approaches often need a big volume of training data and the process of gathering and filtering the contents of said data still requires a significant amount of human effort. Besides this, the researchers often test their approaches with software that is simple, not being representative of real world scenarios. For this, the authors mention the use of a standard fault benchmark, for example, the Defects4J [33] database, that provides real world scenarios of code with a faulty and a fixed version of the software.

When evaluating the results obtained by automatically generated test suites, Shamshiri et al. [56] and Shamshiri [55] find that even though some faults are never executed, most of them are executed but not detected. This might indicate that a better way to generate test oracles is needed. Another problem found, especially in randomly generated tests (e.g. Randoop), is that some of these tests are flaky, which makes them unreliable.

3.1.2 Oracle Automation

A *test oracle* [29] is identified as a mechanism that determines whether a test has passed or failed. Oracles compare the output of a system under test, for a given input, to the expected output. Oracles can be automated, being executed or generated automatically.

Barr et al. [4] do a survey of the existent approaches to solve the test oracle problem. In this study four techniques for oracle automation were identified: (1) *specified test oracles*, (2) *derived test oracles*, (3) *implicit test oracles* and (4) the *human test oracle*.

3.1.2.1 Specified Test Oracles

A specification defines, if possible, using mathematical logic, a test oracle for a determined domain. A specification language allows the definition of a specified test oracle that judges whether the behavior of a system conforms to a formal specification.

Pre-conditions and post-conditions — state-based specifications — impose a necessary condition over the input states and the effect an operation has on the program's state, respectively.

Assertions are fragments of specification languages. An assertion checks the correctness of the software at runtime, according to a boolean expression.

Researchers have developed tools that can generate assertions in a deterministic manner by taking as input the test method and, for some tools, the focal method as well. These tools take advantage of Neural Machine Translation (NMT), which uses an artificial neural network to predict the likelihood of a sequence of tokens. Machine Translation is a task of Natural Language Processing (NLP) where software is applied to translate text from one language to another. Two approaches of NMT have been used: (1) Recurrent Neural Networks (RNNs), which process an input sequence token by token; and (2) the Transformer model, which relies entirely in the attention mechanism to capture the semantic dependency from the global view. Transformers also allow for more parallelization than RNNs, making them more efficient.

Automatic **L**earning of **A**ssert **S**tatements (ATLAS) [65] applies sequence-to-sequence learning with an RNN encoder-decoder model. This tool uses SEQ2SEQ [9], a framework for the implementation of sequence-to-sequence models. ATLAS takes in a pair composed of the contextual test method (test method + focal method) and predicts a meaningful assert statement that replaces the original oracle. Overall, the main challenges ATLAS presents are: (1) it needs the test methods to be written previously; (2) only allows the generation of one assert statement per test method, as it is not as easy to determine which part of the test method's body is relevant to a certain assert statement; and (3) it focuses only in assertion oracles and can not create expected exception oracles, described in Section 2.2.

Since the introduction of the Transformer [63] architecture, various transformer-based techniques have been proposed. BERT [16] (**B**idirectional **E**ncoder **R**epresentations from **T**ransformers) is designed to pre-train deep bidirectional representations of unlabeled text and is used to create state-of-the-art models for a wide variety of tasks. BERT is pre-trained in two tasks: (1) Masked Language Model (MLM), where random tokens are masked and the model has the objective of guessing the original token; and (2) Next Sentence Prediction (NSP), where two sentences are given to the model and it must decide whether the two sentences are related or not. Other researchers have, since then, presented new models with similar pre-training techniques as BERT's. Liu et al. [39] propose an approach based on BERT, RoBERTa (**R**obustly **O**ptimized **BERT** **A**pproach), which trains the model for longer, with more data, removing the NSP task, using longer sequences and dynamically changing the masking pattern. BART [36] pre-trains a model by combining **B**idirectional and **A**uto-**R**egressive **T**ransformers and generalizing BERT's MLM. Similarly, CodeBERT [19] also uses a similar MLM task to pre-train a model. These two last approaches (BART and CodeBERT) can outperform RoBERTa.

Tufano et al. [61] present an approach where a BART [36] transformer is pre-trained on both, an english and source code corpus ($\text{BART}_{\text{ENG+CODE}}$), which is then fine-tuned on the task of generating assert statements for unit tests. To fine-tune this approach for assert generation, the authors rely on the dataset used by ATLAS [65]. This dataset is composed of test methods, focal methods and assert statements. The authors evaluated their approach on how accurately it predicts human-written assert statements by comparing it with ATLAS [65]. $\text{BART}_{\text{ENG+CODE}}$ was able to outperform ATLAS by 80% on top-1 accuracy, where $\text{BART}_{\text{ENG+CODE}}$ achieved 62.47% of prediction accuracy while ATLAS was only able to achieve 26.40%. Moreover, the authors also investigated whether the coverage achieved by tests automatically generated by EVOSUITE [22] would increase with the integration of test oracles generated by $\text{BART}_{\text{ENG+CODE}}$. Their approach increased line and condition coverage of the automatically generated tests for 13 out of 18 test methods. The generated oracles improved the line and condition coverage between 1-3 and 1-4 more lines, respectively. For 4 methods the generated oracles, although being correct, did not improve the coverage, while for 1 method no correct oracle was generated.

REASSERT [68] can use three different models to generate assert statements, which include the Reformer [34], a state-of-the-art Transformer [63] architecture, ATLAS [65] and TESTNMT [67]. The Reformer combines the modeling capacity of a Transformer with a more efficient architecture and less memory use, making it more efficient. As opposed to ATLAS, REASSERT [68] does not require the assert-less test method to be written beforehand and allows the generation of more than one assert per test method. However, it does not do well with small projects as that does not allow the models to generalize maximally.

Mastropaolo et al. [40] pre-train and fine-tune the **T**ext-**t**o-**T**ext **T**ransfer **T**ransformer (T5) [49] model. Similar to the $\text{BART}_{\text{ENG+CODE}}$ [61] approach described before, this model is also pre-trained on a dataset composed of both, english text and source code. This approach is then fine-tuned in four different techniques: (1) fix faults; (2) inject code mutants; (3) generate assert statements; and (4) generate code comments. For the task of fine-tuning the model in the assert generation task, both versions (abstract and raw) of the ATLAS [65] dataset were used.

Mastropaolo et al. [40] compared the results achieved by T5 [49] when fine-tuned to support the assert generation task with the results reported by ATLAS [65].

In this study, the two original ATLAS datasets were used: `AGraw` , which contains the raw source code for the input, and `AGabs` , which contains the abstracted version of input and output.

When considering the abstracted version of the input (`AGabs`), T5 was able to achieve similar accuracy values to the ones reported by ATLAS, for example, for K = 1, the accuracy obtained by T5 was 34%, while ATLAS reported an accuracy of 31%. However, when considering the raw dataset (`AGraw`), T5 achieved a 29% higher accuracy with K = 1 (47% vs. the 18% reported by ATLAS), increasing the performance gap for larger K values between 35-38%. T5 achieved similar results both with and without abstraction.

Dinella et al. [17] developed a new approach, **T**est **O**racle **G**ener**A**tion (TOGA), which can infer exception and assertion test oracles based in a unit's context. This approach uses CodeBERT [19] for the generation of both, exception and assertion oracles. TOGA is pre-trained with a variant of the ATLAS [65] dataset for the task of generating assertion oracles and takes as input a test prefix and a unit context, which can refer to a method's signature or documentation.

Dinella et al. [17] compared their approach (TOGA) with $\text{BART}_{\text{ENG+CODE}}$ [61] in the context of how accurate the generated assert statements are. This was made using a variation of the ATLAS [65] dataset. Results were reported using two test sets: (1) overall set, which consisted of the entire variation of the ATLAS dataset; and (2) in-vocab set, which is the subset of the overall set that can be expressed by the grammar and vocabulary defined by TOGA. For the in-vocab set, TOGA and $\text{BART}_{\text{ENG+CODE}}$ achieved accuracies of 96.0% and 63.0%. However, when considering the overall set, the accuracy achieved by TOGA decreased to 69.0% while $\text{BART}_{\text{ENG+CODE}}$'s accuracy was 62.0%.

3.1.2.2 Derived Test Oracles

Derived test oracles differentiate valid and invalid behavior by using information derived from the system's properties (documentation, system execution results and characteristics of previous versions).

Metamorphic testing [11] exploits properties necessary for the correct functionality of the software in order to assess a system's correctness — *metamorphic relations*. For example, when implementing a sine function, a metamorphic relation to this function is "$sin(x) = sin(\pi - x)$" ([69]). These relations allow for an increased space of inputs used for testing, without the need to know what the output of a given input should be. If these relations do not hold, then we know the software has some defects ([71, 66]).

Segura et al. [54] verify that the interest in metamorphic testing has been increasing in various areas, such as metamorphic relation identification, test case generation, integration with other techniques and validation and evaluation of software. In this study, as well as the ones developed by Liu et al. [38] and Chen et al. [12], researchers find that metamorphic testing can obtain similar fault-detection results to the use of a test oracle, without the need to know what the expected output to a certain input is. The main challenge that metamorphic testing presents is the identification of good metamorphic relations. It is important, not only to use a variety of diverse metamorphic relations, but also to know how to select the most effective ones.

More recently, Ibrahimzada et al. [31] developed SEER, which can determine whether a unit test passes or fails on a given method under test when no oracles are present in the test method. SEER was trained on a large dataset based on DEFECTS4J to learn the semantic correlation between inputs and outputs. To achieve this, SEER uses joint-embeding to semantically separate the representation of correct and buggy MUTs depending on the result of the tests.

Regression test suites assume that the previous implementation of the system is correct and can be used as an oracle. These test suites check the correctness of a system's new implementation based on past functionality.

EVOSUITE [22] generates test suites that are, not only optimized for code coverage, but also enhanced with regression assertions that capture the current behavior of the tested unit.

System executions can be used by exploiting the execution trace to derive test oracles.

Molina et al. [43] developed EVOSPEX, which takes advantage of system executions to generate post-condition assertions. This tool first executes the method under test to obtain valid pre and post-state pairs. Then, these valid pre and post-state pairs are mutated to generate invalid ones. Finally, a genetic algorithm is used to infer assertions in the form of post-conditions that satisfy the valid pre and post-state pairs.

Danglot et al. [14] developed DSPOT, which uses mutation score to try to improve upon developer-written test cases. This approach takes a test case as input and outputs an improved version of the same test case. These new tests can be directly integrated into the main codebase through patches or pull requests.

RANDOOP [45] generates unit tests using feedback-directed random testing, a technique that uses execution feedback gathered from executing test inputs as they are created. Besides this, input sequences that exhibit normal behavior (no exceptions and no contract violations) are output as regression tests.

Textual documentation describes requirements of a system's functionality and can be used as a basis for generating test oracles. Although being, usually, partial and ambiguous, in contrast to specification languages, developers are more likely to write them.

Goffi et al. [27] developed TORADOCU, which automatically generates test oracles for exceptional behaviors by processing Javadoc comments. TORADOCU takes advantage of aspect oriented programming and AspectJ[1] to embed the generated oracles into existing test suites. Similarly, JDOCTOR [7] also parses Javadoc comments by combining pattern, lexical and semantic matching. However, unlike TORADOCU, JDOCTOR creates software contracts, by generating executable procedure specifications for preconditions, and normal and exceptional post-conditions.

3.1.2.3 Implicit Test Oracles

Implicit test oracles rely on common and implicit knowledge of a system's correct and incorrect behavior. An implicit test oracle does not require domain knowledge as it is based on facts, e.g., "the system must not crash".

Exceptions or crashes are, generally, blatant faults. However, there are no universal rules and an exception (or even a crash) can be a fault in one system, but be expected in

[1]`https://www.eclipse.org/aspectj/`

another.

3.1.2.4 Human Test Oracles

When no other solution works, a human tester must be used as an oracle. The human tester can be involved in the process in two ways: (1) writing the test oracles; and (2) evaluating the outcome of tests. Researchers have worked in finding ways to reduce the effort needed by the human tester in both of these. For example, Pastore et al. [46] try to automate the test oracle mechanism by outsourcing the problem to an online service where other people provide the answers — i.e., CrowdSourcing. In this study researchers point out key challenges, e.g. the need for the problem to be exposed in a simple way, provide sufficient and clear documentation of the problem and the need for the workers to have some experience in the area. Despite all of this, the described research still shows promising results (at least 69% correct answers with a qualified crowd).

3.2 Effectiveness of Automatically Generated Oracles

Other researchers have conducted empirical evaluations regarding the effectiveness of automatic oracle generation approaches at detecting real faults [56, 17, 1]. Here we describe the works that have evaluated the effectiveness of automatically generated tests or test oracles at detecting the faulty behavior of the program under test.

Some tools, as is the case for EVOSUITE [22], RANDOOP [45] and AGITARONE [32], are able to generate complete test suites that are complemented with a test oracle, and do not focus specifically on the generation of test oracles. While these tools are not the focus of our work, they are established as state-of-the-art in automated test suite generation and can be seen as a baseline for our analysis.

Shamshiri et al. [56] evaluated and compared the effectiveness of these three state-of-the-art test generation tools (i.e., EVOSUITE, RANDOOP and AGITARONE) at detecting the 357 real faults in the DEFECTS4J v0.1.0 dataset. In this evaluation, researchers reported that the automated test generation tools were able to generate test cases that detected 199 out of the 357 faults (55.7%), but no tool alone could detect more than 40.6%. When considering tools individually, EVOSUITE, AGITARONE and RANDOOP detected 145, 130, and 93 faults, respectively. Furthermore, EVOSUITE and RANDOOP generated 3.4% and 8.3% non-compilable test suites on average, 21.0% of the tests generated by RANDOOP were flaky (i.e., they passed/failed temporarily due to some dependencies), and 46.0% of AGITARONE 's failing tests were false positives.

Almasi et al. [1] evaluated both EVOSUITE [22] and RANDOOP [45] on a real financial application known as LifeCalc, owned and developed by SEB Life & Pension Holding AB Riga Branch. In total, 19 out of 25 faults were detected. On average, EVOSUITE and RANDOOP generated a test suite that detected 50.8% and 36.8% of the faults, respectively.

Some of the researchers for automated test oracle generation tools have evaluated their approaches regarding how effective these approaches are at detecting bugs or at improving test cases in real world projects.

Dinella et al. [17] integrate TOGA with the EvoSuite [22] toolset to determine whether TOGA can generate test oracles capable of detecting real faults. The integration resulted in automatically generated test suites that detected 54 real faults out of the 835 in the Defects4J v2.0.0 dataset.

Danglot et al. [14] evaluate their approach (DSpot) using 40 real-world unit test classes from 10 open-source software projects. DSpot was able to improve 26/40 of the test cases. The researchers also proposed the use of these automatically improved test cases to the projects' lead developers. In the end, 13/19 of these improved test cases were accepted and merged into the main code base.

In this study, as opposed to the approaches described previously, we evaluated an automatic oracle generation approach on manually-written tests. In a nutshell, we investigated (1) how effective are automatically generated oracles at detecting real faults when used in manually-written tests, and (2) how efficient are automatic oracle generation approaches at generating test oracles.

3.3 Summary

Researchers have developed several tools that automatically generate test suites or test oracles, in order to ease the task of writing unitary tests. However, these tools still have trouble distinguishing correct from incorrect behavior — known as the *oracle problem*.

In this chapter, we do a survey of the state-of-the-art on both, test suite automation and oracle automation. First, we review three test suite generation tools that have been used in real-world development as well as a few others and describe the problems raised by these tools. We then identify several oracle automation techniques as well as the type of oracles that each of them generates, and, as we did previously, describe the problems that each of these tools presents. Moreover, we also demonstrate how the oracles generated by these tools perform when compared with manually-written oracles or when compared with the oracles generated by another tool. Finally, we present other works that, similarly to our evaluation, have analyzed how the previously surveyed tools perform when generating test cases and test oracles for real-world projects.

Tools to automatically generate test suites are already used in real development scenarios. However, these tools still fail to identify many of the faults present in the software [56, 55], meaning that a better way to generate test oracles may be needed. Although a lot of effort is being put into the automation of generating better test oracles, most of these tools have only been tested regarding how accurately they can generate an oracle that is similar to a manually-written oracle. However, in reality, how effective are these automatically generated oracles at detecting real faults?

Chapter 4

Empirical Study

To evaluate the effectiveness of automatically generated test oracles at detecting real faults, we performed an empirical evaluation of automatically generated test oracles on manually-written test cases. Despite advances on the automatic generation of test oracles [26, 72, 46], the *oracle problem* is still one of the main challenges of automatic test generation (as without a formal specification or domain knowledge of the program under test, it is almost impossible to know its intended behavior).

In this section we present the methodology and experimental setup of our empirical analysis.

4.1 Experimental Subjects

To automatically perform our experimental analysis, the selection of subject programs used in our empirical evaluations is restricted to the following requirements: (1) the programs used should be developed in Java as the considered tools only support Java; (2) two versions of the same program should be available to automatically validate if an oracle detects or not the change, i.e., the fault; and (3) the difference between two versions should be documented as a patch so that we could automatically validate if an automatically generated test case covers the faulty code[1]. One particular collection of subject programs that meets all requirements is DEFECTS4J [33] 2.0.0 (#74b5b81)[2]. DEFECTS4J is a collection of 835 reproducible and isolated real software faults from 17 large Java open-source programs. For each real fault, DEFECTS4J provides a pair of program versions: a fixed and a faulty version (created by applying a minimal patch to the fixed version) and the corresponding test suites with at least one test case that triggers the fault. In addition, DEFECTS4J has been used in similar studies (e.g., [56, 17]) which allows us to compare our results to the ones reported in the literature.

[1]Note that no test oracle could ever trigger the faulty behavior of the program under test if the test case does not cover the faulty code.

[2]`https://github.com/rjust/defects4j/tree/v2.0.0`

4.2 Experimental Setup

All experiments were executed on a machine at the Faculty of Sciences of the University of Lisbon, Portugal. The machine was running Ubuntu 5.4.0-125-generic (x86_64) with 32 Intel(R) Xeon(R) Silver 4216 CPU at 2.10GHz, and 64 GB of RAM.

4.2.1 Unit Test Cases

In our evaluation, we assess the effectiveness of automatically generated oracles when coupled with developer-written test cases. For this, we used a variation of the fault-revealing manually-written tests available in the DEFECTS4J [33] database, where we removed the lines that trigger the bug, with the intent of replacing these lines with automatically generated oracles.

4.2.2 Test Oracles

In our study we focused on tools capable of either generating a complete oracle (e.g., ATLAS [65], $\text{BART}_{\text{ENG+CODE}}$ [61], T5 [49, 40] and TOGA [17]) or generating augmented test methods (e.g., DSPOT [14]) that can be integrated into the original test suites. Some tools, as is the case for JDOCTOR [7] and TORADOCU [27], are only capable of generating pre and post-conditions, or Java Aspects that enhance parts of a test suite.

Some of these tools, however, proved unusable for various reasons: (1) ATLAS lacks support and documentation that helps to execute it[3]; (2) $\text{BART}_{\text{ENG+CODE}}$ is not publicly available; (3) TOGA requires that we provide it an already formed oracle, which is then improved[4]. This would not allow for a fair comparison or to make any solid conclusions of TOGA's results, as it would already have been provided with an oracle that tested the buggy behavior; and (4) DSPOT can only handle maven projects[5]. As DEFECTS4J contains projects that do no use maven, we opted to exclude DSPOT.

With this in mind, to automatically generate test oracles for manually-written tests, we opted to use only T5[6]. Mastropaolo et al. [40] made two T5 models available (raw and abstract), which were trained for the generation of assert statements and differ in both, complexity and results obtained. In our study, we evaluate only T5's raw model, as it requires a simpler procedure to integrate into a real development scenario. This model is also the one with the best accuracy when comparing the generated oracles with the ones written by developers.

[3]https://gitlab.com/cawatson/atlas---deep-learning-assert-statements/-/issues/2

[4]https://github.com/microsoft/toga/issues/3

[5]https://github.com/STAMP-project/dspot/issues/728

[6]Disclaimer: during the development of this thesis a new version of T5 [41] was proposed that addresses some of the issues we later report in Section 5.1.3. Due to time constraints, we have not included it in our empirical study.

4.2.3 Fault Set

In our study, we used DEFECTS4J 2.0.0 (#74b5b81) [33]. This collection is composed of 17 open-source Java programs (Chart, Cli, Closure, Codec, Collections, Compress, Csv, Gson, JacksonCore, JacksonDatabind, JacksonXml, Jsoup, JxPath, Lang, Math, Mockito, Time), adding up to 835 bugs.

4.3 Experimental Procedure

4.3.1 Fault Detection

To assess whether automatically generated oracles do actually detect a fault, at least two different experiments could be performed: (1) ask a human developer to inspect every generated oracle (expensive and error prone); or (2) execute every generated oracle on a different version of the same program (cheap and can be performed automatically). Ideally, the difference between the program version in which oracles are generated and the program version in which oracles are executed should be minimal and represent a single change. Given that DEFECTS4J already provides such minimal difference between the faulty and fixed version of each fault, we can perform (2) out-of-the-box and most importantly, automatically.

Note that automatically generated test oracles could fail to detect a fault for reasons unrelated to the fault itself. We identified these cases by checking whether the test cases cover any faulty line and assuming the failure is fault-related if and only if it does. Moreover, we also (1) looked at the stacktrace of each failing test case and compared it with the expected stacktrace provided by DEFECTS4J and, (2) compared the generated test oracles with their developer-written counterparts.

4.3.2 Experimental Metrics

To judge the effectiveness and usefulness of the evaluated test oracle generation approach (T5) at detecting real faults we measured a variety of metrics.

- Number of *broken* test oracles, i.e., oracles that do not compile on the program's version they were generated for.
- Number of *flaky* test oracles, i.e., oracles that fail on the program's version they were generated but due to other reason than the fault itself.
- Number of real faults detected.
- Runtime to generate a fault-revealing test oracle. An approach might not be considered as being usable in a real world scenario if it requires a long time to generate test oracles for a given test suite.

We also perform a statistical analysis of our results and discuss whether, e.g., approach X performs statistically better at generating fault-revealing test oracles than approach Y. Depending on the distribution of the results obtained in our experiments, we will either use parametric or non-parametric statistical tests to perform that analysis.

4.4 Threats to Validity

Based on the guidelines described by Wohlin et al. [70], we discuss the threats to validity.

4.4.1 Construct validity

A potential threat to *construct validity* is that we only used one tool (T5) in our study, as we either could not manage to execute other tools or they were not suited for the purpose of this study, as described in Section 4.2.2.

Moreover, the use of all faults in the DEFECTS4J database, without any sort of differentiation on what are the types and severities of the faults used in our study, may present another threat to *construct validity*. As T5 was trained to generate assert statements only and does not know how to handle expected exception cases, our results my not be representative of this approach's capability of generating oracles capable of detecting faults for the type of oracle it was trained on (i.e., assert statements).

Furthermore, the fault set used may not include every bug that was detected in these projects, which may underestimate the capability of the automatically generated oracles at detecting real faults. This is, however, mitigated, as the bugs present in DEFECTS4J are taken from all stages of a project's development.

4.4.2 External validity

A potential threat to *external validity* is that our results may not generalize to other programs with different characteristics.

We mitigate this by using DEFECTS4J 2.0.0 (#74b5b81) [33]. This provides us with (1) a big diversity of projects (17 open-source Java programs), which differ in complexity and in how tests are written (as different developers can write the same test method in different ways), and (2) a wide variety of bugs (835).

Moreover, our study only uses manually-written test cases. It is possible, however, that automatically generated test cases could provide a better set up for the automatic generation of oracles, leading to better results.

4.4.3 Internal validity

A potential threat to *internal validity* is that we may overestimate how good the automatically generated oracles are at detecting real faults. We mitigate this by executing the tests in both versions of the software, i.e., a version where the fault is present and a version where the fault has been fixed through a minimal change to the source code. For the cases where the oracle fails in the faulty version of the software but does not in its fixed counterpart, we (1) manually go through the stacktrace to check if a test case covers a faulty line and, (2) compare the generated oracles to their manually-written counterparts to identify false-positives. Moreover, all scripts developed to perform our experiments and run our statistical analysis were revised by all the authors and are publicly available for others to review and/or reuse.

4.5 Summary

In this chapter we describe the setup used to conduct the evaluation of automatically generated oracles as well as the validity threats that may be raised.

The use of DEFECTS4J eases the execution of our study, as it is composed of various projects written in Java, with documented faults, tests capable of detecting those faults and a version where that fault is fixed.

When considering the automated generation of test oracles, only one was used in our study, while several others were discarded. These tools may be discarded due to unavailability, low documentation, having an unfair advantage over other tools, or simply being out of the scope of our study.

Moreover, we describe the procedure and metrics to be recorded during the execution of our study.

Finally, we identify possible validity threats that our setup presents and how we plan to mitigate these threats.

Following the presented setup, in the following chapter, we perform an empirical study to detect how effective and efficient the selected test oracle generation approach is at detecting real-world software faults.

Chapter 5

Empirical Analysis

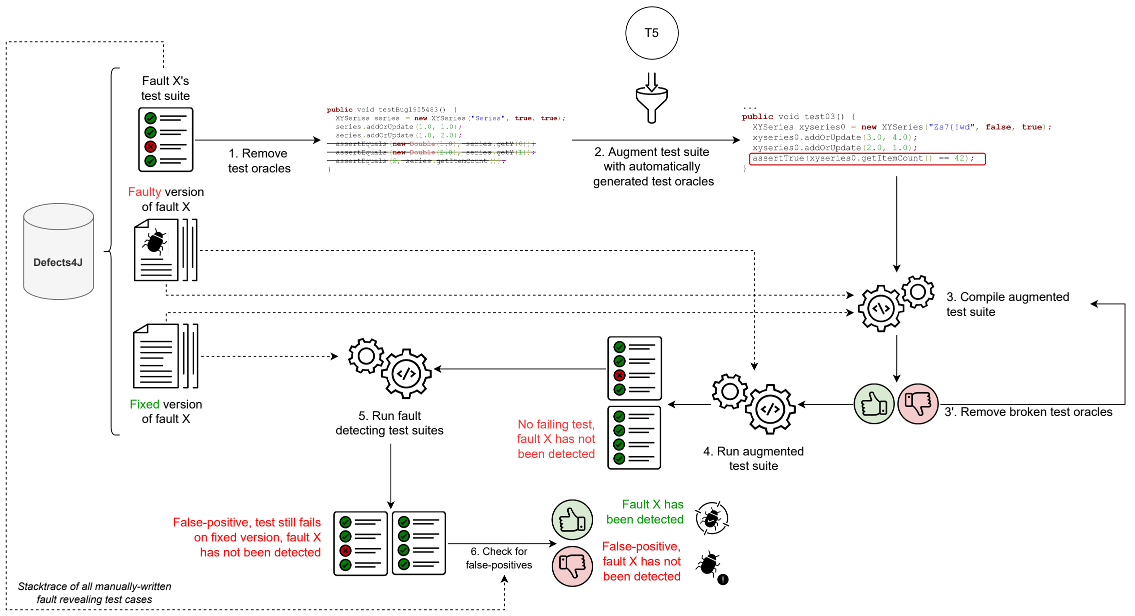
In our empirical analysis, we conduct experiments that allow us to obtain concrete data to evaluate whether automatically generated test oracles are, not only effective at detecting real faults, but also usable in a real development scenario. In particular, our empirical study answers the following research questions:

5.1 RQ1: How effective are manually-written tests augmented with automatically generated test oracles at revealing real faults?

In this research question, we evaluate the effectiveness of an automated oracle generation approach (T5) at producing fault-revealing test oracles for manually-written test cases.

5.1.1 Procedure

Figure 5.1 depicts the procedure we conducted to answer this research question. For each fault X, we ① remove all manually-written test oracles from all fault revealing manually-written tests. ② Augment each test with test oracles generated by T5 [49, 40]. ③ Compile the augmented tests on the faulty and fixed versions of X and remove any test oracle that it is considered broken, i.e., it does not compile. ④ Execute the tests on the faulty version of X to assess its fault detection capability. If no test fails on the faulty version, the fault is considered undetected. ⑤ Otherwise, if at least one test fails, we execute it on the fixed version of X to evaluate for false-positives. A test that fails in both, the version containing the bug and the version that does not contain the bug, it fails due to any other reason that is unrelated to the fault itself and is, therefore, a false-positive. ⑤ If, on the other hand, the test does not fail in the fixed version, we manually compare the automatically generated oracle with the manually-written one. Besides this, we also compare the stacktraces obtained by executing, on the buggy version of the software, the test containing manually-written test oracles and the test with automatically generated test


T5
Fault X's test suite
1. Remove test oracles
2. Augment test suite with automatically generated test oracles
Defects4J
Faulty version of fault X
3. Compile augmented test suite
3'. Remove broken test oracles
Fixed version of fault X
5. Run fault detecting test suites
No failing test, fault X has not been detected
4. Run augmented test suite
False-positive, test still fails on fixed version, fault X has not been detected
6. Check for false-positives
Fault X has been detected
False-positive, fault X has not been detected
Stacktrace of all manually-written fault revealing test cases

```
public void testBug1955483() {
  XYSeries series = new XYSeries("Series", true, true);
  series.addOrUpdate(1.0, 1.0);
  series.addOrUpdate(1.0, 2.0);
  ~~assertEquals(new Double(1.0), series.getY(0));~~
  ~~assertEquals(new Double(2.0), series.getY(1));~~
  ~~assertEquals(2, series.getItemCount());~~
}
```

```
...
public void test03() {
  XYSeries xyseries0 = new XYSeries("Zs7{!wd", false, true);
  xyseries0.addOrUpdate(3.0, 4.0);
  xyseries0.addOrUpdate(2.0, 1.0);
  assertTrue(xyseries0.getItemCount() == 42);
}
```

Figure 5.1: RQ1's procedure.

oracles. If it is clear that the generated oracle tests the same functionality as the manually-written oracle, or both stacktraces report the same error or exceptional message, we then assume the fault is *truly* detected.

(1) Remove all manually-written test oracles from all fault revealing manually-written tests

Figure 5.2 displays the distribution of fault revealing manually-written tests for each project in DEFECTS4J. Overall, in the 835 bugs, there are 1,750 fault revealing manually-written tests in the DEFECTS4J collection. Most bugs are triggered by a single test case (559 out of 835) and on average there are 2.1 fault revealing tests per bug. The bug with the largest number of fault revealing tests is Closure-144 with 84 tests.

To remove all manually-written test oracles from all fault revealing manually-written tests we (1) collected all fault revealing tests and identified the line of code in the test that triggers the buggy behavior of each bug by looking at each test's stacktrace, and (2) removed the oracle and any subsequent lines of code from the test. We remove all lines after the oracle as the execution of these never-executed-before lines could cause the tests to fail for any other reason than the bug itself. The former is an automatic process as that information is already provided by DEFECTS4J. The later was performed manually by all the authors of this document for all bugs and kept in a patch file to ease the validation of the procedure and/or to automatize any procedure related to RQ1.

During this procedure we have identified four types of test oracles that trigger the buggy behavior of each bug in DEFECTS4J (*assert statement*, *auxiliar test method*, *expected exception* and *unexpected exception*). The number of test methods capable of detecting a fault in each project for the identified test oracle types can be seen in Figure 5.3.

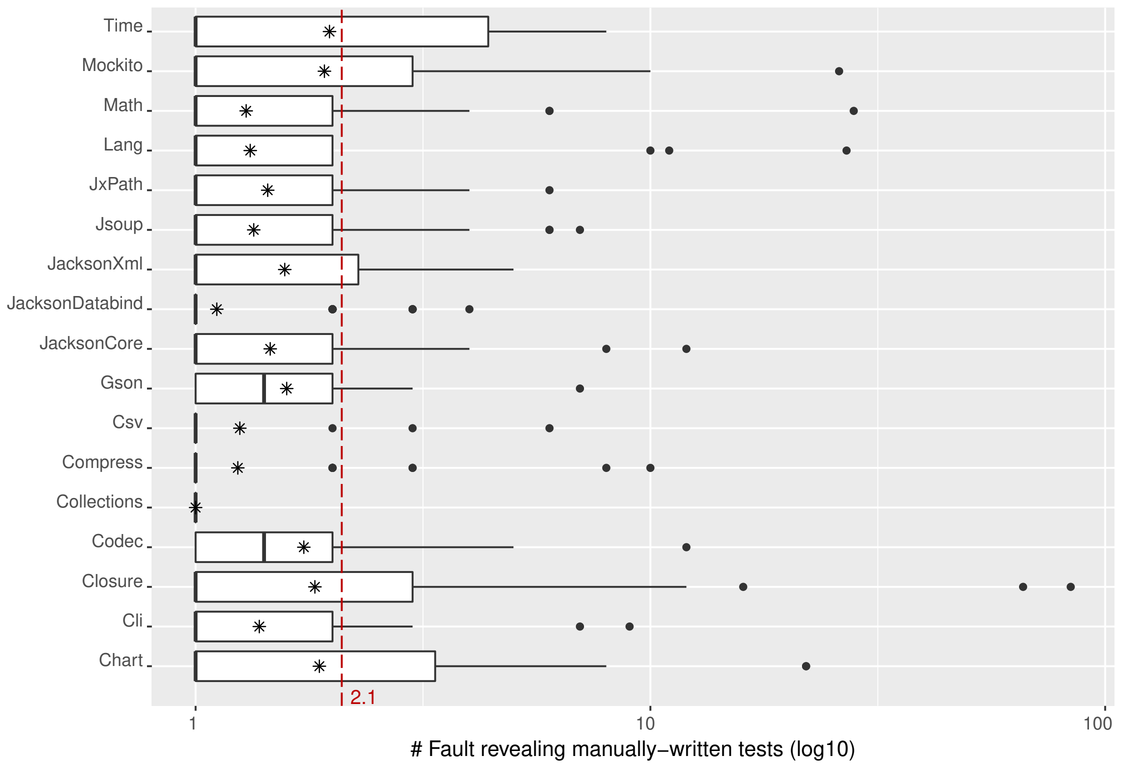


Figure 5.2: Distribution of fault revealing manually-written tests per project.

Figure 5.4 displays the number of bugs triggered per type of test oracle. Note that one bug might be triggered by more than one test, each with a different type of test oracle.

Assert statement: Whereas a test triggers the buggy behavior of a bug through a JUnit assertion, e.g., `assertFalse`. Out of the 1,750 fault revealing manually-written tests in the DEFECTS4J collection, 576 trigger the buggy behavior through an *assert statement*. Regarding number of bugs, 334 bugs are triggered by at least one *assert statement*, where 292 are uniquely triggered by *assert statements*. Chart-26 is the bug with the largest number of *assert statements* (22). In our analysis, we found that *assert statements* could be at the end of a test:

Listing 5.1: Line removed from the test that triggers the buggy behavior of Compress-6.

```
 @Test
 public void testNotEquals() {
    ZipArchiveEntry entry1 = new ZipArchiveEntry("foo");
    ZipArchiveEntry entry2 = new ZipArchiveEntry("bar");
-   assertFalse(entry1.equals(entry2));
 }
```

Or at any line in a test:

Listing 5.2: Lines removed from the test that triggers the buggy behavior of Lang-32.

```
 @Test
 public void testReflectionObjectCycle() {
    ReflectionTestCycleA a = new ReflectionTestCycleA();
    ReflectionTestCycleB b = new ReflectionTestCycleB();
```

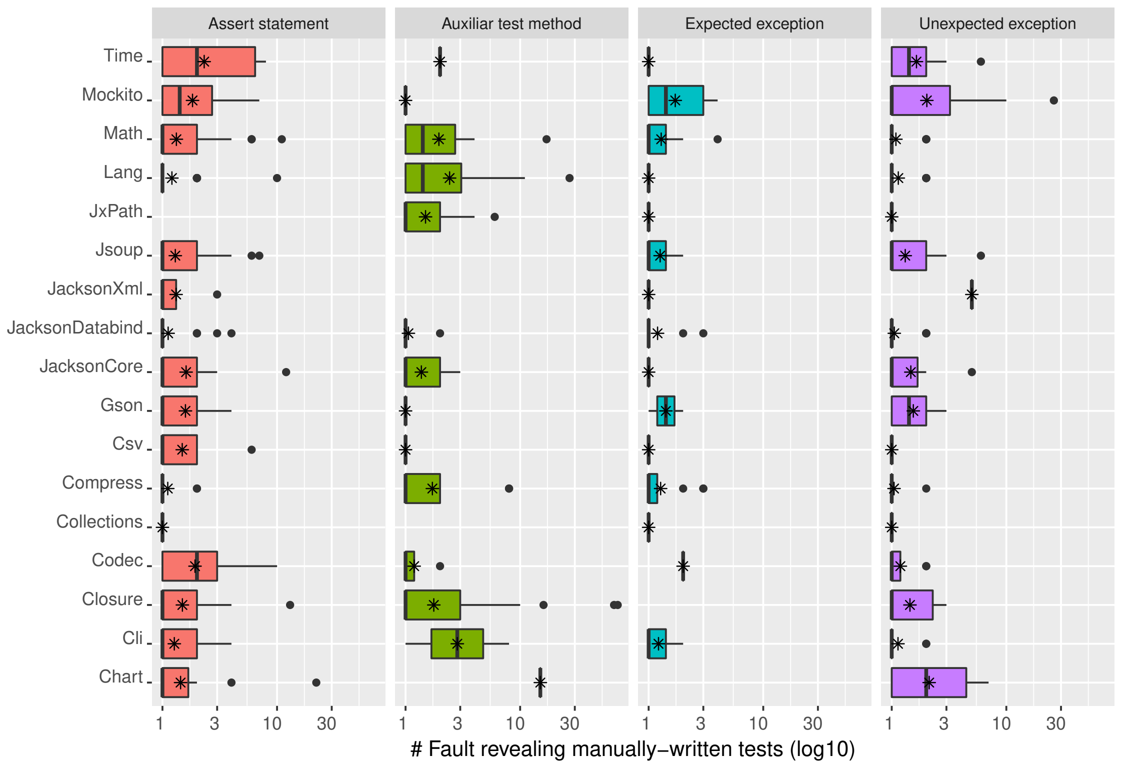


Figure 5.3: Distribution of fault revealing manually-written tests per project and per type of test oracle.

```
    a.b = b;
    b.a = a;

    a.hashCode();
-   assertNull(HashCodeBuilder.getRegistry());
-   b.hashCode();
-   assertNull(HashCodeBuilder.getRegistry());
  }
```

Auxiliar test method: Some tests use auxiliar methods where different conditions are checked for and where the oracles are executed. Out of the 1,750 fault revealing manually-written tests in the DEFECTS4J dataset, 691 trigger the buggy behavior through an *auxiliar test method*. Regarding number of bugs, 241 bugs are triggered by at least one *auxiliar test method*, where 217 are uniquely triggered by *auxiliar test method*. No bug in Jsoup, JacksonXml, and Collections projects is triggered by any auxiliar test method, and Closure-144 is the bug with the largest number of *auxiliar test methods* (71).

For this type of oracle we removed the call to the auxiliar test method in the triggering test. While we acknowledge that this is a simplification of the problem we tackle, we preferred to first investigate the performance of these tools in a simpler scenario. It could, however, be possible to obtain better results if we had generated the oracles for the auxiliar methods instead, maintaining the original call to these methods. It could be interesting to observe the results obtained with this approach in a future work.

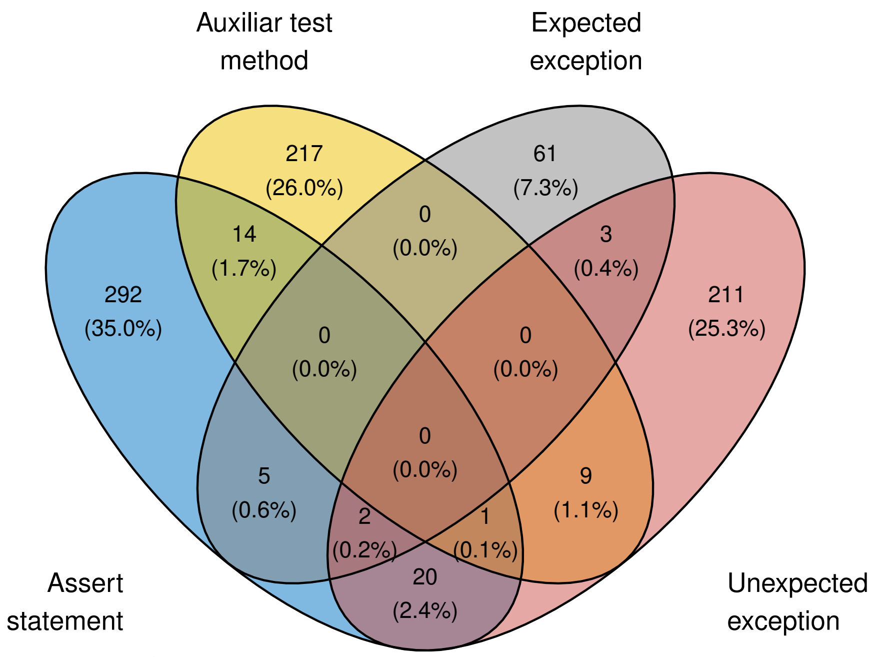


Figure 5.4: Number of bugs triggered per type(s) of test oracle.

Listing 5.3: Lines removed from the test that triggers the buggy behavior of JxPath-6.

```
  @Test
  public void testIterateVariable() throws Exception {
     assertXPathValueIterator(context, "$d", list("a", "b"));
-    assertXPathValue(context, "$d = 'a'", Boolean.TRUE);
-    assertXPathValue(context, "$d = 'b'", Boolean.TRUE);
  }
```

Expected exception: Expected exception oracles execute some sort of action and pass if, an only if, a certain exception is thrown. Out of the 1,750 fault revealing manually-written tests in the DEFECTS4J dataset, 98 trigger the buggy behavior through a *expected exception*. Regarding number of bugs, 71 bugs are triggered by at least one *expected exception*, where 61 are uniquely triggered by *expected exception*. No bug in Chart and Closure projects is triggered by expected exception, and Math-35 and Mockito-4 are the bugs with the largest number of *expected exceptions* (4).

We defined that, for a *try/catch* block to be considered an expected exception oracle, there had to be an action statement (the focal method) and a *fail()* statement in the *try* block. If considered an expected exception, the entire *try/catch* is removed:

Listing 5.4: Lines removed from the test that triggers the buggy behavior of JacksonCore-23.

```
  @Test
  public void testInvalidSubClass() throws Exception
  {
     DefaultPrettyPrinter pp = new MyPrettyPrinter();
-    try {
-       pp.createInstance();
-       fail("Should not pass");
-    } catch (IllegalStateException e) {
```

```
-        verifyException(e, "does not override");
-     }
  }
```

Note that, not all *try/catch* blocks represent an expected exception. The example below shows Compress-17. This test contains a *try/catch*, however, if an exception is thrown, the test will fail.

Listing 5.5: Lines removed from the test that triggers the buggy behavior of Compress-17.

```
  @Test
  public void testCompress197() throws Exception {
     TarArchiveInputStream tar = getTestStream("/COMPRESS-197.tar");
     try {
        TarArchiveEntry entry = tar.getNextTarEntry();
        while (entry != null) {
           entry = tar.getNextTarEntry();
        }
     } catch (IOException e) {
-       fail("COMPRESS-197: " + e.getMessage());
     } finally {
-       tar.close();
     }
  }
```

In JUnit, expected exceptions could also be coded using the *@Test(expected = ...)* annotation. We found 31 cases where the expected exception oracle was a JUnit annotation. In these cases, the entire method executes normally and, once it finishes execution, if the test has not thrown the expected exception, fails. As JUnit does not provide a way to specify what method is expected to throw the exception, it is necessary to have knowledge of the code to know exactly what statement that would be. Because of this, we decided to remove all lines from these test methods, leaving them empty, as well as the '*(expected = ...)*' from the *@Test* annotation.

Listing 5.6: Line removed from the test that triggers the buggy behavior of Csv-8.

```
- @Test(expected = IllegalArgumentException.class)
+ @Test
  public void testDuplicateHeaderElements() {
-    CSVFormat.DEFAULT.withHeader("A", "A").validate();
  }
```

Unexpected exception: In this type of oracle, a test fails due to an exception being thrown during an action in the test. Out of the 1,750 fault revealing manually-written tests in the DEFECTS4J dataset, 385 trigger the buggy behavior through a *unexpected exception*. Regarding number of bugs, 246 bugs are triggered by at least one *unexpected exception*, where 211 are uniquely triggered by *unexpected exception*. Mockito-1 is the bug with the largest number of *unexpected exceptions* (26).

Listing 5.7: Lines removed from the test that triggers the buggy behavior of Compress-20.

```
  @Test
  public void testCpioUnarchiveCreatedByRedlineRpm() throws Exception {
     CpioArchiveInputStream in =
        new CpioArchiveInputStream(new FileInputStream(getFile("redline.cpio")));
     CpioArchiveEntry entry= null;

     int count = 0;
-    while ( (entry = (CpioArchiveEntry) in.getNextEntry())
-        != null) {
-        count++;
-    }
-    in.close();

-    assertEquals(count, 1);
  }
```

Some unexpected exceptions might also occur within a *try/catch* block. Note that, in Listing 5.8, we removed the entire *try/catch* block as it would have been left empty otherwise.

Listing 5.8: Lines removed from the test that triggers the buggy behavior of Jsoup-78.

```
  @Test
  public void handlesEmptyStreamDuringParseRead() throws IOException {
     Connection.Response res = Jsoup.connect(InterruptedServlet.Url)
        .timeout(200)
        .execute();

     boolean threw = false;
-    try {
-        Document document = res.parse();
-        assertEquals("Something", document.title());
-    } catch (IOException e) {
-        threw = true;
-    }
-    assertEquals(true, threw);
  }
```

Miscellaneous: During our procedure, some outliers were found. For these cases, we had to proceed with different strategies, which are described here.

Out of the 835 active bugs in DEFECTS4J, we found two (Chart-4 and Mockito-12) for which the bug is located in the setup method. As this method is invoked by JUnit before any other test method gets executed, removing lines from it could cause other tests to fail. Thus, we did not manage to entirely remove the manually-written test oracles so that every test would pass.

Moreover, some manually-written tests become empty after applying our procedure to remove the manually-written test oracles. In our analysis we found 531 empty tests out of 1696, where Collections is the only project without any empty tests and Closure is the project with the most amount of empty tests (356 out of 545) as well as the highest percentage of empty tests (65.3%), e.g.:

Listing 5.9: Lines removed from the test that triggers the buggy behavior of Gson-11.

```
  @Test
  public void testNumberAsStringDeserialization() {
-    Number value = gson.fromJson("\"18\"", Number.class);
-    assertEquals(18, value.intValue());
  }
```

In some cases, the resulting tests, after applying our procedure, would still fail for another reason. For those cases, we had to follow an additional procedure:

- When a test fails due to a line inside a try block, according to our procedure, all lines inside the finally block (if there is one) are removed. In JacksonDatabind-58, we can not remove the line inside the finally block, as it causes other tests to fail.

Listing 5.10: Patch created to remove the failing line inf JacksonDatabind-58.

```
 @Test
 public void testCauseOfThrowableIgnoral() throws Exception
 {
   final SecurityManager origSecMan = System.getSecurityManager();
   try {
     System.setSecurityManager(new CauseBlockingSecurityManager());
-    _testCauseOfThrowableIgnoral();
   } finally {
     System.setSecurityManager(origSecMan);
   }
 }
```

- When a bug is detected inside a loop, continuing to execute the loop can lead to the test failing in another line. We found three cases where this happened (Closure-74, Lang-40, and Time-25). In these cases, we decided to remove these other failing lines so that the test executes without failing.

Listing 5.11: Patch created to remove the failing lines in Time-25.

```
 @Test
 public void test_getOffsetFromLocal_Moscow_Autumn_overlap_mins()
 {
   for (int min = 0; min < 60; min++) {
     if (min < 10) {
-      doTest_getOffsetFromLocal(10, 28, 2, min,
-          "2007-10-28T02:0" + min + ":00.000+04:00", ZONE_MOSCOW);
     } else {
-      doTest_getOffsetFromLocal(10, 28, 2, min,
-          "2007-10-28T02:" + min + ":00.000+04:00", ZONE_MOSCOW);
     }
   }
 }
```

- In other cases, our procedure might lead to infinite loops. We found two bugs for which this happened (Compress-36 and JacksonCore-4). We took different approaches for each of them. In Compress-36, it was possible to leave one line unchanged, which allows the loop to reach a final state.

Listing 5.12: Patch created for Compress-36 to avoid an infinite loop.

```
 @Test
 public void readEntriesOfSize0() throws IOException {
   final SevenZFile sevenZFile = new SevenZFile(getFile("COMPRESS-348.7z"));
   try {
     int entries = 0;
     SevenZArchiveEntry entry = sevenZFile.getNextEntry();
     while (entry != null) {
       entries++;
-      int b = sevenZFile.read();
-      if ("2.txt".equals(entry.getName()) ||
-          "5.txt".equals(entry.getName()))
-       {
-         assertEquals(-1, b);
-       } else {
-         assertNotEquals(-1, b);
-       }
        entry = sevenZFile.getNextEntry();
      }
-     assertEquals(5, entries);
    } finally {
-     sevenZFile.close();
    }
  }
```

In JacksonCore-4, we did not have this option, therefore, and to avoid adding code (e.g., a `break` statement), in any of the patches, we decided to entirely remove the loop block. We are aware of the problem this raises, however, we thought this would be a preferable approach for the reason described previously.

Listing 5.13: Patch created for JacksonCore-4 to avoid an infinite loop.

```
  @Test
  public void testExpand()
  {
    TextBuffer tb = new TextBuffer(new BufferRecycler());
    char[] buf = tb.getCurrentSegment();

-   while (buf.length < 500 * 1000) {
-     char[] old = buf;
-     buf = tb.expandCurrentSegment();
-     if (old.length >= buf.length) {
-       fail("Expected buffer of " + old.length +
-           " to expand, did not, length now " + buf.length);
-     }
-   }
  }
```

In the end, we generated 1696 patches that remove the statements that cause the tests in DEFECTS4J to fail. Note that, in total, DEFECTS4J had 1,750 failing tests. The difference in these numbers is due to the fact that, in some cases, the failing tests referred to an extended test class.

(2) Augment each test with test oracles generated by either T5

After the process described in (1), the tests present in all of the DEFECTS4J projects should be compilable and execute without failing.

T5, requires an oracle placeholder to know where the generated oracle is to be placed within the source code. We opted to add the line '`// TEST ORACLE`' to the patches, to serve as a placeholder (as demonstrated in Listing 5.14). The use of a comment allows the code to compile and is easy enough to replace by any other placeholder recognized by T5 or any other tool. This way, this dataset can still be used to test any other tool, for example, ATLAS or BART$_{\text{ENG+CODE}}$, in the future.

Listing 5.14: Patch created for Compress-6, containing oracle placeholder.

```
  @Test
  public void testNotEquals() {
      ZipArchiveEntry entry1 = new ZipArchiveEntry("foo");
      ZipArchiveEntry entry2 = new ZipArchiveEntry("bar");
-     assertFalse(entry1.equals(entry2));
+ // TEST ORACLE
  }
```

To be able to provide better oracles, T5 takes as input the test method's source code as well as the source of the focal method being used.

We developed a Java program[1] to help us identify a test's focal method by taking into consideration the strategies described by Qusef et al. [48], Watson et al. [65] and Tufano et al. [62]. Our program takes as input a project's source code path and its tests' path and prints to file the source code of both, a test method and its focal method.

We start by making the assumption that the developers follow JUnit testing's common naming conventions to identify the focal class. According to common practice, the name of a test class is usually composed of the name of the focal class, with the suffix 'Test' (for example, a class that tests `Foo.java` would be named `FooTest.java`). We then loop through the methods called in the test method and remove those that do not belong to the focal class. If a single method remains, then that is our focal method, otherwise, if more than one method fits the criteria, then we attempt to identify the corresponding focal method according to the test method's name, as many times, the test method's name is composed of the name of the focal method plus the prefix/suffix `Test`. If only one method remains, that is the test's focal method, otherwise, if more than one method remains, we select the method that is called last as the focal method.

We are aware this approach is a simplification of the problem and may lead to wrongly identifying a method as being the focal method, however, manually determining each of the test methods' focal method is not feasible.

To increase our chances of detecting the right focal method, we also created another set of patches that, instead of removing the line/oracle that causes a test to fail and every other line after, keeps the line that causes the test to fail and removes every line after that. Doing this means that, if our tool selects the last called method as the focal method, then that method should be the one that made the test fail originally. Note that, in the following examples, we add a '`// TEST ORACLE`' comment at the end of some lines. This is not required, however, helps us to manually identify the line that originally makes a test fail if necessary.

Listing 5.15: Changes made to Codec-2 to keep the buggy behavior.

```
  @Test
  public void testBase64EmptyOutputStream() throws Exception {
    byte[] emptyEncoded = new byte[0];
    byte[] emptyDecoded = new byte[0];
-   testByteByByte(emptyEncoded, emptyDecoded, 76, CRLF);
```

[1]Java tool to find a testś focal method, `https://github.com/jose/meaningful-assert-statements-data/tree/master/tools/test-analysis`

```
-  testByChunk(emptyEncoded, emptyDecoded, 76, CRLF);
+  testByteByByte(emptyEncoded, emptyDecoded, 76, CRLF); // TEST ORACLE
+
  }
```

Listing 5.16: Changes made to Math-106 to keep the buggy behavior.

```
  public void testParseProperInvalidMinus() {
     String source = "2 -2 / 3";
     try {
        Fraction c = properFormat.parse(source);
-       fail("invalid minus in improper fraction.");
+       fail("invalid minus in improper fraction."); // TEST ORACLE
     } catch (ParseException ex) {
        // expected
     }
-    source = "2 2 / -3";
-    try {
-       Fraction c = properFormat.parse(source);
-       fail("invalid minus in improper fraction.");
-    } catch (ParseException ex) {
-       // expected
-    }
  }
```

Moreover, T5 (as well as ATLAS and BART$_{\text{ENG+CODE}}$) expect source code to contain fully-qualified class names (i.e., canonical names) and in a tokenized format. To accomplish this, we used SPOON [47], which analyzes source files and creates a meta-model that allows for a deeper analysis and transformation of the program.

(3) Compile the augmented tests on the faulty version of X

Oracles generated by T5 may not be compilable right away. T5 generates test oracles using only lowercase, therefore, there may be errors in class, method or variable names that must be fixed before we can compile the test suite with the automatically generated oracles.

We developed a Python script[2] to fix the casing of test oracles generated by T5. This script takes as input the test method, the focal method and the test oracle generated by T5 to detect what variables exist in the test method, their type and the focal method's name. Moreover, we also generated JSON files[3] containing all data from Java 8's API that we pass to the script to give it a better chance at correctly fixing the casing of a token.

5.1.2 Metrics

In RQ1 we track and report, for T5: the number of generated test oracles, the number of broken test oracles and the number of failing tests.

[2]Python script to fix oracles generated by T5, https://github.com/jose/meaningful-assert-statements-data/blob/master/oracle-generation/scripts/fix-t5-output.py

[3]Generated JSON files containing Java 8 API's data, https://github.com/jose/meaningful-assert-statements-data/tree/master/utils/java-api

Moreover, we do a false-positive analysis to obtain: the number of false-positives, and the number of faults detected.

A failing test is considered a ‘positive’, however, a ‘positive’ test, does not mean that the test oracle detected a bug and can be one of two things:

- **True-Positive:** The test fails due to the buggy implementation.
- **False-Positive:** The test has flaky test oracle and fails due to a reason unrelated to the bug itself.

To distinguish between these cases, we run the same test on the unit’s buggy and fixed version. If the test fails on both, buggy and fixed versions, we can safely assume the test has an incorrect oracle, and is a False-Positive.

5.1.3 Results

T5 was able to generate a test oracle for every one of the 1696 test methods, even if they were left empty after applying our patch to remove the oracle. We demonstrate a few of these cases here:

Oracles generated by T5 *for empty test methods:* T5 was able to generate oracles for test methods that were left completely empty. Here we show a few of those cases to investigate what kind of oracles can T5 generate in these cases.

- JacksonCore-17. T5 generated one of the most simple oracles possible, verifying that the value ‘true’ is true.

  Listing 5.17: Oracle generated for JacksonCore-17.

  ```
  org. junit. assert. asserttrue ( true )
  ```

- Time-18. T5 verifies whether two class literals are equal, however, it uses the same class in both parameters of the oracle. Moreover, it is worth noting that, most likely, T5 was able to use the ‘gjchronology’ token as it was the return type of the focal method provided.

  Listing 5.18: Oracle generated for Time-18.

  ```
  org. junit. assert. assertequals (
    org. joda. time. chrono. gjchronology. class,
    org. joda. time. chrono. gjchronology. class
  )
  ```

  Listing 5.19: Focal method given for Time-18.

  ```
  public static GJChronology getInstanceUTC() {
    return getInstance(DateTimeZone.UTC, DEFAULT_CUTOVER, 4);
  }
  ```

- Math-5. T5 fails to generate the correct oracle, however, the generated oracle is objectively close to the oracle that had been written by the developers.

Listing 5.20: Oracle generated for Math-5.

```
org. junit. assert. assertequals ( complex. zero, reciprocal ( ) )
```

Listing 5.21: Oracle written by the developers for Math-5.

```
Assert.assertEquals(Complex.ZERO.reciprocal(), Complex.INF);
```

- Codec-16. T5 generated an oracle that is syntactically incorrect. However, it is worth noting that it did try to use, in some way, the focal method that it was given. Moreover, through the use of the 'testcodec' token, T5 may have gotten the idea that this variable existed in the test suite's scope through the name of the test method.

Listing 5.22: Oracle generated for Codec-16.

```
org. junit. assert. assertnotnull ( testcodec. public base32 ( true, null ) )
```

Listing 5.23: Focal method given for Codec-16.

```
public Base32(final boolean useHex, final byte pad) {
  this(0, null, useHex, pad);
}
```

Listing 5.24: Test method for Codec-16.

```
@Test
public void testCodec200() {
  // TEST ORACLE
}
```

Out-of-the-box, none of the oracles generated by T5 compile. The main reasons are: (1) no test oracle includes the '`;`' required by Java at the end of each statement; and (2) T5 generates lowercase test oracles which do not match either the source code under test and test's code. As described in Section 5.1.1, we try to fix the generated oracles through the use of a script.

Despite our efforts to automatically fix the oracles generated by T5, after our post-processing, some oracles still keep badly formatted tokens. These could be variables not defined in the source code, calls to non-existing methods, or oracles with a wrong syntax.

In Listing 5.25, we present a test case from Csv-9 with the post-processed test oracle. As can be seen in Listing 5.26, this test does not compile, as the token '`tomap`' should have been '`toMap`'.

Listing 5.25: Csv-9 test method with automatically generated oracle.

```
@org.junit.Test
public void testToMapWithNoHeader() throws java.lang.Exception {
  final org.apache.commons.csv.CSVParser parser =
      org.apache.commons.csv.CSVParser.parse("a,b",
      org.apache.commons.csv.CSVFormat.newFormat(',','));
  final org.apache.commons.csv.CSVRecord shortRec = parser.iterator().next();
  org.junit.Assert.assertEquals ( 0 , shortRec.tomap().size() );
}
```

Listing 5.26: Stacktrace excerpt from Csv-9's execution with automatically generated oracle.

```
...
[javac] <...>/CSVRecordTest.java:175: error: cannot find symbol
[javac] org.junit.Assert.assertEquals(0, shortRec.tomap().size());
[javac]                                                  ^
[javac] symbol: method tomap()
[javac] location: variable shortRec of type CSVRecord
```

Listing 5.27 and Listing 5.28 show an oracle generated with a wrong syntax and part of the stacktrace from its execution, respectively.

Listing 5.27: Automatically generated oracle for Gson-18.

```
org. junit. assert. assertnotnull ( bigclass. public gson ( ) )
```

Listing 5.28: Stacktrace excerpt from Gson-18's execution with automatically generated oracle.

```
...
[javac] <...>/CollectionTest.java:410: error: <identifier> expected
[javac] org.junit.Assert.assertNotNull ( bigClass.public Gson() );
[javac]                                                   ^
...
```

Despite all of these cases, after the execution of our script, the % of oracles generated by T5 that compile increased to 27.48% (466 oracles out of 1696).

We execute the tests that do compile on the faulty version of the software and, to assess whether a test fails due to a reason unrelated to the bug itself, we then execute these same tests on the fixed version of the software. If the oracle is testing the correct functionality, then it should pass when executed on the version of the software that does not contain the bug. Figure 5.5 reports the number of tests that pass or fail for both versions of the software. Overall, out of the 213 tests that fail in the faulty version, 151 are automatically identified as false-positives, as they also fail in the fixed version.

For the remaining 62 tests, we manually compare the oracle automatically generated by T5 with the oracle written by the developers as well as the stacktraces that originate from executing the tests on the faulty version of the software. When manually evaluating these 62 test oracles, we found that most of the generated oracles (53) are *exactly* the same as those that were manually-written while. For 9 remaining cases, we identified two possibilities: (1) the oracles are different but are correct; and (2) the tests are different and incorrect. We demonstrate these cases here:

Different but correct test oracles: In 5 test cases, despite being semantically different, when comparing the oracles and the context they are executed in, it is simple to say they execute the same verifications:

- Lang-55, in class `org.apache.commons.lang.time.StopWatchTest`, method `testLang315`, the original oracle used `assertTrue` to assess an equality while T5 used `assertEquals` for the same variables.

Listing 5.29: Developer-written oracle for Lang-55.

```
junit.framework.TestCase.assertTrue(suspendTime == totalTime);
```

Listing 5.30: Automatically generated oracle for Lang-55.

```
org.junit.Assert.assertEquals ( suspendTime , totalTime );
```

- Math-37, in class `org.apache.commons.math.complex.ComplexTest`, methods `testTan` and `testTanh`, the original oracle used an extra value for the assert statement. The oracle generated by T5 compares the same variables, however, without the extra value.

Listing 5.31: Developer-written oracle for Math-37.

```
org.apache.commons.math.TestUtils.assertEquals(expected, actual, 1.0E-5);
```

Listing 5.32: Automatically generated oracle for Math-37.

```
org.junit.Assert.assertEquals ( expected , actual );
```

- JacksonDatabind-48, in class `com.fasterxml.jackson.databind.ser.TestFeatures`, method `testVisibilityFeatures`, the original oracle is a `fail` inside an if statement. The oracle generated by T5 tests the opposite of the condition present in the if statement, meaning that, once the execution reaches the generated assert statement, the value being tested is always false, making this assert statement equal to `assertTrue(false)`, or simply a `fail` statement.

Listing 5.33: Developer-written oracle for JacksonDatabind-48.

```
if (props.size() != 1) {
  junit.framework.TestCase.fail((("Should find 1 property, not " + props.size()) +
      "; properties = ") + props);
}
```

Listing 5.34: Automatically generated oracle for JacksonDatabind-48.

```
if (props.size() != 1) {
  org.junit.Assert.assertTrue ( ( ( props.size ( ) ) == 1 ) );
}
```

- Compress-2, in test class `org.apache.commons.compress.archivers.ArTestCase`, method `testArDelete`, the original oracles test whether two variables are equal to 1, while the oracle generated by T5 tests whether those same variables are equal to eachother.

Listing 5.35: Developer-written oracles for Compress-2.

```
junit.framework.Assert.assertEquals(1, copied);
junit.framework.Assert.assertEquals(1, deleted);
```

Listing 5.36: Automatically generated oracle for Compress-2.

```
org.junit.Assert.assertEquals ( copied , deleted );
```

Different and incorrect test oracles: In 4 cases, it is not as easy to distinguish correct from incorrect oracles without knowledge of the system being tested. For this reason, we opted to underestimate the results obtained and identified these cases as false-positives:

- JacksonDatabind-42, in class `com.fasterxml.jackson.databind.deser.TestJdkTypes`, method `testLocale`.

Listing 5.37: Test case with manually-written test oracle for JacksonDatabind-42.

```
@org.junit.Test
public void testLocale() throws java.io.IOException {
  junit.framework.TestCase.assertEquals(new java.util.Locale("en"),
      MAPPER.readValue(quote("en"), java.util.Locale.class));
  junit.framework.TestCase.assertEquals(new java.util.Locale("es", "ES"),
      MAPPER.readValue(quote("es_ES"), java.util.Locale.class));
  junit.framework.TestCase.assertEquals(new java.util.Locale("FI", "fi", "savo"),
      MAPPER.readValue(quote("fi_FI_savo"), java.util.Locale.class));
  // [databind#1123]
  java.util.Locale loc = MAPPER.readValue(quote(""), java.util.Locale.class);
  // TEST ORACLE
  junit.framework.TestCase.assertSame(java.util.Locale.ROOT, loc);
}
```

Listing 5.38: Automatically generated test oracle for JacksonDatabind-42.

```
org.junit.Assert.assertNotNull ( loc );
```

- JacksonDatabind-103, in class `com.fasterxml.jackson.databind.exc.BasicExceptionTest`, method `testLocationAddition`.

Listing 5.39: Test case with manually-written test oracle for JacksonDatabind-103.

```
@org.junit.Test
public void testLocationAddition() throws java.lang.Exception {
  try {
   MAPPER.readValue("{\"value\":\"foo\"}", new
       com.fasterxml.jackson.core.type.TypeReference< java.util.Map<
       com.fasterxml.jackson.databind.BaseMapTest.ABC, java.lang.Integer > >() {
   });
   junit.framework.TestCase.fail("Should not pass");
  } catch (
   com.fasterxml.jackson.databind.exc.MismatchedInputException e
  ) {
   java.lang.String msg = e.getMessage();
   java.lang.String[] str = msg.split(" at \\[");
   if (str.length != 2) {
      // TEST ORACLE
      junit.framework.TestCase.fail((("Should only get one 'at [' marker, got " +
          (str.length - 1)) + ", source: ") + msg);
    }
  }
}
```

Listing 5.40: Automatically generated test oracle for JacksonDatabind-103.

```
org.junit.Assert.assertEquals ( msg , str.length );
```

- Lang-42, class `org.apache.commons.lang.StringEscapeUtilsTest`, method `testEscapeHtmlHighUnicode`.

Listing 5.41: Test case with manually-written test oracle for Lang-42.

```
@org.junit.Test
public void testEscapeHtmlHighUnicode() throws
    java.io.UnsupportedEncodingException {
  // this is the utf8 representation of the character:
  // COUNTING ROD UNIT DIGIT THREE
  // in unicode
  // codepoint: U+1D362
  byte[] data = new byte[] { ((byte) (0xf0)), ((byte) (0x9d)), ((byte) (0x8d)),
      ((byte) (0xa2)) };
  java.lang.String escaped =
      org.apache.commons.lang.StringEscapeUtils.escapeHtml(new
      java.lang.String(data, "UTF8"));
  java.lang.String unescaped =
      org.apache.commons.lang.StringEscapeUtils.unescapeHtml(escaped);
  // TEST ORACLE
  junit.framework.TestCase.assertEquals("High unicode was not escaped correctly",
      "&#119650;", escaped);
}
```

Listing 5.42: Automatically generated test oracle for Lang-42.

```
org.junit.Assert.assertEquals ( escaped , unescaped );
```

- Finally, we identified Jsoup-1, in `org.jsoup.parser.ParserTest`, method `createsStructureFromBodySnippet`.

Listing 5.43: Test case with manually-written test oracle for Jsoup-1.

```
@org.junit.Test
public void createsStructureFromBodySnippet() {
  // the bar baz stuff naturally goes into the body, but the 'foo' goes into root,
      and the normalisation routine
  // needs to move into the start of the body
  java.lang.String html = "foo <b>bar</b> baz";
  org.jsoup.nodes.Document doc = org.jsoup.Jsoup.parse(html);
  // TEST ORACLE
  org.junit.Assert.assertEquals("foo bar baz", doc.text());
}
```

Listing 5.44: Automatically generated test oracle for Jsoup-1.

```
org.junit.Assert.assertEquals ( html , doc.body ( ).html ( ) );
```

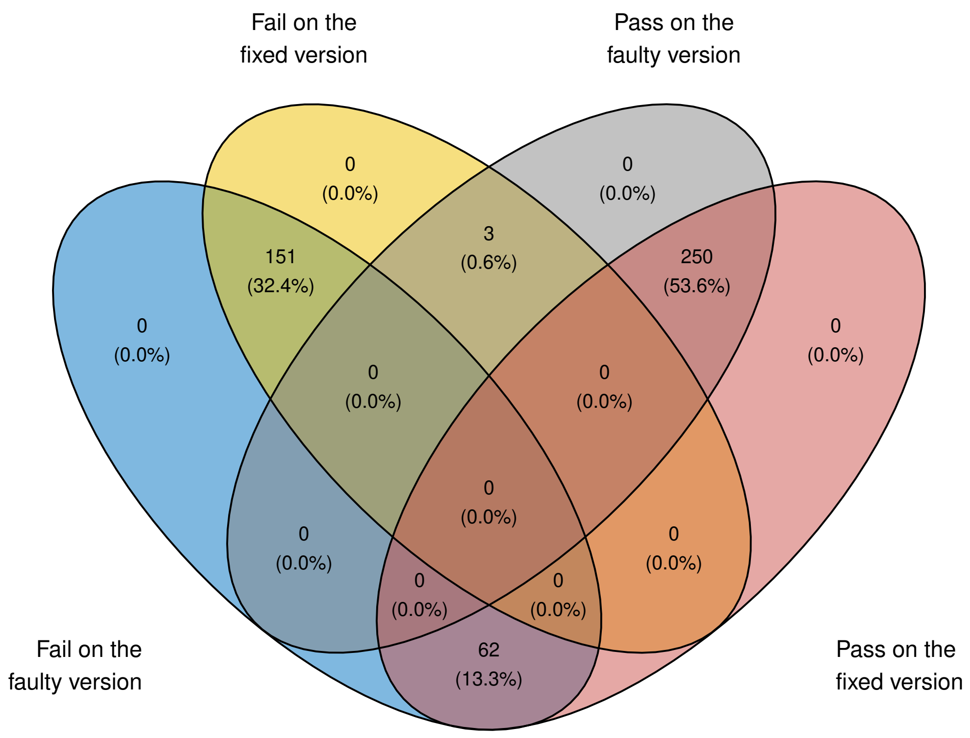


Figure 5.5: Distribution of test oracles that pass/fail on the buggy/fixed version. A test oracle only detects a bug if and only if fails on the buggy version and pass on the fixed version.

Figure 5.6 reports the % as well as the number of oracles generated by T5 per execution result and per project. Figure 5.7 reports the number of faults that can be detected by the oracles generated by T5.

Overall, out of 1696 test oracles generated by T5, we found 58 true-positives, detecting 27 faults out of 835 available on the DEFECTS4J collection. On JacksonXml, no fault was detected out of six; on Math, only 3 faults were detected out of 106; and on Chart, 8 faults were detected out of 26.

It is worth noting that the model used with T5 had only been trained for the generation of assert statements. Therefore, it was expected that this tool could not be able to generate a valid oracle for tests that detected the bug using any of the other techniques mencioned previously. When considering only the bugs in the DEFECTS4J collection that can be detected using an assert statement, out of 334 bugs, T5 was able to detect 8.08%.

RQ1: In a total of 1696 test oracles, 1230 did not compile. Out of the 466 compilable test oracles, 253 do not fail when executed in the faulty version of the software, 155 are false-positives failing due to a reason unrelated to the bug, and only 58 are truly able to detect a fault. T5 is able to detect 27 out of 835 (3.23%) of the faults present in DEFECTS4J.

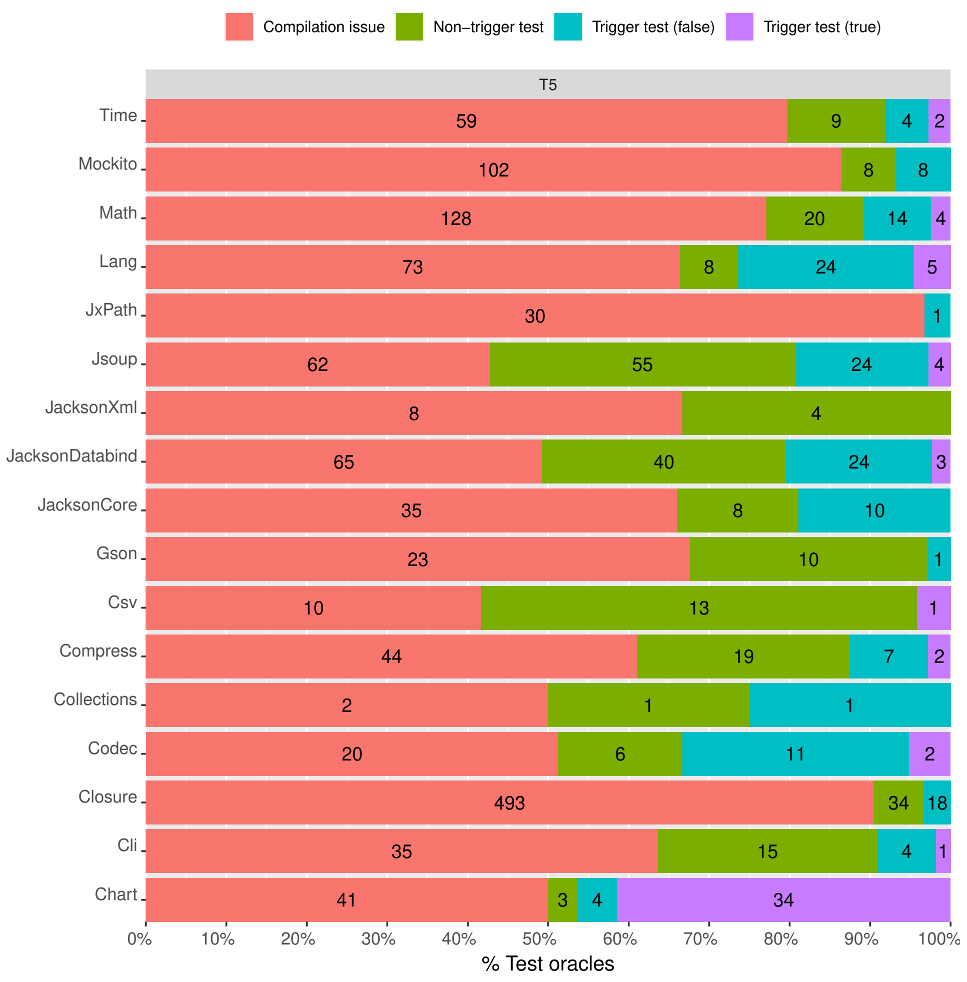


Figure 5.6: Ratio of test oracles' status.

5.2 RQ2: How long does it take to automatically generate fault revealing test oracles?

To assess whether the automated test oracle generation approach might be usable in a real world scenario, in this research question we investigate (using the data generated in the previous research question) the time that T5 spends at generating fault revealing test oracles.

5.2.1 Procedure

For this RQ, we measure the time (in seconds) that T5 takes to generate a test oracle during step (2) of RQ1's procedure.

5.2.2 Metrics

In RQ2 we track and report for the selected automated oracle generation approach (T5) the time, in seconds, it takes to generate a fault revealing test oracle.

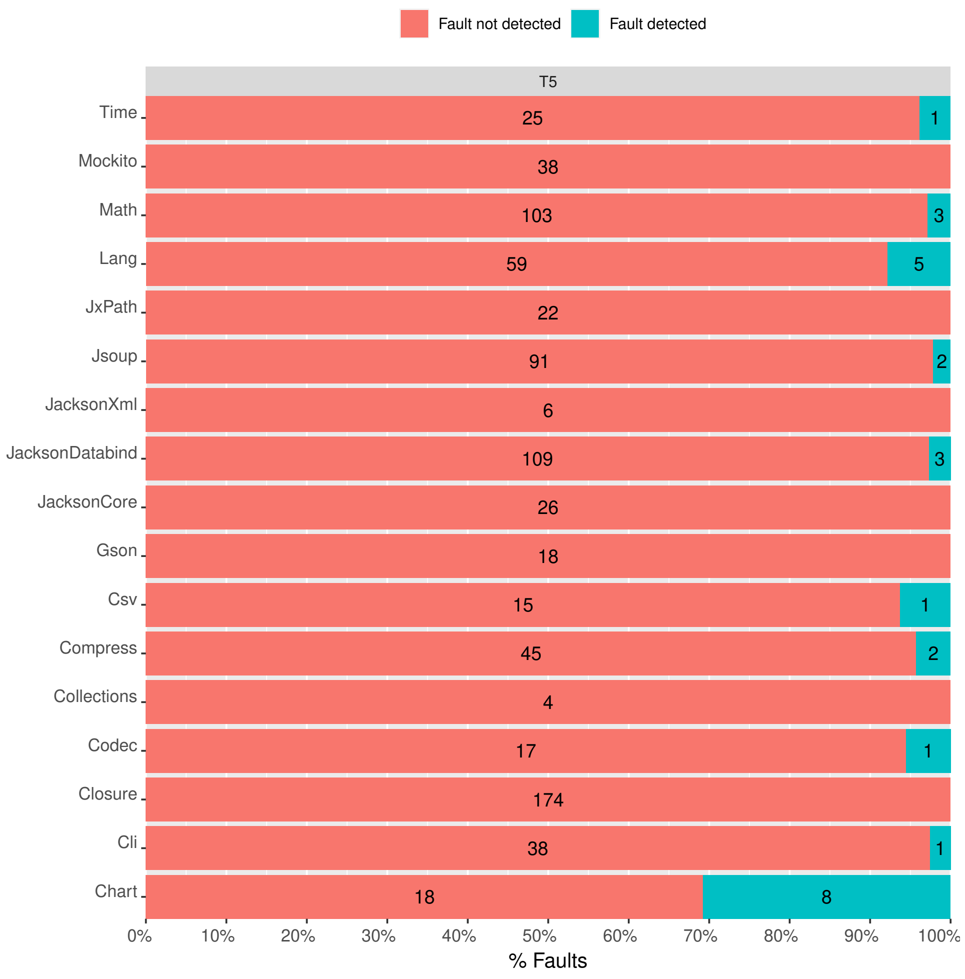


Figure 5.7: Fault detection effectiveness of the automatically generated test oracles for the fault-revealing manually-written test cases in the DEFECTS4J dataset.

5.2.3 Results

Figure 5.8 reports the distribution of T5's execution time per project. Overall, T5 requires 401.3 seconds (6 min. 41 sec.) to generate an oracle. The oracle generated for the test case org.jsoup.parser.ParserTest::handlesNestedImplicitTable in Jsoup-3 took the least time (75.0 seconds). On the other hand, the test case that required the most time was com.google.javascript.jscomp.CrossModuleMethodMotionTest::testIssue600 in Closure-163, and it took 12470.0 seconds for T5 to generate an oracle.

Listing 5.45: Oracle that took T5 the less time to generate

```
org. junit. assert. assertnotnull ( doc )
```

Listing 5.46: Oracle that took T5 the most time to generate

```
org. junit. assert. assertequals ( 1, com. google. javascript. jscomp.
    compilertestcase. static com. google. javascript. jscomp. compilertestcase. static
    com. google. javascript. jscomp. compilertestcase. static com. google. javascript.
    jscomp. compilertestcase. static com. google. javascript. jscomp.
    compilertestcase. static com. google. javascript. jscomp. compilertestcase. ...
```

Table 5.1: Average time required for the generation of a test oracle, for each project.

Project	Time (s)
Chart	341.8
Cli	311.2
Closure	508.7
Codec	434.2
Collections	312.5
Compress	334.8
Csv	340.8
Gson	372.8
JacksonCore	342.8
JacksonDatabind	329.7
JacksonXml	325.5
Jsoup	309.3
JxPath	370.5
Lang	451.0
Math	648.2
Mockito	454.3
Time	634.6
Median	342.8
Average	401.3

As shown in Listing 5.45, the oracle that took T5 the least time to generate was an assertNotNull statement. The fact that this is one of the simplest oracles that can be generated may explain the short amount of time required. On the other hand, Listing 5.46 shows the oracle that took the longest time to be generated. The long time required for this case may be due to an error during the generation of the oracle, as it seems that T5 entered a loop while attempting to generate this oracle.

When analysing the 58 fault-revealing test oracles, we find that their average time to be generated was 331.1 seconds. The shortest time required to generate a fault-revealing test oracle was 103.0 seconds (for org.jsoup.integration.ConnectTest::testBinaryResultThrows in Jsoup-91), while the longest time required was 553.0 seconds (for org.apache.commons .csv.CSVPrinterTest::testMySqlNullOutput in Csv-13).

RQ2: On average, T5 took 401.3 seconds to generate an oracle for each of the test cases and 331.1 seconds when considering only the oracles that were capable of detecting a bug.

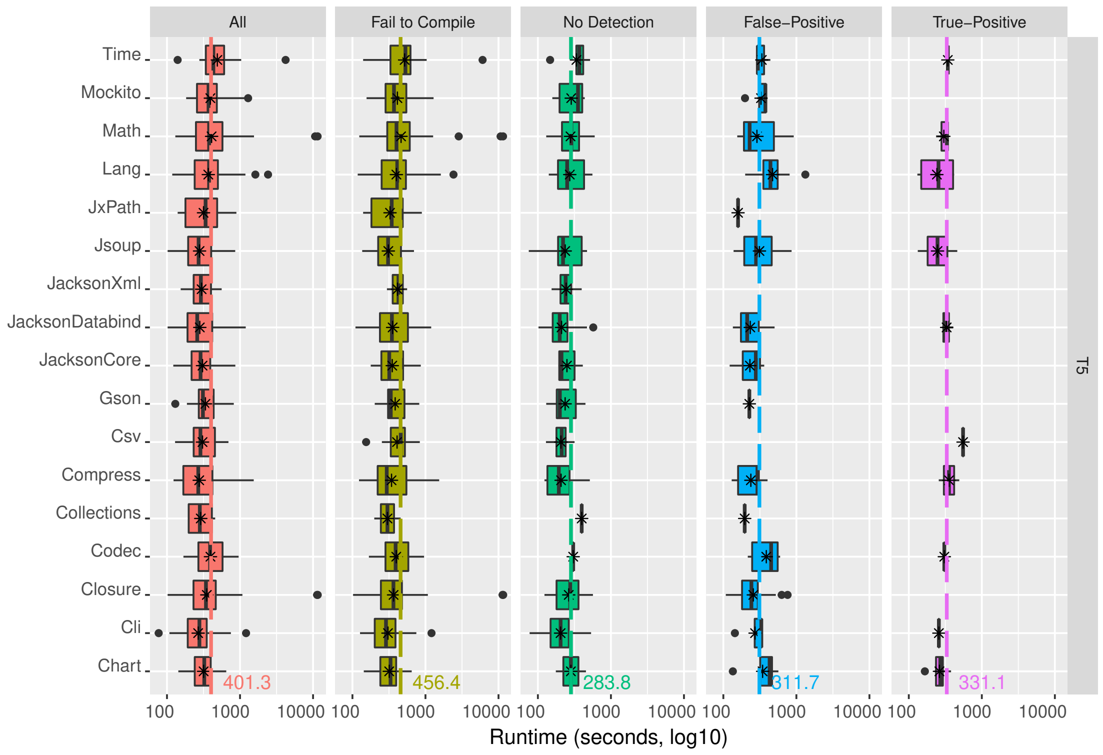


Figure 5.8: Distribution of runtimes.

5.3 Summary

Manually-written test oracles must first be removed from fault-revealing test methods so that we can replace them with automatically generated oracles and execute our evaluation. In this chapter, we describe the procedure we have followed for each of the four identified types of test oracles (assert statement, auxiliar test method, expected exception and unexpected exception) as well as the results obtained when executing our evaluation.

Regarding the evaluated tool to automatically generate test oracles (T5), despite its results when comparing generated test oracles with test oracles written by developers, this tool still fails to achieve a good fault detection rate in real, manually-written, test cases. Our evaluation yielded the following key results:

- Out-of-the-box, none of the generated test oracles compile, as T5 generates oracles only in lower case.
- After a simple post-processing, out of 1696 test oracles, 466 compile and only 58 of these manage to find the real fault.
- Out of the 58 identified true-positives, T5 generated the exact same oracle as the developer in 53 cases.
- Out of the 835 existent bugs in DEFECTS4J, T5 was able to detect 27, i.e., 3.23% of the bugs.

Moreover, when analysing the time required to generate test oracles, T5 took, on average, 401.3 seconds (6 minutes and 41 seconds) to generate an oracle and 331.1 seconds (5 minutes and 31 seconds) to generate a fault-revealing test oracle. It is worth noting that, real test suites usually contain many test methods, meaning that, depending on the size of the test suite, when using T5, one could expect it to take many hours to generate an oracle for each of the methods in the test suite.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Automated oracle generation tools are generally measured in comparison with another tool and on how close the generated oracles are to human-written ones, while automated test suite generation tools are generally measured in terms of code coverage achieved. This work presents an empirical evaluation on the ability of an automated test oracle generation tool to detect real faults in the DEFECTS4J collection. We do this by integrating automatically generated test oracles with manually written test cases and measuring the performance of the generated oracles on their fault detection rate.

The results show that: (1) T5 is able to detect 27 out of the 835 (3.23%) faults present in DEFECTS4J; (2) out of the 466 compilable test oracles, there were 155 false-positives and only 58 are truly able to detect a fault; and, (3) on average, T5 required 401.3 seconds to generate an oracle for each of the test cases.

Moreover, generating compilable oracles remains a problem (out of 1696 test oracles, 1230 did not compile), mainly for two reasons: (1) the tool still needs some work, as sometimes, it appears T5 gets stuck in a loop while generating the oracles; and (2) our procedure to fix the generated oracles still fails in many cases, as it does not have knowledge of all available variables, methods and classes in the system being tested.

As described in Section 4.2.2, some tools were considered to be unusable due various reasons. We hope that these challenges are addressed in future research and that our data can serve to evaluate the progress of research on automated test oracle generation.

6.2 Future Work

In the future, a more in-depth evaluation regarding the state-of-the-art on automated test oracle generation could be done, for example, by: (1) using more tools capable of either automatically generating oracles, or augmenting existing tests; and (2) expanding our dataset to the rest of the test methods in DEFECTS4J, this would provide us some insight into how these tools behave when generating oracles for tests that do not cover the faulty behavior. Moreover, it would be interesting to re-execute our experiments with the new T5 [41] and analyze whether it improves our results in both, ablity to generate compilable oracales and the effectiveness of these oracles. Furthermore, tools like ATLAS, BART$_{\text{ENG+CODE}}$ and T5 can generate more than one oracle for each test method. It would be worth exploring whether their fault-detection rate increases according to how many oracles are generated.

It could also be useful to analyze how effective these tools are when generating oracles for automatically-generated test suites. These test suites can be generated, for example, with state-of-the-art tools like EVOSUITE [22] or RANDOOP [45].

In the future, we aim to investigate two other research questions:

- **RQx: How effective are automatically generated tests augmented with automatically generated test oracles at revealing real faults?** In this research question, automated test generation tools, along with automated oracle generation tools will be used on the faulty version of the software. This is similar to the evaluation done by Dinella et al. [17] and will allow us to better evaluate how the various approaches perform in a non-regression testing scenario.

- **RQy: How effective are automatically generated tests augmented with automatically generated test oracles at revealing regression faults?** As in the previous research question, we will employ automated test generation tools along with automated oracle generation tools, but in a regression testing scenario — the traditional scenario where tools like EVOSUITE and RANDOOP are used.

To answer these research questions we aim to follow the following procedure:

1. For each fault X, automatically generate tests for the faulty and fixed version of X (for RQx and RQy, respectively) by running EVOSUITE or RANDOOP. Note that these tools are usually executed in a regression testing scenario and generate regression test oracles (e.g., [56]). In RQx, regression testing is not performed, therefore, these tools can be executed with their procedure to generate regression oracles disabled.

2. Compile the automatically generated tests on both, the faulty and the fixed versions of X. Any test that does not compile is considered broken and therefore removed.

3. Execute the automatically generated tests on the version of X that they were generated on. At this point, any test that fails when executed on the program version that it was generated on is considered flaky and therefore removed.

4. Collect the code coverage of all generated tests that are not broken or flaky.

5. Check whether the generated tests exercise any faulty line of code If no test exercises any faulty line of code, no test oracle could ever detect the fault and therefore those tests are discarded.

6. Augment the generated tests that do exercise the faulty code with automatically generated test oracles, for example, with tools like T5 [49, 40], BART$_{\text{ENG+CODE}}$ [61], ATLAS [65], TOGA [17], or DSPOT [14]. Note that, for RQy, oracles generated by EVOSUITE, using ALL$^{\alpha}$ and MUTATION$^{\alpha}$ techniques, and RANDOOP, using the REGRESSION$^{\beta}$ approach can also be used.

7. Compile the augmented tests and remove any test oracle that does not compile. This is a safety step as tools might generated uncompilable test oracles.

8. Execute the augmented tests on the faulty version of X to assess its fault detection capability. If no test fails on the faulty version, the fault is considered undetected. Otherwise, if at least one test fails, we then check for false positives.

9. Check whether the tests that fail on the faulty version of X fail due to no other reason than the fault itself. This can be done by executing the tests on the fixed version, as in DEFECTS4J, the difference between the buggy and fixed versions is a minimal patch, therefore, if a test fails on the fixed version, then it is due to some reason other than the bug itself. Moreover, one can manually compare the stacktraces of the manually-written test oracles in the manually-written tests and the stacktrace of the automatically generated test oracles in the automatically generated tests.

We have already modified both EVOSUITE and RANDOOP so that it is possible to:

1. Generate the same test suites to be used for several oracle generation techniques, being only different in the oracles used for each test case. This is particularly important for RQy, where we will use these tools to generate test suites containing oracles generated by them, as well as test suites with no oracles. Although both of these tools claim to be deterministic to some degree, there is always some randomization to the generated test suites. Therefore, generating test suites several times for every selected approach, could mean different test suites, and, therefore, an unfair base of compasison, as some of these test suites could present a better setup for a fault to be detected than others. For this, in both tools, the modified version of EVOSUITE and RANDOOP generates a sequence of statements as the test prefix, then clone this sequence as many times as the number of oracle generation strategies selected and, finally, generate oracles for each of the tests.

2. Generate test suites containing oracle placeholders. For this, a *placeholder strategy* was added in both, EVOSUITE and RANDOOP, which generates placeholder oracles

as follows:

No Exception: In these scenarios the test's action statement does not throw an exception. Generally a statement in the form *assert** is used.

Listing 6.1: Example of a normal execution test method with a placeholder oracle.

```
@Test
public void test1() {
   Foo f = new Foo();
   f.bar();
   // TEST ORACLE
}
```

Expected Exception: In these scenarios the test's action statement is expected to throw an exception. Generally a try/catch block is used.

Note that, as we have done in Chapter 5, in this case, maybe the correct would be to completely remove the try/catch block and let the oracle generation tools choose what oracle to generate. It is also worth noting that, if done this way, it will be harder to identify the focal method, as it will be removed along with the entire try block.

Listing 6.2: Example of an expected exception test method with a placeholder oracle.

```
@Test
public void test2() {
   Foo f = new Foo();
   try {
      f.fooBar();
      // TEST ORACLE
   } catch (Exception e) {
      // pass
   }
}
```

For these two new research questions, we will track and report, for each automated test generation tool and automated oracle generation approach: (1) the number of generated tests, (2) the number of broken and flaky tests, (3) the code coverage of the generated tests, (4) the number of tests that exercise the faulty code, (5) the number of generated test oracles, (6) the number of broken test oracles, (7) the number of false-positives, and (8) the number of true-positives (i.e., the number of faults detected). Further more, we will also conduct a statistical analysis on whether there is an automated oracle generation approach that performs statistically better/worse than any other approach at generating fault revealing test oracles on automatically generated tests. A few examples of comparisons that we plan to investigate are: EVOSUITE + ATLAS vs. RANDOOP + ATLAS, EVOSUITE + ATLAS vs. EVOSUITE + T5, or RANDOOP + REGRESSION$^{\beta}$ vs. RANDOOP + TOGA.

Bibliography

[1] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272, 2017. doi: 10.1109/ICSE-SEIP.2017.27.

[2] Andrea Arcuri, José Campos, and Gordon Fraser. Unit test generation during software development: Evosuite plugins for maven, intellij and jenkins. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 401–408, 2016. doi: 10.1109/ICST.2016.44.

[3] Association for Computing Machinery. Artifact Review and Badging Version 1.1, August 2020. URL `https://www.acm.org/publications/policies/artifact-review-and-badging-current`.

[4] 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, 2015. doi: 10.1109/TSE.2014.2372785.

[5] Kent Beck. *JUnit Pocket Guide*. O'Reilly Media, Inc., 2004.

[6] Kent Beck, Erich Gamma, David Saff, and Kris Vasudevan. JUnit: unit testing framework, 2022. URL `https://junit.org`.

[7] 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, page 242–253, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356992. doi: 10.1145/3213846.3213872. URL `https://doi.org/10.1145/3213846.3213872`.

[8] Ronyérison Braga, Pedro Santos Neto, Ricardo Rabêlo, José Santiago, and Matheus Souza. A machine learning approach to generate test oracles. In *Proceedings*

of the XXXII Brazilian Symposium on Software Engineering, SBES '18, page 142–151, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365031. doi: 10.1145/3266237.3266273. URL `https://doi.org/10.1145/3266237.3266273`.

[9] D. Britz, A. Goldie, T. Luong, and Q. Le. Massive Exploration of Neural Machine Translation Architectures. *ArXiv e-prints*, March 2017.

[10] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 104:207–235, 2018. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2018.08.010. URL `https://www.sciencedirect.com/science/article/pii/S0950584917304858`.

[11] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases, 2020.

[12] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1), jan 2018. ISSN 0360-0300. doi: 10.1145/3143561. URL `https://doi.org/10.1145/3143561`.

[13] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., GBR, 1972. ISBN 0122005503.

[14] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. Automatic test improvement with DSpot: a study with ten mature open-source projects. *Empirical Software Engineering*, 24(4):2603–2635, apr 2019. doi: 10.1007/s10664-019-09692-y. URL `https://doi.org/10.1007%2Fs10664-019-09692-y`.

[15] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting Repairs for Broken Unit Tests. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 433–444, Nov 2009. doi: 10.1109/ASE.2009.17.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[17] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu Lahiri. Toga: A neural method for test oracle generation. In *ICSE 2022*. ACM, May 2022. URL `https://www.microsoft.com/en-us/research/publication/toga-a-neural-method-for-test-oracle-generation/`.

[18] Zhiyu Fan. A Systematic Evaluation of Problematic Tests Generated by EvoSuite. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE '19, page 165–167. IEEE Press, 2019. doi: 10.1109/ICSE-Companion.2019.00068. URL `https://doi.org/10.1109/ICSE-Companion.2019.00068`.

[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[20] Afonso Fontes and Gregory Gay. Using machine learning to generate test oracles: A systematic literature review. In *Proceedings of the 1st International Workshop on Test Oracles*, TORACLE 2021, page 1–10, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386265. doi: 10.1145/3472675.3473974. URL `https://doi.org/10.1145/3472675.3473974`.

[21] Afonso Fontes, Gregory Gay, Francisco Gomes de Oliveira Neto, and Robert Feldt. Automated support for unit test generation: A tutorial book chapter, 2021.

[22] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304436. doi: 10.1145/2025113.2025179. URL `https://doi.org/10.1145/2025113.2025179`.

[23] Gordon Fraser and Andrea Arcuri. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2), dec 2014. ISSN 1049-331X. doi: 10.1145/2685612. URL `https://doi.org/10.1145/2685612`.

[24] Gordon Fraser and Andrea Arcuri. 1600 Faults in 100 Projects: Automatically Finding Faults While Achieving High Coverage with EvoSuite. *Empirical Softw. Engg.*, 20(3):611–639, jun 2015. ISSN 1382-3256. doi: 10.1007/s10664-013-9288-2. URL `https://doi.org/10.1007/s10664-013-9288-2`.

[25] Gordon Fraser and José Miguel Rojas. *Software Testing*, pages 123–192. Springer International Publishing, Cham, 2019. ISBN 978-3-030-00262-6. doi: 10.1007/978-3-030-00262-6_4. URL `https://doi.org/10.1007/978-3-030-00262-6_4`.

[26] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing*

and Analysis, ISSTA '10, page 147–158, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605588230. doi: 10.1145/1831708.1831728. URL `https://doi.org/10.1145/1831708.1831728`.

[27] 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, page 213–224, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931061. URL `https://doi.org/10.1145/2931037.2931061`.

[28] K. Gwet. Computing inter-rater reliability and its variance in the presence of high agreement. *British Journal of Mathematical and Statistical Psychology*, 61(1):29–48, 2008.

[29] W.E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, SE-4(4):293–298, 1978. doi: 10.1109/TSE.1978.231514.

[30] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2019. URL `https://arxiv.org/abs/1909.09436`.

[31] Ali Reza Ibrahimzada, Yigit Varli, Dilara Tekinoglu, and Reyhaneh Jabbarvand. Perfect is the enemy of test oracle. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 70–81, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549086. URL `https://doi.org/10.1145/3540250.3549086`.

[32] Agitar Technologies Inc. AgitarOne JUnit Generator. `http://www.agitar.com/solutions/products/automated_junit_generation.html`, 2014. Last visited on 2021-12-14.

[33] 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, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2628055. URL `https://doi.org/10.1145/2610384.2628055`.

[34] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer, 2020.

[35] Claus Klammer and Albin Kern. Writing unit tests: It's now or never! In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–4, 2015. doi: 10.1109/ICSTW.2015.7107469.

[36] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019.

[37] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *AI Open*, 3:111–132, 2022. ISSN 2666-6510. doi: https://doi.org/10.1016/j.aiopen.2022.10.001. URL `https://www.sciencedirect.com/science/article/pii/S2666651022000146`.

[38] 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. doi: 10.1109/TSE.2013.46.

[39] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[40] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347, 2021. doi: 10.1109/ICSE43902.2021.00041.

[41] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Using transfer learning for code-related tasks, 2022. URL `https://arxiv.org/abs/2206.08574`.

[42] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Using Transfer Learning for Code-Related Tasks. *IEEE Transactions on Software Engineering*, pages 1–20, 2022. doi: 10.1109/TSE.2022.3183297.

[43] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias. Evospex: An evolutionary algorithm for learning postconditions, 2021.

[44] Glenford J. Myers, Tom Badgett, and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2012.

[45] 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, page 815–816, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938657. doi: 10.1145/1297846.1297902. URL https://doi.org/10.1145/1297846.1297902.

[46] Fabrizio Pastore, Leonardo Mariani, and Gordon Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 342–351, 2013. doi: 10.1109/ICST.2013.13.

[47] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015. doi: 10.1002/spe.2346. URL https://hal.archives-ouvertes.fr/hal-01078532/document.

[48] Abdallah Qusef, Rocco Oliveto, and Andrea De Lucia. Recovering traceability links between unit tests and classes under test: An improved method. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010. doi: 10.1109/ICSM.2010.5609581.

[49] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.*, 21(1), jan 2020. ISSN 1532-4435.

[50] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. A Detailed Investigation of the Effectiveness of Whole Test Suite Generation. *Empirical Software Engineering*, 2016.

[51] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA, 1986. ISBN 026268053X.

[52] Arvinder Saini. How much do bugs cost to fix during each phase of the sdlc?, Jan 2017. URL https://www.synopsys.com/blogs/software-security/cost-to-fix-bugs-during-each-sdlc-phase/. [Online; accessed 19-January-2022].

[53] Amanda Schwartz, Daniel Puckett, Ying Meng, and Gregory Gay. Investigating faults missed by test suites achieving high code coverage. *Journal of Systems*

and Software, 144:106–120, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.06.024. URL `https://www.sciencedirect.com/science/article/pii/S0164121218301201`.

[54] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016. doi: 10.1109/TSE.2016.2532875.

[55] 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, page 1038–1041, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2803196. URL `https://doi.org/10.1145/2786805.2803196`.

[56] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211, 2015. doi: 10.1109/ASE.2015.86.

[57] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Comput. Surv.*, 55(6), dec 2022. ISSN 0360-0300. doi: 10.1145/3530811. URL `https://doi.org/10.1145/3530811`.

[58] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *CoRR*, abs/1812.08693, 2018.

[59] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, 2019.

[60] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context, 2020. URL `https://arxiv.org/abs/2009.05617`.

[61] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating Accurate Assert Statements for Unit Test Cases using Pretrained Transformers. *CoRR*, abs/2009.05634, 2020. URL `https://arxiv.org/abs/2009.05634`.

[62] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context, 2021.

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[64] Ham Vocke. The practical test pyramid, Feb 2018. URL `https://martinfowler.com/articles/practical-test-pyramid.html`. [Online; accessed 08-December-2021].

[65] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1398–1409, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380429. URL `https://doi.org/10.1145/3377811.3380429`.

[66] Hillel Wayne. Metamorphic testing, Mar 2019. URL `https://www.hillelwayne.com/post/metamorphic-testing/`. [Online; accessed 29-November-2021].

[67] Robert White and Jens Krinke. Testnmt: Function-to-test neural machine translation. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering*, NL4SE 2018, page 30–33, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360555. doi: 10.1145/3283812.3283823. URL `https://doi.org/10.1145/3283812.3283823`.

[68] Robert White and Jens Krinke. Reassert: Deep learning for assert generation. *CoRR*, abs/2011.09784, 2020. URL `https://arxiv.org/abs/2011.09784`.

[69] Wikipedia contributors. Metamorphic testing — Wikipedia, the free encyclopedia, 2021. URL `https://en.wikipedia.org/w/index.php?title=Metamorphic_testing&oldid=1039383920`. [Online; accessed 27-November-2021].

[70] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012.

[71] Pomin Wu. Test your machine learning algorithm with metamorphic testing, Nov 2017. URL `https://medium.com/trustableai/testing-ai-with-metamorphic-testing-61d690001f5c`. [Online; accessed 29-November-2021].

[72] Tao Xie. Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, pages 380–403, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-35727-8.

[73] Runze Yu, Youzhe Zhang, and Jifeng Xuan. MetPurity: A Learning-Based Tool of Pure Method Identification for Automatic Test Generation. In *Proceedings of the ACM/IEEE 35th International Conference on Automated Software Engineering*, ASE '20, New York, NY, USA, 2020. Association for Computing Machinery.